3

Handling the UI State in Jetpack Compose and Using Hilt

All Android applications display the state to users, which helps inform users on what the outcome is and when. The **state** in an Android application is any value that changes over time, and a good example is a toast that shows a message when there is an error. In this chapter, readers will learn how to handle the UI state better with the new Jetpack library.

It is fair to say with great power comes great responsibility, and managing the state of any Composable component requires a distinct approach compared to using the older way of building Android views, or as many might call it, the imperative way. This means Jetpack's library, Compose, is entirely different from XML layouts.

Handling the UI state in the XML **View System** is very straightforward. The process entails setting the properties of the views to reflect the current state – that is, showing or hiding the views accordingly. For instance, when loading data from an API, you can hide the loading view, show the content view, and populate it with the desired views.

In Compose, however, it is impossible to change a Composable component once the application has drawn it. You can, however, change the values passed to each Composable by changing the state each Composable receives. Hence, learning about managing the state better when building robust Android applications will be handy.

In this chapter, we'll be covering the following recipes:

- Implementing **Dependency Injection** (**DI**) with Jetpack Hilt
- Implementing `ViewModel` classes and understanding the state in Compose
- Implementing Compose in existing an XML layout-based project
- Understanding and handling recomposition in Jetpack Compose
- Writing UI tests for your Compose views
- Writing tests for your `ViewModel` classes

Technical requirements

The complete source code for this chapter can be found at **https://github.com/PacktPublishing/Modern-Android-13-Development-Cookbook/tree/main/chapter_three**.

Implementing DI with Jetpack Hilt

In object-oriented programming, DI is vital. Some people use it, and some prefer not to use it for their own reasons. However, DI is the practice of designing objects in a manner where they receive instances of the object from other pieces of code instead of constructing them internally.

If you know of the SOLID principles, you know their primary goal is to make software design easier to maintain, read, test, and build upon. In addition, DI helps us follow some of the SOLID principles. The dependency inversion principle allows the code base to be easily expanded and extended with new functionalities and improves reusability. In Modern Android Development, DI is essential, and we will implement it in our application in this recipe.

There are different types of libraries that you can use in Android for DI, such as Koin, Dagger, and Hilt; Hilt harnesses the power of Dagger and benefits from compile-time correctness, good runtime performance, Android studio support, and scalability. For this recipe, we will use Hilt, which provides containers for every Android class in our project and automatically manages their life cycle.

Getting ready

Just like in previous recipes, we will use the project we have used in previous recipes to add DI.

How to do it...

Hilt uses Java features; make sure your project is in the **app/build.gradle**, and you have the following compile options:

```
android {
    ...
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_11
        targetCompatibility JavaVersion.VERSION_11
    }
}
```

This is already added automatically, but make sure you check that you have it just in case. Let's get started:

1. First, we must add the **Hilt-android-gradle-plugin** plugin into our project's root file, **build.gradle(Project:SampleLogin)**:

```
plugins {
    id 'com.google.dagger.Hilt.android' version '2.44'
apply false
}
```

2. Then, in our **app/build.gradle** file, add these dependencies, and sync the project. It should run without any issues:

```
plugins {
    id 'kotlin-kapt'
    id 'dagger.Hilt.android.plugin'
}
dependencies {
    implementation "com.google.dagger:Hilt-
        android:2.44"
    kapt "com.google.dagger:Hilt-compiler:2.44"
}
```

3. Now, let's go ahead and add the **Application** class. All apps that use Hilt must have an **Application** class annotated with **@HiltAndroidApp**, and we need to call

the **`Application`** class that we create in **`Manifest`**:

```
@HiltAndroidApp
class LoginApp : Application()
```

4. In our **`Manifest`** folder, let's add **`LoginApp`**:

```
<application
    android:name=".LoginApp"
  ...
```

5. Now that we have the setup done, we need to start working with Hilt by adding the required annotations to our class. In **`MainActivity.kt`**, we need to add the **`@AndroidEntryPoint`** annotation:

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
  ...
```

6. Let's go ahead and display what we did by running the **`./gradlew :app:dependencies`** command, and we will see something similar to *Figure 3.1*.

```
madonasyombua@Madonas-MacBook-Pro Sample Login % ./gradlew :app:dependencies
Starting a Gradle Daemon, 1 incompatible Daemon could not be reused, use --status for details

> Task :app:dependencies

------------------------------------------------------------
Project ':app'
------------------------------------------------------------

