

Chapter 9: Implementing Dependency Injection with Jetpack Hilt

In this chapter, we're continuing our journey of improving the architectural design of the Restaurants app. More precisely, we will be incorporating **dependency injection (DI)** in our project.

In the first section, *What is DI?*, we will start by defining DI and understanding its basic concepts, from what a dependency is, the types of dependencies, and what injection represents, through to concepts such as dependency containers and manual injection.

Afterward, in the *Why is DI needed?* section, we will focus in more detail on the benefits that DI brings to our projects.

In the last section, *Implementing DI with Hilt*, we will first understand how the Jetpack Hilt DI library works, and how to use it, and finally, with its help, we will incorporate DI in our Restaurants app.

To summarize, in this chapter we will be covering the following sections:

- What is DI?
- Why is DI needed?
- Implementing DI with Hilt

Before jumping in, let's set up the technical requirements for this chapter.

Technical requirements

Building Compose-based Android projects for this chapter would usually require your day-to-day tools; however, to follow along smoothly, make sure you have the following:

- The Arctic Fox 2020.3.1 version of Android Studio. You can also use a newer Android Studio version or even Canary builds but note that IDE interface and other generated code files might differ from the ones used throughout this book.
- Kotlin 1.6.10 or newer plugin installed in Android Studio
- The Restaurants app code from the previous chapter

The starting point for this chapter is represented by the Restaurants app developed in the previous chapter. If you haven't followed the implementation from the previous chapter, access the starting point for this chapter by navigating to the **Chapter_08** directory of the repository and importing the Android project entitled **chapter_8_restaurants_app**.

To access the solution code for this chapter, navigate to the **Chapter_09** directory: https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_09/chapter_9_restaurants_app.

What is DI?

In simple terms, **DI** represents the concept of providing the instances of the dependencies that a class needs, instead of having it construct them itself. But, what are dependencies?

Dependencies are other classes that a certain class *depends on*. For example, an **ExampleViewModel** class could contain a **repository** variable of type **Repository**:

```
class ExampleViewModel {  
    private val repository: Repository = Repository()  
    fun doSomething() {
```

```
        repository.use()  
    }  
}
```

That's why **ExampleViewModel** depends on **Repository**, or **Repository** is a dependency for **ExampleViewModel**. Most of the time, classes have many more dependencies, but we'll stick with only one for the sake of simplicity. In this case, the **ExampleViewModel** provides its own dependencies so it's very easy to create an instance of it:

```
fun main() {  
    val vm = ExampleViewModel()  
    vm.doSomething()  
}
```

Now, the previous example doesn't incorporate DI, mainly because **ExampleViewModel** provides instances for its own dependencies. It does that by instantiating a **Repository** instance (through the **Repository()** constructor) and by passing it to the **repository** variable.

To incorporate DI, we must create a component that provides **ExampleViewModel** with its dependencies:

```
object DependencyContainer {  
    val repo: Repository = Repository()  
}
```

The **DependencyContainer** class will act, as the name suggests, as a **dependency container**, as it will provide instances for all the dependencies our classes need. When a class needs an instance for its dependency, this container will provide it. This way, we centralize the creation of the instances of dependencies so we can handle this process (which can become elaborate for complex projects where each dependency has other dependencies, for example) within a single place in our project.

NOTE

*Apart from the DI technique, you can also use the **service locator** pattern to construct classes. Unlike DI, if you try to follow the service locator pattern, then the class that needs to be constructed will be responsible for creating its own dependencies with the help of a **ServiceLocator** component. Both DI and the service locator pattern are useful; however, we will only cover DI in this chapter.*

Getting back to incorporating DI, we then must allow **DependencyContainer** to provide a **Repository** instance to **ExampleViewModel**:

```
class ExampleViewModel {  
    private val repository: Repository =  
        DependencyContainer.repo  
    fun doSomething() {  
        repository.use()  
    }  
}
```

This technique of having dependencies declared as variables (for example, **ExampleViewModel** contains a **repository** variable) and then providing their instances through a container is a form of DI called **field injection**.

There are several issues with this approach, mainly caused by the fact that we have declared dependencies as field variables. The most notable ones are as follows:

- The **ExampleViewModel** class is tightly coupled to our **DependencyContainer** and we cannot use the **ViewModel** without it.
- The dependencies are **implicit**, which means they are hidden from the outside world. In other words, whoever is instantiating **ExampleViewModel** doesn't know about the **ViewModel** class's dependencies or their creation.

This won't allow us to reuse the same **ExampleViewModel** with other implementations of its dependencies (given its dependencies, such as **Repository**, are interfaces that can be implemented by different classes).

- Since **ExampleViewModel** has hidden dependencies, it becomes hard for us to test it. As we will instantiate the **ExampleViewModel** and put it under test, it will create its own **Repository** instance that will probably make real I/O requests for every test. We want our tests to be fast and reliable and not dependent on real third-party APIs.

To alleviate these issues, we must first refactor **ExampleViewModel** to expose its dependencies through its public API to the outside world. The most appropriate way to do that is through its public **constructor**:

```
class ExampleViewModel constructor(private val repo:
Repository) {
    fun doSomething() { repo.use() }
}
```

Now, **ExampleViewModel** exposes its dependencies to the outside world through its constructor, making those dependencies **explicit**. Yet, who's going to provide the dependencies from outside?

When we need to instantiate **ExampleViewModel**, **DependencyContainer** will provide it with the necessary dependencies from the outside:

```
fun main() {
    val repoDependency =
DependencyContainer.repository
    val vm = ExampleViewModel(repoDependency)
    vm.doSomething()
}
```

In the previous example, instead of field injection, we have used **constructor injection**. This is because we have provided and injected the dependencies to **ExampleViewModel** from the outside world through its constructor.

As opposed to field injection, constructor injection allows us to do the following:

- Decouple our classes from the DI container, just like `ExampleViewModel` no longer depends on `DependencyContainer`.
- The dependencies are exposed to the outside world, so we can reuse the same `ExampleViewModel` with other implementations of `Repository` (given `Repository` is an interface).
- The `ExampleViewModel` class can no longer decide which dependency implementation to get and use as was the case with field injection, so we have now inverted this responsibility from `ExampleViewModel` to the outside world.
- `ExampleViewModel` is easier to test, as we can easily pass a mock or a fake `Repository` implementation (given `Repository` is an interface) that will behave the way we're expecting it to in a test.

