Chapter 6: Assembling a Repository

In this chapter, we will begin by discussing the application's data layer and the components that make up this layer, including repositories and data sources. We will then move on to the topic of repositories, one of the application layer's components, and the role they play in managing the data of an application. In this chapter's exercise, we will continue the project started in the previous chapter by providing the repository implementations for the abstractions defined there and also introducing new abstractions for the different types of data sources.

In this chapter, we will cover the following topics:

- Creating the data layer
- Creating repositories

By the end of the chapter, you will have learned what the data layer is and how we can create repositories for an Android application.

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/Chapter6 ↗️.

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

Creating the data layer

In this section, we will look at the data layer of an Android application and the components that typically form part of the data layer.

The data layer is the layer in which data is created and managed. This means that this layer is responsible for creating, reading, updating, and deleting data, as well as for managing and ensuring that data from the internet is synced with persistent data.

In the previous chapter, we have seen that use cases depend on an abstraction of a repository class, and there can be multiple repositories for different data types. Repositories represent the entry point into the data layer and are responsible for managing multiple data sources and centralizing the data. The data sources represent the other component of the data layer and are responsible for managing the data of a particular source (internet, Room, data store, and suchlike).

An example of what the data layer for a particular set of data, which uses two data sources, might look like is shown in the following figure:

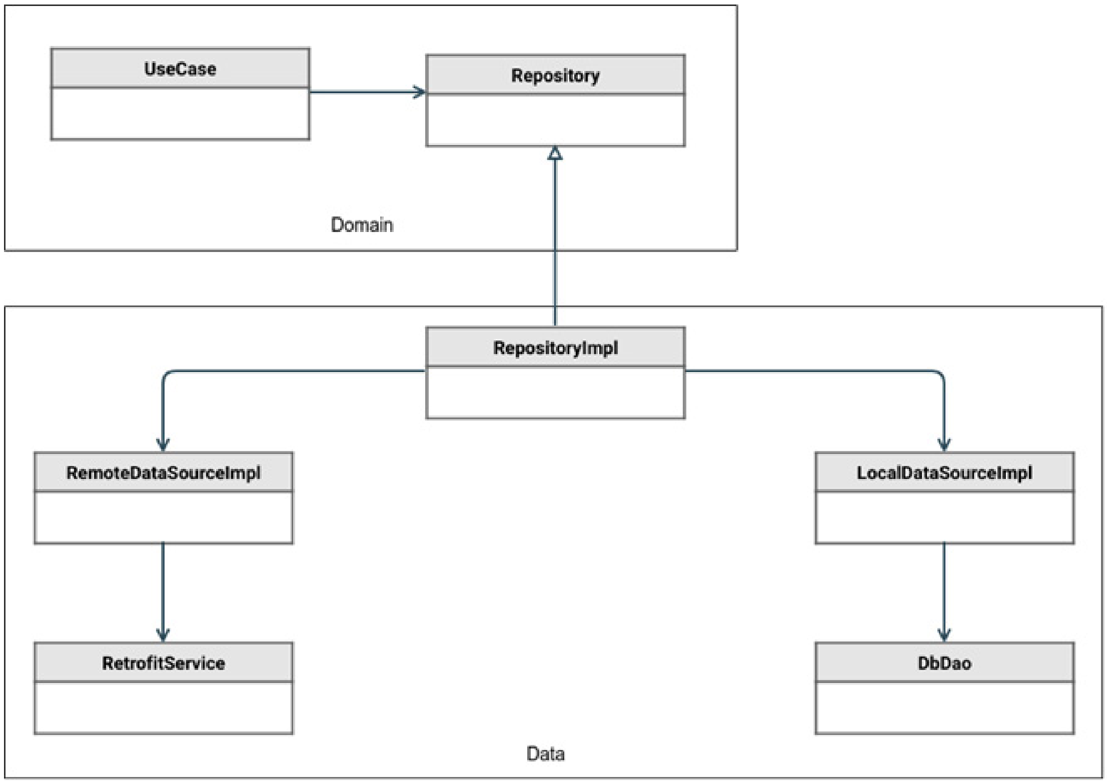


Figure 6.1 – Data layer example

In the preceding diagram, we have an example of what a data layer would look like when it is connected to the domain layer. We can observe that the **`UseCase`** class depends on a **`Repository`** abstraction, which represents the domain layer. The data layer is represented by **`RepositoryImpl`**, which is the implementation of the **`Repository`** abstraction. The **`RepositoryImpl`** class depends on the two data source implementations: **`RemoteDataSourceImpl`** and **`LocalDataSourceImpl`**. Each data source then depends on a particular implementation for managing data from the internet using Retrofit in the case of **`RetrofitService`**, or using a particular data access class that uses Room in the case of **`DbDao`**.