_agp_internal_javaPreCompileDebugAndroidTest_kaptClasspath
\--- com.google.dagger:hilt-compiler:2.42
     +--- com.google.dagger:dagger:2.42
     |    \--- javax.inject:javax.inject:1
     +--- com.google.dagger:dagger-compiler:2.42
     |    +--- com.google.dagger:dagger:2.42 (*)
     |    +--- com.google.dagger:dagger-producers:2.42
     |    |    +--- com.google.dagger:dagger:2.42 (*)
     |    |    +--- com.google.guava:failureaccess:1.0.1
     |    |    +--- com.google.guava:guava:31.0.1-jre
     |    |    |    +--- com.google.guava:failureaccess:1.0.1
     |    |    |    +--- com.google.guava:listenablefuture:9999.0-empty-to-avoid-conflict-with-guava
     |    |    |    +--- com.google.code.findbugs:jsr305:3.0.2
     |    |    |    +--- org.checkerframework:checker-qual:3.12.0
     |    |    |    +--- com.google.errorprone:error_prone_annotations:2.7.1
     |    |    |    \--- com.google.j2objc:j2objc-annotations:1.3
     |    |    +--- javax.inject:javax.inject:1
     |    |    \--- org.checkerframework:checker-compat-qual:2.5.5
```

Figure 3.1 – Dagger Hilt dependency tree

You can also view the dependency in Android Studio. That is by clicking on the **Gradle** tab on the right-hand side and selecting

expand:yourmodule | **Tasks** | **android**. Then, finally, double-click on **androidDependencies** to run it.

Finally, compile and run the project; it should run successfully.

How it works...

`@HiltAndroidApp` triggers Hilt's code generation, including a base class for our application, which acts as the application-level dependency container. The **`@AndroidEntryPoint`** annotation adds a DI container to the Android class annotated with it. When using Hilt, the generated Hilt component is attached to the Application object's life cycle and provides its dependencies. Hilt currently supports the following Android classes:

- **`ViewModel`** annotated as **`@HiltViewModel`**
- **`Application`** annotated as **`@HiltAndroidApp`**
- **`Activity`**
- **`Fragment`**
- **`View`**
- **`Service`**
- **`BroadcastReceiver`**

We will use other necessary annotations in Hilt later, for instance, the **`@Module`** annotation, **`@InstallIn`**, and **`@Provides`**. The **`@Module`** annotation means the class in which you can add binding for types that cannot be injected in the constructor. **`@InstallIn`** indicates which Hilt-generated DI container (or singleton component) has to be available in the code module binding.

Finally, `@Provides` binds a type that cannot be constructor injected. Its return type is the binding type, it can take dependency parameters, and every time you need an instance, the function body is executed if the type is not scoped.

Implementing ViewModel classes and understanding the state in Compose

In Android, a `ViewModel` is a class responsible for consciously managing the UI-related data life cycle. There is also a lot of debate in the community about whether developers should use `ViewModel` in Compose or not. However, Manuel Vivo, a senior Android developer relations engineer at Google, says:

"I'd include them if their benefits apply to your app. No need to use them if you handle all configuration changes yourself and don't use Navigation Compose. Otherwise, use ViewModels not to reinvent the wheel."

"On the other hand, the debate as to why one should not use ViewModels is based on the argument that in pure Compose, since Compose handles configuration changes, having your Composable functions reference the ViewModel is unnecessary."

You can also refer to this tweet by Jim Sproch: **https://twitter.com/JimSproch/status/139716**

9679647444993.

NOTE

You can find more info about the benefits of using the ViewModel here: https://developer.android.com/jetpack/compose/state#viewmodels-source-of-truth.

This means using `ViewModel` to handle the state in your application will be a question of personal choice with Jetpack Compose. The currently recommended architecture pattern in Android is **Model-View-View-Model** (**MVVM**), and many applications use it.

Jetpack Compose uses a unidirectional data flow design pattern; this means the data or state only streams down while the events stream up. Hence, a clear understanding of how we can utilize the unidirectional pattern to make our code more readable, maintainable, and testable as much as possible using the `ViewModel` class will be helpful.

In addition, `ViewModel` is suitable for providing your application with access to business logic, preparing the data for presentation on the screen, and making your code testable.

Getting ready

In this recipe, we will work with a pre-built skeleton `SampleLogin` project, which you can download from the *Technical requirements* section. We will use Hilt in this recipe since the project uses Hilt, but we will explain Hilt in a next recipe.

How to do it...

You will now create a **ViewModel** class and modify most of the code in the **LoginContent** Kotlin file:

1. To keep our classes and files well organized, let's go ahead and first create a package to hold our UI and view models. Navigate to the main **Package** folder, right-click to open a prompt, then go down to **Package**, and a dialog with a package name will appear.

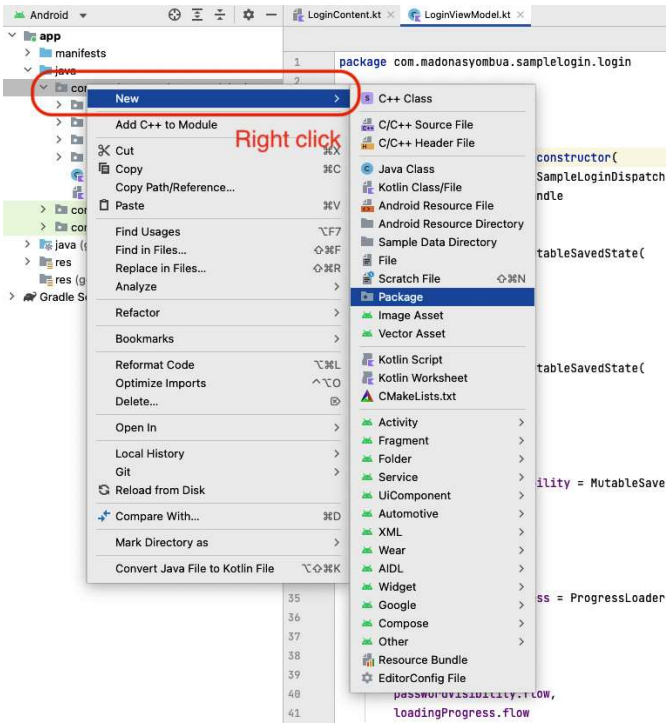

Figure 3.2 – How to create a package

2. Name the package **Login**; inside **Login**, we will move our **LoginContent** file and add the new class, **LoginViewModel**, there. Go ahead and create a **ViewModel** class:

```
class LoginViewModel {...}
```

3. Now that we have created a **LoginViewModel** class, we need to add a DI

annotation of **`HiltViewModel`** and ensure we extend the **`ViewModel`** class:

```
@HiltViewModel
class LoginViewModel @Inject constructor(
): ViewModel(){. . .}
```

4. In our **`ViewModel`** constructor, we will need to add **`stateHandle: SavedStateHandle`**, which will help us maintain and retrieve objects to and from the saved state. These values persist even after the system kills the process and remain available through the same object:

```
@HiltViewModel
class LoginViewModel @Inject constructor(
    stateHandle: SavedStateHandle
) : ViewModel() {...}
```

5. Before we build our **`ViewModel`**, let's go ahead and create a data class, **`AuthenticationState()`**. This class will come in handy during our testing since we need to be able to test most of our validation cases. A **`View`** state class, plus having a single source of truth, has many advantages and is one of the principles of **Model-View-Intent** (**MVI**):

```
data class AuthenticationState(
    val userName: String = "",
    val password: String = "",
    val loading: Boolean = false,
    var togglePasswordVisibility: Boolean = true
) {
    companion object {
        val EMPTY_STATE = AuthenticationState()
    }
}
```

6. Now, let's go ahead and create a helper class, **`MutableSavedState<T>()`**, which will

take in **savedStateHandle**, a key, and a default value. This class acts as a **MutableStateFlow()** but saves the data and value and retrieves it upon the application's death with the help of **SavedStateHandle**:

```
class MutableSavedState<T>(
    private val savedStateHandle: SavedStateHandle,
    private val key: String,
    defValue: T,
) {
 . . .
}
```

7. Now, let's go ahead and create callbacks that will be invoked when a user enters their username and password in our **LoginViewModel**:

```
private val username = MutableSavedState(
    stateHandle,
    "UserName",
    defValue = ""
)
fun userNameChanged(userName: String){
    username.value = userName
}
```

8. Go ahead and do the same for password and password toggle visibility.
9. Now, we need to create a **combineFlows** helper class. You can combine more than two flows in Kotlin; a coroutine **flow** is a type that emits multiple values sequentially, as opposed to the **suspend** function, which returns a single value. More details on how to combine flows can be found at **combine(flow1, flow2, flow3, flow4) {t1, t2, t3, t4 -> resultMapper}.stateIn(scope)**:

