



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

Day 18

Dependency Injection

(Part 2 of 2)

Diving Deep into Hilt and Dagger for Mastery over
Scopes, Bindings, Qualifiers, and Testing 🚀



Esmaeel Moradi

@esmaeel-moradi700

Day 18



Advanced Dependency Injection in Android: Diving Deep into Hilt and Dagger for Mastery over Scopes, Bindings, Qualifiers, and Testing 🚀

**Greetings once again to all professional colleagues,
engineers, and system architects!**

In our Day 17 article, we laid the foundation for understanding Dependency Injection (DI). We grasped why DI is crucial for solving the Tight Coupling problem and boosting Testability, became familiar with manual implementation, and got an initial look at the powerful Hilt and Dagger frameworks. You now know that constructor injection (*@Inject constructor()*) is a great starting point. However, in real-world applications, managing dependencies is often more complex than simply injecting one dependency into another class's constructor. We face the need to inject interfaces, handle multiple instances of the same type, and control the lifecycle of dependencies. Today, on the eighteenth day of our challenge, we are going to dive deep into the advanced arsenal of Hilt and Dagger and discover the tools needed to build truly robust, scalable, and testable applications that go beyond simple examples and demos. You know why DI is vital and are familiar with the basics of injection. Now, let's unleash the full power of DI frameworks!



Esmaeel Moradi

@esmaeel-moradi700

Introduction:

Beyond *@Inject constructor()* – Unleashing the Full Power of DI Frameworks.

In the real world of Android development, you will encounter scenarios where merely using the *@Inject constructor()* annotation is not enough. You will have dependencies that are interfaces, and the framework needs to know which specific implementation to inject. You might need to provide multiple different instances of the same type (e.g., two different base URLs). Or it might be necessary to tie the lifecycle of an expensive dependency to the lifecycle of a specific part of your application. This is where advanced DI concepts come into play.



Esmaeeil Moradi

@esmaeeil-moradi700



Advanced Binding Types in Hilt/Dagger:

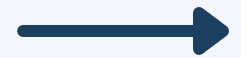
@Binds:

- **"Why" do we need @Binds?** When you want to bind an interface to a specific implementation. For example, you want to inject *AnalyticsService* based on its interface type, but at runtime, an instance of *FirebaseAnalyticsService* (which implements that interface) should be injected. *@Inject constructor()* only works on concrete classes.
- **How it Works:** *@Binds* is an abstract annotation placed on an abstract method within a *@Module*. This method must have one parameter, which is the type of the concrete implementation, and its return type must be the interface type.
- **Difference from @Provides:** *@Binds* is used to connect an interface to an existing implementation, whereas *@Provides* is used for constructing a new instance (Third-party class, framework class, or complex construction). Using *@Binds*, when possible, is slightly more performant as it generates less code.



Esmaeel Moradi

@esmaeeil-moradi700



```
// Example: @Binds
import dagger.Binds
import dagger.Module
import dagger.hilt.InstallIn
import dagger.hilt.components.SingletonComponent
import javax.inject.Inject
import javax.inject.Singleton

interface AnalyticsService {
    fun trackEvent(eventName: String)
}

// Break FirebaseAnalyticsService definition more aggressively
class FirebaseAnalyticsService
    @Inject constructor()
    : AnalyticsService {
    override fun trackEvent(eventName: String) { /* ... */ }
}

@Module
@InstallIn(SingletonComponent::class)
abstract class AnalyticsBindingModule { // Must be abstract for @Binds methods

    @Binds // Binds the interface to the concrete implementation
    @Singleton
        // Scope the binding (Scoped in the component where the module is installed)
    abstract fun bindAnalyticsService(
        firebaseAnalyticsService: FirebaseAnalyticsService
        // The implementation type
    ): AnalyticsService // The interface type
}
```



Esmaeel Moradi

@esmaeeil-moradi700



@Provides (Revisited with Advanced Use Cases):

This annotation is placed on methods within a `@Module` that tell the framework how to construct Dependencies that cannot be injected with `@Inject constructor()`.

