# *Chapter 7*: Building Data Sources

In this chapter, we will continue focusing on the data layer by discussing how we can implement local and remote data sources and the roles they play in clean architecture. First, we will look at how remote data sources can be built and how they can fetch data from the internet through calls to Retrofit. Then, we will look at implementing local data sources and how they can interact with Room and Data Store to persist data locally. In the chapter's exercises, we will continue the previous exercises and add the data sources discussed in the chapter, seeing how we can connect them to Room and Retrofit.

In this chapter, we will cover the following topics:

- Building and using remote data sources
- Building and integrating local data sources

By the end of the chapter, you will have learned the role of data sources, how to implement remote and local data sources that use Retrofit, Room, and Data Store to manage an application's data, and how we can separate these data sources in separate library modules.

## 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/Chapter7 ↗.

Check out the following video to see the Code in Action:
https://bit.ly/3yOa7jE ↗️

## Building and using remote data sources

In this section, we will look at how we can build remote data sources and how we can use them in combination with Retrofit to fetch and manipulate data from the internet.

In the previous chapters, we defined abstractions for data sources that repositories depend on to manipulate data. This was because we wanted to avoid the repositories having dependencies on the data sources and instead have the data sources depend on the repositories. For remote data sources, this looks something like the following figure:

Figure 7.1 – A remote data source class diagram

The implementation of the remote data source has two roles. It will invoke the networking layer to fetch and manipulate data, and it will convert the data to either the domain entity or, if necessary, intermediary data required by the repository.

Let's look at the entity defined in the previous chapters:

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

Here, we have the same **`User`** data class that was defined as part of the domain. Now let's assume we are fetching the following data from the internet in JSON format:

```
data class UserApiModel(
    @Json(name = "id") val id: String,
    @Json(name = "first_name") val firstName: String,
    @Json(name = "last_name") val lastName: String,
    @Json(name = "email") val email: String
)
```

Here, we have a **`UserApiModel`** class in which we define the same fields as the **`User`** class and annotate them with the **`@Json`** annotation, which is part of the Moshi library.

The remote data source abstraction looks like the following:

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

This is the abstraction we defined in the previous chapter. Before we write the implementation of this class, we will first need to specify our Retrofit service:

```
interface UserService {
    @GET("/users/{userId}")
    suspend fun getUser(@Path("userId") userId:
String):
        UserApiModel
}
```

This is a typical Retrofit service class, which will fetch an **`UserApiModel`** class from the **`/users/{userId}`** endpoint. We can now create the implementation of the data source to fetch the user from **`UserService`**:

```
data class UserRemoteDataSourceImpl(private val
userService: UserService) : UserRemoteDataSource {
    override fun getUser(id: String): Flow<User> {
        return flow {
            emit(userService.getUser(id))
        }.map {
            User(it.id, it.firstName, it.lastName,
                it.email)
        }
    }
}
```

Here, we implement the **UserRemoteDataSource** interface, and in the **getUser** method, we invoke the **getUser** method from the **UserService** dependency. Once **UserApiModel** is obtained, we then convert it to the **User** class.

In this section, we looked at how we can build a remote data source with the help of the Retrofit library to manipulate data from the internet. In the section that follows, we will look at an exercise that shows how we can implement a remote data source.

## Exercise 07.01 – Building a remote data source

Modify *Exercise 06.01 – Creating repositories* so that a new library module is created in Android Studio. Name the module **data-remote**. This module will depend on **domain** and **data-repository**. The module will be responsible for fetching users and posts as JSON from https://jsonplaceholder.typicode.com/ ↗.

The user will have the following JSON representation:

```
    {
        "id": 1,
        "name": "Leanne Graham",
        "username": "Bret",
        "email": "Sincere@april.biz"
    }
```

The post will have the following JSON representation:

```
{
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident
        occaecati excepturi optio reprehenderit",
    "body": "quia et suscipit\nsuscipit recusandae
consequuntur expedita et cum\nreprehenderit molestiae
ut ut quas totam\nnostrum rerum est autem sunt rem
eveniet architecto"
}
```

The module will need to implement the following:

