Chapter 10: Putting It All Together

In this chapter, we will analyze what we have done so far in the previous chapters and look at different ways we can improve the layers of the application. Later, we will explore the benefit of clean architecture when we integrate instrumented testing into the application, where we will swap the data source dependencies with mock dependencies to ensure the reliability of the tests.

In this chapter, we will cover the following topics:

- Inspecting module dependencies
- Instrumentation testing

By the end of the chapter, you will be able to identify and remove external dependencies in the use case layer of the application to enforce the **Common Closure Principle** (**CCP**) and know how to create instrumented tests on Android with mock data sources.

Technical requirements

The hardware and software requirements are as follows:

- Android Studio Arctic Fox 2020.3.1 Patch 3

The code files for this chapter can be found here: https://github.com/PacktPublishing/Clean-Android-Architecture/tree/main/Chapter10 ↗️.

Check out the following video to see the Code in Action: https://bit.ly/3sLr0HS ↗️

Inspecting module dependencies

In this section, we will analyze the dependencies used across the different modules in the application created in the previous chapters.

Following *Exercise 09.01 – Transitioning to MVI* from *Chapter 9*, *Implementing an MVI Architecture*, we now have a fully functioning application split into separate modules, representing different layers. We can analyze the relationship between the different modules by looking at the **`dependencies`** block in the **`build.gradle`** file in each module and focusing in particular on the **`implementation(project(path: "{module}"))`** lines. If we were to draw a diagram, it would look like the following:

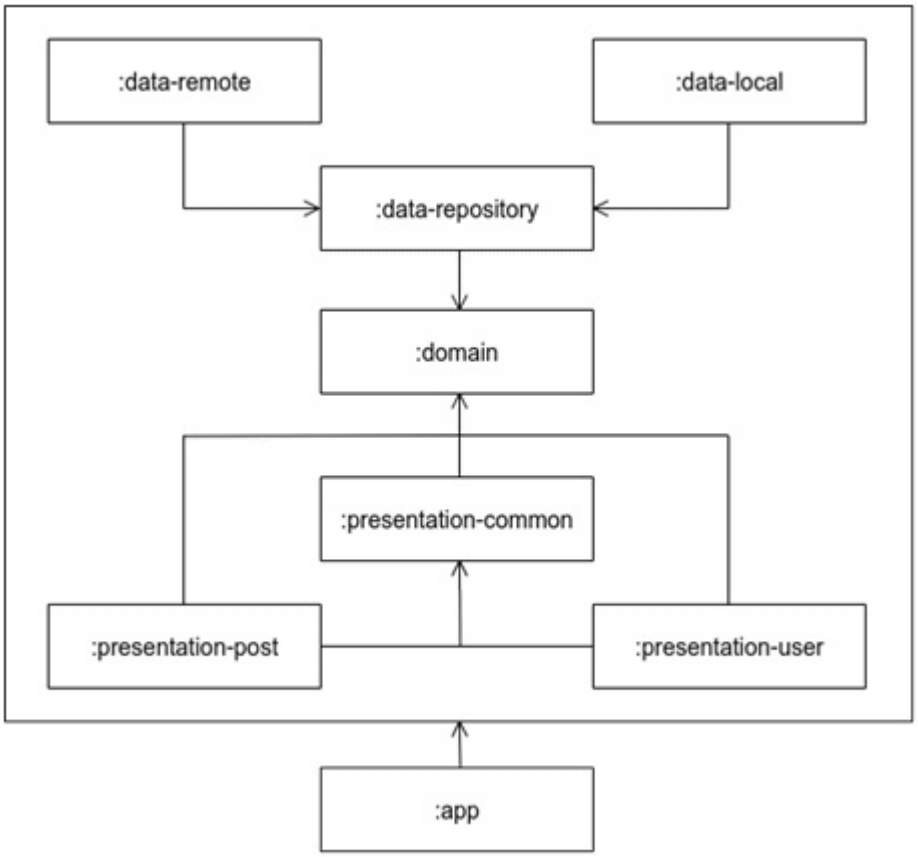

Figure 10.1 – A module dependency diagram for exercise 09.01

In the preceding figure, we can see that the **`:domain`** module, which is part of the domain layer, is at the center, with the modules from the other layers having a dependency toward it. The **`:app`** module is responsible for as-

sembling all of the dependencies, and this means that it will have a dependency on all the other modules. This means that we are in a good clean architecture position because we want the entities and use cases to have minimal dependencies on other components. If we continue analyzing the **`build.gradle`** files for each module and include the external dependencies as well, we will see additional dependencies on external libraries for each module:

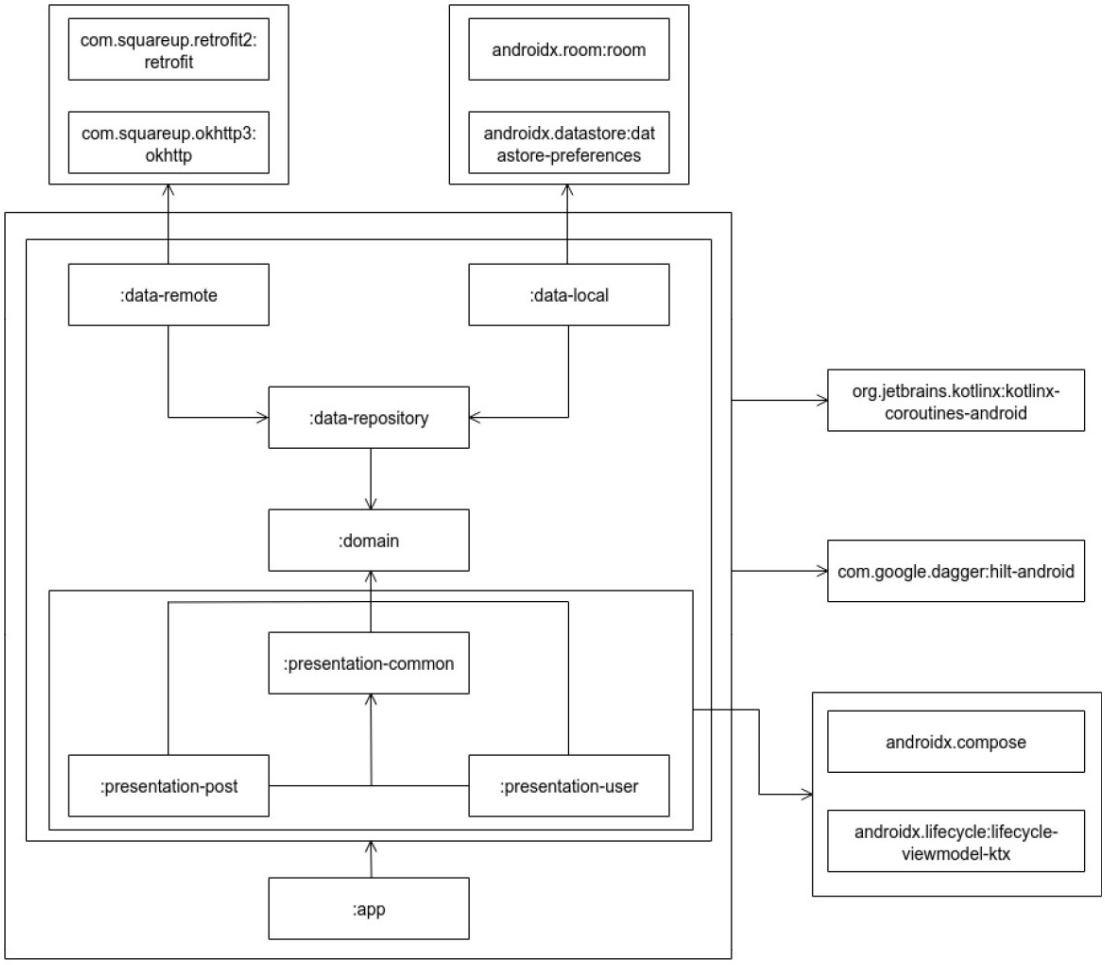

Figure 10.2 – A module dependency diagram with external dependencies for exercise 09.01

In *Figure 10.2*, we can see some of the relevant external dependencies our modules use. **`:data-remote`** uses dependencies toward Retrofit and OkHttp for networking, the **`:data-local`** module has dependencies toward Room and DataStore, while the presentation layer modules depend on things such as Compose, ViewModel, and the Android framework. Dependencies

that were used used across the entire project were coroutines, flows, and Hilt.

Having dependencies on Hilt and coroutines might pose a problem for the **`:domain`** and **`:data-repository`** modules. We want these two modules to be as stable as possible, and having external dependencies will create problems every time we update the versions for those libraries. We decided to use flows because of their threading benefits, the reactive approach, and because they were developed as an extension to the Kotlin framework. They might still pose a problem if we wanted to adapt our use cases for multiple platforms using Kotlin Multiplatform. A solution for this would be to develop a reactive plugin that would abstract the usage of flows and use this abstraction across the different modules. This would allow us to swap different reactive libraries without changing the code inside the module. While this solution would fix the problem, it comes with lots of baggage because we would need to abstract both the streams of data and the operators that the project requires from the flows framework, which would give us more code to maintain.

When it comes to the Hilt dependency, we can remove the references to Hilt from the **`:domain`** and **`:data-repository`** modules and move the Hilt modules into **`:app`**. Another solution would be to create new Gradle modules that would be responsible for providing the necessary dependency. For example, a **`:domain-hilt`** module could be created, where it would have a **`@Module`** annotated class that would provide all of the dependencies that the **`:domain`** module would need to expose. This approach can be used for other modules that we wish to export into applications that use different dependency injection frameworks to avoid the dependency on Hilt in those projects.

Module dependencies will increase as applications develop new features and evolve; this means we should take time and assess the dependencies used in a project. This will help us identify potential issues and whether we can scale an application properly. We should also account for external dependencies and analyze the influence they have over our project. In

the following section, we will look at an exercise on how to reduce the dependencies that the domain and repository modules have on Hilt.

Exercise 10.01 – Reduce dependencies

Modify *Exercise 09.01 – transitioning to MVI* in *Chapter 9*, *Implementing an MVI Architecture*, so that the **`domain`** and **`data-repository`** modules will no longer depend on Hilt and instead provide the dependencies from those modules inside the **`app`** module.

Before completing this exercise, you will need to do the following:

1. Remove Hilt from the **`domain`** module.
2. Remove the **`@Inject`** annotation from the **`GetPostsWithUsersWithInteractionUseCase`**, **`GetPostUseCase`**, **`GetUserUseCase`**, and **`UpdateInteractionUseCase`** classes.
3. Rename the **`AppModule`** class **`UseCaseModule`** and use **`@Provides`** to provide dependencies to the preceding objects.
4. Remove Hilt from the **`data-repository`** module and delete the use of the **`@Inject`** annotation.
5. Move **`RepositoryModule`** from the **`data-repository`** module into **`app`** and use **`@Provides`** to provide the dependencies to **`PostRepository`**, **`UserRepository`**, and **`InteractionRepository`**.

Follow these steps to complete the exercise:

1. In the **`build.gradle`** file of the **`domain`** module, remove the use of the **`kapt`** and Hilt plugins:
```
plugins {
    id 'com.android.library'
    id 'kotlin-android'
}
```
2. In the same file, delete the usages of Hilt from the **`dependencies`** block:
```
dependencies {
    implementation coroutines.coroutinesAndroid
```

```
        testImplementation test.junit
        testImplementation test.coroutines
        testImplementation test.mockito
    }
```

3. Delete the use of **`@Inject`** from **`GetPostsWithUsersWithInteractionUseCase`**:

```
    class GetPostsWithUsersWithInteractionUseCase(
        configuration: Configuration,
        private val postRepository: PostRepository,
        private val userRepository: UserRepository,
        private val interactionRepository:
            InteractionRepository
    ) :
    UseCase<GetPostsWithUsersWithInteractionUseCase.
        Request,
            GetPostsWithUsersWithInteractionUseCase.
                Response>(configuration) {
        …
    }
```

4. Delete the use of **`@Inject`** from **`GetPostUseCase`**:

```
    class GetPostUseCase(
        configuration: Configuration,
        private val postRepository: PostRepository
    ) : UseCase<GetPostUseCase.Request, GetPostUseCase.
        Response>(configuration) {
        …
    }
```

5. Delete the use of **`@Inject`** from **`GetUserUseCase`**:

```
    class GetUserUseCase(
        configuration: Configuration,
        private val userRepository: UserRepository
    ) : UseCase<GetUserUseCase.Request, GetUserUseCase.
        Response>(configuration) {
        …
    }
```

6. Delete the use of **`@Inject`** from **`UpdateInteractionUseCase`**:

```
class UpdateInteractionUseCase(
    configuration: Configuration,
    private val interactionRepository:
        InteractionRepository
) : UseCase<UpdateInteractionUseCase.Request,
   UpdateInteractionUseCase.Response>
(configuration) {
    …
}
```

7. In the app module, rename **`AppModule`** **`UseCaseModule`**.
8. In the app module in the **`UseCaseModule`** class, provide a dependency to **`GetPostsWithUsersWithInteractionUseCase`**:

```
@Module
@InstallIn(SingletonComponent::class)
class UseCaseModule {
    …
    @Provides
    fun
    provideGetPostsWithUsersWithInteractionUseCase(
        configuration: UseCase.Configuration,
        postRepository: PostRepository,
        userRepository: UserRepository,
        interactionRepository:
InteractionRepository
    ): GetPostsWithUsersWithInteractionUseCase =
       GetPostsWithUsersWithInteractionUseCase(
        configuration,
        postRepository,
        userRepository,
        interactionRepository
    )
}
```

Here, we need to use **`@Provides`** because we are no longer in the same module, which means we should treat this as an external dependency, which needs the **`@Provides`** annotation, similar to how we provided the Room and Retrofit dependencies.