- **Common Use Cases:**

1. **Providing classes from Third-party libraries:** Like *OkHttpClient*, *Retrofit*, *Gson*.
2. **Providing framework classes:** Like *Context*, *Resources*, *FragmentManager* (although Hilt provides many of these internally).
3. **Providing complex objects that require specific setup:** Like a database instance requiring initial configuration.
4. **Providing multiple instances of the same type (with the help of Qualifiers).**

In the example below, *UserRepository* needs the *ApiService* interface, but `@Provides` in *NetworkModule* tells Hilt how to construct its implementation (via *Retrofit*, which itself needs an *OkHttpClient*).



Esmaeel Moradi

@esmaeel-moradi700

```
// Example: @Provides for a Third-party Library
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.components.SingletonComponent
import javax.inject.Singleton
import okhttp3.OkHttpClient
import okhttp3.logging.HttpLoggingInterceptor
import okhttp3.logging.HttpLoggingInterceptor.Level
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
import retrofit2.create // Extension function

// Assuming ApiService interface is defined elsewhere
interface ApiService { /* ... */ }

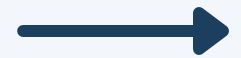
@Module
@InstallIn(SingletonComponent::class)
object NetworkModule {

    @Provides // Provides an OkHttpClient instance
    @Singleton // Scope the instance
    fun provideOkHttpClient(): OkHttpClient {
        // Complex setup logic
        val logging = HttpLoggingInterceptor().apply { level = Level.BODY }
        return OkHttpClient.Builder()
            .addInterceptor(logging)
            .build()
    }

    @Provides // Provides a Retrofit instance
    @Singleton
    fun provideRetrofit(okHttpClient: OkHttpClient): Retrofit { // Dependencies
        return Retrofit.Builder()
            .baseUrl("https://api.example.com/")
            .client(okHttpClient)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }

    @Provides // Provides an ApiService instance (interface)
    @Singleton
    fun provideApiService(retrofit: Retrofit): ApiService { // ApiService inter
        return retrofit.create(ApiService::class.java)
    }
}

// Assuming UserRepository exists and needs ApiService
// class UserRepository @Inject constructor(private val apiService: ApiService)
```



Esmaeel Moradi

@esmaeel-moradi700

Qualifiers:

Distinguishing Multiple Bindings of the Same Type.

The problem arises when you need to provide two or more different instances of the same type. The DI framework doesn't know which instance to inject. Qualifiers solve this problem.

- **The Problem:**

```
// Problem: Two Strings needed, but how to distinguish?
// @Inject lateinit var apiKey: String // Which string?
// @Inject lateinit var baseUrl: String // Which string?

// Module attempting to provide multiple Strings (would result in a binding error)
// @Module
// @InstallIn(SingletonComponent::class)
// object AppConfigModule {
//     @Provides fun provideApiKey(): String = "MY_API_KEY"
//     @Provides fun provideBaseUrl(): String = "https://myapi.com/"
//     // Error: Duplicate String binding!
// }
```

- **The *@Qualifier* Annotation:**

A marker annotation placed on another annotation to turn it into a Qualifier:

```
// Define custom Qualifiers
import javax.inject.Qualifier

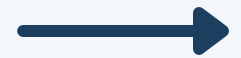
@Qualifier // Marker annotation
@Retention(AnnotationRetention.RUNTIME) // Keep at runtime (useful for reflection)
annotation class ApiKey

@Qualifier
@Retention(AnnotationRetention.RUNTIME)
annotation class BaseUrl
```



Esmaeel Moradi

@esmaeel-moradi700



- **Using Qualifiers:**

Apply the Qualifier annotation on both the *@Provides/@Binds* method and the *@Inject* injection point:

```
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.components.SingletonComponent
import javax.inject.Inject

@Module
@InstallIn(SingletonComponent::class)
object AppConfigModule {

    @Provides
    @ApiKey // Apply the Qualifier to the provider method
    fun provideApiKey(): String = "MY_API_KEY"

    @Provides
    @BaseUrl // Apply the Qualifier to the provider method
    fun provideBaseUrl(): String = "https://myapi.com/"
}

// Injecting using Qualifiers
class NetworkManager @Inject constructor(
    @ApiKey private val apiKey: String, // Specify which String binding to inject
    @BaseUrl private val baseUrl: String // Specify which String binding to inject
) { /* ... */ }
```

- **Built-in Qualifiers:**

@Named is a commonly used built-in Qualifier that you can use instead of defining custom annotations, especially for simple string-based distinctions (*@Named("apiKey")*).