- **`UserApiModel`** and **`PostApiModel`**, which will hold the data from the JSON.
- **`UserService`**, which will return a list of **`UserApiModel`** from the **`/users`** URL and **`UserApiModel`** based on the ID from the **`/users/{userId}`** URL.
- **`PostService`**, which will return a list of **`PostApiModel`** from the **`/posts`** URL and **`PostApiModel`** based on the ID from the **`/post/{postId}`** URL.
- **`RemoteUserDataSourceImpl`**, which will implement **`RemoteUserDataSource`**, call **`UserService`**, and return **`Flow`**, which emits a list of **`User`** objects or **`UseCaseException.UserException`** if there is an error in the call to **`UserService`**. The same approach will be taken for returning **`User`** based on the ID.
- **`RemotePostDataSourceImpl`** which will implement **`RemotePostDataSource`**, call **`PostService`**, and return **`Flow`**, which emits a list of **`Post`** objects or **`UseCaseException.PostException`** if there is an error in the call to **`PostService`**. The same approach will be taken for returning a post based on the ID.

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

1. Create the **`data-remote`** module.
2. Create the **`UserApiModel`** and **`UserService`** classes.

3. Create the **`PostApiModel`** and **`PostService`** classes.
4. Create the remote data sources implementations for **`RemoteUserDataSource`** and **`RemotePostDataSource`**.

Follow these steps to complete the exercise:

1. Create a new module named **`data-remote`**, 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 same file, add the networking libraries to the library mappings:

```
    ext {
        …
        versions = [
                …
                okHttp             : "4.9.0",
                retrofit           : "2.9.0",
                moshi              : "1.13.0",
                …
        ]
        …
        network = [
                okHttp       :
"com.squareup.okhttp3:
                    okhttp:${versions.okHttp}",
                retrofit     :
"com.squareup.retrofit2
```

```
                    :retrofit:${versions.retrofit}"
,
                retrofitMoshi:
"com.squareup.retrofit2
                    :converter-moshi:$
                        {versions.retrofit}",
                moshi        : "com.squareup.moshi:
                    moshi:${versions.moshi}",
                moshiKotlin  : "com.squareup.moshi:
                    moshi-kotlin:${versions.moshi}"
        ]
        …
    }
```

4. In the **build.gradle** file of the **data-remote** 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'
}
```

5. 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
    }
}
```

Here, we are making sure that the new module will use the same configurations with regards to compilation and the minimum and maximum Android version as the rest of the project, making it easier to change the configuration across all the modules.

6. In the same file, add the dependencies to the networking libraries and the **`data-repository`** and **`domain`** modules:

```
dependencies {
    implementation(project(path: ":domain"))
    implementation(project(path: ":data-
repository"))
    implementation coroutines.coroutinesAndroid
    implementation network.okHttp
    implementation network.retrofit
    implementation network.retrofitMoshi
    implementation network.moshi
    implementation network.moshiKotlin
    implementation di.hiltAndroid
    kapt di.hiltCompiler
    testImplementation test.junit
    testImplementation test.coroutines
    testImplementation test.mockito
}
```

7. In the top-level **`gradle.properties`**, add the following configuration for **`moshi`**:

```
android.jetifier.ignorelist=moshi-1.13.0
```

8. In the **`AndroidManifest.xml`** file in the **`data-remote`** module, add the internet permission:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/a
```

```
ndroid"
    package="com.clean.data_remote">
    <uses-permission
android:name="android.permission.INTERNET" />
</manifest>
```

9. In the **data-remote** module, create a new package called **networking**.
10. In the **networking** package, create a new package called **user**.
11. In the **user** package, create a new class called **UserApiModel**:

```
data class UserApiModel(
    @Json(name = "id") val id: Long,
    @Json(name = "name") val name: String,
    @Json(name = "username") val username: String,
    @Json(name = "email") val email: String
)
```

12. In the same package, create a new interface called **UserService**:

```
interface UserService {
    @GET("/users")
    suspend fun getUsers(): List<UserApiModel>
    @GET("/users/{userId}")
    suspend fun getUser(@Path("userId") userId:
Long):
        UserApiModel
}
```

13. In the **networking** package, create a new package called **post**.
14. In the **post** package, create a new class called **PostApiModel**:

```
data class PostApiModel(
    @Json(name = "id") val id: Long,
    @Json(name = "userId") val userId: Long,
    @Json(name = "title") val title: String,
    @Json(name = "body") val body: String
)
```

15. In the same package, create a new interface called **PostService**:

```
interface PostService {
    @GET("/posts")
    suspend fun getPosts(): List<PostApiModel>
```