9. In the same class, provide a dependency to **`GetPostUseCase`**:

```
@Module
@InstallIn(SingletonComponent::class)
class UseCaseModule {
    …
    @Provides
    fun provideGetPostUseCase(
        configuration: UseCase.Configuration,
        postRepository: PostRepository
    ): GetPostUseCase = GetPostUseCase(
        configuration,
        postRepository
    )
}
```

In this snippet, we follow the approach of the previous step.

10. In the same class, provide a dependency to **`GetUserUseCase`**:

```
@Module
@InstallIn(SingletonComponent::class)
class UseCaseModule {
    …
    @Provides
    fun provideGetUserUseCase(
        configuration: UseCase.Configuration,
        userRepository: UserRepository
    ): GetUserUseCase = GetUserUseCase(
        configuration,
        userRepository
    )
```

```
}
```

In this snippet, we follow the approach of the previous step.

11. In the same class, provide a dependency to **`UpdateInteractionUseCase`**:

```
@Module
@InstallIn(SingletonComponent::class)
class UseCaseModule {
    …
    @Provides
    fun provideUpdateInteractionUseCase(
        configuration: UseCase.Configuration,
        interactionRepository:
InteractionRepository
    ): UpdateInteractionUseCase =
       UpdateInteractionUseCase(
        configuration,
        interactionRepository
    )
}
```

In this snippet, we follow the approach of the previous step.

12. In the **`build.gradle`** file of the **`data-repository`** module, remove the use of the **`kapt`** and Hilt plugins:

```
plugins {
    id 'com.android.library'
    id 'kotlin-android'
}
```

13. In the same file, delete the usages of Hilt from the **`dependencies`** block:

```
dependencies {
    implementation(project(path: ":domain"))
    implementation coroutines.coroutinesAndroid
    testImplementation test.junit
    testImplementation test.coroutines
```

```
    testImplementation test.mockito
}
```

14. Move the **`RepositoryModule`** class from the injection package in the **`data-repository`** module into the injection package in the **`app`** module and make the class not abstract.
15. Delete the use of **`@Inject`** from **`InteractionRepositoryImpl`**:

```
class InteractionRepositoryImpl(
    private val interactionDataSource:
LocalInteractionDataSource
) : InteractionRepository {
    …
}
```

16. Delete the use of **`@Inject`** from **`PostRepositoryImpl`**:

```
class PostRepositoryImpl(
    private val remotePostDataSource:
        RemotePostDataSource,
    private val localPostDataSource:
        LocalPostDataSource
) : PostRepository {
    …
}
```

17. Delete the use of **`@Inject`** from **`UserRepositoryImpl`**:

```
class UserRepositoryImpl(
    private val remoteUserDataSource:
        RemoteUserDataSource,
    private val localUserDataSource:
        LocalUserDataSource
) : UserRepository {
    …
}
```

18. In the **`RepositoryModule`** class, replace the **`bindPostRespository`** method with a **`@Provides`** method:

```
@Module
@InstallIn(SingletonComponent::class)
abstract class RepositoryModule {
```

```
        @Provides
        fun providePostRepository(
            remotePostDataSource: RemotePostDataSource,
            localPostDataSource: LocalPostDataSource
        ): PostRepository = PostRepositoryImpl(
            remotePostDataSource,
            localPostDataSource
        )
        …
    }
```

Here, we are no longer able to use the **`@Binds`** annotation because we removed the **`@Inject`** annotation from the **`PostRepositoryImpl`** class, and because it is an external dependency, we will need to use **`@Provides`**.

19. In the same file, replace the **`bindUserRepository`** method with a **`@Provides`** method:

```
    @Module
    @InstallIn(SingletonComponent::class)
    abstract class RepositoryModule {
        …
        @Provides
        fun provideUserRepository(
            remoteUserDataSource: RemoteUserDataSource,
            localUserDataSource: LocalUserDataSource
        ): UserRepository = UserRepositoryImpl(
            remoteUserDataSource,
            localUserDataSource
        )
        …
    }
```

20. In the same file, replace **`bindInteractionRepositorymethod`** with a **`@Provides`** method:

```
    @Module
    @InstallIn(SingletonComponent::class)
```

```
abstract class RepositoryModule {
    …
    @Provides
    fun provideInteractionRepository(
        interactionDataSource:
            LocalInteractionDataSource
    ): InteractionRepository =
       InteractionRepositoryImpl(
        interactionDataSource
    )
    …
}
```

If we run the application, we should see the same output that we got in *Exercise 09.01 – Transitioning to MVI*:

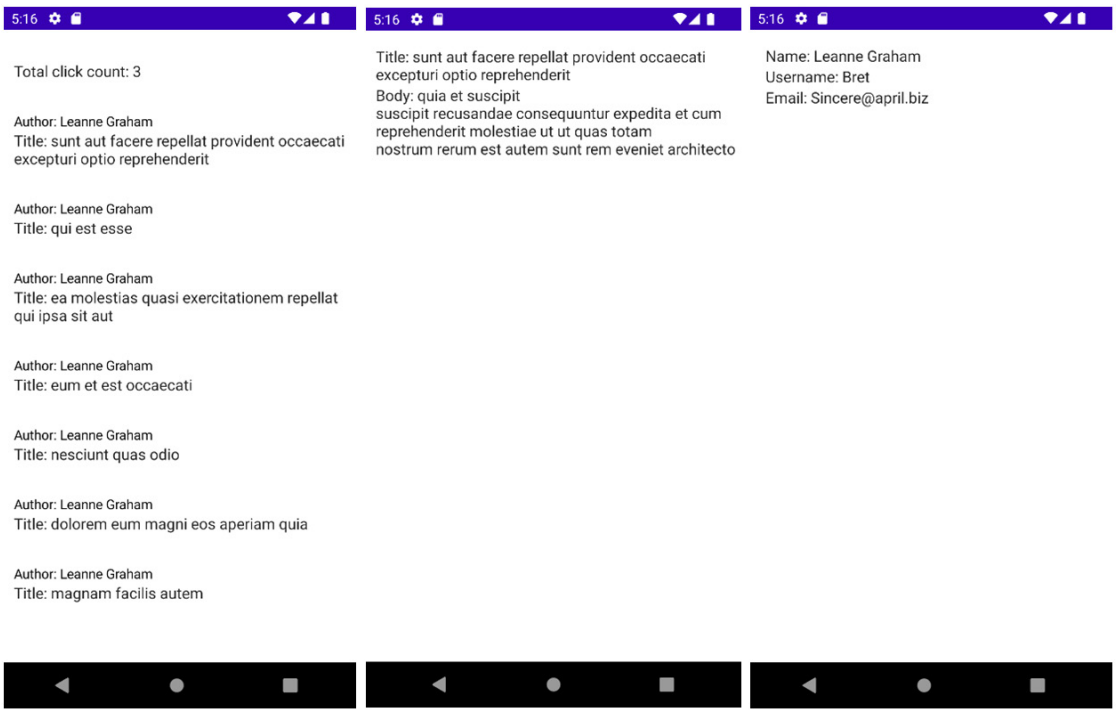

Figure 10.3 – The output of exercise 10.01

The project is now in a state where the **`domain`** and **`data-repository`** modules no longer depend on Hilt. This means that all the other modules that depend on these two will be less exposed to potential issues caused by updates to Hilt. It also means that in the future, if we want to change the de-

pendency injection framework used across the application, the **`domain`** and **`data-repository`** modules will remain unaffected by the change. In the section that follows, we will look at how we can create instrumentation tests with mock data to test that the modules are well integrated and that the data being passed is processed appropriately.