Esmaeel Moradi

@esmaeeil-moradi700

Scoping: Managing the Lifecycle of Dependencies.



Scoping allows you to control how many instances of a Dependency are created and how long they live. The primary goal is performance optimization (avoiding repeated creation of expensive objects) and resource management.

- **Hilt's Built-in Scopes:** Hilt provides several built-in Scopes tied to Android component lifecycles:
- **@Singleton:** A single instance throughout the lifecycle of *SingletonComponent* (the entire application).
- **@ActivityRetainedScoped:** An instance lives as long as the *ActivityRetainedComponent* (survives configuration changes, similar to a *ViewModel*).
- **@ActivityScoped:** An instance lives as long as the *ActivityComponent*.
- **@FragmentScoped:** An instance lives as long as the *FragmentComponent*.
- And other scopes for *View*, *Service*, *ViewModel*. You apply these scopes by annotating the *@Provides* method or the *@Inject constructor()* annotated class.



Esmaeel Moradi

@esmaeel-moradi700

- **Custom Scopes (In Dagger and their relevance to understanding Hilt):**

In raw Dagger, you can define your own custom Scopes using the @Scope annotation and tie them to Components with custom lifecycles. This concept is helpful in understanding the underlying mechanism of Hilt's scopes and how components like ActivityComponent are implicitly scoped to the Activity's lifecycle.

```
// Conceptual Dagger Custom Scope Example
import javax.inject.Scope
import dagger.Component // Dagger annotation

@Scope // Define a custom scope annotation
@Retention(AnnotationRetention.RUNTIME)
annotation class UserSessionScoped // Example scope annotation

// This scope is applied to a Dagger Component
// @UserSessionScoped // Apply the custom scope to the component
// @Component(modules = [...])
// interface UserSessionComponent {
//     // ... provide dependencies for the user session
// }

// And then dependencies provided within this component can be scoped
// @Provides
// @UserSessionScoped // Apply the custom scope to the provider
// fun provideUserSessionManager(): UserSessionManager { ... }
```

Hilt automates this process for standard Android scopes, but understanding that Scopes are fundamentally tied to Components and their lifecycles is important.



Esmaeel Moradi

@esmaeel-moradi700

• Why is Scoping Important?



1. **Optimization:** Dependencies like *OkHttpClient* or large database instances should not be created anew every time they are injected. Scoping them to the Application or Activity lifecycle can significantly improve performance.
2. **Resource Management:** Ensure that expensive resources (like network connections or databases) live as long as the relevant Component is alive and are properly released afterwards.



Esmaeel Moradi

@esmaeel-moradi700

Components and Subcomponents

(Structure of Dagger and Hilt):



- **Components:** A *@Component* in Dagger, or the Components generated by Hilt, represent a complete or partial graph of dependencies. These Components are the access points for retrieving dependencies.
- **Subcomponents:** A *@Subcomponent* in raw Dagger is a Component that inherits dependencies from its parent Component. Subcomponents are used to create subgraphs of dependencies for specific sections of the application (like the scope of a particular feature or an Activity) and can have their own Scope.
- **Understanding the Hilt Component Hierarchy (@InstallIn):**
The *@InstallIn* annotation in Hilt directly maps to the internal Dagger Component hierarchy that Hilt builds for Android. *SingletonComponent* is the root parent, and *ActivityComponent* and *ActivityRetainedComponent* are its *subcomponents*. Understanding this hierarchy helps you decide where to install your Modules and what the scope of your dependencies will be.



