Chapter 4: Managing Dependencies in Android Applications

In this chapter, we will analyze the concept of **dependency injection** (**DI**) and the benefits it provides and look at how this was done in the past in Android applications either through manual injection or using Dagger 2. We will go over some of the libraries used in Android applications, stopping and looking in more detail at the Hilt library and how it simplifies DI for an Android application.

In this chapter, we will cover the following topics:

- Introduction to DI
- Using Dagger 2 to manage dependencies
- Using Hilt to manage dependencies

By the end of this chapter, you will be familiar with the DI pattern and libraries such as Dagger and Hilt, which can be used to manage dependencies in Android applications.

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/Chapter4 ↗️.

Check out the following video to see the Code in Action: https://bit.ly/38yFDHz ↗️

Introduction to DI

In this section, we will look at what DI is, the benefits it provides, and how this concept is applied to an Android application. We will then look at some DI libraries and how they work.

When a class depends on functionality from another class, a dependency is created between the two classes. To invoke the functionality on the class you depend on, you will need to instantiate it, as in the following example:

```
class ClassA() {
    private val b: ClassB = ClassB()
    fun executeA() {
        b.executeB()
    }
}
class ClassB() {
    fun executeB() {

    }
}
```

In this example, **`ClassA`** creates a new instance of **`ClassB`**, and then when **`executeA`** is invoked, it will invoke **`executeB`**. This poses a problem because **`ClassA`** will have the extra responsibility of creating **`ClassB`**. Let's see what happens if **`ClassB`** needs to change to something such as the following:

```
class ClassB(private val myFlag: Boolean) {

    fun executeB() {
        if (myFlag) {
            // Do something
        } else {
            // Do something else
```

```
        }
    }
}
```

Here, we added the **`myFlag`** variable to **`ClassB`**, which is used in the **`executeB`** method. This change would cause a compile error because now **`ClassA`** will need to be modified to make the code compile.

```
class ClassA() {
    private val b: ClassB = ClassB(true)
    fun executeA() {
        b.executeB()
    }
}
```

Here, we will need to supply a Boolean value when we create **`ClassB`**.

Making these types of changes to an application as its code base increases will make it hard to maintain. A solution to this problem is to separate how we use dependencies and how we create them and delegate the creation to a different object. Continuing from the preceding example, we can rewrite **`ClassA`** as the following:

```
class ClassA(private val b: ClassB) {
    fun executeA() {
        b.executeB()
    }
}
```

Here, we removed the instantiation of **`ClassB`** and moved the variable in the constructor of **`ClassA`**. Now, we can create a class that will be responsible for creating the instances of both classes that looks like the following:

```
class Injector() {
    fun createA(b: ClassB) = ClassA(b)
    fun createB() = ClassB(true)
}
```

Here, we have a new class that will create an instance of **`ClassA`** with **`ClassB`** as a parameter and a separate method for creating an instance of **`ClassB`**. Ideally, when the program is initialized, we would need to initialize all the dependencies and pass them appropriately:

```
fun main(args : Array<String>) {
    val injector = Injector()
    val b = injector.createB()
    val a = injector.createA(b)
}
```

Here, we created **`Injector`**, which is responsible for creating our instances, and then invoked the appropriate methods on **`Injector`** to retrieve the appropriate instances of each class. What we have done here is called DI. Instead of **`ClassA`** creating the instance of **`ClassB`**, it will have an instance of **`ClassB`** injected through the constructor, also known as *constructor injection*.

In **`ClassB`**, we have an **`if-else`** statement in the **`executeB`** method. We can introduce an abstraction there, so we split the **`if-else`** statement into two separate implementations:

```
class ClassA(private val b: ClassB) {
    fun executeA() {
        b.executeB()
    }
}
interface ClassB {
    fun executeB()
}
class ClassB1() : ClassB {
    override fun executeB() {
        // Do something
    }
}
class ClassB2() : ClassB {
```

```
        override fun executeB() {
            // Do something else
        }
    }
```

Here, **`ClassA`** remains the same and **`ClassB`** has become an interface with two implementations, called **`ClassB1`** and **`ClassB2`**, representing the implementations of the **`if-else`** branch. Here, we can use the **`Injector`** class as well to inject one of the two implementations without requiring any change on **`ClassA`**:

```
    class Injector() {
        fun createA(b: ClassB) = ClassA(b)
        fun createB() = ClassB1()
    }
```