So far, with the help of a dependency container, we have incorporated DI by ourselves by allowing `DependencyContainer` to provide instances of dependencies to our classes (that is, an instance of `ExampleViewModel`). This technique is called **manual DI**.

Apart from manual DI, you can have DI done automatically through frameworks that relieve you from the burden of the following:

- Providing instances of dependencies to the classes that need them. More specifically, frameworks help you wire up complex object relationships for the required dependencies, so you don't have to write boilerplate code to generate instances and pass them to appropriate objects. This infrastructure code is often cumbersome for large-sized apps, so a framework that automates that for you can be quite handy.
- Scoping dependencies to certain lifetime scopes, such as the **Application** scope or **Activity** scope. For example, if you want a certain dependency to be a singleton (to be scoped to the lifetime of the application), you must manually make sure that only one instance is created in memory while also avoiding concurrency issues due to concurrent access. A framework can do that for you behind the scenes.

In Android, a very simple DI library is **Hilt**, and we will explore it in the *Implementing DI with Hilt* section. But until then, let's better understand why DI is needed in the first place.

Why is DI needed?

DI is not a must for all projects. Until now, our Restaurants app worked just fine without any DI incorporated. Yet, while not including DI might not seem like a big issue, by incorporating it you bring a lot of benefits to your project; the most notable advantages are that you can do the following:

- Write less boilerplate code.
- Write testable classes.

Let's cover these two next.

Write less boilerplate code

Let's circle back to our Restaurants app, and let's have a look at how we instantiate the Retrofit interface within the **RestaurantsRepository** class:

```
class RestaurantsRepository {  
    private var restInterface: RestaurantsApiService  
    =  
        Retrofit.Builder()  
            .addConverterFactory(  
                GsonConverterFactory.create()  
            ).baseUrl("your_firebase_database_url")  
            .build()  
            .create(RestaurantsApiService::class.java  
        )  
    [...]  
}
```

Now, let's have a look at how we similarly instantiate the Retrofit interface within the **RestaurantsDetailsViewModel** class:

```
class RestaurantDetailsViewModel(...): ViewModel() {  
    private var restInterface: RestaurantsApiService  
    [...]  
    init {  
        val retrofit: Retrofit = Retrofit.Builder()  
            .addConverterFactory(GsonConverterFactory  
            .create())  
            .baseUrl("your_firebase_database_url")  
            .build()  
        restInterface = retrofit  
            .create(RestaurantsApiService::class.java  
        )  
        [...]  
    }  
    [...]  
}
```

While the code seems different, in essence, it's the same code needed to instantiate a concrete instance of **RestaurantsApiService**. Unfortunately, we have duplicated this instantiation code in two places, both in the **RestaurantsRepository** class and in the **RestaurantsDetailsViewModel** class.

In medium to large-sized production apps, the relationship between objects is often much more complex, making such infrastructure code plague every class, mostly because, without any DI, every class builds the instances of the dependencies it needs. Such code is often duplicated throughout the project and ultimately becomes difficult to manage.

DI will help us centralize this infrastructure code and will eliminate all the duplicated code needed to provide instances of dependencies, wherever we need them throughout the project.

Going back to our Restaurants app, if we were to use manual DI, all this instantiation code could be extracted into a **DependencyContainer** class that would provide us with a **RestaurantsApiService** instance wherever we need it, so we would have no more duplicated code! Don't worry, we will incorporate DI soon, in the upcoming *Implementing DI with Hilt* section.

Now that we touched upon how DI helps us with containing and organizing the code related to building instances of classes, it's time to check out another essential advantage of DI.

Write testable classes

Let's suppose that we want to test the behavior of **RestaurantsRepository** to make sure that it performs as expected. But first, let's have a quick look at the existing implementation of **RestaurantsRepository**:

```
class RestaurantsRepository {  
    private var restInterface: RestaurantsApiService  
    =  
        Retrofit.Builder()  
            .[...]  
            .create(RestaurantsApiService::class.java  
        )  
    private var restaurantsDao = RestaurantsDb  
        .getDaoInstance(  
            RestaurantsApplication.getAppContext()  
        )  
    suspend fun toggleFavoriteRestaurant(...) = {...}  
    suspend fun getRestaurants(): List<Restaurant>  
    {...}  
    [...]  
}
```

We can see that no DI is currently incorporated, as **RestaurantsRepository** has two implicit dependencies: an instance of **RestaurantsApiService** and

an instance of **RestaurantsDao**. The **RestaurantsRepository** provides instances to its own dependencies, first by constructing a **Retrofit.Builder()** object and creating the concrete instance by calling **.create(...)**.

Now, let's say we want to test this **RestaurantsRepository** class, and make sure that it behaves correctly by running different verifications. Let's imagine how such a test class would look:

```
class RestaurantsRepositoryTest {  
    @Test  
    fun repository_worksCorrectly() {  
        val repo = RestaurantsRepository()  
        assertNotNull(repo)  
        // Perform other verifications  
    }  
}
```

The previous test structure is simple: we created a **RestaurantsRepository** instance by using its constructor and then we saved it inside a **repo** variable. We then asserted that the instance of the **Repository** is not **null**, so we can proceed with testing its behavior.