Instrumentation testing

In this section, we will look at how to perform instrumentation testing for an Android application and how we can take advantage of dependency injection to inject either mock data or add test-related logic without modifying the structure of an application's code.

Instrumentation testing is a set of tests that are run on an Android device or emulator and is represented by the tests written in the **`androidTest`** directory. Just like other parts of Android development, instrumentation testing evolved across the years to improve the quality of test code and to provide the ability to create better tests and assertions. Initially, testing was done using test classes such as **`ActivityTestCase`**, **`ContentProviderTestCase`**, and **`ServiceTestCase`**, which were mainly used to test individual components of an application in isolation. The addition of the Espresso testing libraries allows us to easily test multiple activities as part of the journey a user would undertake.

In order to add Espresso and the associated libraries into a project, the following will need to be added to any module's **`build.gradle`** file:

```
dependencies {
    …
    androidTestImplementation
"androidx.test:core:1.4.0"
    androidTestImplementation
"androidx.test:runner:1.4.0 "
```

```
        androidTestImplementation
    "androidx.test:rules:1.4.0 "
        androidTestImplementation
            "androidx.test.ext:junit:1.1.3 "
        androidTestImplementation
            "androidx.test.espresso:espresso-core:3.4.0 "
        androidTestImplementation
    "androidx.test.espresso.
            idling:idling-concurrent:3.4.0 "
    }
```

The following is an example of a test written using Espresso:

```
        @Test
        fun myTest(){
            ActivityScenario.launch(MainActivity::class.j
    ava).
                moveToState(Lifecycle.State.RESUMED)
            onView(withId(R.id.my_id))
                .perform(click())
                .check(isDisplayed())
        }
```

In the preceding example, we use the **`ActivityScenario`** launch method to start **`MainActivity`** and transition **`Activity`** to the **`RESUMED`** state. We then use **`onView`**, which requires **`ViewMatcher`**, and **`withId`** looks up **`View`** by its ID and returns **`ViewMatcher`** holding that information. We then have the option to use **`perform`**, which requires **`ViewAction`**. This is for when we want to interact with certain views. We can also perform **`ViewAssertion`** using the **`check`** method. In this case, we are checking whether a view is displayed.

Another useful addition to help with testing is the orchestrator. The orchestrator is useful when we want to delete the data generated by the tests that might be kept in memory or persisted on the device and that in turn might impact other tests and cause them to malfunction. What the orchestrator does is uninstall the application before each executed test so

that every test will be on a freshly installed app. In order to add the orchestrator to the application, you will need to add it to the module's **`build.gradle`** file:

```
android {
    …
    defaultConfig {
        …
        testInstrumentationRunnerArguments
            clearPackageData: 'true'
        testOptions {
            execution 'ANDROIDX_TEST_ORCHESTRATOR'
        }
        …
    }
}
```

This will add the orchestrator configuration into the test execution and pass the instruction to delete the application data after each test. To add the orchestrator dependency into the project, the following is required:

```
dependencies {
    …
    androidTestUtil "androidx.test:orchestrator:
1.4.1"
}
```

Espresso also comes with many extensions, one of which is the concept of **`IdlingResource`**. When both local tests (tests that are run on the development machine) and instrumented tests are run, they are run on a dedicated set of threads for testing. The Espresso testing library will monitor the main thread of the application, and when it is idle, it will make the assertions required. If the application uses background threads, Espresso will need a way to be informed by this. We can use **`IdlingResource`** to indicate to Espresso wait for an action to complete before continuing its execution. An example of **`IdlingResource`** is **`CountingIdlingResource`**, which will hold a counter for each operation Espresso will need to wait for. The

counter is incremented before each long-running operation and then decremented after the operation is completed. Before each test, **`IdlingResource`** will need to be registered and then unregistered when the test finishes:

```
class MyClass(private val countingIdlingResource:
    CountingIdlingResource) {
    fun doOperation() {
        countingIdlingResource.increment()
        // Perform long running operation
        countingIdlingResource.decrement()
    }
}
```

In the preceding example, we have **`CountingIdlingResource`** being incremented at the beginning of the **`doOperation`** method and decremented after the long operation we intend to perform. To register and unregister **`IdlingResource`**, we can perform the following:

```
    lateinit var countingIdlingResource :
CountingIdlingResource
    @Before
    fun setUp(){
        IdlingRegistry.getInstance().register
            (countingIdlingResource)
    }
    @After
    fun tearDown(){
        IdlingRegistry.getInstance().
            unregister(countingIdlingResource)
    }
```

In this example, we register **`IdlingResource`** in the **`setUp`** method, which is called before each test because of the **`@Before`** annotation, and unregister it in the **`tearDown`** method, which is called after each test because of the **`@After`** annotation.

Because **`IdlingResource`** is a part of Espresso but needs to be used when operations inside the application's code are executed, we want to avoid using **`IdlingResource`** alongside that code. A solution for this is to decorate the class that contains the operation and then use dependency injection to inject the decorated dependency into the test. To decorate the code, we will need to have an abstraction for the operation. An example of this is as follows:

```
interface MyInterface {
    fun doOperation()
}
class MyClass : MyInterface {
    override fun doOperation() {
        // Implement long running operation
    }
}
```

In the preceding example, we have created an interface that defines the **`doOperation`** method, and then we implement the interface with the long-running operation into a class. We can now create a class that will belong to the **`androidTest`** folder, which will decorate the current implementation of the class:

```
class MyDecoratedClass(
    private val myInterface: MyInterface,
    private val countingIdlingResource:
        CountingIdlingResource
) : MyInterface {
    override fun doOperation() {
        countingIdlingResource.increment()
        myInterface.doOperation()
        countingIdlingResource.decrement()
    }
}
```

Here, we have another implementation of **`MyInterface`**, which will hold a reference to the abstraction and **`CountingIdlingResource`**. When **`doOpera-`**

tion is called, we will increment **IdlingResource**, call the operation, and then, when it's done, decrement **IdlingResource**.