In the **`createB`** method, we return an instance of **`ClassB1`**, which will then be later injected into **`ClassA`**. This represents another benefit of DI, where we can make our code depend on abstractions rather than concretions and provide different concretions for different purposes. Based on this, we can define the following roles when it comes to DI:

- **Service**: Represents the object that contains useful functionality (**`ClassB1`** and **`ClassB2`** in our example)
- **Interface**: Represents the service abstraction (**`ClassB`** in our example)
- **Client**: Represents the object that depends on the service (**`ClassA`** in our example)
- **Injector**: Represents the object responsible for constructing the services and injecting them into the client (**`Injector`** in our example)

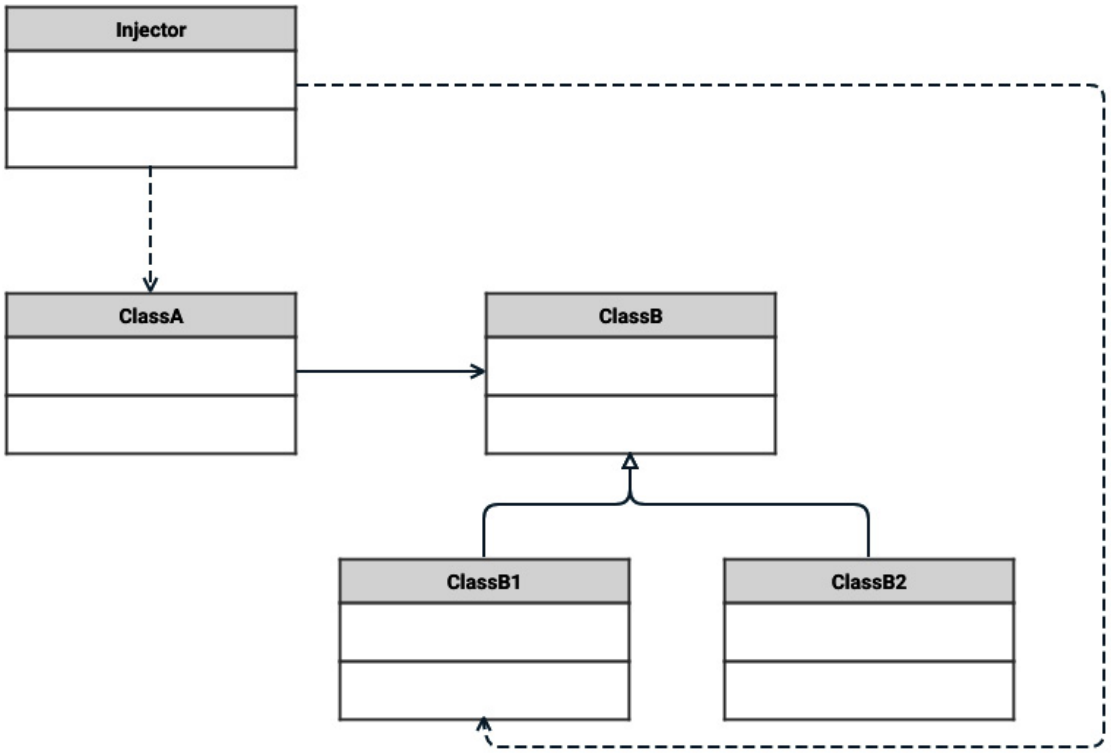

Figure 4.1 – DI class diagram

The preceding figure shows the class diagram of our example and the DI pattern. We can observe how the `Injector` class is responsible for creating and injecting the dependencies, `ClassA` is the client receiving a dependency to `ClassB`, which is the interface, and `ClassB1` and `ClassB2` represent the services.

There are multiple classifications of the types of DI, and they mainly revolve around two ways of injecting dependencies:

- **Constructor injection**: Where dependencies are passed through the constructor.
- **Field injection**: Where dependencies are passed through setter methods or by changing the instance variables. This can also be referred to as **setter injection** and it can also be expanded to **interface injection** in which the setter method is abstracted to an interface.

Another benefit of DI is the fact that it makes the code more testable. When dependencies are injected into an object, it makes the class easier

to test, because in the test code, we can inject objects that allow us to mimic various behaviors, called **mocks**.

In this section, we have introduced the DI pattern, how it works, and the problems it is solving. Developers can manage an application's dependencies and injection manually, by setting up injectors. But as an application grows, it becomes hard to maintain, especially when we want certain objects to live only as long as other objects and not as long as the application, or handle different instances of the same class. There are various DI frameworks and libraries that can manage all these cases and in Android, one of the most commonly used ones is Dagger 2.