This approach poses a problem owing to the direct dependency between **`RepositoryImpl`** and **`RemoteDataSourceImpl`**, and the problem arises when we might want to swap out Retrofit or Room for alternatives. If we might want to swap out these libraries for others, we risk changes in the **`RepositoryImpl`** class, which violates the single-responsibility principle. The solution for this is like the solution we had for solving the dependencies between the use cases and the repositories, and that is to invert the dependencies between the repository and the data sources. This would look like the following:

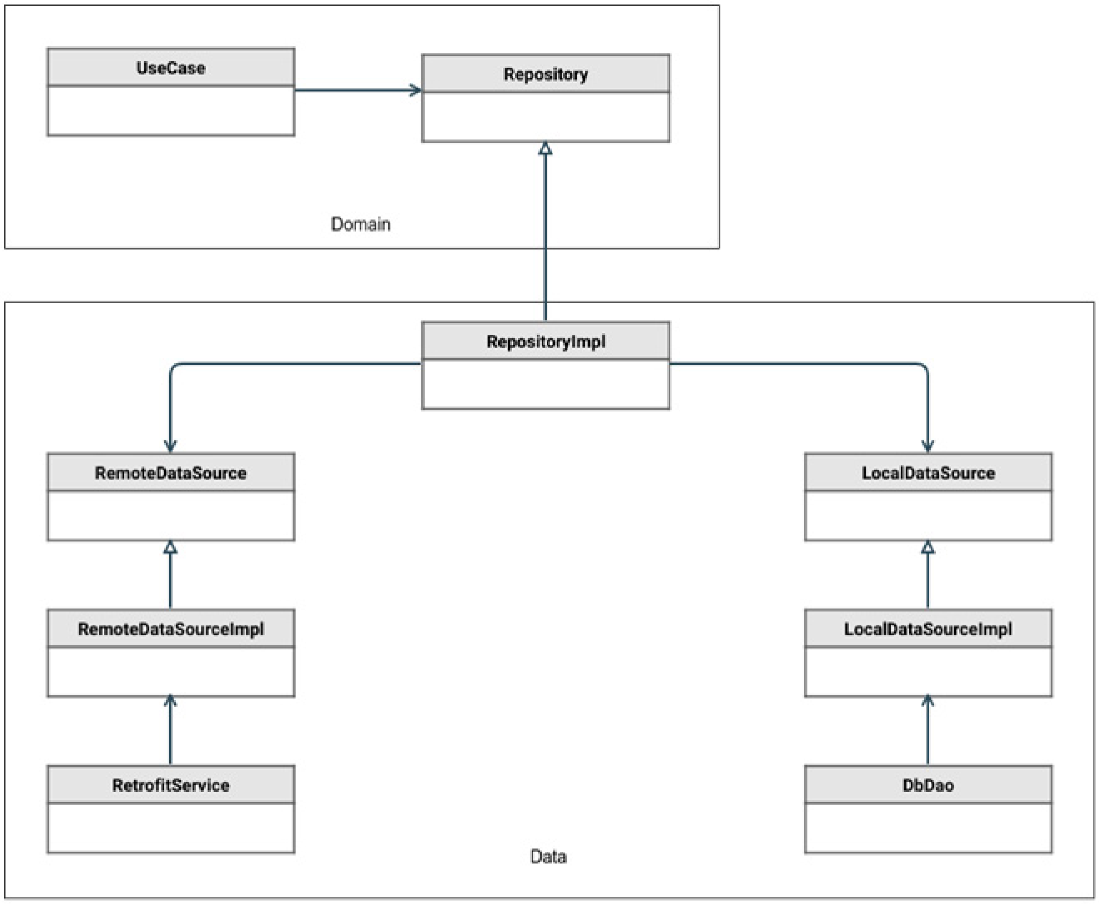


Figure 6.2 – Data layer with inverted dependencies

In the preceding diagram, we have introduced two abstractions for each data source, named **`RemoteDataSource`** and **`LocalDataSource`**. **`RepositoryImpl`** now depends on these two abstractions and all the conversions between Retrofit- or Room-related objects and domain entities should now be placed in **`RemoteDataSourceImpl`** or **`LocalDataSourceImpl`**, which inherit the new abstractions and will continue to handle the data from Retrofit or Room. If we want to split the data layer into different Gradle modules, we will have the following:

Figure 6.3 – Data layer modules

The preceding diagram shows the Gradle module dependencies between the repository and local and remote data sources. Here we can see the benefit of dependency inversion, which allows us to have a separate repository module without depending on Retrofit or Room.