```
fun <T1, T2, T3, T4, T5, T6, R> combine(
    flow: Flow<T1>,
    flow2: Flow<T2>,
    flow3: Flow<T3>,
    flow4: Flow<T4>,
    flow5: Flow<T5>,
    flow6: Flow<T6>,
    transform: suspend (T1, T2, T3, T4, T5, T6) -> R
): Flow<R> = combine(
        combine(flow, flow2, flow3, ::Triple),
        combine(flow4, flow5, flow6, ::Triple)
) { t1, t2 ->
    transform(
        t1.first,
        t1.second,
        t1.third,
        t2.first,
        t2.second,
        t2.third
    )
}
```

Read more here – **https://stackoverflow.com/questions/67939183/kotlin-combine-more-than-2-flows**:

```
val state = combineFlows(
    username.flow,
    password.flow,
    passwordVisibilityToggle.flow,
    loadingProgress.flow
) { username, password, passwordToggle, isLoading ->
    AuthenticationState(
        userName = username,
        password = password,
        togglePasswordVisibility = passwordToggle,
        loading = isLoading
    )
}.stateIn(. . .)
```

10. Now, let's go ahead and create our helper class for using coroutines and call it **`SampleLoginDispatchers()`**; it will help us in testing our code and ensuring our code is easily readable. In addition, we use coroutine dispatchers that help determine what thread the corresponding coroutine should use for execution:

```
.stateIn(
   coroutineScope = viewModelScope + dispatchers.main,
   initialValue = AuthenticationState.EMPTY_STATE
)
```

`SharedFlow` represents a read-only state with a single updatable data value, which emits any updates to its collectors. On the other hand, a state flow is a hot flow because its active instance exists independently of the presence of collectors.

11. The **`SharingStarted`** coroutine flow operator in Android is used to share the execution of a flow among multiple collectors. It is commonly used to create a "hot" flow, which means that the flow starts emitting data as soon as it is created, and the data is shared among all the active collectors of the flow. These can be back-to-back emissions of the same command and have no effect:

```
fun <T> Flow<T>.stateIn(
    coroutineScope: CoroutineScope,
    initialValue: T
): StateFlow<T> = stateIn(
    scope = coroutineScope,
    started = SharingStarted.WhileSubscribed(5000),
    initialValue = initialValue
)
```

12. There are four types of dispatchers. In our example, we will only use three. In addition, you can inject a single dispatcher and achieve the same result without the class; hence, this can be preference-based. See how it works for the four types of dispatchers:

```
class SampleLoginDispatchers(
    val default: CoroutineDispatcher,
    val main: CoroutineDispatcher,
    val io: CoroutineDispatcher
) {
    companion object {
        fun createTestDispatchers(coroutineDispatcher:
        CoroutineDispatcher): SampleLoginDispatchers {
            return SampleLoginDispatchers(
                default = coroutineDispatcher,
                main = coroutineDispatcher,
                io = coroutineDispatcher
            )
        }
    }
}
```

Now that we have created our helper class, we must provide the dispatcher via DI. We have an entire recipe dedicated to Hilt, so we will look at the concepts and what the annotations mean in the Hilt recipe.

13. Create a new package and call the package **`di`**. In this package, create a new object and call it **`AppModule`**; we will provide our dispatcher to the **`ViewModel`** constructor via the dependency graph:

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
    @Provides
```

```
        fun provideSlimeDispatchers():
        SampleLoginDispatchers {
            return SampleLoginDispatchers(
            default = Dispatchers.Default,
            main = Dispatchers.Main,
            io = Dispatchers.IO
            )
        }
    }
```

14. We will now need to go to **`LoginContent`** and modify the code – that is, by adding callbacks that will correspond to our **`ViewModel`**, and whenever we have a view – for example, **`UserNameField()`** – we will use the callback. See the sample code:

```
@Composable
fun LoginContent(
    modifier: Modifier = Modifier,
    uiState: AuthenticationState,
    onUsernameUpdated: (String) -> Unit,
    onPasswordUpdated: (String) -> Unit,
    onLogin: () -> Unit,
    passwordToggleVisibility: (Boolean) -> Unit
){
. . .
UserNameField(authState = uiState, onValueChanged =
onUsernameUpdated)
PasswordInputField(
    text = stringResource(id = R.string.password),
    authState = uiState,
    onValueChanged = onPasswordUpdated,
    passwordToggleVisibility =
        passwordToggleVisibility
)
LoginButton(
    text = stringResource(id = R.string.sign_in),
    enabled = if (uiState.isValidForm()) {
        !uiState.loading
    } else {
        false
    },
    onLoginClicked = {
```

```
            onLogin.invoke()
        },
        isLoading = uiState.loading
    ). . .}
```

15. Now, in our **LoginContentScreen** Composable function, we will pass our **LoginViewModel**:

```
@Composable
fun LoginContentScreen(
    loginViewModel: LoginViewModel,
    onRegisterNavigateTo: () -> Unit
) {
    val viewState by
        loginViewModel.state.collectAsState()
    LoginContent(
        uiState = viewState,
        onUsernameUpdated =
            loginViewModel::userNameChanged,
         onPasswordUpdated =
            loginViewModel::passwordChanged,
         onLogin = loginViewModel::login,
         passwordToggleVisibility =
             loginViewModel::passwordVisibility,
         onRegister = onRegisterNavigateTo
    )
}
```

16. Finally, in **MainActivity**, we can now go ahead and call **LoginContentScreen**, pass in our **ViewModel**, and also specify what action we want when the user clicks **onRegister**:

```
LoginContentScreen(loginViewModel = HiltViewModel(), onRegisterNavigateTo = {. . .}
```