This is optional, yet if you're trying to write the previous test class and follow this process, make sure that the **RestaurantsRepositoryTest** class is placed inside the **test** directory of the application:

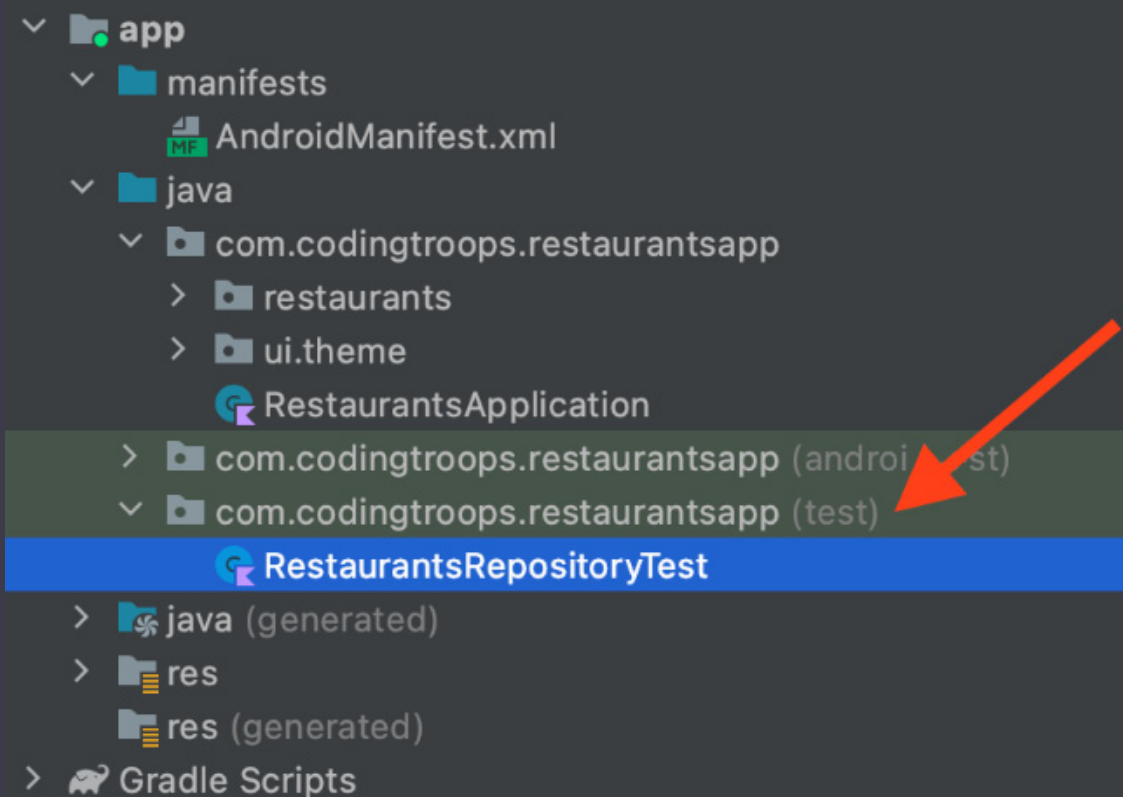


Figure 9.1 – Placement of test classes in the project structure

Now, if we would to run this test, it will throw an exception before having the chance to verify anything. The stack trace would look like this:

```
lateinit property app has not been initialized
kotlin.UninitializedPropertyAccessException: lateinit property app has not been initialized
    at com.codingtroops.restaurantsapp.RestaurantsApplication$Companion.getAppContext(RestaurantsApplication.kt:10)
    at com.codingtroops.restaurantsapp.restaurants.data.RestaurantsRepository.<init>(RestaurantsRepository.kt:25)
    at com.codingtroops.restaurantsapp.RestaurantsRepositoryTest.addition_isCorrect(RestaurantsRepositoryTest.kt:16)
```

Figure 9.2 – Stack trace of running invalid test

This happens because we're trying to write a small test for **RestaurantsRepository** but this class is not yet testable (in fact, we're trying to perform a Unit test – we will tackle this in more detail in [Chapter 10, Test Your App with UI and Unit Tests](#)).

But, why is our simple test throwing **UninitializedPropertyAccessException**?

If we have a look at the stack trace, we can see that the crash is caused because our test is trying to obtain the application context through **getAppContext()** from the **RestaurantsApplication** class.

It makes sense because, if we have another look at **RestaurantsRepository**, we can see that to obtain the **restaurantsDao** instance, the **Repository** calls **RestaurantsDb.getDaoInstance()** that initializes the Room database, and it needs an instance of **Context** to do that:

```
class RestaurantsRepository {  
    [...]  
    private var restaurantsDao = RestaurantsDb  
        .getDaoInstance(  
            RestaurantsApplication.getAppContext()  
        )  
    suspend fun toggleFavoriteRestaurant(...) = {...}  
    suspend fun getRestaurants(): List<Restaurant>  
    {...}  
    [...]  
}
```

Our small test shouldn't need a **Context** object, simply because it should neither try to create a Room database nor to create a Retrofit client instance; it shouldn't even depend on these concrete implementations. This is not efficient for small tests simply because such operations are memory-expensive and will do nothing but slow down our tests.

Moreover, we don't want our small test (that should run with much ease and very fast, several times in a short time frame) to make Room queries or, even worse, network requests through Retrofit, simply because the tests are dependent on the external world and so they become expensive and difficult to automate.

If, however, we would have had DI in place with constructor injection, we could have created our own classes that *fake* the behavior, ultimately making our **Repository** class easy to test and independent of concrete implementations that perform heavy I/O operations. We'll cover more about tests and *faking* in [Chapter 10, Test Your App with UI and Unit Tests](#).

Going back to our app, we're not yet ready to write tests, because as you could see, we're lacking DI in our project. Now that we've seen that, without DI, life is somehow tough, let's learn how we can incorporate DI in the Restaurants app with the help of the Hilt library!

Implementing DI with Hilt

DI libraries are often used to simplify and accelerate the incorporation of DI in our projects, especially when the infrastructure code required by manual DI gets difficult to manage in large projects.

Hilt is a DI library that is part of Jetpack, and it removes the unnecessary boilerplate involved in manual dependency injection in Android apps by generating the code and the infrastructure that you otherwise would have had to develop manually.

NOTE

*Hilt is a DI library based on another popular DI framework called **Dagger**, meaning that they are strongly related, so we will often refer to Hilt as **Dagger Hilt** in this chapter. Due to the steep learning curve of the Dagger APIs, Hilt was developed as an abstraction layer over Dagger to allow easier adoption of automated DI in Android projects.*

Dagger Hilt relies on annotation processors to automatically generate code at build time, making it able to create and optimize the process of managing and providing dependencies throughout your project. Because of that, its core concepts are strongly connected to the use of annotations, so before we start adding and implementing Hilt in our Restaurants app, we must first cover a few concepts to better understand how Dagger Hilt works.