If we want to inject the new dependency into the test, we will need first to define a new class that extends **Application**, which will hold the dependency graph containing the test dependencies. If we are using Hilt, it already provides such a class in the form of **HiltTestApplication**. If we want to integrate Hilt into the instrumented tests, we will need the following dependencies to be added to the module's **build.gradle** file:

```
dependencies {
    androidTestImplementation
"com.google.dagger:hilt-
        android-testing:2.40.5"
    kaptAndroidTest "com.google.dagger:hilt-android-
        compiler: 2.40.5"
}
```

To provide the **HiltTestApplication** class to the test, we will need to change the instrumented test runner. An example of a new test runner will look like the following:

```
class MyTestRunner : AndroidJUnitRunner() {
    override fun newApplication(cl: ClassLoader?,
name:
        String?, context: Context?): Application {
        return super.newApplication(cl,
            HiltTestApplication::class.java.name,
context)
    }
}
```

In this example, we are extending from **AndroidJUnitRunner**, and in the **newApplication** method, we invoke the **super** method, and we pass **HiltTestApplication** as the **name**. This means that when the test is executed, **HiltTestApplication** will be used instead of the **Application** class

we defined in our main code. We will now need to change the configuration in the module's **`build.gradle`** file to use the preceding runner:

```
android {
    …
    defaultConfig {
        …
        testInstrumentationRunner
"com.test.MyTestRunner"
        …
        }
    }
}
```

This allows the instrumented test to use the runner we have created. Let's now assume that we have the following module, which will provide the initial dependency:

```
@Module
@InstallIn(SingletonComponent::class)
abstract class MyModule {
    @Binds
    abstract fun bindMyClass(myClass: MyClass):
MyInterface
}
```

Here, we are using a simple binding to connect an implementation to the abstraction. In the **`androidTest`** folder, we can create a new module in which we replace this instance with the decorated one:

```
@Module
@TestInstallIn(
    components = [SingletonComponent::class],
    replaces = [MyModule::class]
)
class MyDecoratedModule {
    @Provides
```

```
    fun provideIdlingResource() =
        CountingIdlingResource("my-idling-resource")
    @Provides
    fun
provideMyDecoratedClass(countingIdlingResource:
        CountingIdlingResource) =
        MyDecoratedClass(MyClass(),
countingIdlingResource)
}
```

In this example, we use the **`@TestInstallIn`** annotation, which will make the dependencies in this module live as long as the test application and replace the dependencies in the previous module. We can then provide dependencies for **`IdlingResource`** and **`MyDecoratedClass`**, which will wrap **`MyClass`** and use **`IdlingResource`**. If we want these changes to take effect in the tests, we will need the following changes:

```
@HiltAndroidTest
class MyActivityTest {
    @get:Rule(order = 0)
    var hiltAndroidRule = HiltAndroidRule(this)
    @Inject
    lateinit var idlingResource:
CountingIdlingResources
    @Before
    fun setUp(){
        hiltAndroidRule.inject()
        IdlingRegistry.getInstance().register
            (idlingResource)
    }
    @After
    fun tearDown(){
       IdlingRegistry.getInstance().unregister
           (idlingResource)
    }
}
```

In this example, we have used the **`@HiltAndroidTest`** annotation because we want to inject **`CountingIdlingResources`** into the test. We then used **`HiltAndroidTestRule`** to perform the injection. We also gave it the highest priority in terms of the order of execution for test rules. Finally, we were able to register and unregister **`CountingIdlingResources`** for each test in the class.

Jetpack Compose comes with its own testing libraries, which require the following configuration to the module's **`build.gradle`** file:

```
dependencies {
    androidTestImplementation
"androidx.compose.ui:ui-test-
        junit4:1.0.5"
    debugImplementation "androidx.compose.ui:ui-test-
        manifest:1.0.5"
}
```

To write tests for Jetpack Compose components, we will need to define a Compose test rule using **`createComposeRule`** when we want to test individual composable methods, or **`createAndroidComposeRule`** if we want to test the Compose content of an entire activity. An example would look like the following:

```
class MyTest {
    @get:Rule
    var composeTestRule = createAndroidComposeRule
        (MyActivity::class.java)
}
```

In the preceding example, we have defined a test rule that will be responsible for testing the Compose content inside **`MyActivity`**. If we want the test to interact with the user interface or assert that it displays the correct information, we have the following structure:

```
    @Test
    fun testDisplayList() {
```

```
        composeTestRule.onNode()
            .assertIsDisplayed()
            .performClick()
    }
```

In this example we use the **`onNode`** method to locate a particular element, such as **`Text`** or **`Button`**. We then have the **`assertIsDisplayed`** method, which is used to check whether the node is displayed. Finally, we have the **`performClick`** method, which will click on the element. Jetpack Compose uses its own **`IdlingResource`** type, which can be registered in the Compose test rule, similar to the following example:

```
    lateinit var idlingResource: IdlingResource
    @Before
    fun setUp() {
        composeTestRule.registerIdlingResource
            (idlingResource)
    }
    @After
    fun tearDown() {
        composeTestRule.unregisterIdlingResource
            (idlingResource)
    }
```

From a clean architecture perspective, we should strive to make our application's code as testable as possible. This applies to both local tests such as unit tests and instrumented tests. We want to be able to ensure that the tests are reliable; this usually means that we will need to remove the dependency on network calls, which means we will need to provide a way to inject mock data into the application without modifying the application's code. We also need to be able to either inject **`IdlingResources`** into the application or use decorated dependencies to verify that the data inserted by the user is the correct data received in the data layer. This also involves the ability to decorate these dependencies to add extra logic without modifying the application's code. In the following section, we will look at an exercise in which we will inject various depen-

dencies containing testing logic into the application and assess the difficulty it takes to introduce them.