```
        @GET("/posts/{postId}")
        suspend fun getPost(@Path("postId") id: Long):
            PostApiModel
    }
```

16. In the **`data-remote`** module, create a new package called **`source`**.
17. In the **`source`** package, create a new class called **`RemoteUserDataSourceImpl`**:

```
    class RemoteUserDataSourceImpl @Inject
    constructor(private val userService: UserService) :
        RemoteUserDataSource {
        override fun getUsers(): Flow<List<User>> =
    flow {
            emit(userService.getUsers())
        }.map { users ->
            users.map { userApiModel ->
                convert(userApiModel)
            }
        }.catch {
            throw UseCaseException.UserException(it)
        }
        override fun getUser(id: Long): Flow<User> =
    flow {
            emit(userService.getUser(id))
        }.map {
            convert(it)
        }.catch {
            throw UseCaseException.UserException(it)
        }

        private fun convert(userApiModel: UserApiModel)
    =
            User(userApiModel.id, userApiModel.name,
                userApiModel.username,
    userApiModel.email)
    }
```

Here, we invoke the **getUsers** and **getUser** methods from **UserService** and then convert the **UserApiModel** objects to **User** objects to avoid the other layers depending on the networking-related data. The same principle applies to error handling. If there is a network error, such as an **HTTP 404** code, the exception will be **HttpException**, which is part of the Retrofit library.

18. In the **source** package, create a new class called **RemotePostDataSourceImpl**:

```
class RemotePostDataSourceImpl @Inject
constructor(private val postService: PostService) :
    RemotePostDataSource {

    override fun getPosts(): Flow<List<Post>> =
flow {
        emit(postService.getPosts())
    }.map { posts ->
        posts.map { postApiModel ->
            convert(postApiModel)
        }
    }.catch {
        throw UseCaseException.PostException(it)
    }
    override fun getPost(id: Long): Flow<Post> =
flow {
        emit(postService.getPost(id))
    }.map {
        convert(it)
    }.catch {
        throw UseCaseException.PostException(it)
    }
    private fun convert(postApiModel: PostApiModel)
=
        Post(postApiModel.id, postApiModel.userId,
            postApiModel.title, postApiModel.body)
}
```

Here, we follow the same principle as with the **RemoteUserDataSourceImpl** class.

19. In the **data-remote** module, create a new package called **injection**:
20. In the **injection** package, create a new class called **NetworkModule**:

```
@Module
@InstallIn(SingletonComponent::class)
class NetworkModule {
    @Provides
    fun provideOkHttpClient(): OkHttpClient =
        OkHttpClient
        .Builder()
        .readTimeout(15, TimeUnit.SECONDS)
        .connectTimeout(15, TimeUnit.SECONDS)
        .build()
    @Provides
    fun provideMoshi(): Moshi = Moshi.Builder().add
        (KotlinJsonAdapterFactory()).build()
    @Provides
    fun provideRetrofit(okHttpClient: OkHttpClient,
        moshi: Moshi): Retrofit =
Retrofit.Builder()
        .baseUrl
             ("https://jsonplaceholder.typicode.com
/")
        .client(okHttpClient)
        .addConverterFactory
            (MoshiConverterFactory.create(moshi))
        .build()
    @Provides
    fun provideUserService(retrofit: Retrofit):
        UserService =
        retrofit.create(UserService::class.java)
    @Provides
    fun providePostService(retrofit: Retrofit):
        PostService =
        retrofit.create(PostService::class.java)
}
```

Here, we provide the Retrofit and **`OkHttp`** dependencies required for networking.

21. In the **`injection`** package, create a class named **`RemoteDataSourceModule`**:

```
    @Module
    @InstallIn(SingletonComponent::class)
    abstract class RemoteDataSourceModule {
        @Binds
        abstract fun
    bindPostDataSource(postDataSourceImpl:
    RemotePostDataSourceImpl): RemotePostDataSource
        @Binds
        abstract fun bindUserDataSource
            (userDataSourceImpl:
                RemoteUserDataSourceImpl):
    RemoteUserDataSource
    }
```

Here, we use Hilt to bind the implementations from this module with the abstractions defined in the **`data-repository`** module.

22. To unit-test the code, we now need to create a new folder called **`re-sources`** in the **`test`** folder of the **`data-remote`** module.
23. Inside the **`resources`** folder, create a folder called **`mockito-extensions`**; inside this folder, create a file named **`org.mockito.plugins.MockMaker`**; and inside this file, add the following text – **`mock-maker-inline`**.
24. Create a test class named **`RemoteUserDataSourceImplTest`**, which will test the success scenarios for the methods inside

**RemoteUserDataSourceImpl**:

```
class RemoteUserDataSourceImplTest {
    private val userService = mock<UserService>()
    private val userDataSource =
        RemoteUserDataSourceImpl(userService)
    @ExperimentalCoroutinesApi
    @Test
    fun testGetUsers() = runBlockingTest {
        val remoteUsers = listOf(UserApiModel(1,
            "name", "username", "email"))
        val expectedUsers = listOf(User(1, "name",
            "username", "email"))
        whenever(userService.getUsers()).
            thenReturn(remoteUsers)
        val result =
userDataSource.getUsers().first()
        Assert.assertEquals(expectedUsers, result)
    }
    @ExperimentalCoroutinesApi
    @Test
    fun testGetUser() = runBlockingTest {
        val id = 1L
        val remoteUser = UserApiModel(id, "name",
            "username", "email")
        val user = User(id, "name", "username",
            "email")
        whenever(userService.getUser(id))
            .thenReturn(remoteUser)
        val result = userDataSource.getUser(id).
            first()
        Assert.assertEquals(user, result)
    }
}
```

Here, we are mocking the **UserService** interface and providing mock user data, which will then be obtained and converted by **RemoteDataSourceImpl**.

25. In the same test class, add the error scenarios:

```
class RemoteUserDataSourceImplTest {
    …
    @ExperimentalCoroutinesApi
    @Test
    fun testGetUsersThrowsError() = runBlockingTest
{
        whenever(userService.getUsers()).thenThrow
            (RuntimeException())
        userDataSource.getUsers().catch {
            Assert.assertTrue(it is
UseCaseException.
                UserException)
        }.collect()
    }
    @ExperimentalCoroutinesApi
    @Test
    fun testGetUserThrowsError() = runBlockingTest
{
        val id = 1L
        whenever(userService.getUser(id)).thenThrow
            (RuntimeException())
        userDataSource.getUser(id).catch {
            Assert.assertTrue(it is
UseCaseException.
                UserException)
        }.collect()
    }
}
```

Here, we are mocking an error that is thrown by **UserService**, which will then be converted by **RemoteUserDataSourceImpl** into

**UseCaseException.UserException**.

26. Create a test class named **RemotePostDataSourceImplTest**, which will have similar test methods as **RemoteUserDataSourceImplTest** for posts:

```
class RemotePostDataSourceImplTest {
    private val postService = mock<PostService>()
    private val postDataSource =
        RemotePostDataSourceImpl(postService)
    @ExperimentalCoroutinesApi
    @Test
    fun testGetPosts() = runBlockingTest {
        val remotePosts = listOf(PostApiModel(1, 1,
            "title", "body"))
        val expectedPosts = listOf(Post(1, 1,
"title",
            "body"))
        whenever(postService.getPosts()).thenReturn
            (remotePosts)
        val result =
postDataSource.getPosts().first()
        Assert.assertEquals(expectedPosts, result)
    }
    @ExperimentalCoroutinesApi
    @Test
    fun testGetPost() = runBlockingTest {
        val id = 1L
        val remotePost = PostApiModel(id, 1,
"title",
            "body")
        val expectedPost = Post(id, 1, "title",
            "body")
        whenever(postService.getPost(id)).thenRetur
n
            (remotePost)
        val result = postDataSource.getPost(id).
```

```
            first()
        Assert.assertEquals(expectedPost, result)
    }
}
```

Here, we are doing for posts what we did for users in **`RemoteUserDataSourceImplTest`**.

27. Add the error scenarios in **`RemotePostDataSourceImplTest`**:

```
class RemotePostDataSourceImplTest {
    …
    @ExperimentalCoroutinesApi
    @Test
    fun testGetPostsThrowsError() = runBlockingTest
{
        whenever(postService.getPosts()).thenThrow
            (RuntimeException())
        postDataSource.getPosts().catch {
            Assert.assertTrue(it is
UseCaseException.
                PostException)
        }.collect()
    }
    @ExperimentalCoroutinesApi
    @Test
    fun testGetPostThrowsError() = runBlockingTest
{
        val id = 1L
        whenever(postService.getPost(id)).thenThrow
            (RuntimeException())
        postDataSource.getPost(id).catch {
            Assert.assertTrue(it is
UseCaseException.
                PostException)
        }.collect()
        }
    }
```

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

Figure 7.2 – Output of the remote data source unit tests

In this exercise, we have added a new module to the application, in which we can see how we can add a remote data source to the application. To fetch the data, we are using libraries such as OkHttp and Retrofit and combining them with the data source implementation for fetch users and posts. In the following section, we will expand the application to introduce local data sources, in which we will persist the data we are fetching here.

## Building and integrating local data sources

In this section, we will analyze how we can build local data sources and integrate them with libraries such as Room and Data Store.

Local data sources have a similar structure to remote data sources. The abstractions are provided by the layers sitting above, and the implementations are responsible for invoking methods from persistence frameworks and converting data into entities, like the following figure:

Figure 7.3 – A local data source diagram

Let's assume we have the same **`UserEntity`** defined in the previous chapters:

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

Let's make the same assumption about **`UserLocalDataSource`**:

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

We now need to provide an implementation for this data source that will manipulate the data from Room. First, we need to define a user entity for Room:

```
@Entity(tableName = "user")
data class UserEntity(
    @PrimaryKey @ColumnInfo(name = "id") val id:
String,
    @ColumnInfo(name = "first_name") val firstName:
String,
    @ColumnInfo(name = "last_name") val lastName:
String,
    @ColumnInfo(name = "email") val email: String
)
```

Now, we can define **`UserDao`**, which queries a user by an ID and inserts a user:

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user where id = :id")
    fun getUser(id: String): Flow<UserEntity>
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertUser(users: UserEntity)
}
```

Finally, the implementation of the data source looks like this:

```
class UserLocalDataSourceImpl(private val userDao:
UserDao) : UserLocalDataSource {
    override suspend fun insertUser(user: User) {
        userDao.insertUser(UserEntity(user.id,
            user.firstName, user.lastName,
user.email))
    }
    override fun getUser(id: String): Flow<User> {
        return userDao.getUser(id).map {
            User(it.id, it.firstName, it.lastName,
                it.email)
        }
    }
}
```

Here, the local data source invokes **`UserDao`** to insert and retrieve a user and converts the domain entity into a Room entity.

If we want to use Data Store instead of Room with a local data store implementation, we can have something like the following example:

```
private val KEY_ID = stringPreferencesKey("key_id")
private val KEY_FIRST_NAME =
    stringPreferencesKey("key_first_name")
private val KEY_LAST_NAME =
    stringPreferencesKey("key_last_name")
private val KEY_EMAIL =
stringPreferencesKey("key_email")
class UserLocalDataSourceImpl(private val dataStore:
    DataStore<Preferences>) : UserLocalDataSource {
    override suspend fun insertUser(user: User) {
        dataStore.edit {
            it[KEY_ID] = user.id
            it[KEY_FIRST_NAME] = user.firstName
            it[KEY_LAST_NAME] = user.lastName
            it[KEY_EMAIL] = user.email
        }
    }
    override fun getUser(id: String): Flow<User> {
        return dataStore.data.map {
            User(
                it[KEY_ID].orEmpty(),
                it[KEY_FIRST_NAME].orEmpty(),
                it[KEY_LAST_NAME].orEmpty(),
                it[KEY_EMAIL].orEmpty()
            )
        }
    }
}
```

Here, we use a key for each of the fields of the **`User`** object to store the data. The **`getUser`** method doesn't use the ID to search for a user, which shows that for this particular use case, Room is the more appropriate method.

In this section, we looked at how we can build a local data source with the help of the Room and Data Store libraries to be able to query and persist data locally on a device. Next, we will look at an exercise to show how we can implement a local data store.

## Exercise 07.02 – Building a local data source

Modify *Exercise 07.01 – Building a remote data source* so that a new Android library module named **`data-local`** is created. This module will depend on **`domain`** and **`data-repository`**.

The module will implement the following:

- **`UserEntity`** and **`PostEntity`**, which will hold data to be persisted from **`User`** and **`Post`** using Room

- **UserDao** and **PostDao**, which will be responsible for persisting and fetching a list of **UserEntity** and **PostEntity**
- **LocalUserDataSourceImpl** and **LocalPostDataSourceImpl**, which will be responsible for invoking the **UserDao** and **PostDao** objects to persist data and for converting data to **User** and **Post** objects
- **LocalInteractionDataSourceImpl**, which will be responsible for persisting the **Interaction** object

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

1. Create the **data-local** module.
2. Create the **UserEntity** and **PostEntity** classes.
3. Create the DAOs for users and posts.
4. Create the data source implementations.