To summarize, in this section we will be doing the following:

- Understanding the basics of Dagger Hilt

- Setting up Hilt
- Using Hilt for DI

Let's begin!

Understanding the basics of Dagger Hilt

Let's analyze the three most important concepts and their corresponding annotations that we're required to work with to enable automatic DI in our project:

- Injection
- Modules
- Components

Let's start with injection!

Injection

Dagger Hilt needs to know the type of instances we want it to provide us with. When we discussed manual constructor injection, we initially wanted `ExampleViewModel` to be injected wherever we needed it, and we used a `DependencyContainer` class for that.

If we want Dagger Hilt to inject instances of a class somewhere, we must first declare a variable of that type and annotate it with the `@Inject` annotation.

Let's say that inside the `main()` function used for the manual DI example, we no longer want to use manual DI to get an instance of `ExampleViewModel`. Instead, we want Dagger to instantiate this class. That's why we will annotate the `ExampleViewModel` variable with the Java `@Inject` annotation and refrain from instantiating the `ViewModel` class by ourselves. Dagger Hilt should do that for us now:

```
import javax.inject.Inject
```

@Inject

```
val vm: ExampleViewModel
fun main() {
    vm.doSomething()
}
```

Now, for Dagger Hilt to know how to provide us with an instance of the **ExampleViewModel** class, we must also add the **@Inject** annotation to the dependencies of **ExampleViewModel** so that Dagger knows how to instantiate the **ViewModel** class.

Since the dependencies of **ExampleViewModel** are inside the constructor (from when we used manual constructor injection), we can directly add the **@Inject** annotation to **constructor**:

```
class ExampleViewModel @Inject constructor(private
val repo:Repository) {
    fun doSomething() { repo.use() }
}
```

Now, Dagger Hilt also needs to know how to inject the dependencies of **ExampleViewModel**, more precisely the **Repository** class.

Let's consider that **Repository** has only one dependency, a **Retrofit** constructor variable. For Dagger to know how to inject a **Repository** class, we must annotate its constructor with **@Inject** as well:

```
class Repository @Inject constructor(val retrofit:
Retrofit){
    fun use() { retrofit.baseUrl() }
}
```

Until now, we got away with **@Inject** annotations because we had access to the classes and dependencies that we were trying to inject, but now, how can Dagger know how to provide us with a **Retrofit** instance? We have no way of tapping inside the **Retrofit** class and annotating its constructor with **@Inject**, since it's in an external library.

To instruct Dagger on how to provide us with specific dependencies, let's learn a bit about modules!

Modules

Modules are classes annotated with `@Module` that allow us to instruct Dagger Hilt on how to provide dependencies. For example, we need Dagger Hilt to provide us with a **Retrofit** instance in our **Repository**, so we could define a **DataModule** class that tells Dagger Hilt how to do so:

```
@Module
object DataModule {
    @Provides
    fun provideRetrofit(): Retrofit {
        return
        Retrofit.Builder().baseUrl("some_url").build()
    }
}
```

To tell the library how to provide us with a dependency, we must create a method inside the `@Module` annotated class where we manually build that class instance.

Since we don't have access to the **Retrofit** class and we need it injected, we've created a `provideRetrofit()` method (you can call it any way you want) annotated with the `@Provides` annotation, and that returns a **Retrofit** object. Inside the method, we manually created the **Retrofit** instance the way we needed it to be built.

Now, Dagger Hilt knows how to provide us with all the dependencies our **ExampleViewModel** needs (its direct **Repository** dependency and **Repository Retrofit** dependency). Yet, Dagger will complain that it needs a component class in which the module we've created must be installed.

Let's have a brief look at components next!

Components

Components are interfaces that represent the container for a certain set of dependencies. A component takes in modules and makes sure that the injection of its dependencies happens with respect to a certain lifecycle.

For our example with the **ExampleViewModel**, **Repository**, and **Retrofit** dependencies, let's say that we create a component that manages the creation for these dependencies.

With Dagger Hilt, you can define a component with the **@DefineComponent** annotation:

```
@DefineComponent()  
interface MyCustomComponent(...) { /* component build  
code */ }
```

Afterward, we could install our **DataModule** in this component:

```
@Module  
@InstallIn(MyCustomComponent::class)  
object DataModule {  
    @Provides  
    fun provideRetrofit(): Retrofit { [...] }  
}
```

In practice though, the process of defining and building a component is more complex than that. This is because a component must scope its dependencies to a certain lifetime scope (such as the lifetime of the application) and have a pre-existent parent component.

Luckily, Hilt provides components for us out of the box. Such predefined components allow us to install modules in them and to scope dependencies to their corresponding lifetime scope.

Some of the most important predefined components are as follows:

- **SingletonComponent**: Allows us to scope dependencies to the lifetime of the application, as singletons, by annotating them with the **@Singleton**

annotation. Every time a dependency annotated with `@Singleton` is requested, Dagger will provide the same instance.

- **ActivityComponent**: Allows us to scope dependencies to the lifetime of an **Activity**, with the `@ActivityScoped` annotation. If the **Activity** is recreated, a new instance of the dependency will be provided.
- **ActivityRetainedComponent**: Allows us to scope dependencies to the lifetime of an **Activity**, surpassing its recreation upon orientation change, with the `@ActivityRetainedScoped` annotation. If the **Activity** is recreated upon orientation change, the same instance of the dependency is provided.
- **ViewModelComponent**: Allows us to scope dependencies to the lifetime of a **ViewModel**, with the `@ViewModelScoped` annotation.