Esmaeel Moradi

@esmaeel-moradi700



```
// Hilt Component Hierarchy (Conceptual relationship via @InstallIn)
// SingletonComponent (App Scope, @Singleton)
// |
// +--- ActivityRetainedComponent
//      (Activity survives config changes, @ActivityRetainedScoped)
// |
// | +--- ViewModelComponent (ViewModel Scope, @ViewModelScoped)
// |
// +--- ActivityComponent (Activity Scope, @ActivityScoped)
// |
// | +--- FragmentComponent (Fragment Scope, @FragmentScoped)
// |
// | +--- ViewComponent (View Scope, @ViewScoped)
// |
// | +--- ServiceComponent (Service Scope, @ServiceScoped)
// |
// | +--- ServiceComponent (Service Scope, @ServiceScoped)
// |
//      // For foreground/background services
```



Esmaeel Moradi

@esmaeeil-moradi700

Injection in Testing:



One of the strongest arguments for using DI is that it dramatically simplifies testing. You can easily replace real dependencies with Mocks or Fake Implementations.

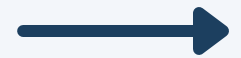
- **Testing Strategies with Hilt:**

1. **@HiltAndroidTest:** Annotate your test class (usually an Instrumented test class) with this.
2. **@UninstallModules:** Use this annotation on your test class to replace specific production Modules with test-specific Modules (e.g., a module providing a mock *ApiService*).
3. **Injecting Mocks in tests:** You can inject Mocks or Fake dependencies directly into your test class using *@Inject* or provide them via test Modules.
4. **Test Modules:** Write specific Modules for your tests that provide Fake or Mock implementations of Dependencies using *@Provides* and install them in the appropriate test Component (e.g., *SingletonComponent::class* for Instrumented tests).



Esmaeel Moradi

@esmaeel-moradi700



```
// Example: Hilt Test Setup with Mocking
import dagger.hilt.android.testing.HiltAndroidRule
import dagger.hilt.android.testing.HiltAndroidTest
import dagger.hilt.android.testing.UninstallModules
import org.junit.Before
import org.junit.Rule
import org.junit.Test
import org.mockito.Mockito.* // For mocking

// Assuming UserRepository and ApiService exist from Day 17 example
// Assuming NetworkModule provides ApiService

@HiltAndroidTest // Mark test class for Hilt
@UninstallModules(NetworkModule::class) // Replace the real network module
class UserRepositoryTest {

    @get:Rule // Hilt's test rule
    val hiltRule = HiltAndroidRule(this)

    @Inject
    // Inject the UserRepository (which will now get the mocked dependency)
    lateinit var userRepository: UserRepository

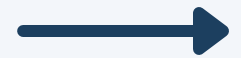
    @Inject
    lateinit var mockApiService: ApiService
    // This must be provided by a test module

    // A test module to provide the mock ApiService
    @Module
    @InstallIn(SingletonComponent::class)
    // Or another test component appropriate for the test scope
    object TestNetworkModule { // Name doesn't have to match the uninstalled one
```



Esmaeel Moradi

@esmaeel-moradi700



```
@Provides // Tells Hilt how to provide the mock ApiService
@Singleton // Scope the mock if needed
fun provideMockApiService(): ApiService {
    // Return a mock instance
    return mock(ApiService::class.java) // Using Mockito to create a mock
}
// Provide other mock/fake dependencies needed for testing UserRepository
}

@Before // Setup method
fun setup() {
    // Inject dependencies into the test class instance
    hiltRule.inject()
}

@Test // Your test method
fun getUser_success_returnsUser() {
    // Define the behavior of the mock dependency
    val dummyUser = User("123", "Test User")
    // Assuming User data class exists
    `when`(mockApiService.fetchUser("123")).thenReturn(dummyUser)

    // Now test the userRepository which uses the mock
    val user = userRepository.getUser("123")

    // Assertions
    assertEquals("Test User", user.name)
    // Verification: Ensure the dependency was called as expected
    verify(mockApiService).fetchUser("123")
}
}
```

Testing raw Dagger-based code follows similar principles but requires manual management of test Components and injecting them.