Follow these steps to complete the exercise:

1. Create a new module named **data-local**, 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 same file, add the persistence libraries to the library mappings:
```
    ext {
        …
        versions = [
                …
                room               : "2.4.0",
                datastore          : "1.0.0",
                …
        ]
        …
        persistence = [
                roomRuntime : "androidx.room:room-
                    runtime:${versions.room}",
                roomKtx     : "androidx.room:room-
                     ktx:${versions.room}",
                roomCompiler: "androidx.room:room-
                    compiler:${versions.room}",
                datastore   : "androidx.datastore:
                    datastore-preferences:$
                       {versions.datastore}"
        ]
        …
    }
```

4. In the **build.gradle** file of the **data-local** 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'
}
```

5. 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
    }
}
```

6. In the same file, add the dependencies to the networking libraries and the **data-repository** and **domain** modules:

```
    dependencies {
        implementation(project(path: ":domain"))
        implementation(project(path: ":data-
    repository"))
        implementation coroutines.coroutinesAndroid
        implementation persistence.roomRuntime
        implementation persistence.roomKtx
        kapt persistence.roomCompiler
        implementation persistence.datastore
        implementation di.hiltAndroid
        kapt di.hiltCompiler
        testImplementation test.junit
        testImplementation test.coroutines
        testImplementation test.mockito
    }
```

7. In the **data-local** module, create a new package called **db**.
8. In the **db** package, create a new package called **user**.
9. In the **user** package, create the **UserEntity** class:

```
    @Entity(tableName = "user")
    data class UserEntity(
        @PrimaryKey @ColumnInfo(name = "id") val id:
    Long,
        @ColumnInfo(name = "name") val name: String,
        @ColumnInfo(name = "username") val username:
            String,
        @ColumnInfo(name = "email") val email: String
```

```
)
```

10. In the same package, create the **UserDao** interface:

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getUsers(): Flow<List<UserEntity>>
    @Insert(onConflict =
OnConflictStrategy.REPLACE)
    fun insertUsers(users: List<UserEntity>)
}
```

11. In the **db** package, create a new package called **post**.
12. In the **post** package, create a new class called **PostEntity**:

```
@Entity(tableName = "post")
data class PostEntity(
    @PrimaryKey @ColumnInfo(name = "id") val id:
Long,
    @ColumnInfo(name = "userId") val userId: Long,
    @ColumnInfo(name = "title") val title: String,
    @ColumnInfo(name = "body") val body: String
)
```

13. In the same package, create a new interface called **PostDao**:

```
@Dao
interface PostDao {
    @Query("SELECT * FROM post")
    fun getPosts(): Flow<List<PostEntity>>
    @Insert(onConflict =
OnConflictStrategy.REPLACE)
    fun insertPosts(users: List<PostEntity>)
}
```

14. In the **db** package, create the **AppDatabase** class:

```
@Database(entities = [UserEntity::class,
PostEntity::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    abstract fun postDao(): PostDao
}
```

15. In the **`data-local`** module, create a new package called **`source`**.
16. In the **`source`** package, create a new class called **`LocalUserDataSourceImpl`**:

```
class LocalUserDataSourceImpl @Inject
constructor(private val userDao: UserDao) :
    LocalUserDataSource {
    override fun getUsers(): Flow<List<User>> =
        userDao.getUsers().map { users ->
        users.map {
            User(it.id, it.name, it.username,
                it.email)
        }
    }
    override suspend fun addUsers(users:
List<User>) =
        userDao.insertUsers(users.map {
        UserEntity(it.id, it.name, it.username,
            it.email)
    })
}
```

Here, in the **`getUsers`** method, we retrieve a list of **`UserEntity`** objects from **`UserDao`** and convert them into **`User`** objects. In the **`addUsers`** method, we do the opposite, by taking a list of **`User`** objects to be inserted and converting them into **`UserEntity`** objects.

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