As the lifetime scope of these components varies, this also translates into the fact that each component derives its lifetime scope from each other, from the widest `@Singleton` lifetime scope (of the application) to narrower scopes such as `@ActivityScoped` (of an **Activity**):

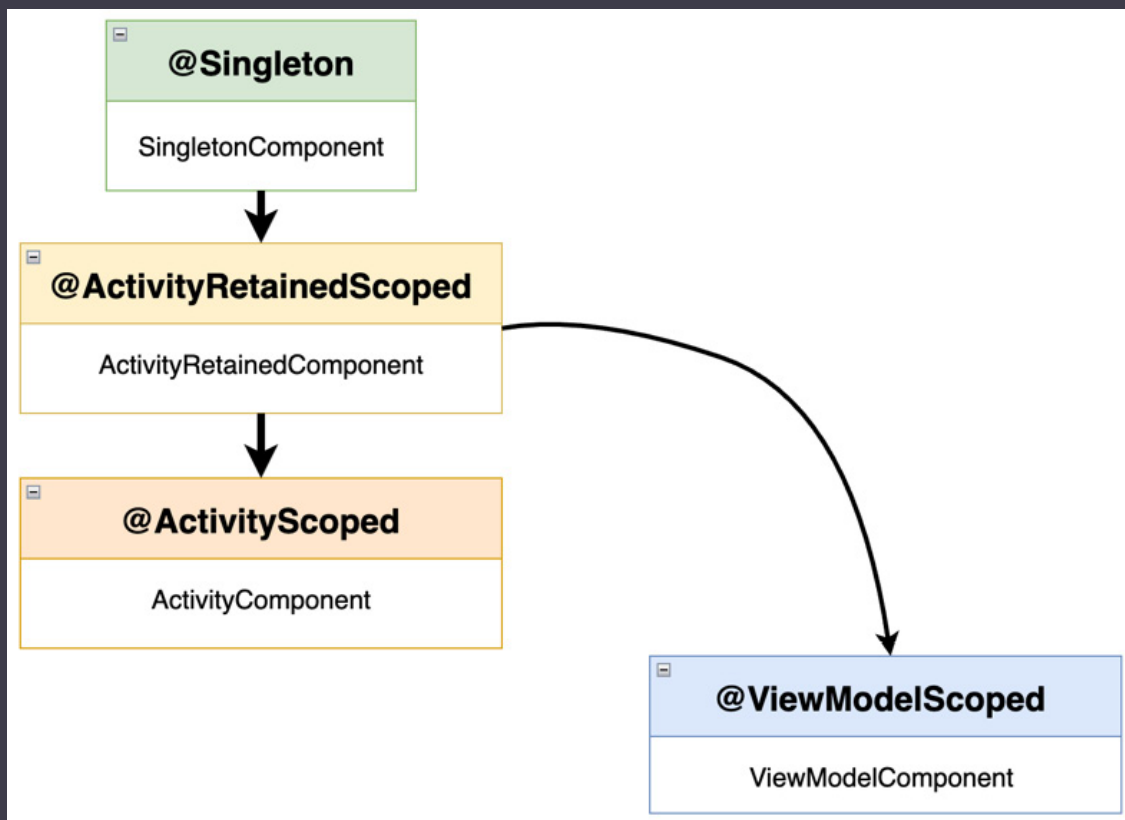


Figure 9.3 – Simplified version of Dagger Hilt scope annotations and their corresponding components

While in our Restaurants app, we will mostly be using `SingletonComponent` and its `@Singleton` scope annotation; it's important to note that Dagger Hilt exposes a broader variety of predefined components and scopes. Check them out in the documentation here:

<https://dagger.dev/hilt/components.html>.

Now that we've briefly covered components, it's time to add Hilt to our Restaurants app!

Setting up Hilt

Before injecting dependencies with Hilt, we must first set up Hilt. Let's begin!

1. In the project-level `build.gradle` file, inside the `dependencies` block, add the Hilt-Android Gradle dependency:

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

2. Moving inside the application-level `build.gradle` file, add the Dagger Hilt plugin inside the `plugins` block:

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

3. Still inside the application-level `build.gradle`, inside the `dependencies` block, add the Android-Hilt dependencies:

```
dependencies {  
    [...]
```

```
implementation "com.google.dagger:hilt-  
    android:2.40.5"  
kapt "com.google.dagger:hilt-compiler:2.40.5"  
}
```

The **kapt** keyword stands for **Kotlin Annotation Processor Tool** and is required by Dagger Hilt to generate code based on the annotations we will be using.

After updating the **build.gradle** files, make sure to sync your project with its Gradle files. You can do that by pressing on the **File** menu option and then by selecting **Sync Project with Gradle Files**.

4. Annotate the **RestaurantsApplication** class with the **@HiltAndroidApp** annotation:

```
@HiltAndroidApp  
class RestaurantsApplication: Application() { [...] }
```

To make use of automated DI with Hilt, we must annotate our **Application** class with the **HiltAndroidApp** annotation. This annotation allows Hilt to generate DI-related boilerplate code, starting with the application-level dependency container.

5. Build the project to trigger Hilt's code generation.
6. Optionally, if you want to check out the generated classes, first, expand the **Project** tab on the left, and then expand the package for the generated code. These classes are the proof that Hilt generates a lot of code behind the scenes so we can incorporate DI much easier:

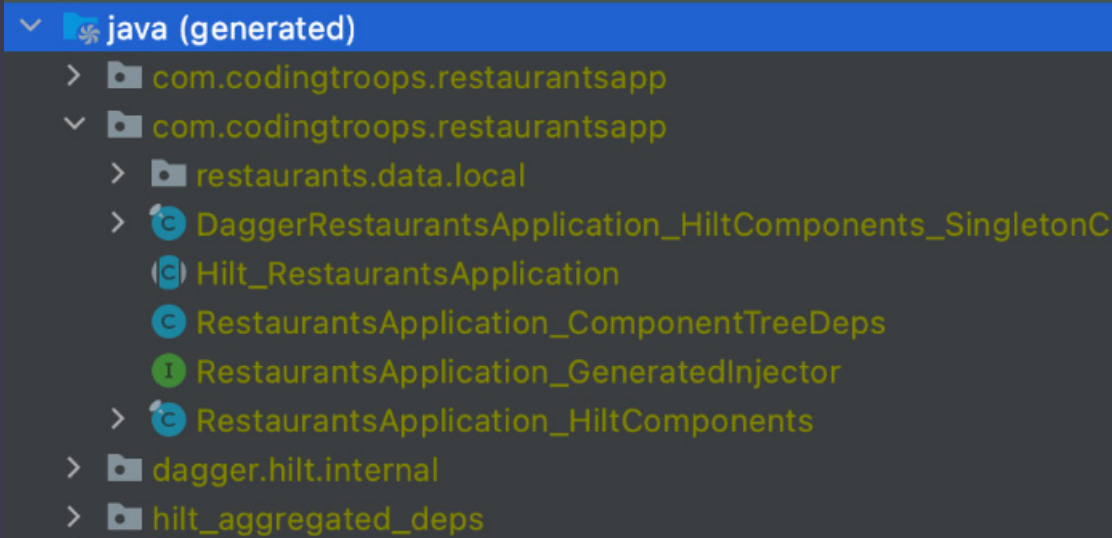


Figure 9.4 – Automatically generated classes by Hilt

Let's move on to the actual implementation!

Using Hilt for DI

In this sub-section, we will implement DI with Hilt for the first screen of our app where the list of restaurants is displayed. In other words, we want to inject all the dependencies that `RestaurantsScreen()` needs or depends on.

To have a starting point, let's have a look inside the `RestaurantsApp()` composable for the `RestaurantsScreen()` destination and see what we have to inject first:

```
@Composable
private fun RestaurantsApp() {
    val navController = rememberNavController()
    NavHost(navController, startDestination =
"restaurants") {
        composable(route = "restaurants") {
            val viewModel: RestaurantsViewModel =
viewModel()
            RestaurantsScreen(state =
viewModel.state.value, [...])
        }
    }
}
```

```
    }  
    composable(...) { RestaurantDetailsScreen() }  
}  
}
```

It's clear that **RestaurantsScreen()** depends on **RestaurantsViewModel** to obtain its state and consume it.

This means that we must first inject an instance of **RestaurantsViewModel** inside the **composable()** destination where the **RestaurantsScreen()** resides:

1. Since we cannot add the **@Inject** annotation inside a composable function, we must use a special composable function to inject a **ViewModel**. To do that, first, add the **hilt-navigation-compose** dependency inside the **dependencies** block of the app-level **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'  
}
```

After updating the **build.gradle** file, make sure to sync your project with its Gradle files. You can do that by pressing on the **File** menu option and then by selecting **Sync Project with Gradle Files**.

2. Then, going back inside the **RestaurantsApp()** composable, in the DSL **composable()** destination for our **RestaurantsScreen()** composable, replace the **viewModel()** constructor of **RestaurantsViewModel** with the **hiltViewModel()** composable:

```
@Composable  
private fun RestaurantsApp() {  
    val navController = rememberNavController()
```

```
NavHost(navController, startDestination =
    "restaurants") {
    composable(route = "restaurants") {
        val viewModel: RestaurantsViewModel =
            hiltViewModel()
        RestaurantsScreen(...)
    }
    composable(...) { RestaurantDetailsScreen() }
}
```

The `hiltViewModel()` function injects an instance of `RestaurantsViewModel` scoped to the lifetime of the `RestaurantsScreen()` navigation component destination.

3. Since now our composable hierarchy injects a `ViewModel` at some point with the help of Hilt, we must annotate the Android component that is the host of the `RestaurantsApp()` root composable with the `@AndroidEntryPoint` annotation. In our case, the `RestaurantsApp()` composable is hosted by the `MainActivity` class, so we must annotate it with the `@AndroidEntryPoint` annotation:

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            RestaurantsAppTheme { RestaurantsApp()
        }
    }
}
```

The `@AndroidEntryPoint` annotation generates another component for our `Activity` with a lifetime narrower than the lifetime of the application.

More precisely, this component allows us to scope dependencies to the lifetime of our **Activity**.

4. In the **RestaurantsViewModel** class, first refactor it to explicitly declare its dependencies by moving them inside its constructor so that testability is promoted through constructor injection:

```
class RestaurantsViewModel constructor(  
    private val getRestaurantsUseCase:  
        GetInitialRestaurantsUseCase,  
    private val toggleRestaurantsUseCase:  
        ToggleRestaurantUseCase  
) : ViewModel() {  
    private val _state = mutableStateOf(...) [ ... ]  
}
```

Notice that, while we extracted the two Use Case variables into the constructor, we're no longer instantiating them – we will leave that to Hilt.

5. To get Hilt to inject **RestaurantsViewModel** for us, mark the **ViewModel** with the **@HiltViewModel** annotation, while also annotating its constructor with the **@Inject** annotation so that Hilt understands which dependencies of the **ViewModel** must be provided:

```
@HiltViewModel  
class RestaurantsViewModel @Inject constructor(  
    private val getRestaurantsUseCase: [...] ,  
    private val toggleRestaurantsUseCase: [...]) :  
    ViewModel() {  
    [...] [ ... ]  
}
```

Now that our **ViewModel** is annotated with **@HiltViewModel**, instances of **RestaurantsViewModel** will be provided by **ViewModelComponent** that respects the lifecycle of a **ViewModel** (bound to the lifetime of the composable destination while also surviving configuration changes).

6. Now that we instructed Hilt how to provide **RestaurantsViewModel**, we might think we're done; yet, if we build the application, we will get this exception:

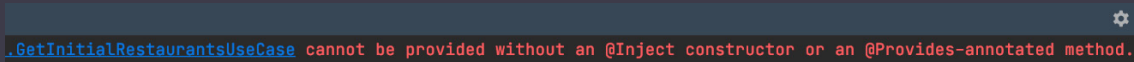


Figure 9.5 – Hilt compilation error

The issue lies in the fact that, while we instructed Hilt to inject **RestaurantsViewModel** and its dependencies, we never made sure that Hilt knows how to provide those dependencies: neither the **GetInitialRestaurantsUseCase** dependency nor the **ToggleRestaurantsUseCase** dependency.

In other words, if we want **RestaurantsViewModel** to be injected, we need to make sure that its dependencies can be provided by Hilt, and their dependencies too, and so on.