Esmaeel Moradi

@esmaeeil-moradi700

Advanced Use Cases & Best Practices (Review):



- **Optional Bindings** (*@BindsOptionalOf*): In Dagger and Hilt, you can define a Dependency as optional. If a binding for that type exists in the graph, it's injected; otherwise, *null* or an empty *Optional* is injected.
- **Multibindings** (*@IntoMap*, *@IntoSet*): For providing a collection of Dependencies of a specific type (e.g., a set of plugins, handlers, or factories) as a *Map* or *Set*. Very useful for plugin-based architectures. (This is a more advanced topic).
- **Lazy Injection (*Lazy<T>*) and Provider Injection (*Provider<T>*):**
 1. ***Lazy<T>***: Lazy injection. The Dependency instance is created only on the first call to the *.get()* method on the Lazy object. Useful for expensive Dependencies that are not always needed.
 2. ***Provider<T>***: Provider injection. A new instance of the Dependency is created every time the *.get()* method is called on the Provider object (unless the Dependency is scoped). Useful for Dependencies where you need multiple distinct instances during the Component's lifecycle.



Esmaeel Moradi

@esmaeel-moradi700

```
// Example: Lazy and Provider Injection
import javax.inject.Inject
import dagger.Lazy // Dagger's Lazy type
import javax.inject.Provider // Standard JSR 330 Provider

class ExpensiveObject @Inject constructor() {
    init { Log.d("DIExample", "ExpensiveObject created!") }
    // To show when it's created
    fun doWork() { /* ... */ }
} // Assume this is costly to create

class SomeManager @Inject constructor(
    private val lazyExpensiveObject: Lazy<ExpensiveObject>, // Lazy Injection
    private val expensiveObjectProvider: Provider<ExpensiveObject>
    // Provider Injection
) {
    fun useLazyFirstTime() {
        Log.d("DIExample", "Calling useLazyFirstTime...")
        val obj = lazyExpensiveObject.get()
        // Object created HERE on first call to this method
        obj.doWork()
    }

    fun useLazySecondTime() {
        Log.d("DIExample", "Calling useLazySecondTime...")
        val obj = lazyExpensiveObject.get()
        // Object IS NOT created again (reused)
        obj.doWork()
    }

    fun useProviderMultipleTimes() {
        Log.d("DIExample", "Calling useProviderMultipleTimes...")
        val obj1 = expensiveObjectProvider.get()
        // New object created HERE
        val obj2 = expensiveObjectProvider.get()
        // ANOTHER new object created HERE
        obj1.doWork()
        obj2.doWork()
        // obj1 != obj2 unless ExpensiveObject is scoped to a scope
        // where new instances are provided
    }
}
}
```



Esmaeel Moradi
@esmaeeil-moradi700



Choosing Injection Strategy (Constructor vs. Field vs. Method):

- **Constructor Injection (*@Inject constructor()*):** The first and best choice. Makes dependencies explicit and code highly testable.
- **Field Injection (*@Inject lateinit var*):** Used when you don't have direct control over the class constructor (like Activity, Fragment in Android) or for testing. Try to keep the number of fields injected this way low.
- **Method Injection (*@Inject fun set...()*):** Less common, used for injecting dependencies needed after the initial object construction.



Esmaeel Moradi

@esmaeeil-moradi700

Conclusion:

Mastering DI for Robust Android Architecture.



Advanced Dependency Injection concepts with Hilt and Dagger are powerful tools that enable you to confront and solve the most complex dependency management scenarios in real-world projects. Understanding how *@Binds* and advanced *@Provides* work, the importance of Qualifiers, how to manage lifecycles with Scoping, the structure of Components and Subcomponents, and correctly implementing injection in tests, defines the boundary between an average developer and a skilled Android architect.

Mastering these topics with tools like Hilt (for its simplicity and integration with Android) and a foundational understanding of Dagger (for grasping the underlying mechanisms) is crucial for building production-grade, maintainable, testable, and scalable Android applications. Don't shy away from practicing and experimenting with these patterns. This knowledge is a significant investment in your professional future.



Esmaeel Moradi

@esmaeel-moradi700

Looking forward to your
insights and experiences!
Let's elevate Android
development together.



LEAVE A COMMENT BELOW



Esmaeeil Moradi

@esmaeeil-moradi700