```
class LocalPostDataSourceImpl @Inject
constructor(private val postDao: PostDao) :
    LocalPostDataSource {
    override fun getPosts(): Flow<List<Post>> =
       postDao.getPosts().map { posts ->
        posts.map {
            Post(it.id, it.userId, it.title,
it.body)
        }
    }
    override suspend fun addPosts(posts:
List<Post>) =
        postDao.insertPosts(posts.map {
        PostEntity(it.id, it.userId, it.title,
            it.body)
    })
}
```

Here, we follow the same approach we used for **`LocalUserDataSourceImpl`**.

18. In the same package, create the **`LocalInteractionDataSourceImpl`** class:

```
    internal val KEY_TOTAL_TAPS =
    intPreferencesKey("key_total_taps")
    class LocalInteractionDataSourceImpl @Inject
    constructor(private val dataStore:
    DataStore<Preferences>) :
        LocalInteractionDataSource {
        override fun getInteraction():
    Flow<Interaction> {
            return dataStore.data.map {
                Interaction(it[KEY_TOTAL_TAPS] ?: 0)
            }
        }
        override suspend fun
    saveInteraction(interaction:
            Interaction) {
            dataStore.edit {
                it[KEY_TOTAL_TAPS] =
                    interaction.totalClicks
            }
        }
```

```
}
```

Here, we use the Preference Data Store library to persist the Interaction object, by holding different keys for each field in the **`Interaction`** class, and in this case, it will be just one key for the total clicks.

19. In the **`data-local`** module, create a new package named **`injection`**.
20. In the **`injection`** package, create a new class named **`PersistenceModule`**:

```
val Context.dataStore: DataStore<Preferences> by
preferencesDataStore(name = "my_preferences")
@Module
@InstallIn(SingletonComponent::class)
class PersistenceModule {
    @Provides
    fun provideAppDatabase(@ApplicationContext
        context: Context): AppDatabase =
        Room.databaseBuilder(
            context,
            AppDatabase::class.java, "my-database"
        ).build()
    @Provides
    fun provideUserDao(appDatabase: AppDatabase):
        UserDao = appDatabase.userDao()
    @Provides
    fun providePostDao(appDatabase: AppDatabase):
        PostDao = appDatabase.postDao()
    @Provides
    fun provideLocalInteractionDataSourceImpl
        (@ApplicationContext context: Context) =
        LocalInteractionDataSourceImpl(context.data
Store)
}
```

Here, we provide all the Data Store and Room dependencies.

21. In the same package, create a new class called **LocalDataSourceModule**, in which we connect the abstractions to the bindings:

```
@Module
@InstallIn(SingletonComponent::class)
abstract class LocalDataSourceModule {
    @Binds
    abstract fun bindPostDataSource
       (lostDataSourceImpl:
LocalPostDataSourceImpl):
           LocalPostDataSource
    @Binds
    abstract fun bindUserDataSource
        (userDataSourceImpl:
LocalUserDataSourceImpl):
            LocalUserDataSource
    @Binds
    abstract fun bindInteractionDataStore
        (interactionDataStore:LocalInteractionData
            SourceImpl): LocalInteractionDataSource
}
```

22. To unit-test the code, we will now need to create a new folder called **resources** in the test folder of the **data-local** module.
23. Inside the **resources** folder, create a folder called **mockito-extensions**; inside this folder, create a file named **org.mockito.plugins.MockMaker**; and inside this file, add the following text – **mock-maker-inline**.
24. Create the **LocalUserDataSourceImplTest** test class:

```
class LocalUserDataSourceImplTest {
    private val userDao = mock<UserDao>()
    private val userDataSource =
        LocalUserDataSourceImpl(userDao)
    @ExperimentalCoroutinesApi
    @Test
    fun testGetUsers() = runBlockingTest {
        val localUsers = listOf(UserEntity(1,
"name",
            "username", "email"))
        val expectedUsers = listOf(User(1, "name",
            "username", "email"))
        whenever(userDao.getUsers()).thenReturn
            (flowOf(localUsers))
        val result =
userDataSource.getUsers().first()
        Assert.assertEquals(expectedUsers, result)
    }
    @ExperimentalCoroutinesApi
    @Test
    fun testAddUsers() = runBlockingTest {
        val localUsers = listOf(UserEntity(1,
"name",
            "username", "email"))
        val users = listOf(User(1, "name",
"username",
            "email"))
        userDataSource.addUsers(users)
        verify(userDao).insertUsers(localUsers)
    }
}
```

Here, we are mocking the **`UserDao`** class and using it to provide mock data to **`LocalUserDataSourceImpl`**, which will then convert the data to and from the **`User`** objects.

25. Create the **`LocalPostDataSourceImplTest`** test class:

```
    class LocalPostDataSourceImplTest {
        private val postDao = mock<PostDao>()
        private val postDataSource =
            LocalPostDataSourceImpl(postDao)
        @ExperimentalCoroutinesApi
        @Test
        fun testGetPosts() = runBlockingTest {
```

```
        val localPosts = listOf(PostEntity(1, 1,
            "title", "body"))
        val expectedPosts = listOf(Post(1, 1,
"title",
            "body"))
        whenever(postDao.getPosts()).thenReturn
            (flowOf(localPosts))
        val result =
postDataSource.getPosts().first()
        Assert.assertEquals(expectedPosts, result)
    }
    @ExperimentalCoroutinesApi
    @Test
    fun testAddUsers() = runBlockingTest {
        val localPosts = listOf(PostEntity(1, 1,
            "title", "body"))
        val posts = listOf(Post(1, 1, "title",
            "body"))
        postDataSource.addPosts(posts)
        verify(postDao).insertPosts(localPosts)
    }
}
```

Here, we perform the same type of tests for posts as we did in **`LocalUserDataSourceImplTest`** for users.

26. Create the **`LocalInteractionDataSourceImplTest`** test class:

```
class LocalInteractionDataSourceImplTest {
    private val dataStore = mock<DataStore
        <Preferences>>()
    private val interactionDataSource =
        LocalInteractionDataSourceImpl(dataStore)
    @ExperimentalCoroutinesApi
    @Test
    fun testGetInteraction() = runBlockingTest {
        val clicks = 10
        val interaction = Interaction(clicks)
        val preferences = mock<Preferences>()
        whenever(preferences[KEY_TOTAL_TAPS]).
            thenReturn(clicks)
        whenever(dataStore.data).thenReturn
            (flowOf(preferences))
        val result = interactionDataSource.
            getInteraction().first()
        assertEquals(interaction, result)
    }
    @ExperimentalCoroutinesApi
    @Test
    fun testSaveInteraction() = runBlockingTest {
        val clicks = 10
        val interaction = Interaction(clicks)
        val preferences = mock<MutablePreferences>
()
        whenever(preferences.toMutablePreferences()
)
            .thenReturn(preferences)
        whenever(dataStore.updateData(any())).
            thenAnswer {
            runBlocking {
                it.getArgument<suspend
(Preferences) -
                 > Preferences>
(0).invoke(preferences)
            }
            preferences
        }
        interactionDataSource.saveInteraction(inter
action)
        verify(preferences)[KEY_TOTAL_TAPS] =
clicks
```

```
        }
    }
```

Here, in the **`testSaveInteraction`** method, we need to mock the **`updateData`** method instead of the **`edit`** method from the **`DataStore`** class. This is because the **`edit`** method is an extension function that can't be mocked with the current libraries we have and instead must rely on the method it invokes, which is **`updateData`**.

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

Figure 7.4 – Output of the local data source unit tests

If we draw a diagram of the modules in the exercise, we will see something like the following figure:

Figure 7.5 – The exercise 07.02 module diagram

We can see that the **`:data-remote`** and **`:data-local`** modules are isolated from each other. The two modules have different responsibilities and deal with different dependencies. **`:data-remote`** deals with fetching data from the internet, while **`:data-local`** deals with persisting data locally into SQLite using Room and files using Data Store. This gives our code more flexibility because we are able to change how we fetch data – for example, without impacting how we persist the data.

In this exercise, we have created a new module in the application in which we deal with local data sources. To persist data, we have used libraries such as Room and Data Store, and we have integrated them with the local data store.

## Summary

In this chapter, we looked at the concept of data sources and the different types of data sources we have available in an Android application. We started with remote data sources and saw some examples of how we can build a data source and combine it with libraries such as Retrofit and OkHttp. The local data source followed similar principles as the remote one, and here, we have used libraries such as Room and Data Store to implement this.

In the exercises, we implemented the data sources as part of different modules. This was to avoid creating any unnecessary dependencies between the other layers of the application and the specific frameworks we have used for the data sources. In the next chapter, we will look at how we can build the presentation layer and show data to the user. We will also explore how we can split the presentation layer into separate modules and navigate from a screen in one module to a screen in another module, through the introduction of modules that can be shared by other presentation modules.