7. Let's first make sure that Hilt knows how to provide **GetInitialRestaurantsUseCase** to **RestaurantsViewModel**. Inside the **GetInitialRestaurantsUseCase** class, move its dependencies inside the constructor and mark it with **@Inject**, just like we did with **RestaurantsViewModel**:

```
class GetInitialRestaurantsUseCase @Inject
constructor(
    private val repository: RestaurantsRepository,
    private val getSortedRestaurantsUseCase:
        GetSortedRestaurantsUseCase) {
    suspend operator fun invoke(): List<Restaurant>
    { ... }
}
```

After you add the **repository** and **getSortedRestaurantsUseCase** variables inside the constructor, remember to remove the old member variables as well as their instantiation code from the body of **GetInitialRestaurantsUseCase**.

Note that we aren't annotating the **GetInitialRestaurantsUseCase** class with any Hilt scope annotations, simply because we don't want it to be tied to a certain lifetime scope.

Now, Hilt knows how to inject the **GetInitialRestaurantsUseCase** class, yet we must also instruct Hilt how to provide its dependencies as well:

RestaurantsRepository and **GetSortedRestaurantsUseCase**.

We need to make sure that Hilt knows how to provide instances of **RestaurantsRepository**. We can see that its dependencies are **RestaurantsApiService** (the Retrofit interface) and **RestaurantsDao** (the Room Data Access Object interface):

```
class RestaurantsRepository {
    private var restInterface: RestaurantsApiService
    =
        Retrofit.Builder()
        [...]
        .create(RestaurantsApiService::class.java)
    private var restaurantsDao = RestaurantsDb
        .getDaoInstance(
            RestaurantsApplication.getAppContext()
        )
    [...]
}
```

The issue here is that once we place these dependencies inside the constructor and inject them, Hilt will have no idea how to provide them – simply because we cannot tap into the internal workings of Room or Retrofit and inject their dependencies too, like we did with **RestaurantsViewModel**, **GetInitialRestaurantsUseCase**, and now with **RestaurantsRepository**.

For Hilt to know how to provide dependencies out of our reach, we must create a **module** class where we will instruct Hilt on how to provide us with instances of **RestaurantsApiService** and **RestaurantsDao**:

8. Expand the **restaurants** package, then right click on the **data** package, and create a new package called **di** (short for dependency injection). Inside this package, create a new **object** class called **RestaurantsModule** and add the following code inside:

```
@Module
@InstallIn(SingletonComponent::class)
object RestaurantsModule { }
```

RestaurantsModule will allow us to instruct Hilt on how to provide Room and Retrofit dependencies to **RestaurantsRepository**. Since this is a Hilt module, we have done the following:

- Annotated it with **@Module** so that Hilt recognizes it as a module that provides instances of dependencies.
- Annotated it with **@InstallIn()** and passed the predefined **SingletonComponent** component provided by Hilt. Since our module is installed in this component, the dependencies that are contained can be provided anywhere throughout the application since **SingletonComponent** is an application-level dependency container.

9. Next up, inside **RestaurantsModule**, we need to tell Hilt how to provide our dependencies, so we will start with **RestaurantsDao**. For us to obtain an instance to **RestaurantsDao**, we must first instruct Hilt on how to instantiate a **RestaurantsDb** class.

Add a **provideRoomDatabase** method annotated with **@Provides** that will instruct Hilt how to provide an **RestaurantsDb** object by borrowing part of the instantiation code of the **database** class from the **companion object** of the **RestaurantsDb** class:

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

```

fun provideRoomDatabase(
    @ApplicationContext appContext: Context
): RestaurantsDb {
    return Room.databaseBuilder(
        appContext,
        RestaurantsDb::class.java,
        "restaurants_database"
    ).fallbackToDestructiveMigration().build()
}
}

```

First off, we've annotated the `provideRoomDatabase()` method with the `@Singleton` instance so that Hilt will create only one instance of `RestaurantsDb` for the whole application, allowing us to save memory.

Then, we can see that the `provideRoomDatabase()` method builds a `RestaurantsDb` instance, yet for this to work, we needed to provide the application-wide context to the `Room.databaseBuilder()` method. To achieve this, we have passed a `Context` object as a parameter of `provideRoomDatabase()` and annotated it with `@ApplicationContext`.

To understand how Hilt provides us with the application `Context` object, we must first note that each Hilt container comes with a set of default bindings that we can inject as dependencies. The `SingletonComponent` container provides us with the application-wide `Context` object wherever we need it by defining the `@ApplicationContext` annotation.

- Now that Hilt knows to provide us with `RestaurantsDb`, we can create another `@Provides` method that takes in a `RestaurantsDb` variable (which Hilt will now know how to provide) and return a `RestaurantsDao` instance:

```

@Module
@InstallIn(SingletonComponent::class)
object RestaurantsModule {
    @Provides
    fun provideRoomDao(database: RestaurantsDb):

```

```

    RestaurantsDao {
        return database.dao
    }
    @Singleton
    @Provides
    fun provideRoomDatabase(
        @ApplicationContext appContext: Context
    ): RestaurantsDb { ... }
}

```

11. Still inside **RestaurantsModule**, we now have to tell Hilt how to provide us with an instance of **RestaurantsApiService**. Do the same as before, but this time add a **@Provides** method for an instance of **Retrofit**, and one for an instance of **RestaurantsApiService**. Now, **RestaurantsModule** should look like this:

```

@Module
@InstallIn(SingletonComponent::class)
object RestaurantsModule {
    @Provides
    fun provideRoomDao(database: RestaurantsDb):
    [...] {
        return database.dao
    }
    @Singleton
    @Provides
    fun provideRoomDatabase(@ApplicationContext
        appContext: Context): RestaurantsDb
    { [...] }
    @Singleton
    @Provides
    fun provideRetrofit(): Retrofit {
        return Retrofit.Builder()
            .addConverterFactory([...])
            .baseUrl("[...]")
            .build()
    }
}

```

```
@Provides
fun provideRetrofitApi(retrofit: Retrofit):
    RestaurantsApiService {
    return retrofit
        .create(RestaurantsApiService::class.java)
    }
}
```

Remember that all this instantiation code resides in **RestaurantsRepository**, so you can get it from there.