17. For the entire code, please ensure you check out the link in the *Technical requirements* section.

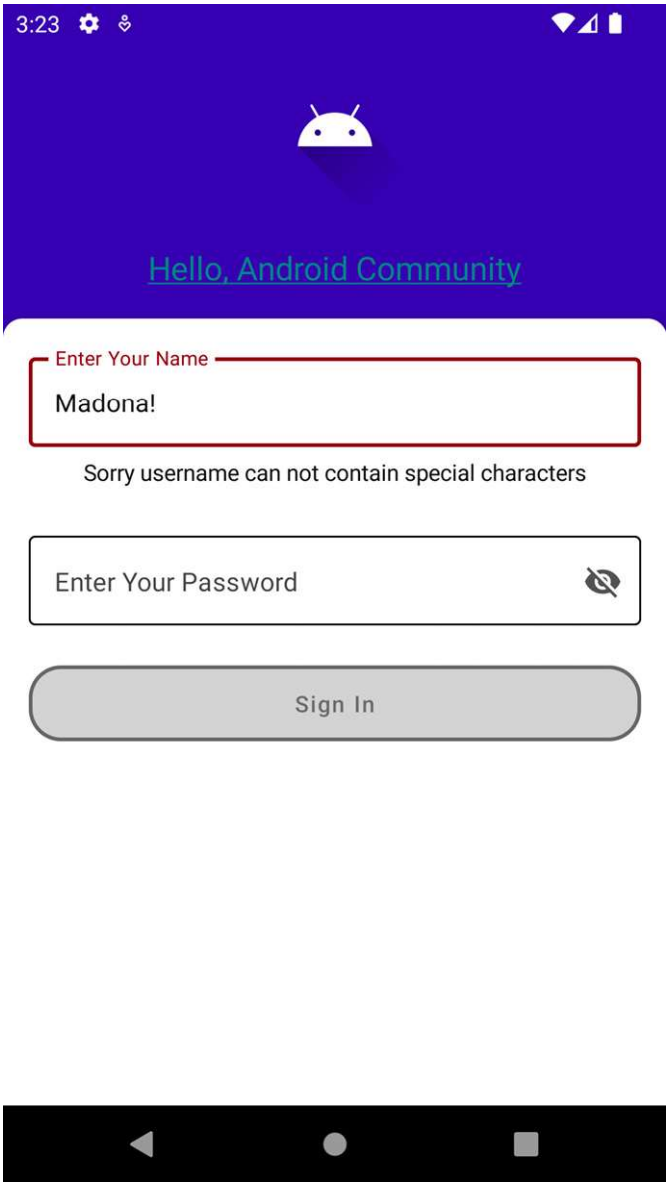

Figure 3.3 – Error state display when we enter a special character !

How it works...

Jetpack Compose uses a unidirectional data flow design pattern. This means the data or state only streams down while the events stream up, as shown in *Figure 3.4*.

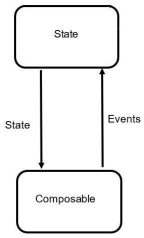

Figure 3.4 – A unidirectional data flow

That is to say, Composable functions receive the state and then display it on the screen. On the other hand, an event can cause the state to need to be updated and come from either a Composable or any other part of your application. Furthermore, whatever handles the state, in our case, `ViewModel`, receives the event and adjusts the state for us.

We also use coroutines, which are nothing but lightweight threads, and they help us handle synchronous and asynchronous programming easily. Furthermore, coroutines allow execution to be suspended and resumed later. The main advantages are that they are lightweight, have built-in cancellation support, have lower chances of memory leaks, and the Jetpack libraries provide coroutine support.

There are four types of dispatchers:

- **The Main dispatcher**: The `Main` dispatcher executes in the main thread, which is usually used when your application needs to perform some UI operations within a coroutine. This is because the UI can only be changed from the main thread. Another name for the main thread is the UI thread.
- **The IO dispatcher**: The `IO` dispatcher starts the coroutine in the I/O thread; I/O simply means input and output in programming. This is also used to perform all data work, such as networking, reading, or writing from the database. You can simply say fetching data from the I/O operation is done in the I/O thread.

- **The Default dispatcher**: The `Default` dispatcher starts in the default state. Your application can utilize this if you plan on doing complex long-running calculations, which can block the UI/main thread and make your UI freeze or cause an **Application Not Responding** (**ANR**) error. In addition, the default dispatchers are launched in `GlobalScope`, and you can use it by simply calling `GlobalScope.launch{...}`.
- **The Unconfined dispatcher**: Finally, `Unconfined`, as the name suggests, is a dispatcher not confined to any specific thread. This executes the dispatcher to perform its work in a current call frame and lets the coroutine resume whatever threads that are used by the corresponding function.

See also...

A lot was covered in this chapter, and it is fair to acknowledge that just this simple recipe cannot explain `ViewModel` in its entirety. To find out more, see the following link: **https://developer.android.com/topic/libraries/architecture/viewmodel**.

Implementing Compose in an existing XML layout-based project

Since Compose is a new UI framework, many code bases still rely heavily on XML layouts.

However, many companies are opting to build new screens using Compose, and this is achievable by utilizing existing XML layouts and adding unique views using **`ComposeView`** XML tags. This recipe will look into adding a Compose view to an XML layout.

Getting ready

In this recipe, we can create a new project or opt to use an existing project that does not heavily rely on Compose. We will try to display **`GreetingDialog`** and use an XML layout to show how we can use the **`ComposeView`** tag in XML layouts. If you already have a project, you do not need to set this up; you can skip to *step 4* in the preceding *How to do it...* section.

How to do it...

Now let us go ahead and explore how we can utilize existing XML layouts with Compose:

1. Let's start by creating a new project or using a preexisting one; if you create a new activity that is not Compose, you can use **`EmptyActivity`**, and give it any name.
2. If you already have a project set up, you can skip this step. If you opt to create a new project, you will have **`MainActivity`**, and since this is the old way of creating views, you will notice an XML layout in the **`resource`** folder with a TextView that has **`Hello world`**. We can go ahead and remove that since we will not use it.
3. If you already have a project ready, you can launch **`GreetingDialog`** on any screen you want. Also, if you opt to create a button in-

stead of a dialog, that is fine, too since the idea is to showcase how we can use XML tags in Jetpack Compose.

4. Now, let us go ahead and add an XML tag inside **`activity_main.xml`** and give our Compose view an **`id`** value. The first time you add **`ComposeView`**, you will see an error message if you still need to add the dependency. Go ahead and click **Add dependency on android.compose.ui:ui** and the project will sync as shown in *Figure 3.5*.