In this section, we have discussed the data layer and the components inside it and how to manage the dependencies between all the components. In the following section, we will take a closer look at repositories and how to implement them.

Creating repositories

In this section, we will look at what a repository is and the role it plays in the data layer of an application, and how we can create repositories with various data sources.

The repository represents an abstraction for the data than an application uses, and it is responsible for managing and centralizing the data from one or multiple data sources.

In the previous chapter, we defined the following entity:

```
data class User(
    val id: String,
    val firstName: String,
    val lastName: String,
    val email: String
) {
    fun getFullName() = "$firstName $lastName"
}
```

Here we have a simple **`User`** data class with a few relevant fields. The repository abstraction for the **`User`** data is as follows:

```
interface UserRepository {
    fun getUser(id: String): Flow<User>
}
```

Here we have an interface named **`UserRepository`** that is responsible for fetching the user information in a Kotlin flow.

If we want to fetch data from the internet, we must first define a **`UserRemoteDataSource`** abstraction:

```
interface UserRemoteDataSource {
    fun getUser(id: String): Flow<User>
}
```

In this case, we have an interface similar to how **`UserRepository`** is defined with a simple method to retrieve a **`User`** object. We can now implement **`UserRepository`** to use this data source:

```
class UserRepositoryImpl(private val
userRemoteDataSource:
    UserRemoteDataSource) : UserRepository {

    override fun getUser(id: String): Flow<User> =
        userRemoteDataSource.getUser(id)

}
```

Here we have a dependency on **`UserRemoteDataSource`** and invoke the **`getUser`** method. If we want to persist the remote user data locally, we will need to define a **`UserLocalDataSource`** abstraction, which will be responsible for inserting the user:

```
interface UserLocalDataSource {
    suspend fun insertUser(user: User)
}
```

Here we have a method for inserting a user into the local store. We can now update **`UserRepositoryImpl`** to connect the data sources and insert a user after it was retrieved:

```
class UserRepositoryImpl(
    private val userRemoteDataSource:
UserRemoteDataSource,
    private val userLocalDataSource:
UserLocalDataSource
) : UserRepository {
    override fun getUser(id: String): Flow<User> =
        userRemoteDataSource.getUser(id)
        .onEach {
            userLocalDataSource.insertUser(it)
        }
}
```

This represents a simple use case for data sources, but we can use repositories to improve the user experience for the user. For instance, we can change the repository implementation to return the saved data and have a separate method for fetching the data remotely. We can take advantage of flows, which can emit multiple users in a stream:

```
interface UserLocalDataSource {
    suspend fun insertUser(user: User)
    fun getUser(id: String): Flow<User>
}
```

In the preceding example, we have added the **`getUser`** method to retrieve a **`User`** object, which was persisted locally. We will need to modify the repository abstraction as follows:

```
interface UserRepository {
    fun getUser(id: String): Flow<User>
    fun refreshUser(id: String): Flow<User>
}
```

Here, we have added the **`refreshUser`** method, which, when implemented, will be responsible for fetching a new user from the internet. The implementation will be as follows:

```
class UserRepositoryImpl(
    private val userRemoteDataSource:
UserRemoteDataSource,
    private val userLocalDataSource:
UserLocalDataSource
) : UserRepository {
    override fun getUser(id: String): Flow<User> =
        userLocalDataSource.getUser(id)
    override fun refreshUser(id: String): Flow<User>
=
        userRemoteDataSource.getUser(id)
        .onEach {
            userLocalDataSource.insertUser(it)
        }
}
```

Here, we return the persisted user in the **`getUser`** method and, in the **`refreshUser`** method, we now fetch the remote data and insert it locally. If we are using libraries such as Room, this will trigger the emission of a new **`User`** object, which will come from **`UserLocalDataSource`**. This means that all subscribers of the **`getUser`** method will be notified of a change and receive a new **`User`** object.