Exercise 10.02 – Instrumented testing

Add one instrumented test to *Exercise 10.01 – Reduce dependencies*, which will assert that the following data is displayed onscreen:

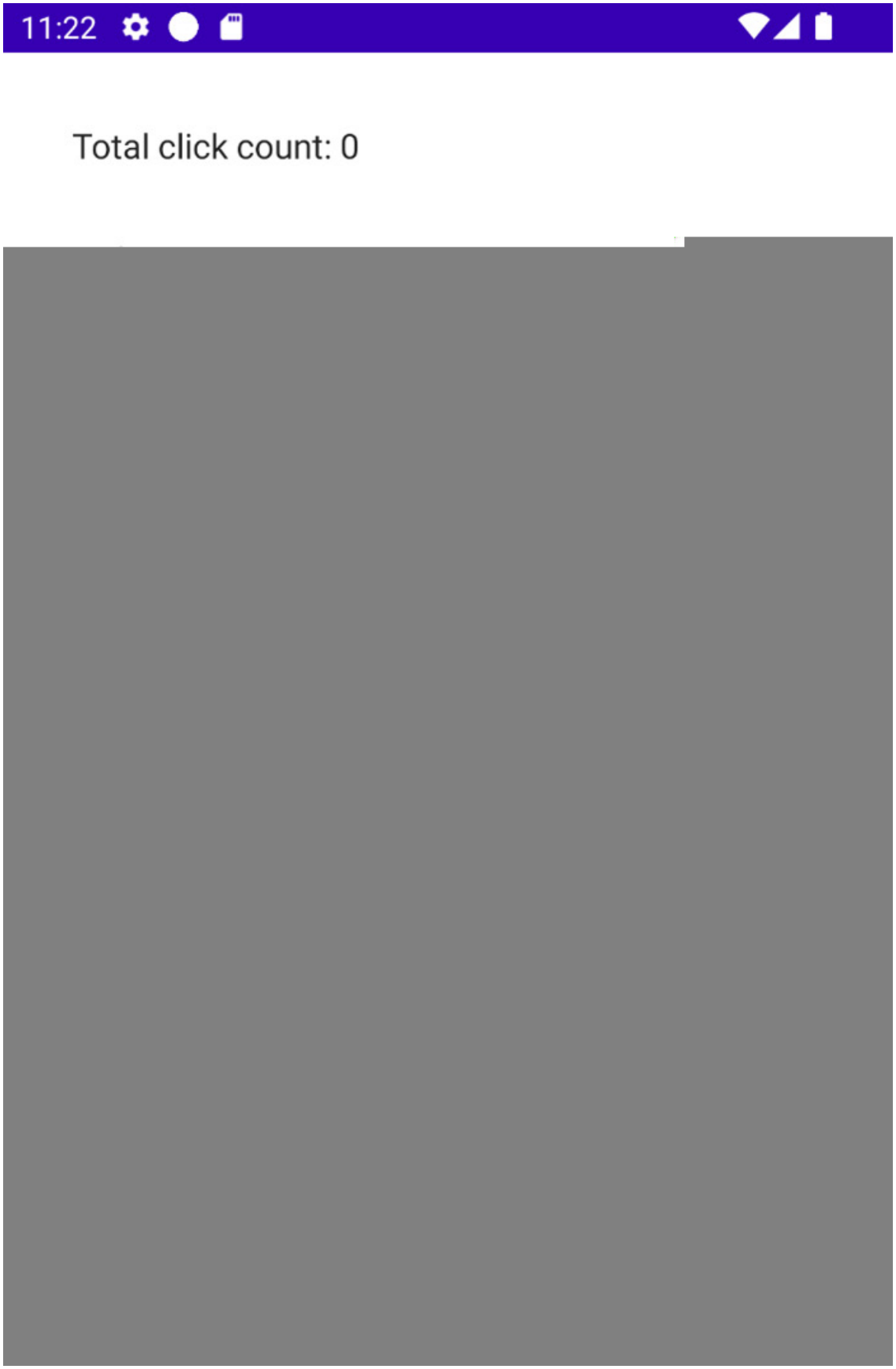

Figure 10.4 – The expected output of exercise 10.02

To achieve this, you will need to create a new implementation of **`RemotePostDataSource`**, which will return a list of four posts; two posts will

belong to one user and the other two will belong to another user. The same thing will need to be done for **`RemoteUserDataSource`**, which will return the two users. These implementations will need to be injected into the test. To ensure that the test will wait for the background work to complete, you will need to decorate each repository with **`IdlingResource`**, which will also need to be injected into the test.

Before completing this exercise, you will need to do the following:

1. Integrate the testing libraries into the app module.
2. Create **`PostAppTestRunner`**, which will be used to provide **`HiltTestApplication`** to the Android instrumentation test runner.
3. Create a **`ComposeCountingIdlingResource`** class, which will wrap an Espresso **`CountingIndlingResource`** and implement the Compose **`IdlingResource`**.
4. Create **`MockRemotePostDataSource`** and **`MockRemoteUserDataSource`**, which will be responsible for returning the users and posts in presented in *Figure 10.4*.
5. Create **`IdlingInteractionRepository`**, **`IdlingUserRepository`**, and **`IdlingPostRespository`**, which will decorate **`InteractionRepository`**, **`UserRepository`**, and **`PostRepository`**, and use the **`ComposeCountingIdlingResource`**, which will be incremented when new data is loaded and decremented when data loading is done.
6. Create **`IdlingRepositoryModule`** and **`MockRemoteDataSourceModule`**, which will replace **`RepositoryModule`** and **`RemoteDataSourceModule`** respectively in the tests.
7. Create **`MainActivityTest`**, which will have one test, and use **`createAndroidComposeRule`** to assert that the list of mock data is displayed.

Follow these steps to complete the exercise:

1. In the top-level **`build.gradle`** file, add the following library versions:

```
buildscript {
    ext {
        …
```

```
        versions = [
                …
                androidTestCore      : "1.4.0",
                androidTestJunit     : "1.1.3",
                orchestrator         : "1.4.1"
        ]
        …
}
```

2. In the same file, make sure that the following **`androidTest`** dependencies are added:

```
buildscript {
    ext {
       …
        androidTest = [
                junit             :
"androidx.test.ext
                    :junit:${versions.espressoJunit
}",
                espressoCore      : "androidx.test.
                    espresso:espresso-
core:${versions.
                        espressoCore}",
                idlingResource    : "androidx.test.
                    espresso:espresso-idling-
resource
                        :${versions.espressoCore}",
                composeUiTestJunit:
"androidx.compose.
                    ui:ui-test-junit4:$
                        {versions.compose}",
                composeManifest   :
"androidx.compose
                    .ui:ui-test-manifest:$
                        {versions.compose}",
                hilt              : "com.google.
```

```
                    dagger:hilt-android-testing:$
                        {versions.hilt}",
                hiltCompiler      : "com.google.
                     dagger:hilt-android-compiler:$
                        {versions.hilt}",
                core              : "androidx.test:
                    core:${versions.androidTestCore
}",
                runner            : "androidx.test:
                    runner:$
                        {versions.androidTestCore}"
,
                rules             : "androidx.test:
                   rules:${versions.androidTestCore
}",
                orchestrator      : "androidx.test:
                    orchestrator:$
                        {versions.orchestrator}"
            ]
    }
    …
}
```

Here, we are defining the mappings for all the testing libraries we will be using so that they will be available across multiple modules.