Using Dagger 2 to manage dependencies

In this section, we will analyze the Dagger 2 library, how it handles DI, how it works, how it is integrated into an Android application, and what issues it might create.

The Dagger 2 library relies on code generation based on annotation processing, which will generate the boilerplate code that is required to perform DI. The library is written in Java, and it is used for various projects outside of Android applications. Because it is written in Java, it provides compatibility for apps written in Java, Kotlin, or both. The library is built using **Java Specification Requests** (**JSR**) **330**, which provide a set of useful annotations for DI (**`@Inject`**, **`@Named`**, **`@Qualifier`**, **`@Scope`**, and **`@Singleton`**).

When integrating Dagger 2, there are three main concepts that we will need to consider:

- **Provider**: This is represented by the classes responsible for providing the dependencies, using the **`@Module`** annotation for the classes and **`@Provides`** for the methods. To avoid many **`@Module`** definitions, we can

use the **`@Inject`** annotation on a constructor, which will provide the object as a dependency.

- **Consumer**: This is represented by the classes where the dependencies are required using the **`@Inject`** annotation.
- **Connector**: This is represented by the classes that connect the providers with the consumers and is annotated with the **`@Component`** annotation.

In order to add Dagger 2 to an Android application, you will first need to add the Kotlin annotation processor plugin to the **`build.gradle`** file of the module in which Dagger 2 is used:

```
plugins {
    …
    id 'kotlin-kapt'
    …
}
```

Here, we added the **`kotlin-kapt`** plugin to allow Dagger 2 to generate the code necessary for DI. Next, we will need the Dagger 2 dependencies:

```
dependencies {
    …
    implementation 'com.google.dagger:dagger:2.40.5'
    kapt 'com.google.dagger:dagger-compiler:2.40.5'
    …
}
```

Here, we are adding a dependency to the Dagger 2 library and a dependency to the annotation processing library, which has the role of code generation. The library version should ideally be the latest stable one available in the library repository.

Let's now re-introduce the example from the previous section:

```
class ClassA(private val b: ClassB) {
    fun executeA() {
```

```
            b.executeB()
        }
    }
    interface ClassB {
        fun executeB()
    }
    class ClassB1() : ClassB {
        override fun executeB() {
            // Do something
        }
    }
    class ClassB2() : ClassB {
        override fun executeB() {
            // Do something else
        }
    }
```

Here, we have the same classes with the same dependencies. Instead of defining an **`Injector`** class, we can use Dagger 2 to define an **`@Module`**:

```
    @Module
    class ApplicationModule {
        @Provides
        fun provideClassA(b: ClassB): ClassA = ClassA(b)
        @Provides
        fun provideClassB(): ClassB = ClassB1()
    }
```

Here, we annotated the class with **`@Module`** and for each instance, we used the **`@Provides`** annotation. We can further simplify this with the **`@Inject`** annotation and delete the **`@Provides`** methods from **`ApplicationModule`**:

```
    class ClassA @Inject constructor(private val b:
    ClassB) {
        …
    }
    class ClassB1 @Inject constructor() : ClassB {
```

```
        …
    }
    class ClassB2 @Inject constructor() : ClassB {
        …
    }
```

In the preceding code, we have added **`@Inject`** for each constructor. In the case of **`ClassA`**, it will have both the role of injecting **`ClassB`** and providing **`ClassA`** to other objects as a dependency. There is, however, an issue because **`ClassA`** has a dependency on the abstraction rather than the concretion, so Dagger will not know which instance to provide to **`ClassA`**. We can now add an **`@Binds`** annotated method to **`ApplicationModule`**, which will connect the abstraction with the implementation:

```
    @Module
    abstract class ApplicationModule {
        @Binds
        abstract fun bindClassB(b: ClassB1): ClassB
    }
```

Here, we added the **`bindClassB`** abstract method, which is annotated with **`@Binds`**. This method will tell Dagger 2 to connect the **`ClassB1`** implementation with the **`ClassB`** abstraction. To avoid large **`@Provides`** annotations, we should try to use the annotation for dependencies where we cannot modify the code and instead rely on **`@Inject`** on the constructors and using **`@Binds`** where possible.

Now, we will need to create the connector:

```
    @Singleton
    @Component(modules = [ApplicationModule::class])
    interface ApplicationComponent
```

Here, we are defining an **`@Component`** in which we specify the module the application will use. The **`@Singleton`** annotation tells Dagger that all the dependencies in this component will live as long as the application. At this point, we should trigger a build on the application. This will trigger the compilation, which will generate a **`DaggerApplicationComponent`** class.