We can also use repositories for caching data in the memory. An example of this would be as follows:

```
class UserRepositoryImpl(
    private val userRemoteDataSource:
UserRemoteDataSource,
    private val userLocalDataSource:
UserLocalDataSource
) : UserRepository {
    private val usersFlow = MutableStateFlow
        (emptyMap<String, User>().toMutableMap())
    override fun getUser(id: String): Flow<User> =
        usersFlow.flatMapLatest {
        val user = it[id]
        if (user != null) {
            flowOf(user)
        } else {
            userLocalDataSource.getUser(id)
                .onEach { persistedUser ->
                    saveUser(persistedUser)
                }
        }
    }
    override fun refreshUser(id: String): Flow<User>
=
        userRemoteDataSource.getUser(id)
        .onEach {
            saveUser(it)
            userLocalDataSource.insertUser(it)
        }
    private fun saveUser(user: User) {
        val map = usersFlow.value
        map[user.id] = user
        usersFlow.value = map
    }
    }
```

Here, we have added a new **`MutableStateFlow`** object, which will hold a map in which the keys are represented by the user IDs and the values are the users. In the **`getUser`** method, we check whether the user is stored in memory and return the memory value if present, otherwise we get the persisted data, which we will store in memory after. In the **`refreshUser`** method, we persist the value in memory and persist the data locally.

Because we defined the repository abstraction to return entities, we should try as much as possible to use entities across the repository and the data source abstractions. However, we might need specific object definitions to handle processing the data from the data sources. We can define these specific classes in this layer and then convert them to entities in the repository implementation.

In this section, we have seen how we can create repositories and how they can be used to manage data in an application. In the section that follows, we will look at an exercise in which we will create repositories for an application.

Exercise 06.01 – Creating repositories

Modify *Exercise 05.01: Building a domain layer*, so that a new library module is created in Android Studio. The module will be named **`data-repository`** and will have a dependency on the **`domain`** module. In this module, we will implement the repository classes from the domain module as follows:

- **`UserRepositoryImpl`** will have dependencies on the following data sources: **`UserRemoteDataSource`**, which will fetch a list and a user by ID, and **`UserLocalDataSource`**, which will have methods for inserting a list of users and obtaining a list of the same. **`UserRepositoryImpl`** will always load the remote users and insert them locally.
- **`PostRepositoryImpl`** will have dependencies on the following data sources: **`PostRemoteDataSource`**, which will fetch a list of users and a

user by ID, and **`PostLocalDataSource`**, which will have methods for inserting a list of posts and obtaining a list of the same. **`PostRepositoryImpl`** will always load the remote posts and insert them locally.
- **`InteractionRepositoryImpl`** will have a dependency on a single data source, **`LocalInteractionDataSource`**, which will be responsible for loading an interaction and saving it. **`InteractionRepositoryImpl`** will load the interaction and save a new interaction.

To complete this exercise, you will need to do the following:

- Create the data repository module in Android Studio
- Create the user's data sources and repository
- Create the post's data sources and repository
- Create the interaction data source and repository

Follow these steps to complete the exercise:

1. Create a new module named **`data-repository`**, which will be an Android Library module.
2. Make sure that in the top-level **`build.gradle`** file, the following dependencies are set:
```
buildscript {
     …
    dependencies {
        classpath gradlePlugins.android
        classpath gradlePlugins.kotlin
        classpath gradlePlugins.hilt
    }
}
```
3. In the **`build.gradle`** file of the **`data-repository`** module, make sure that the following plugins are present:
```
plugins {
    id 'com.android.library'
    id 'kotlin-android'
```

```
        id 'kotlin-kapt'
        id 'dagger.hilt.android.plugin'
    }
```

4. In the same file, change the configurations to the ones defined in the top-level **build.gradle** file:

```
    android {
        compileSdk defaultCompileSdkVersion
        defaultConfig {
            minSdk defaultMinSdkVersion
            targetSdk defaultTargetSdkVersion
            …
        }
        …
        compileOptions {
            sourceCompatibility javaCompileVersion
            targetCompatibility javaCompileVersion
        }
        kotlinOptions {
            jvmTarget = jvmTarget
        }
    }
```

5. In the same file, make sure that the following dependencies are specified:

```
    dependencies {
        implementation(project(path: ":domain"))
        implementation coroutines.coroutinesAndroid
        implementation di.hiltAndroid
        kapt di.hiltCompiler
        testImplementation test.junit
        testImplementation test.coroutines
        testImplementation test.mockito
    }
```