3. In the **`build.gradle`** file of the app module, add the required test dependencies:

```
dependencies{
    …
    androidTestImplementation androidTest.junit
    androidTestImplementation
androidTest.espressoCore
    androidTestImplementation
        androidTest.idlingResource
```

```
        androidTestImplementation androidTest.core
        androidTestImplementation androidTest.rules
        androidTestImplementation androidTest.runner
        androidTestImplementation androidTest.hilt
        kaptAndroidTest androidTest.hiltCompiler
        androidTestImplementation
            androidTest.composeUiTestJunit
        debugImplementation androidTest.composeManifest
        androidTestUtil androidTest.orchestrator
    }
```

4. In the **androidTest** folder of the app module, create the **PostAppTestRunner** class inside the **java/{package-name}** folder:

```
class PostAppTestRunner : AndroidJUnitRunner() {
    override fun newApplication(cl: ClassLoader?,
        name: String?, context: Context?):
Application {
        return super.newApplication(cl,
            HiltTestApplication::class.java.name,
                context)
    }
}
```

5. In the **build.gradle** file of the app module, set the following test configuration. Make sure to replace **{package-name}** with the package that the **PostAppTestRunner** is in:

```
android {
    …
    defaultConfig {
        …
        testInstrumentationRunner "{package-name}.
            PostAppTestRunner"
        testInstrumentationRunnerArguments
            clearPackageData: 'true'
        testOptions {
            execution 'ANDROIDX_TEST_ORCHESTRATOR'
        }
```

```
        }
    }
```

6. In the **androidTest** folder of the app module, create the following packages inside the **java/{package-name}** folder – **idling**, **injection**, **remote**, **repository**, and **test**.
7. Inside the **idling** package, create a new class called **ComposeCountingIdlingResource**:

```
class ComposeCountingIdlingResource(name: String) :
    IdlingResource {
    private val countingIdlingResource =
        CountingIdlingResource(name)
    override val isIdleNow: Boolean
        get() = countingIdlingResource.isIdleNow
    fun increment() = countingIdlingResource.
        increment()
    fun decrement() = countingIdlingResource.
        decrement()
}
```

Here, we have used the **CountingIdlingResource** class from Espresso to perform the logic for incrementing, decrementing, and providing its current idling state through the **isIdleNow** method, which is used by Jetpack Compose.

8. In the same package, create a file called **IdlingUtils** with the following method:

```
fun <T> Flow<T>.attachIdling(
    countingIdlingResource:
        ComposeCountingIdlingResource
): Flow<T> {
    return onStart {
        countingIdlingResource.increment()
    }.onEach {
        countingIdlingResource.decrement()
    }
```

```
}
```

This is an extension function that we can use to increment **`IdlingResource`** before **`Flow`** is collected and decrement it when the first value of **`Flow`** is being emitted.

9. In the **`repository`** package, create a class called **`IdlingInteractionRepository`**:

```
class IdlingInteractionRepository(
    private val interactionRepository:
InteractionRepository,
    private val countingIdlingResource:
ComposeCountingIdlingResource
) : InteractionRepository {
    override fun getInteraction():
Flow<Interaction> {
        return
interactionRepository.getInteraction()
            .attachIdling(countingIdlingResource)
    }
    override fun saveInteraction(interaction:
        Interaction): Flow<Interaction> {
        return interactionRepository.
            saveInteraction(interaction)
            .attachIdling(countingIdlingResource)
    }
}
```

This class has a reference to the **`ComposeCountingIdlingResource`** object and the **`attachIdling`** method created previously to increment when the data is loaded or saved and to decrement when it's done performing these operations.

10. In the same package, create a class called **`IdlingPostRepository`**:

```
class IdlingPostRepository(
```

```
        private val postRepository: PostRepository,
        private val countingIdlingResource:
            ComposeCountingIdlingResource
    ) : PostRepository {
        override fun getPosts(): Flow<List<Post>> =
            postRepository.getPosts().attachIdling
                (countingIdlingResource)
        override fun getPost(id: Long): Flow<Post> =
            postRepository.getPost(id).
                attachIdling(countingIdlingResource)
    }
```

In this snippet, we follow the approach of the previous step.

11. In the same package, create a class called **`IdlingUserRepository`**:

```
    class IdlingUserRepository(
        private val userRepository: UserRepository,
        private val countingIdlingResource:
            ComposeCountingIdlingResource
    ) : UserRepository {
        override fun getUsers(): Flow<List<User>> =
            userRepository.getUsers()
                .attachIdling(countingIdlingResource)
        override fun getUser(id: Long): Flow<User> =
            userRepository.getUser(id)
                .attachIdling(countingIdlingResource)
    }
```

In this snippet, we follow the approach of the previous step.

12. In the **`injection`** package, create the **`IdlingRepositoryModule`** class:

```
    @Module
    @TestInstallIn(
        components = [SingletonComponent::class],
        replaces = [RepositoryModule::class]
```

```
)
class IdlingRepositoryModule {
}
```

13. In the **`IdlingRepositoryModule`** class, provide a dependency for **`ComposeCountingIdlingResource`**, which will be a single instance across all the repositories:

```
@Module
@TestInstallIn(
    components = [SingletonComponent::class],
    replaces = [RepositoryModule::class]
)
class IdlingRepositoryModule {
    @Singleton
    @Provides
    fun provideIdlingResource():
        ComposeCountingIdlingResource =
        ComposeCountingIdlingResource
            ("repository-idling")
}
```

In this snippet, we are providing a single instance of **`ComposeCountingIdlingResource`** so that when multiple repositories load data at the same time, the same counter will be used for all of them.

14. In the same file, provide a dependency for **`IdlingPostRepository`**:

```
@Module
@TestInstallIn(
    components = [SingletonComponent::class],
    replaces = [RepositoryModule::class]
)
class IdlingRepositoryModule {
    …
    @Provides
    fun providePostRepository(
        remotePostDataSource: RemotePostDataSource,
```

```
            localPostDataSource: LocalPostDataSource,
            countingIdlingResource:
                ComposeCountingIdlingResource
        ): PostRepository = IdlingPostRepository(
            PostRepositoryImpl(
                remotePostDataSource,
                localPostDataSource
            ),
            countingIdlingResource
        )
    }
```

In this snippet, we are providing an instance of the **`IdlingPostRepository`**, which will wrap an instance of **`PostRepositoryImpl`** and have a reference to the **`ComposeCountingIdlingResource`** instance defined previously.

15. In the same file, provide a dependency for **`IdlingUserRepository`**:

```
    @Module
    @TestInstallIn(
        components = [SingletonComponent::class],
        replaces = [RepositoryModule::class]
    )
    class IdlingRepositoryModule {
        …
        @Provides
        fun provideUserRepository(
            remoteUserDataSource: RemoteUserDataSource,
            localUserDataSource: LocalUserDataSource,
            countingIdlingResource:
                ComposeCountingIdlingResource
        ): UserRepository = IdlingUserRepository(
            UserRepositoryImpl(
                remoteUserDataSource,
                localUserDataSource
            ),
```

```
            countingIdlingResource
        )
    }
```

In this snippet, we are providing an instance of **`IdlingUserRepository`**, which will wrap an instance of **`UserRepositoryImpl`** and have a reference to the **`ComposeCountingIdlingResource`** instance defined previously.