This is an implementation of **`ApplicationComponent`** that Dagger 2 will handle. This class will be used to create the entire dependency graph. In Android, we need an entry point for this, which is represented by the **`Application`** class:

```
class MyApplication : Application() {
    lateinit var component: ApplicationComponent
    override fun onCreate() {
        super.onCreate()
        component =
DaggerApplicationComponent.create()
    }
}
```

Here, in the **`MyApplication`** class, we are using **`DaggerApplicationComponent`** and creating the dependency graph. This will go over all the modules in the graph and invoke all the **`@Provides`** methods. The **`@Component`** annotation has another role, which is to define member injection when constructor injection is not possible. In Android, this situation occurs when dealing with life cycle components such as activities and fragments, because we are not allowed to modify the default constructors of these classes. To do this, we can do the following:

```
@Singleton
@Component(modules = [ApplicationModule::class])
interface ApplicationComponent {
    fun inject(mainActivity: MainActivity)
}
```

In **`ApplicationComponent`**, we add a method called **`inject`** and the **`Activity`** where we want the injection to be performed. In the **`MainActivity`** class, we will need to do the following:

```
class MainActivity : AppCompatActivity() {
    @Inject
    lateinit var a: ClassA
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        (application as
            MyApplication).component.inject(this)
        a.executeA()
    }
}
```

Here, we will need to access the **`ApplicationComponent`** instance created in **`MyApplication`** and then invoke the **`inject`** method from **`ApplicationComponent`**. This will then initialize variable **`a`** with the instance Dagger 2 created. This approach has a problem, however, because all the dependencies will live as long as the application. This means that Dagger 2 will need to keep dependencies in memory when they are not required. Dagger 2 offers a solution for this in the form of scopes and subcomponents. We can create a new Scope, which will tell Dagger 2 to only keep certain dependencies as long as an Activity is alive, and then apply this Scope to a Subcomponent, which will handle a smaller graph of dependencies.

```
@Scope
@MustBeDocumented
@kotlin.annotation.Retention(AnnotationRetention.RUNT
IME)
annotation class ActivityScope
```

Here, we created a new **`@Scope`** annotation, which will indicate that dependencies will live as long as activities. We will next use **`@ActivityScope`** to create an **`@Subcomponent`** annotated class:

```
@ActivityScope
@Subcomponent(modules = [ApplicationModule::class])
interface MainSubcomponent {
    fun inject(mainActivity: MainActivity)
}
```

Here, we have defined a subcomponent that will use **`ApplicationModule`** and has an **`inject`** method for field injection into **`MainActivity`**. After that, we will need to tell Dagger 2 to create **`MainSubcomponent`**, by modifying **`ApplicationComponent`**:

```
@Singleton
@Component
interface ApplicationComponent {
    fun createMainSubcomponent(): MainSubcomponent
}
```

Here, we have removed **`ApplicationModule`** from **`@Component`** and replaced the **`inject`** method with a **`createMainSubcomponent`** method, which will allow Dagger to create **`MainSubcomponent`**. Finally, we will need to access **`MainSubcomponent`** in **`MainActivity`** and inject the **`ClassA`** dependency:

```
class MainActivity : AppCompatActivity() {
    @Inject
    lateinit var a: ClassA
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        (application as MyApplication).component.
            createMainSubcomponent().inject(this)
        a.executeA()
    }
}
```

Here, we access the **`ApplicationComponent`** instance from **`MyApplication`**, then create **`MainSubcomponent`** and then inject the **`ClassA`** dependency into the **`a`** variable. The code generated by Dagger 2 can be seen in the **`{module}/build/generated/source/kapt/{build type}`** folder and will look something similar to the following figure:

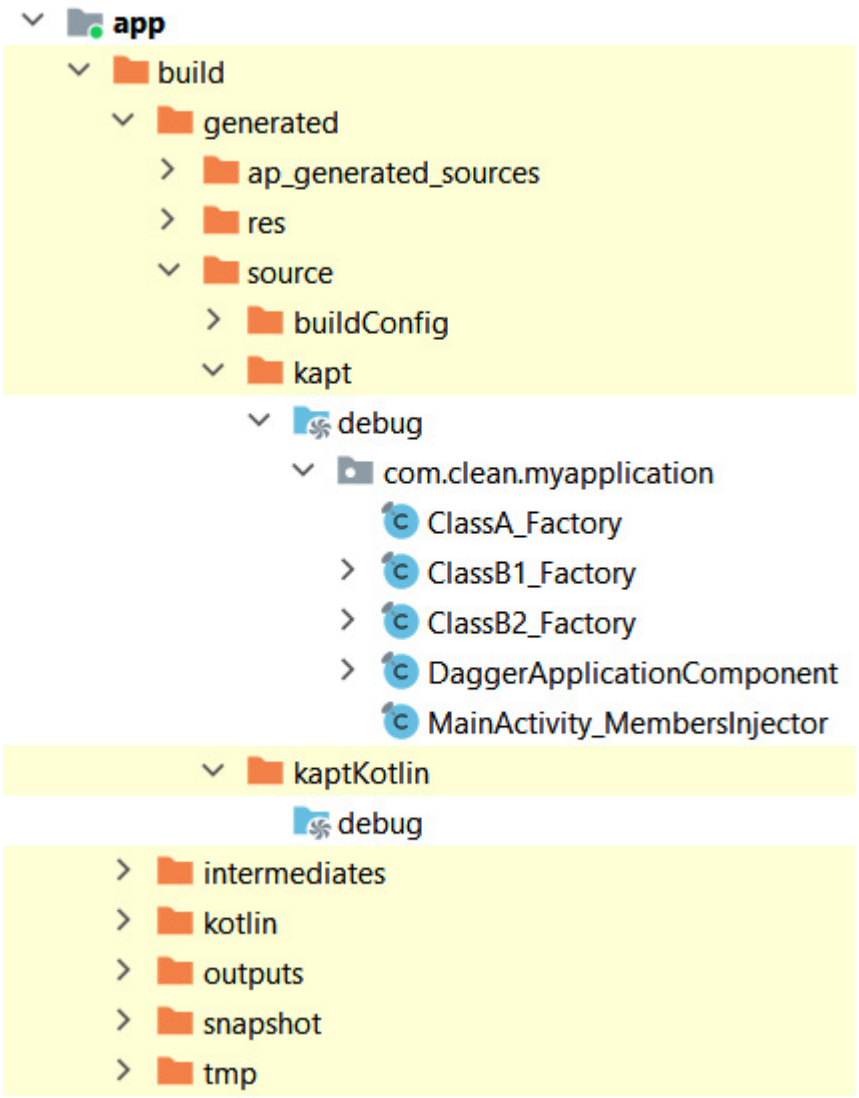

Figure 4.2 – Generated Dagger Classes

In the preceding figure, we can see Dagger will generate the implementation for the **`ApplicationComponent`** interface as well as the **`MainSubcomponent`** implementation. For dependencies that will need to be injected, it will generate a **`Factory`** class to create the dependency. Where we are injecting through the members, it will create an **`Injector`** class, which will be responsible for setting the value on the member variable, like the **`MainActivity`** class.

In this section, we have discussed the Dagger 2 library and how it can be used to provide and inject dependencies. Because it is a library used in other frameworks other than Android, it requires specific workarounds for injecting in activities and fragments, using member injectors and Subcomponents. An attempt at fixing this was through the introduction of

the Dagger Android library, which handled the creation of **`@Subcomponent`** annotated classes and introduced new annotations to indicate how Subcomponents should be created. More recently, the introduction of the Hilt library was more effective at solving these problems by further simplifying the amount of code developers needed to write and providing better compatibility with components such as ViewModel. In the section that follows, we will look at the Hilt library and how it solves these problems.

Using Hilt to manage dependencies

In this section, we will discuss the Hilt DI library, how we can use it in an Android application, and the extra features it provides on top of the Dagger 2 library.

Hilt is a library built on top of Dagger 2 with a specific focus on Android applications. This is to remove the extra boilerplate code that was required to use Dagger 2 in an application. Hilt removes the need to use **`@Component`** and **`@Subcomponent`** annotated classes and in turn offers new annotations:

- When injecting dependencies in Android classes, we can use **`@HiltAndroidApp`** for **`Application`** classes, **`@AndroidEntryPoint`** for activities, fragments, services, broadcast receivers, and views, and **`@HiltViewModel`** for **`ViewModels`**.
- When using the **`@Module`** annotation, we now have the option to use **`@InstallIn`** and specify an **`@DefineComponent`** annotated class, which represents the component the module will be added to. Hilt provides a set of useful components to install modules in:
  - **`@SingletonComponent`**: This will make the dependencies live as long as the application.
  - **`@ViewModelComponent`**: This will make the dependencies live as long as a **`ViewModel`**.