Here, we are using the **implementation** method to add a dependency to the **:domain** module, in the same way as other libraries are referenced. In

Gradle we also have the option of using the **api** method. This makes a module's dependencies public to other modules. This, in turn, might have potential side effects, such as leaking dependencies that should be kept private. In this example, we might be better served by using the **api** method for the **:domain** module because of the close relationship between the two modules (which would make all modules that depend on **:data-repository** not have to add the dependency to **:domain**). However, dependencies such as Hilt and Coroutines should be kept with the implementation method because we would want to avoid exposing these libraries in modules that do not use them.

6. In the **data-repository** module, create a new package named **data_source**.
7. In the **data_source** package, create a new package named **remote**.
8. In the **remote** package, create the **RemoteUserDataSource** interface:
```
interface RemoteUserDataSource {
    fun getUsers(): Flow<List<User>>
    fun getUser(id: Long): Flow<User>
}
```
9. In the **remote** package, create the **RemotePostDataSource** interface:
```
interface RemotePostDataSource {
    fun getPosts(): Flow<List<Post>>
    fun getPost(id: Long): Flow<Post>
}
```
10. In the **data_source** package, create a new package called **local**.
11. In the **local** package, create the **LocalUserDataSource** interface:
```
interface LocalUserDataSource {
    fun getUsers(): Flow<List<User>>
    suspend fun addUsers(users: List<User>)
}
```
12. In the **local** package, create the **LocalPostDataSource** interface:
```
interface LocalPostDataSource {
    fun getPosts(): Flow<List<Post>>
    suspend fun addPosts(posts: List<Post>)
}
```

13. In the **`local`** package, create the **`LocalInteractionDataSource`** package:

    ```
    interface LocalInteractionDataSource {
        fun getInteraction(): Flow<Interaction>
        suspend fun saveInteraction(interaction:
    Interaction)
    }
    ```

14. Next to the **`data_source`** package, create a new package named **`reposi-tory`**.
15. In the **`repository`** package, create the **`UserRepositoryImpl`** class:

    ```
    class UserRepositoryImpl @Inject constructor(
        private val remoteUserDataSource:
            RemoteUserDataSource,
        private val localUserDataSource:
            LocalUserDataSource
    ) : UserRepository {
        override fun getUsers(): Flow<List<User>> =
            remoteUserDataSource.getUsers()
            .onEach {
                localUserDataSource.addUsers(it)
            }
        override fun getUser(id: Long): Flow<User> =
    remoteUserDataSource.getUser(id)
            .onEach {
                localUserDataSource.addUsers(listOf(it)
    )
            }
    }
    ```

Here, we fetch the user data from the remote data source and store it locally.

16. In the same package, create the **`PostRepositoryImpl`** class:

    ```
    class PostRepositoryImpl @Inject constructor(
        private val remotePostDataSource:
            RemotePostDataSource,
    ```

```
        private val localPostDataSource:
            LocalPostDataSource
    ) : PostRepository {
        override fun getPosts(): Flow<List<Post>> =
            remotePostDataSource.getPosts()
            .onEach {
                localPostDataSource.addPosts(it)
            }
        override fun getPost(id: Long): Flow<Post> =
            remotePostDataSource.getPost(id)
            .onEach {
                localPostDataSource.addPosts(listOf(it)
    )
            }
    }
```

Here, we are fetching the post data from the remote data source and using the local data source to persist the data.

17. In the same package, create the **`InteractionRepositoryImpl`** class:

```
    class InteractionRepositoryImpl @Inject
    constructor(
        private val interactionDataSource:
            LocalInteractionDataSource
    ) : InteractionRepository {
        override fun getInteraction():
    Flow<Interaction> =
            interactionDataSource.getInteraction()
        override fun saveInteraction(interaction:
           Interaction): Flow<Interaction> = flow {
            interactionDataSource.saveInteraction(inter
    action)
            this.emit(Unit)
        }.flatMapLatest {
            getInteraction()
        }
    }
```

Here, we are just interacting with the local data source to read and store the data.