16. In the same file, provide a dependency for **`IdlingInteractionRepository`**:

```
    @Module
    @TestInstallIn(
        components = [SingletonComponent::class],
        replaces = [RepositoryModule::class]
    )
    class IdlingRepositoryModule {
        …
        @Provides
        fun provideInteractionRepository(
            interactionDataSource:
                LocalInteractionDataSource,
            countingIdlingResource:
                ComposeCountingIdlingResource
        ): InteractionRepository =
           IdlingInteractionRepository(
            InteractionRepositoryImpl(
                interactionDataSource
            ),
            countingIdlingResource
        )
    }
```

In this snippet, we are providing an instance of **`IdlingInteractionRepository`**, which will wrap an instance of

InteractionRepositoryImpl and have a reference to the **ComposeCountingIdlingResource** instance defined previously.

17. In the **remote** package, create a class called **MockRemoteUserDataSource** and create a list of **User** objects representing the test data:

```
class MockRemoteUserDataSource @Inject
constructor() :
    RemoteUserDataSource {
    private val users = listOf(
        User(
            id = 1L,
            name = "name1",
            username = "username1",
            email = "email1"
        ),
        User(
            id = 2L,
            name = "name2",
            username = "username2",
            email = "email2"
        )
    )
    override fun getUsers(): Flow<List<User>> =
flowOf
        (users)
    override fun getUser(id: Long): Flow<User> =
        flowOf(users[0])
}
```

Here, we have created a list in which we return two users and put it into **Flow** for the **getUsers** method.

18. In the same package, create a class called **MockRemotePostDataSource** and create a list of **Post** objects representing the test data:

```
class MockRemotePostDataSource @Inject
constructor() :
    RemotePostDataSource {
    private val posts = listOf(
        Post(
            id = 1L,
            userId = 1L,
            title = "title1",
            body = "body1"
        ),
        Post(
            id = 2L,
            userId = 1L,
            title = "title2",
            body = "body2"
        ),
        Post(
            id = 3L,
            userId = 2L,
            title = "title3",
            body = "body3"
        ),
        Post(
            id = 4L,
            userId = 2L,
            title = "title4",
            body = "body4"
        )
    )
    override fun getPosts(): Flow<List<Post>> =
        flowOf(posts)
    override fun getPost(id: Long): Flow<Post> =
        flowOf(posts[0])
}
```

Similar to what we did with the users, we create a list of posts and connect the first two posts to the first user and the last two posts to the second user.

19. In the **`injection`** package, create a class called **`MockRemoteDataSourceModule`**, which will be responsible for binding the previous two implementations to the abstractions:

```
@Module
@TestInstallIn(
    components = [SingletonComponent::class],
    replaces = [RemoteDataSourceModule::class]
)
abstract class MockRemoteDataSourceModule {
    @Binds
    abstract fun bindPostDataSource(
        postDataSourceImpl:
MockRemotePostDataSource):
            RemotePostDataSource
    @Binds
    abstract fun
bindUserDataSource(userDataSourceImpl
          : MockRemoteUserDataSource):
              RemoteUserDataSource
}
```

20. In the **`test`** package, create a class called **`MainActivityTest`**:

```
@HiltAndroidTest
class MainActivityTest {
    @get:Rule(order = 0)
    var hiltAndroidRule = HiltAndroidRule(this)
    @get:Rule(order = 1)
    var composeTestRule = createAndroidComposeRule
        (MainActivity::class.java)
    @Inject
    lateinit var idlingResource:
        ComposeCountingIdlingResource
```

```
    @Before
    fun setUp() {
        hiltAndroidRule.inject()
        composeTestRule.
            registerIdlingResource(idlingResource)
    }
    @After
    fun tearDown() {
        composeTestRule.unregisterIdlingResource
            (idlingResource)
    }
}
```

Here, we are initializing our test rules, which are for Hilt and Compose, in that exact order. Then, we inject **`ComposeCountingIdlingResource`** into the test class so that we can register it into the Compose test rule.

21. In the **`MainActivityTest`** class, add a test that will assert that the required data is displayed on the screen:

```
@HiltAndroidTest
class MainActivityTest {
    …
    @Test
    fun testDisplayList() {
        composeTestRule.onNodeWithText("Total click
            count: 0")
            .assertIsDisplayed()
        composeTestRule.onAllNodesWithText("Author:
            name1")
            .assertCountEquals(2)
        composeTestRule.onAllNodesWithText("Author:
            name2")
            .assertCountEquals(2)
        composeTestRule.onNodeWithText("Title:
            title1")
```

```
                .assertIsDisplayed()
            composeTestRule.onNodeWithText("Title:
                title2")
                .assertIsDisplayed()
            composeTestRule.onNodeWithText("Title:
                title3")
                .assertIsDisplayed()
            composeTestRule.onNodeWithText("Title:
                title4")
                .assertIsDisplayed()
        }
    }
```

Here, we have added a test that asserts that the header text is displayed, the two users are displayed for each of their posts, and that each post is displayed.

If we run the test, we should see the following output:

Figure 10.5 – The test output for exercise 10.02

As part of this exercise, we were able to provide mock data to the application without changing any of its existing code by changing the remote data sources and building upon existing functionality, by adding **`IdlingResources`** to our repositories. Both techniques were possible using dependency injection, and because of abstractions, we introduced different layers of the application when we performed dependency inversion. This makes the application's code testable and provides us with the op-

portunity to test different scenarios and create various types of tests to ensure the integration of different components.

Summary

In this chapter, we analyzed the exercises we've done in previous chapters and found potential issues with the dependencies that the modules of the application have. We looked at potential solutions for these problems. Then, we looked at a practical application of clean architecture, which is the implementation of instrumented tests, and how we can change the data sources of an application to ensure testing reliability. We looked at how we can implement instrumented tests using Jetpack Compose and Hilt to provide dependency injection, and then we applied them in an exercise in which we changed the dependencies for the tests. This serves as just one example of the benefits of clean architecture. Other benefits will come in situations where multiple flavors are used to publish similar applications and want to inject different implementations or configurations for each application we want to build. Another benefit comes when dealing with multiple platforms (such as Android and iOS), where we can define entities, use cases and repositories agnostically of the platforms using cross platform frameworks and then inject the implementations for retrieving and persisting the data for each platform.

In *Chapter 9*, *Implementing an MVI Architecture*, we showed how we can change an application's presentation layer without impacting other layers. In a clean application, this should be possible for the data layer as well. We saw how libraries have changed and evolved over time. When networking libraries change, we should be able to transition to new libraries without causing issues in the other modules of an application. The same principle can be applied to local storage. We should be able to change from Room to other ways of persisting data locally. A good rule of thumb for how modules should be created is to view each module as a library that can be released and imagine yourself as the end user. You should have now a good idea of how clean architecture is supposed to

work, the problems it is trying to solve, and how you can apply it to an Android application.