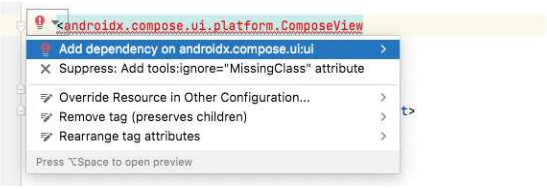

Figure 3.5 – A Compose view in XML

5. Once you have synced your project, the error will disappear, and you should be able to use this view in **`MainActivity`**, or where you want to use **`ComposeView`**:

```
<androidx.Compose.ui.platform.ComposeView
    android:id="@+id/alert_dialog"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

6. Let's also add **`viewBinding`** to our **`build.gradle(Module:app)`** so that we can easily access our view in **`MainActivity`**. Also, if you already have **`viewBinding`** set up, you can skip this part:

```
buildFeatures{
    viewBinding true
}
```

7. Once we have synced the project, we can go ahead and, in **`MainActivity`**, access **`ComposeView`** through binding.

Furthermore, it will have a **`setContent{}`** method where you can set all your Composables and wrap it into your Theme:

```
class MainActivity : AppCompatActivity() {
    private lateinit var activityBinding:
    ActivityMainBinding
    override fun onCreate(savedInstanceState: Bundle?)
    {
        super.onCreate(savedInstanceState)
        activityBinding =
           ActivityMainBinding.inflate(layoutInflater)
        setContentView(activityBinding.root)
        activityBinding.alertDialog.setContent {
            GreetingAlertDialog()
        }
    }
}
```

8. Our **`GreetingAlertDialog()`** will have an **`AlertDialog()`** Composable, a title, and text, which will provide our message as a simple text element. The title will say **`Hello`** since this is a greeting, and the message will be **`Hello, and thank you for being part of the Android community`**. You can customize this to fit your needs:

```
@Composable
fun SimpleAlertDialog() {
    AlertDialog(
        onDismissRequest = { },
        confirmButton = {
            TextButton(onClick = {})
            { Text(text = "OK") }
        },
        dismissButton = {
            TextButton(onClick = {})
            { Text(text = "OK") }
        },
        title = { Text(text = "Hello") },
        text = { Text(text = "Hello, and thank you for
        being part of the Android community") }
```

```
        )
    }
```

9. To create Compose components, you will need to add the Compose Material Design dependency to your gradle app. Depending on what your application supports, you can utilize Compose Material 3 components, which is the next evolution of Material Design and comes with updated theming.
10. You can easily customize features such as dynamic color and more. We look into Material 3 in *Chapter 11, GUI Alerts – What's New in Menus, Dialog, Toast, Snackbars, and More in Modern Android Development*. Hence, for now, since the application that I am using has not migrated to Material 3, I will use this import – **`implementation "androidx.Compose.material:material:1.x.x"`**. Here, you can use any import that fits your need.
11. You can also create a custom view that extends from **`AbstractComposeView`**:

```
class ComposeAlertDialogComponent @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
    defStyle: Int = 0
) : AbstractComposeView(context, attrs, defStyle) {
        @Composable
    override fun Content() {
        GreetingAlertDialog()
    }
}
```

12. Finally, when you run your application, you should have a dialog that has a title and text; *Figure 3.6* shows a dialog from an already pre-existing project, so this will definitely vary based on what steps you took:

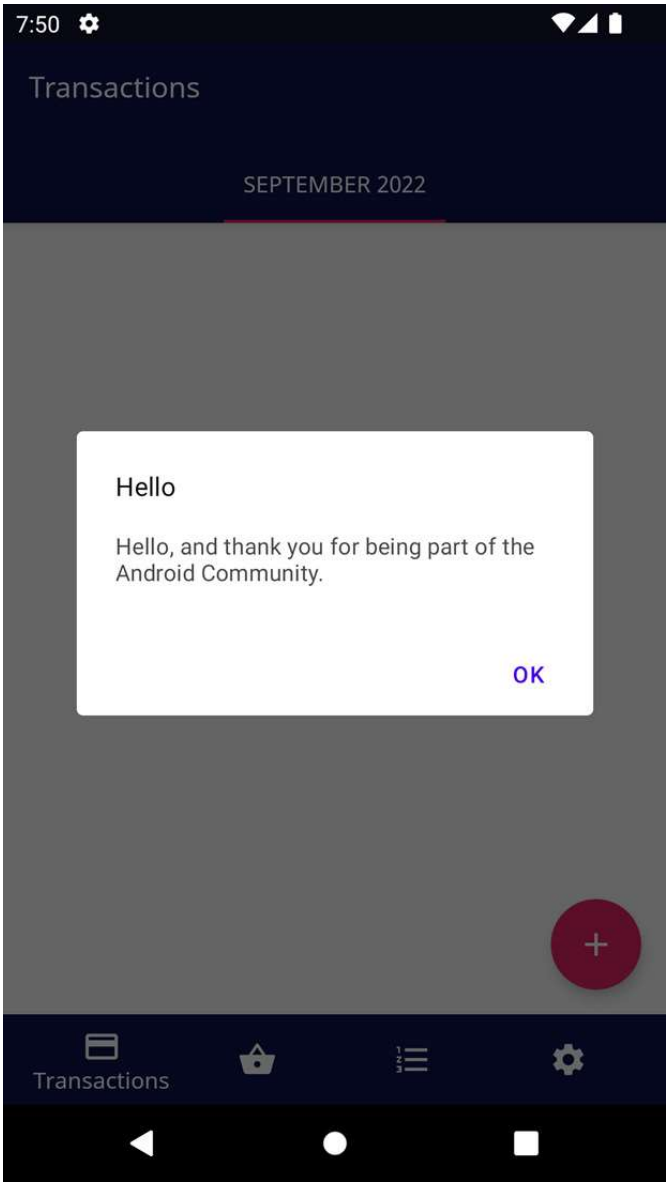

Figure 3.6 – Dialog Compose view in XML

How it works...

First, we inflate the XML layout, which we define in our layout **`resource`** folder. Then, using binding, we got **`ComposeView`** using the created XML ID, set a Compose strategy that works best for our host view, and called **`setContent`** to use Compose. In your activity, to be able to create any Compose-based screen, you have to ensure that you call the **`setContent{}`** method and pass whatever Composable function you have created.

To further explore the **`setContent`** method, it is written as an extension function of **`ComponentActivity`**, and it expects a Composable function as the last parameter. There is also a better way to demonstrate how **`setContent{}`** works to integrate a Composable tree into your Android application.

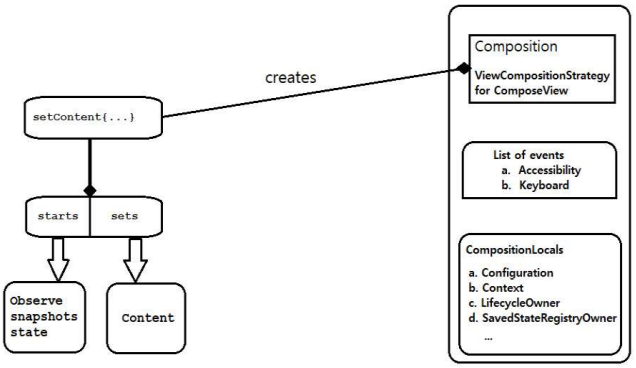

Figure 3.7 – This happens when you call setContent{}

`ViewCompositionStrategy` helps determines when to dispose of the composition; hence, Compose UI views such as **`ComposeView`** and the **`AbstractComposeView`** use **`ViewCompositonStrategy`**, which helps define this behavior.

You can learn more by following this link to learn more about the interoperability APIs: **https://developer.android.com/jetpack/compose/interop/interop-apis#composition-strategy**.

Understanding and handling

recomposition in Jetpack Compose

Jetpack Compose is still very new, and many companies are starting to use it. Furthermore, Google has done a great job by giving developers significant documentation to help them embrace this new UI toolkit. However, despite all the documentation, one concept needs to be clarified. And that is recomposition.

Fair enough, all new software has its ups and downs, and as many people start using it, more people start giving feedback – hence, the need for more improvement. Recomposition, in Compose, involves calling your Composable again when the input changes. Or you can think of it when the composition structure and relation change.

Unless its parameters change, we want to avoid a Composable function being re-invoked in most use cases. So, in this recipe, we look into how recomposition happens and how you can debug and solve any recomposition in your application.

How to do it...

Since our view system is simple, we will be checking whether we have any recomposition in our **`Login`** project:

1. We can look at a simple example and how recomposition will occur:

```
@Composable
fun UserDetails(
    name: String,
    gender: String,
) {
    Box() {
        Text(name)
        Spacer()
        Text(gender)
    }}
```

In our given example, the **`Text`** function will recompose when **`name`** changes and not when **`gender`** changes. In addition, the **`gender:String`** input value will recompose only when **`gender`** changes.

2. You can also launch and utilize **Layout Inspector** to debug recomposition. If it is not on your Android Studio dock, you can start it by going to **View** | **Tool Windows** | **Layout Inspector**. We will look into **`LoginContent`** and see whether we have any recomposition.

Figure 3.8 – Layout Inspector

3. Once you launch **Layout Inspector**, you need to ensure you have your emulator hooked to it.

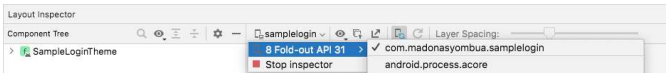

Figure 3.9 – Linking the inspector

4. Go ahead and expand the `SampleLoginTheme` entry point, and you will notice our current view system is not complex. As you can see, **Layout Inspector** does not show any recomposition counts.

That is to say, if our application had any recomposition counts, they would show up in **Layout Inspector**.

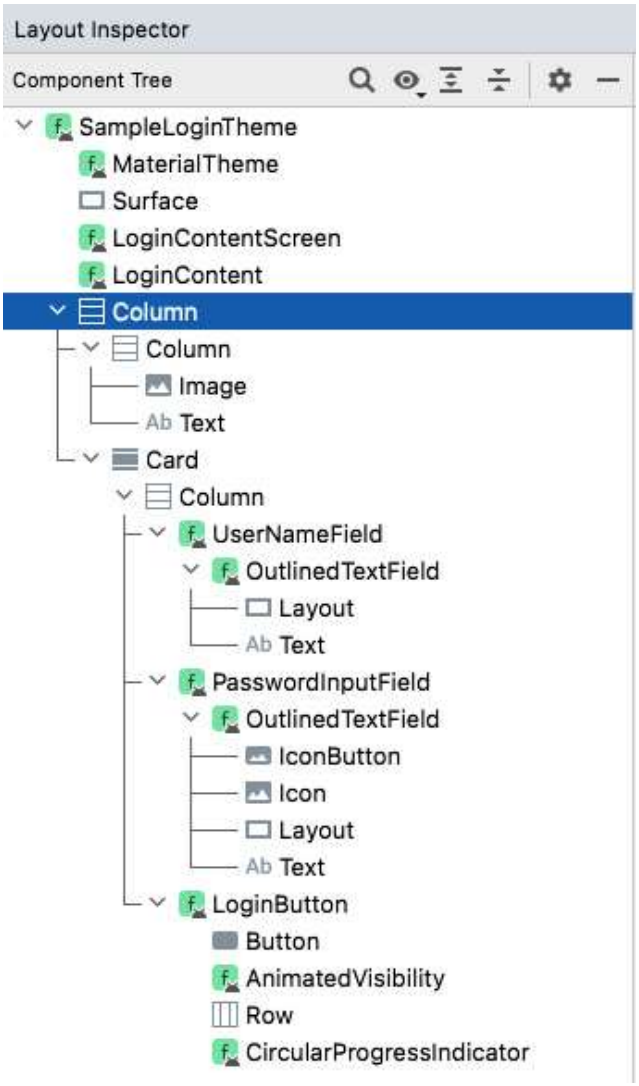

Figure 3.10 – The component tree

5. Finally, as you have seen, our application does not have any recomposition happening, but it is always beneficial to check your application to know what might be causing the recomposition and fix it.

IMPORTANT NOTE

Using side-effects might lead to users of your application experiencing strange and unpredictable behavior in your app. In addition, a side-effect is any change that is visible to the rest of your application. For instance, writing to a property of a shared object, updating an ob-

servable in **`ViewModel`***, and updating shared preferences are all dangerous side effects.*

How it works...

For adaptability, Compose skips **`lambda`** calls or any child function that does not have any changes to its input. This better handling of resources makes sense since, in Compose, animations and other UI elements can trigger recomposition in every frame.

We can go in depth and use a diagram to showcase how the Jetpack composition life cycle works. In short, the life cycle of a Composable function is defined by three significant events:

- Being composed
- Getting recomposed or not getting recomposed
- No longer being composed

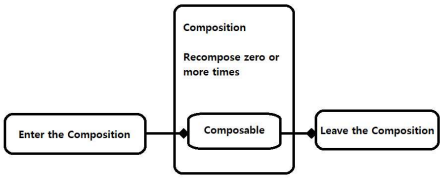

Figure 3.11 – The composition life cycle of a Composable

To fathom how Compose works, it's good to know what constitutes the Compose architectural layer. A high-level overview of the Jetpack Compose architectural layer includes **Material**, **Foundation**, **UI**, and **Runtime** aspects.

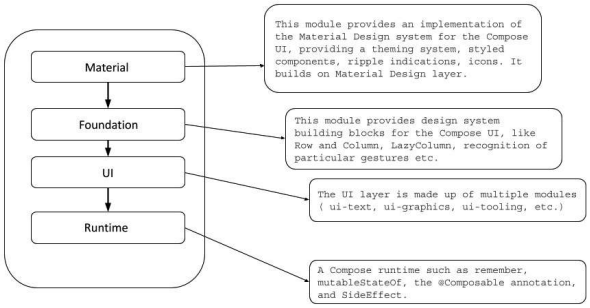

Figure 3.12 – A diagram showing Jetpack Compose Architectural Layers

In **Material**, this module implements the Material Design system for the Compose UI.

Furthermore, it provides a theming system, styled components, and more. **Foundation** is where we have the design system building blocks such as the UI, **`Row`**, **`Column`**, and more. The **UI** layer is made of multiple modules that implement the fundamentals of the UI toolkit.

See also

The Compose team is launching Jetpack Compose Composition Tracing, the first alpha that will help developers trace their composition easily; you can read more here:

- **https://medium.com/androiddevelopers/jetpack-Compose-composition-tracing-9ec2b3aea535**
- **https://developer.android.com/jetpack/compose/lifecycle**

Writing UI tests for your Compose views

It is essential to test your code when developing Android applications, especially if your applications have many users. Furthermore, when you write tests for your code, you basically verify the functions, behavior, correctness, and versatility of the Android application. The most popular UI testing tools in Android are Espresso, UI Automator, Calabash, and Detox.

In this book, however, we will use Espresso. The most notable advantages of Espresso are as follows:

- It is easy to set up
- It has highly stable test cycles
- It supports JUnit 4
- It is made purely for Android UI testing
- It is suitable for writing black-box tests
- It supports testing activities outside the application as well

Getting ready

You will need to have completed previous recipes to follow along with this one.

How to do it...

As with the other recipes in this chapter, we will use the new project we created in *Chapter 1*, *Getting Started with Modern Android Development Skills*:

1. Let's go ahead and navigate into the **`androidTest`** package in our project folder.
2. Start by creating a new class in the **`androidTest`** package and call it

LoginContentTest.kt. In Jetpack Compose, testing is made more accessible, and we need to have unique tags for our views.

3. So, for this step, let's go back to our main package (**com.name.SampleLogin**) and create a new package and call it **util**. Inside **util**, let's create a new class and call it **TestTags**, which will be an object. Here, we will have another object, name it **LoginContent**, and create constant values that we can call in our view:

```
object TestTags {
    object LoginContent {
        const val SIGN_IN_BUTTON = "sign_in_button"
        const val LOGO_IMAGE = "logo_image_button"
        const val ANDROID_TEXT = "community_text"
        const val USERNAME_FIELD = "username_fields"
        const val PASSWORD_FIELD = "password_fields"
    }
}
```

4. Now that we have created the test tags, let's go back to our **LoginContent** and add them to all views in the **Modifier()** so that when we test, it is easier to identify the view using the test tag we have added. See the following code snippet:

```
Image(
    modifier = modifier.testTag(LOGO_IMAGE),
    painter = painterResource(id =
        R.drawable.ic_launcher_foreground),
    contentDescription = "Logo"
)
```

5. Inside our **LoginCotentTest** class, let's now go ahead and set up our testing environment. We will need to create **@get:Rule**, which annotates fields that reference rules or methods that return a rule. Under the

rule, let's create **`ComposeRuleTest`** and initialize it:

```
@get:Rule
val ComposeRuleTest = createAndroidComposeRule<MainActivity>()
```

6. Add the following function to help us set up the content. We should call this function in our **`Test`** annotated function:

```
private fun initCompose() {
    ComposeRuleTest.activity.setContent {
        SampleLoginTheme {
            LoginContent()
        }
    }
}
```

7. Finally, let's go ahead and add our first test. For the tests we will write, we will verify that the views are displayed on the screen as we expect them to be:

```
@Test
fun assertSignInButtonIsDisplayed(){
    initCompose()
    ComposeRuleTest.onNodeWithTag(SIGN_IN_BUTTON,
    true).assertIsDisplayed()
}
@Test
fun assertUserInputFieldIsDisplayed(){
    initCompose()
    ComposeRuleTest.onNodeWithTag(USERNAME_FIELD,
    true).assertIsDisplayed()
}
```

8. **`SIGN_IN_BUTTON`** and **`USERNAME_FIELD`** are imported from the test tags that we have created and are already used by only one view, the sign-in button.
9. Go ahead and run the tests, and a dialog will pop up showing the running process; if successful, the tests will pass. In our case, the tests should pass.

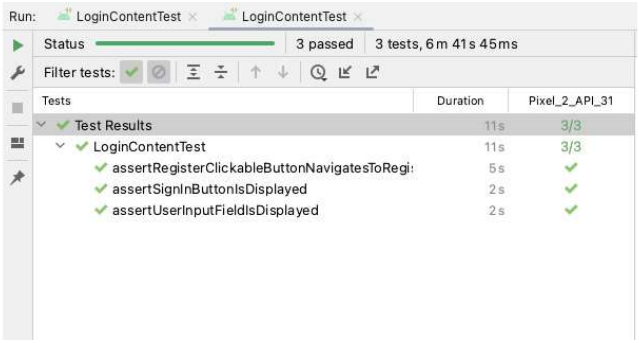

Figure 3.13 – A screenshot showing passing tests

IMPORTANT NOTE

For these tests, you will not need to add any dependencies; everything we need is already available for our use.

How it works...

We use **`createAndroidComposeRule<>()`** when accessing an activity. Testing and ensuring your applications display the expected outcome is essential. This is why Android Studio uses the emulator to help developers test their code to ensure their application functions as it would on standard devices.

Furthermore, Android phones come with a developers' option ready for developers to use, making it even easier for the different number of devices that Android supports and helping reproduce bugs that are hard to find in emulators.

When we test our Compose code, we improve our app's quality by catching errors early on in the development process. In this chapter, we touched on creating more views to demonstrate how Jetpack Compose works; further-

more, our test cases need to address user action since we did not implement any.

In a different setting, we can write more crucial tests to confirm the intended action, and we will do this in later chapters. Furthermore, Compose provides testing APIs to find elements, verify their attributes, and perform user actions. Moreover, they also include advanced features such as time manipulation, among others.

Explicitly calling the `@Test` annotation is very important when writing tests since this annotation tells JUnit that the function to which it is attached is to run as a `Test` function. In addition, UI tests in Compose use **Semantics** to interact with the UI hierarchy. And semantics, as the name implies, give meaning to a piece of UI, for example, `.onNodeWithTag`.

A UI portion or element can mean anything from a single Composable to a full screen. If you try to access the wrong node, the semantics tree, which is generated alongside the UI hierarchy, will complain.

There's more...

There are other testing tools as follows:

- **Espresso Test Recorder** provides developers with a faster, interactive way to test their app's everyday user input behavior and visual elements.
- **App Crawler** undoubtedly uses a more hands-off approach to help you test user actions without needing to maintain or write

any code. With this tool, you can easily configure your inputs, such as entering your username and password credentials.

- **Monkey** is a command-line device that also stress-tests your app by sending a random flow of user validation/input or tap actions into the device or emulator instance.

To learn more about testing and semantics in Compose, read the following: **https://developer.android.com/jetpack/compose/semantics**.

Writing tests for your ViewModels

Unlike **Model-View-Controller** (**MVC**) and **Model-View-Presenter** (**MVP**), MVVM is the favored design pattern in Modern Android Development because of its unidirectional data and dependency flow. Furthermore, it becomes more accessible to unit test, as you will see in this recipe.

Getting ready

We will use our previous recipe, *Implementing ViewModel and understanding the state in Compose*, to test our logic and state changes.

How to do it...

In this recipe, we will write unit tests to verify our authentication state changes since that is what we have implemented so far:

1. Start by creating a **`LoginViewModelTest`** class in the **`test`** package:

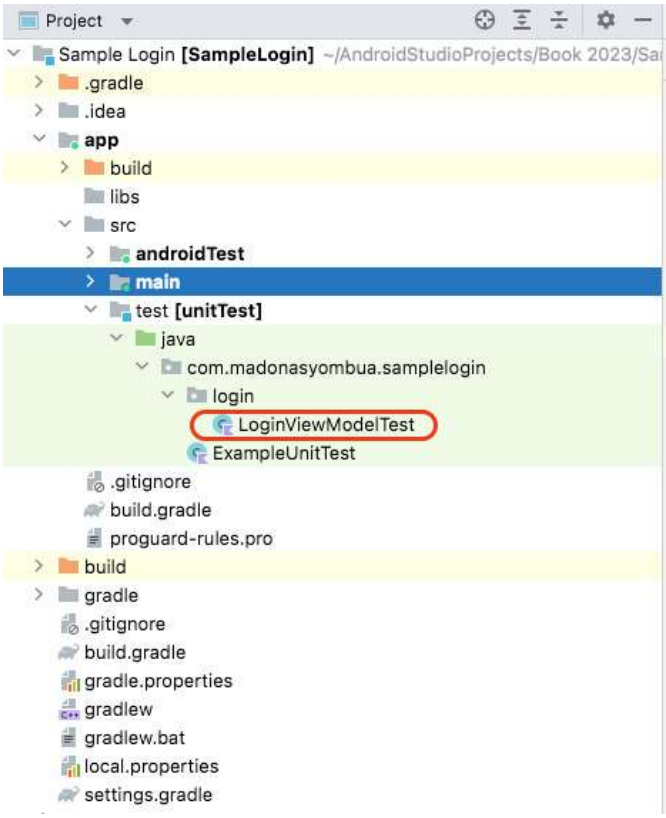

Figure 3.14 – Created unit test

2. We will use the **`cashapp/turbine`** testing library for coroutine flows to test the flow we have created. Hence, you will need to include the processing code snippet in **`build.gradle`**:

```
repositories {
  mavenCentral()
}
dependencies {
  testImplementation 'app.cash.turbine:turbine:0.x.x'
}
```

3. Once you have created the class, go ahead and set up **`@Before`**, which will run before each test:

```
class LoginViewModelTest {
   private lateinit var loginViewModel: LoginViewModel
   @Before
```

```
    fun setUp(){
        loginViewModel = LoginViewModel(
            dispatchers =
            SampleLoginDispatchers.createTestDispatchers(
            UnconfinedTestDispatcher()),
            stateHandle = SavedStateHandle()
        )
    }
}
```

4. As you can see, we utilized **`SampleLoginDispatchers.createTestDispatchers`**. For **`UnconfinedTestDispatcher`**, you must include the testing dependencies and import, **`import kotlinx.coroutines.test.UnconfinedTestDispatcher`**.
5. Now that we have our setup ready let us go ahead and create our test, verifying the authentication state changes:

```
@Test
fun `test authentication state changes`() = runTest {...}
```

6. Inside our **`Test`** function, we will now need to access the **`loginViewModel`** functions and pass fake values to the parameters:

```
@Test
fun `test authentication state changes`() = runTest {
    loginViewModel.userNameChanged("Madona")
    loginViewModel.passwordChanged("home")
    loginViewModel.passwordVisibility(true)
    loginViewModel.state.test {
        val stateChange = awaitItem()
        Truth.assertThat(stateChange).isEqualTo(
            AuthenticationState(
                userName = "Madona",
                password = "home",
                togglePasswordVisibility = true
            )
        )
    }
}
```

7. Finally, go ahead and run the test, and it should pass.

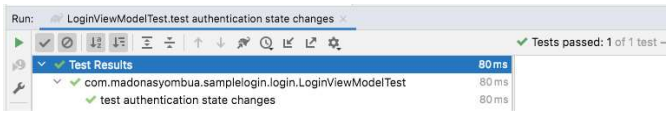

Figure 3.15 – Unit test passing

How it works...

As mentioned before, the most notable advantage of MVVM is being able to write code you can quickly test. In addition, architecture in Android is all about selecting the trade-offs. Each architecture has its pros and cons; based on your company's needs, you might work with a different one.

We create **`lateint var loginViewModel`** to set up a class for testing, and this is because the logic to be tested is in **`ViewModel`**.

We use **`UnconfinedDispatcher`**, which creates an instance of an **`Unconfined`** dispatcher. That means the tasks it executes are not confined to any particular thread and form an event loop. It is different in that it skips delays, as all **`TestDispatcher`** instances do. And by default, **`runTest()`** provides **`StandardTestDispatcher`**, which does not execute child coroutines immediately.

We use **`Truth`** for our assertation to help us make more readable code, and the significant advantages of **`Truth`** are as follows:

- It aligns the actual values to the left
- It gives us more detailed failure messages

- It offers richer operations to help with testing

There are also other alternatives, such as Mockito, Mockk, and more, but in this section, we have used `Truth`. We have also used a library by Cashapp that helps us test coroutine flows. You can learn more about the `turbine` library here: **https://github.com/cashapp/turbine**.