18. We now want to use Hilt to bind the repository abstraction with the implementation, so we will need to create a package named **`injection`** next to the **`data_source`** and **`repository`** packages.
19. Inside the **`injection`** package, create a class named **`RepositoryModule`**:

```
    @Module
    @InstallIn(SingletonComponent::class)
    abstract class RepositoryModule {
        @Binds
        abstract fun
    bindPostRepository(postRepositoryImpl
            : PostRepositoryImpl): PostRepository
        @Binds
        abstract fun bindUserRepository
            (userRepositoryImpl: UserRepositoryImpl):
                UserRepository
        @Binds
        abstract fun bindInteractionRepository
            (interactionRepositoryImpl:
                InteractionRepositoryImpl):
                    InteractionRepository
    }
```

Here, we are using the **`@Binds`** Hilt annotation, which maps the implementation of a repository annotated with **`@Inject`** with the abstraction.

20. To unit test the code, we will now need to create a new folder called **`resources`** in the test folder of the **`data-repository`** module.
21. Inside the resources folder, create a folder called **`mockito-extensions`** and, inside this folder, create a file named

org.mockito.plugins.MockMaker, and, inside this file, add the following text: **mock-maker-inline**.

22. Create a **UserRepositoryImplTest** class for unit testing the **UserRepositoryImpl** methods:

```
class UserRepositoryImplTest {
    private val remoteUserDataSource =
        mock<RemoteUserDataSource>()
    private val localUserDataSource =
        mock<LocalUserDataSource>()
    private val repositoryImpl = UserRepositoryImpl
        (remoteUserDataSource, localUserDataSource)


}
```

23. In the **UserRepositoryImplTest** class, add a test method for each repository method:

```
class UserRepositoryImplTest {
     …
    @ExperimentalCoroutinesApi
    @Test
    fun testGetUsers() = runBlockingTest {
        val users = listOf(User(1, "name",
"username",
            "email"))
        whenever(remoteUserDataSource.getUsers()).
            thenReturn(flowOf(users))
        val result =
repositoryImpl.getUsers().first()
        assertEquals(users, result)
        verify(localUserDataSource).addUsers(users)
    }
    @ExperimentalCoroutinesApi
    @Test
    fun testGetUser() = runBlockingTest {
        val id = 1L
        val user = User(id, "name", "username",
            "email"
)
        whenever(remoteUserDataSource.getUser(id))
            .thenReturn(flowOf(user))
        val result = repositoryImpl.getUser(id).
            first()
        assertEquals(user, result)
        verify(localUserDataSource).addUsers(listOf
(user))
    }
}
```

In this class, we unit test each of the methods in the **`UserRepositoryImpl`** class by mocking the local data and remote data sources and verifying that the data obtained from the remote data source is inserted into the local data source.

24. Create a **`PostRepositoryImplTest`** class to test the **`PostRepositoryImpl`** class:

```
class PostRepositoryImplTest {
    private val remotePostDataSource =
        mock<RemotePostDataSource>()
    private val localPostDataSource =
        mock<LocalPostDataSource>()
    private val repositoryImpl = PostRepositoryImpl
        (remotePostDataSource, localPostDataSource)
}
```

25. Create unit tests for each of the methods in the **`PostRepositoryImpl`** class:

```
class PostRepositoryImplTest {
    …
    @ExperimentalCoroutinesApi
    @Test
    fun testGetPosts() = runBlockingTest {
        val posts = listOf(Post(1, 1, "title",
            "body"))
        whenever(remotePostDataSource.getPosts())
            .thenReturn(flowOf(posts))
        val result =
repositoryImpl.getPosts().first()
        Assert.assertEquals(posts, result)
        verify(localPostDataSource).addPosts(posts)
    }
    @ExperimentalCoroutinesApi
    @Test
    fun testGetPost() = runBlockingTest {
        val id = 1L
        val post = Post(id, 1, "title", "body")
        whenever(remotePostDataSource.getPost(id)).
thenReturn(flowOf(post))
        val result =
            repositoryImpl.getPost(id).first()
        Assert.assertEquals(post, result)
        verify(localPostDataSource).addPosts(listOf
(post))
    }
}
```

In this class, we perform the same tests that we did for **UserRepositoryImpl**.