12. Now that Hilt knows how to provide both dependencies of **RestaurantsRepository**, head back in the **RestaurantsRepository** class and apply constructor injection with Hilt by adding the **@Inject** annotation to the constructor while moving its **RestaurantsApiService** and **RestaurantsDao** dependencies inside the constructor:

```
@Singleton
class RestaurantsRepository @Inject constructor(
    private val restInterface:
    RestaurantsApiService,
    private val restaurantsDao: RestaurantsDao
) {
    suspend fun toggleFavoriteRestaurant(...) = [...]
        [...]
}
```

Usually, **Repository** classes have a static instance so that only one instance is re-used throughout the app. This is useful when different data is stored in memory globally in **Repository** classes (be cautious with system-initiated process death because that will wipe anything in memory!).

Finally, to have only one instance of **RestaurantsRepository** that can then be reused across the app, we have annotated the class with the **@Singleton** annotation. This annotation is provided by the Hilt

SingletonComponent container and allows us to scope instances of classes to the lifetime of the application.

13. Now that Hilt knows how to inject **RestaurantsRepository**, let's get back to the other remaining dependency of **GetInitialRestaurantsUseCase**: the **GetSortedRestaurantsUseCase** class. Head inside this class and make sure to inject its dependencies by moving the **repository** variable inside the constructor as we did before with other classes:

```
class GetSortedRestaurantsUseCase @Inject
constructor(
    private val repository: RestaurantsRepository
) {
    suspend operator fun invoke(): List<Restaurant>
    {
        return repository.getRestaurants()
            .sortedBy { it.title }
    }
}
```

While we have annotated **RestaurantsRepository** with a scope annotation, we haven't added any scope annotation for this Use Case class simply because we don't want the instance to be preserved across a specific lifetime.

Now, we have instructed Hilt how to provide all the dependencies for the first dependency of **RestaurantsViewModel**, which is **GetInitialRestaurantsUseCase**!

14. Next up, let's tell Hilt how to provide the dependencies for the second and last dependency of **RestaurantsViewModel**, the **ToggleRestaurantUseCase** class. Head inside this class and make sure to inject its dependencies by moving the **repository** and **getSortedRestaurantsUseCase** variables inside the constructor as we did before with other classes:

```
class ToggleRestaurantUseCase @Inject constructor(
    private val repository: RestaurantsRepository,
    private val getSortedRestaurantsUseCase:
        GetSortedRestaurantsUseCase
) {
    suspend operator fun invoke(id: Int, oldValue:
        Boolean): List<Restaurant> {
        val newFav = oldValue.not()
        repository.toggleFavoriteRestaurant(id,
newFav)
        return getSortedRestaurantsUseCase()
    }
}
```

15. Optionally, you can head inside the **RestaurantsDb** class and delete the entire **companion object** that was in charge of providing a singleton instance for our **RestaurantsDao**. The **RestaurantsDb** class should now be much slimmer and look like this:

```
@Database(
    entities = [LocalRestaurant::class],
    version = 3,
    exportSchema = false
)
abstract class RestaurantsDb : RoomDatabase() {
    abstract val dao: RestaurantsDao
}
```

It's safe to delete this instantiation code because from now on, Hilt will do that for us out of the box.

16. Also, if you followed the previous step of cleaning up the **RestaurantsDb** class, inside **RestaurantsApplication**, you can also remove all the logic inside this class that was related to obtaining the application-wide **Context** object. From now on, Hilt will do that for us out of the box.

The **RestaurantsApplication** class should be much slimmer and look like this:

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

17. Build and run the application. Now, the build should be successful because Hilt is in charge of providing the dependencies that we required it to provide.

With the help of DI, we have now promoted testability while also extracting the boilerplate associated with building class instances.

ASSIGNMENT

*We have integrated DI with Hilt for the first screen of **RestaurantsApplication**. However, the project is still not incorporating DI entirely because the second destination of our app (represented by the **RestaurantDetailsScreen()** composable) has neither its **RestaurantDetailsViewModel** injected nor this **ViewModel** class's dependencies injected. As a take-home assignment, incorporate DI in this second screen. This will allow you to get rid of the redundant Retrofit client instantiation inside **RestaurantDetailsViewModel** – remember that you can now inject a **RestaurantsApiService** instance directly with Hilt!*

Summary

In this chapter, we improved the architecture of the Restaurants App by incorporating DI.

We discussed what DI is and covered its basic concepts: dependency with its implicit or explicit types, injection, dependency containers, and manual injection.

We then examined the main benefits that DI brings to our projects: testable classes and less boilerplate code.

Finally, we covered how DI frameworks can help us with the injection of dependencies, and explored the Jetpack Hilt library as a viable solution for DI on Android. Afterward, we practiced what we learned as we incorporated DI with Hilt in our Restaurants app.

Since we incorporated DI, it's a bit clearer that our classes can be easily tested, so it's time we start writing some tests in the next chapter!

Further reading

Knowing how to work with the basics of Hilt is usually enough for most projects. However, sometimes you might need to use more advanced features of Hilt or Dagger. To learn more about Dagger and how the framework automatically creates the dependencies for you by building a dependency graph, check this article: <https://medium.com/android-news/dagger-2-part-i-basic-principles-graph-dependencies-scopes-3dfd032ccd82>.

On the same note, apart from the `@Singleton` scope that was the most used scope throughout our app, Dagger Hilt exposes a broader variety of predefined components and scopes that allow you to scope different classes to various lifecycles. Check out more about components and their scopes in the official documentation: <https://dagger.dev/hilt/components.html>.

Leaving components and their scopes aside, in some projects, you might need to allow injection of dependencies in other Android classes than **Activity**. To see which Android classes can be annotated with `@AndroidEntryPoint`, check out the documentation: <https://dagger.dev/hilt/android-entry-point>.

[Support](#) | [Sign Out](#)

©2022 O'REILLY MEDIA, INC. [TERMS OF SERVICE](#) | [PRIVACY POLICY](#)