- **`@ActivityComponent`**: This will make the dependencies live as long as an **`Activity`**.
- **`@FragmentComponent`**: This will make the dependencies live as long as a **`Fragment`**.
- **`@ServiceComponent`**: This will make the dependencies live as long as a **`Service`**.

In order to use Hilt in a project, it will require a Gradle plugin, which will need to be added as a dependency to the root **`build.gradle`** file in the project:

```
buildscript {
    repositories {
        …
    }
    dependencies {
        …
        classpath 'com.google.dagger:hilt-android-
gradle-
            plugin:2.40.5'
    }
}
```

We will then need to add the annotation processor plugin and the Hilt plugin to the **`build.gradle`** file of the Gradle module that we want to use the Hilt library in:

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

The combination of these two plugins is what allows Hilt to generate the necessary source code for injecting the dependencies. Finally, we will need to add the dependency to the Hilt library:

```
dependencies {
    …
    implementation 'com.google.dagger:hilt-
android:2.40.5'
    kapt 'com.google.dagger:hilt-compiler:2.40.5'
    …
}
```

Here, we need the dependency on the library itself and a dependency on the annotation processor like how it was necessary for Dagger 2.

Let's now re-introduce the example from the previous section:

```
class ClassA @Inject constructor(private val b:
ClassB) {
    fun executeA() {
        b.executeB()
    }
}
interface ClassB {
    fun executeB()
}
class ClassB1 @Inject constructor() : ClassB {
    override fun executeB() {
        // Do something
    }
}
class ClassB2 @Inject constructor() : ClassB {
    override fun executeB() {
        // Do something else
    }
}
```

Here, we can keep the same structure of our classes and use the **`@Inject`** annotation like previously. The **`@Module`** annotated class that will provide these dependencies will look similar to a Dagger 2 Module:

```
@Module
@InstallIn(SingletonComponent::class)
abstract class ApplicationModule {
    @Binds
    abstract fun bindClassB(b: ClassB1): ClassB
}
```

In the **`ApplicationModule`** class, we keep the same implementation as before but now we have added the **`@InstallIn`** annotation, which will make the dependencies provided by this module live as long as the application will. Next, we will need to trigger the generation of components:

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

Here, we no longer need to use **`DaggerApplicationComponent`** to manually trigger the creation of the dependency graph and instead use **`@HiltAndroidApp`**, which will do this for us, as well as providing the ability to inject dependencies into the **`MyApplication`** class. Finally, we will need to inject the dependencies into an **`Activity`**:

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
    @Inject
    lateinit var a: ClassA
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        a.executeA()
    }
}
```

Here, we use the **`@AndroidEntry`** point to inform Hilt that we want to inject a dependency into an Activity and then use the **`@Inject`** annotation like how it worked in Dagger 2. The code generated by Hilt will look similar to the following figure and can be found in **`{module}/build/generated/source/kapt/{build type}`**:

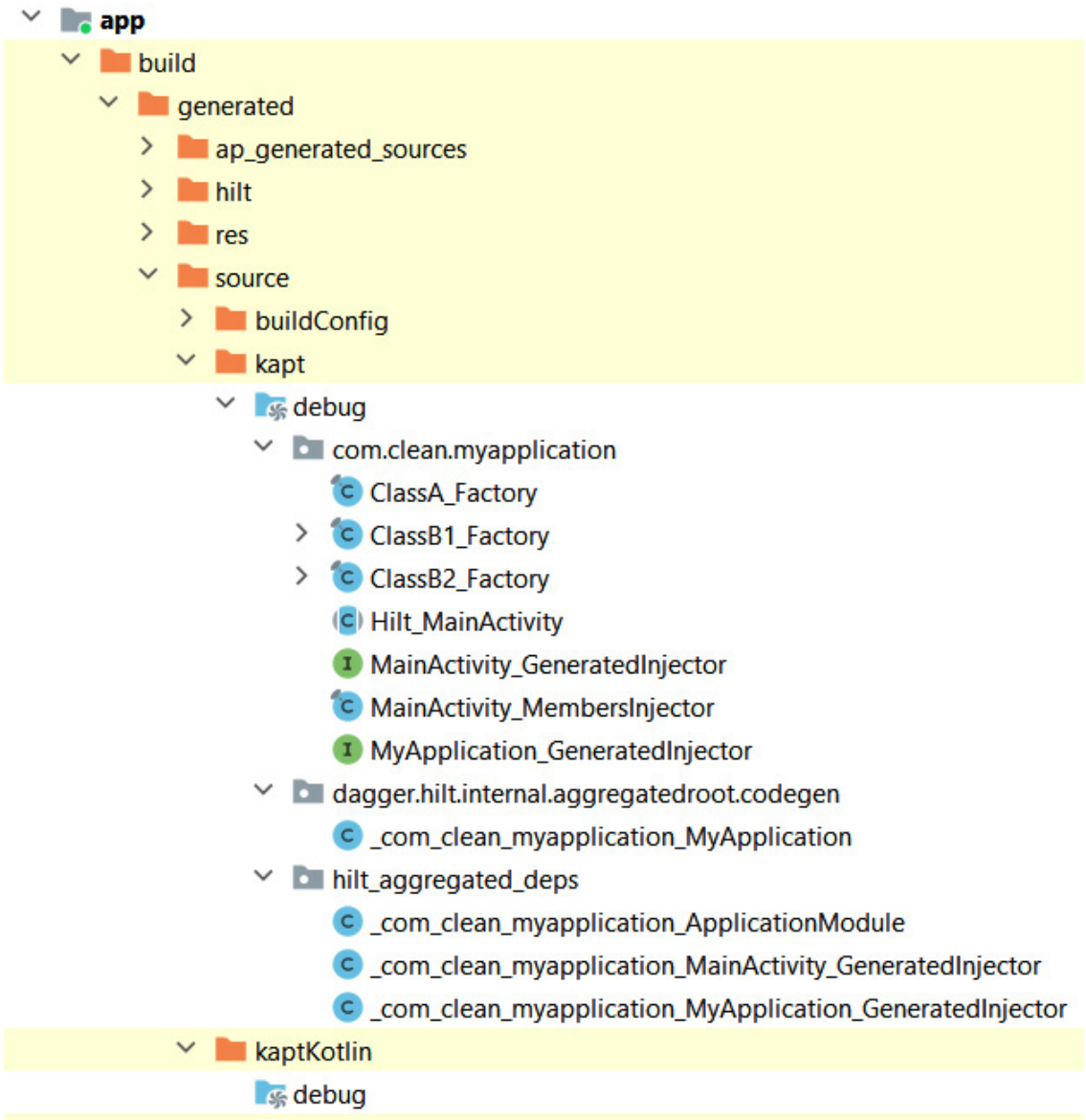

Figure 4.3 – Generated Hilt Classes

In the preceding figure, we can see **`Factory`** classes like the ones that Dagger 2 generates, but extra classes that Hilt will generate to handle the boilerplate work that was required to work with Dagger 2, such as handling the injection into activities and fragments or creating the dependency graph in the **`Application`** class.

In this section, we discussed the Hilt library, how we can use it to manage dependencies in an Android application, and how it removes boilerplate code that Dagger 2 required. In the following section, we will look at an exercise on integrating Hilt into an application together with other libraries.

Exercise 04.01 – Using Hilt to manage dependencies

Modify *Exercise 03.02 – navigating using Jetpack Compose* from *Chapter 3*, *Understanding Data Presentation on Android*, so that it will use Hilt to manage the dependencies across the application.

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

1. Add the Hilt library to the project.
2. Create a **`NetworkModule`** class that will provide the Retrofit dependencies.
3. Create a **`PersistenceModule`** class that will provide the Room and Data Store dependencies.
4. Clean up the **`MyApplication`** class, delete the **`MainViewModelFactory`** class, and instead use the **`@HiltViewModel`** annotation.
5. Modify **`MainActivity`** to instead obtain an instance of the **`MainView`** model from the Hilt Compose Navigation library.

Follow these steps to complete the exercise:

1. Add the Hilt Gradle plugin to the root project **`build.gradle`** file:
```
buildscript {
    repositories {
        …
    }
    dependencies {
        …
        classpath 'com.google.dagger:hilt-android-
            gradle-plugin:2.40.5'
    }
}
```
2. Apply the Gradle plugin to the **`build.gradle`** file in the app module:
```
plugins {
    …
```

```
        id 'dagger.hilt.android.plugin'
    }
```

3. Add the Hilt library dependency to the app module's **`build.gradle`** file:

```
    dependencies {
        …
        implementation 'com.google.dagger:hilt-android
            :2.40.5'
        kapt 'com.google.dagger:hilt-compiler:2.40.5'
        implementation 'androidx.hilt:hilt-navigation-
            compose:1.0.0-rc01'
        …
    }
```

Here, we added a dependency that allows Hilt to work with the Jetpack Compose Navigation library.

4. Create a **`NetworkModule`** class in which the networking dependencies are provided:

```
    @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
)
}
```

Here, we have moved all the dependencies for networking and split them into separate methods for **`OkHttplClient`**, **`Moshi`**, **`Retrofit`**, and finally, the **`UserService`** class.

5. Next, create a **`PersistenceModule`** class, which will return all the persistence-related dependencies:

```
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 provideAppDataStore(@ApplicationContext
        context: Context) = AppDataStore
            (context.dataStore)
}
```

Here, we have moved all the Room-related classes and the Data Store classes. For **`DataStore`**, we are required to declare the **`Context.dataStore`** file at the top level of the file, so we will need to keep it here. The usage of **`@ApplicationContext`** is meant to denote that the **`Context`** object is represented by context of the application and not other Context objects such as an **`Activity`** object or **`Service`** object. The annotation is a **`Qualifier`**, which is meant to distinguish between different instances of the same class (in this case, it's to distinguish between the application context and activity context).

6. Add the **`@Inject`** annotation to the constructor of the **`MainTextFormatter`** class:

```
class MainTextFormatter @Inject
constructor(@ApplicationContext private val
applicationContext: Context) {
    fun getCounterText(count: Int) =
        applicationContext.getString(R.string.total
_
            request_count, count)
}
```

This will let Hilt provide a new instance of **`MainTextFormatter`** every time it will be used as a dependency. Here, again, we will need to use the **`@ApplicationContext`** annotation to use the application **`Context`** object.

7. Delete all the dependencies in the **`MyApplication`** class and add the **`@HiltAndroidApp`** annotation:

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

8. Delete the **`MainViewModelFactory`** class.
9. Add the **`@HiltViewModel`** annotation to the **`MainViewModel`** class and **`@Inject`** to the constructor:

```
@HiltViewModel
class MainViewModel @Inject constructor(
    private val userService: UserService,
    private val userDao: UserDao,
    private val appDataStore: AppDataStore,
    private val mainTextFormatter:
MainTextFormatter
) : ViewModel() {
   …
}
```

10. Delete the reference to **`MainViewModelFactory`** in the **`Users`** **`@Composable`** method in **`MainActivity`**:

```
@Composable
fun Users(
    navController: NavController,
    viewModel: MainViewModel
) {
    …
}
```

11. Change the **`@Composable`** **`App`** method in **`MainActivity`** so that it provides a **`MainViewModel`** instance when it invokes the **`Users`** method:

```
@Composable
fun App(navController: NavHostController) {
    NavHost(navController, startDestination =
AppNavigation.Users.route) {
        composable(route =
AppNavigation.Users.route) {
            Users(navController, hiltViewModel())
        }
        composable(
            route = AppNavigation.User.route,
            arguments = listOf(navArgument
```

```
                    (AppNavigation.User.argumentName) {
                    type = NavType.StringType
                })
            ) {
                User(it.arguments?.getString(AppNavigat
ion.User.
    argumentName).orEmpty())
            }
    }
}
```

Here, we are using the **`hiltViewModel`** method, which is from the Hilt compatibility library with the Navigation library.

12. Add the **`@AndroidEntryPoint`** annotation to **`MainActivity`**:

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    …
}
```

13. If you encounter a **`Records requires ASM8`** error when building the application, then add the following to the root project's **`gradle.properties`** file:

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

This error is caused by an incompatibility that currently exists in the Android build tools and should be resolved when later updates are available.

If we run the application covered in this exercise, the functionality and user interface should remain the same as before. The role of Hilt here was to simplify how we manage dependencies, shown by how we have simplified the **`MyApplication`** class, leaving it with a simple annotation, and the fact that we have removed **`MainViewModelFactory`**, which itself had to depend on the **`MyApplication`** class. We can also see how easy it is to integrate Hilt with the rest of the libraries we used in the exercise.

Summary

In this chapter, we looked at the DI pattern and some of the more popular libraries that are available to apply this pattern to an Android application. We looked initially at Dagger 2 and how it can be integrated into an application, and then we analyzed the Hilt library, which is built on top of Dagger 2 and solves further problems that are specific to Android development.

There are other libraries that can be used to manage dependencies, such as Koin, which uses the Service Locator pattern (in which a registry is created and dependencies can be obtained) and is developed for Kotlin development. The exercise in this chapter showed how Hilt can be integrated with other libraries into an Android application. The problem is that the application still has no shape; there isn't anything we can point to that indicates what the use cases are. In the chapters that follow, we will look further into how we can structure our code to give it a shape using the Clean Architecture principles, starting with defining entities and use cases.