26. Create an **InteractionRepositoryImplTest** class to test the **InteractionRepositoryImpl** class:

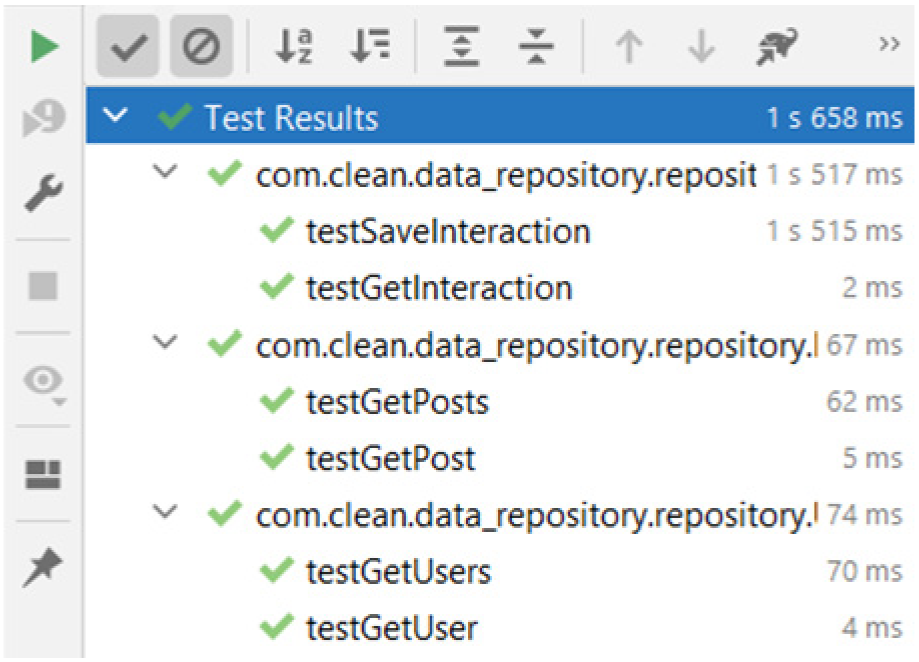
```
    class InteractionRepositoryImplTest {
        private val localInteractionDataSource =
            mock<LocalInteractionDataSource>()
        private val repositoryImpl =
            InteractionRepositoryImpl
            (localInteractionDataSource)
    }
```

27. Create unit tests for each of the methods in the **InteractionRepositoryImpl** class:

```
class InteractionRepositoryImplTest {
    …
    @ExperimentalCoroutinesApi
    @Test
    fun testGetInteraction() = runBlockingTest {
        val interaction = Interaction(10)
        whenever(localInteractionDataSource.
            getInteraction()).
                thenReturn(flowOf(interaction))
        val result =
repositoryImpl.getInteraction()
            .first()
        assertEquals(interaction, result)
    }
    @ExperimentalCoroutinesApi
    @Test
    fun testSaveInteraction() = runBlockingTest {
        val interaction = Interaction(10)
        whenever(localInteractionDataSource.
            getInteraction()).thenReturn
                (flowOf(interaction))
        val result = repositoryImpl.saveInteraction
            (interaction).first()
        veriy(localInteractionDataSource).
             saveInteraction(interaction)
        assertEquals(interaction, result)
    }
}
```

In this class, we mock the local data source and then we verify that the repository has the appropriate invocations on the **LocalInteractionDataStore** mock.

If we run the tests, we should see something like the following screenshot:



```
Test Results                                      1 s 658 ms
  com.clean.data_repository.reposit               1 s 517 ms
    testSaveInteraction                           1 s 515 ms
    testGetInteraction                                  2 ms
  com.clean.data_repository.repository.l               67 ms
    testGetPosts                                       62 ms
    testGetPost                                         5 ms
  com.clean.data_repository.repository.l               74 ms
    testGetUsers                                       70 ms
    testGetUser                                         4 ms
```

Figure 6.4 – Output of the repository unit tests

In this exercise, we have created a new module in which we implemented our repositories and defined new abstractions for the data sources that the repositories will use. Here, we have continued the integration with other libraries, such as Hilt for dependency injection, and Kotlin flows to handle the data in a reactive approach. The use of dependency injection made the unit tests simple to write because we could easily provide mocks.

Summary

In this chapter, we started looking into the data layer of an Android application and provided an overview of the components that are part of this layer. We also looked at the Repository component, which is responsible for managing the data provided by one or more data sources, and provided examples of how we could build different repositories. We also looked at the relationship between repositories and data sources and how we can further decouple the components with dependency inversion, to keep our repositories unaffected by changes in libraries used to fetch data. Finally, we looked at an exercise on how we can build repositories with local and remote data sources. In the following chapter, we will continue with the data layer and how we can integrate the remote and local data sources with libraries such as Room and Retrofit.