



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

Day 17

Dependency Injection

(Part 1 of 2)

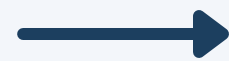
From the Coupling Problem to First Steps with
Manual DI, Hilt, and Dagger 🚀



Esmaeel Moradi

@esmaeel-moradi700

Day 17



Fundamentals of Dependency Injection in Android: From the Coupling Problem to First Steps with Manual DI, Hilt, and Dagger 🚀

Greetings once again to all professional colleagues, engineers, and system architects! I hope you are energized and ready for one of the most crucial topics in software architecture: Dependency Injection (DI).

Throughout my years as an Android developer, I've repeatedly encountered code that was a nightmare to test, required major surgery in different parts of the application to change, and was exhausting to maintain. The root of many of these problems traces back to one concept: Tight Coupling.

Today, on the seventeenth day of our challenge, we want to address this fundamental issue and introduce its remedy: Dependency Injection. Tired of classes tightly coupled to their dependencies, making testing a nightmare? Let's unleash the power of Dependency Injection – the pattern that transforms your codebase from rigid and inflexible to flexible and testable! This article is your starting point for a deep understanding of the "why" and "how" of DI fundamentals in Android.



Esmaeel Moradi

@esmaeel-moradi700

Introduction: The Tight Coupling Problem – Why Dependencies Are Painful (and Why DI is the Cure) →

Imagine you have a class that needs another dependency to perform its task, for example, an ApiService for communicating with a server:

```
// The Problem: Tight Coupling
class UserRepository {
    private val apiService = ApiService() // UserRepository creates its dependency

    fun getUser(userId: String) {
        apiService.fetchUser(userId) // Uses the dependency
    }
}

class ApiService {
    fun fetchUser(userId: String) {
        // Network call logic...
    }
}
```

In this example, *UserRepository* creates the *ApiService* itself using `new ApiService()`. This approach seems simple at first glance, but it has serious consequences:



Esmaeel Moradi
@esmaeel-moradi700

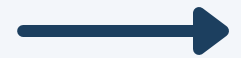


- **Tight Coupling:** *UserRepository* is directly dependent on a specific implementation of *ApiService*. You cannot easily replace it with another implementation (like a *MockApiService* for testing).
- **Hard to Test:** To test *UserRepository*, the real *ApiService* and its network logic are also implicitly involved! You cannot control (Mock) the behavior of *ApiService* to test different scenarios (success, error, specific data).
- **Reduced Flexibility:** If you wanted to use a different *ApiService* (e.g., for Development or Staging environments), you would have to modify the *UserRepository* code.
- **Harder Maintenance and Changes:** A change in *ApiService* might require changes in all classes that directly create it.

Dependency Injection (DI) is a design pattern introduced to solve this problem. The core idea is simple: instead of a class creating its own dependencies, the dependencies are "injected" into that class from the outside.



Esmaeel Moradi
@esmaeel-moradi700



```
// The Solution: Dependency Injection (Manual Example)
class UserRepository(private val apiService: ApiService) { // Dependency injected

    fun getUser(userId: String) {
        apiService.fetchUser(userId) // Uses the dependency
    }
}

// In another part of the code (Composition Root)
// val apiService = ApiService()
// val userRepository = UserRepository(apiService) // Inject the dependency
```

Key advantages of this approach:

- **Significantly Increased Testability:** Now you can easily test *UserRepository* with a *MockApiService* (or any other implementation of the *ApiService* interface) without needing the actual network.
- **Increased Flexibility:** You can inject different *ApiService* implementations without changing the *UserRepository* code.
- **Simpler Maintenance:** Classes are less dependent on implementation details.
- **Clearer Code Structure:** A class's dependencies are explicitly stated in its constructor.



Esmaeel Moradi
@esmaeel-moradi700



The Core Principle: Inversion of Control (IoC)

DI is closely related to the Inversion of Control (IoC) principle. **IoC means the control of the program's flow (or in this case, the control of creating and managing dependencies) is taken away from the class in question and delegated to an external framework or mechanism.** In the example above, *UserRepository* is no longer responsible for creating *ApiService*; this responsibility is delegated to external code (the Composition Root). DI is a specific and very common form of implementing IoC.



Esmaeel Moradi

@esmaeeil-moradi700



Manual Dependency Injection ("Do It Yourself!"):

You can implement DI without using any frameworks. This helps in understanding the core concept. There are three main ways to inject dependencies manually:

- **Constructor Injection:** The recommended and cleanest approach. Dependencies are injected into the class via its constructor parameters. This method clearly shows a class's dependencies in its definition.

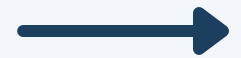
```
class AuthRepository(private val authService: AuthService) // Constructor Inject
class AuthService { /* ... */ }

// Usage
// val authService = AuthService()
// val authRepository = AuthRepository(authService)
```



Esmaeel Moradi
@esmaeel-moradi700

- **Setter Injection:**



Dependencies are injected via public setter methods. This method is used when you don't have direct control over the class's constructor (like Activity or Fragment in Android, which the system instantiates) or when the dependency is optional.

```
class SettingsFragment : Fragment() { // Android component
    lateinit var settingsPresenter: SettingsPresenter // Dependency field

    // Setter method (often called from outside)
    fun setSettingsPresenter(presenter: SettingsPresenter) {
        this.settingsPresenter = presenter
    }

    // ... usage of settingsPresenter ...
}

// Formatted for better readability
class SettingsPresenter(private val settingsRepository: SettingsRepository) {
    /* ... */ // Placeholder for the actual class body (methods, etc.)
}

// Usage (e.g., in containing Activity or a factory)
// val presenter = SettingsPresenter(...)
// fragment.setSettingsPresenter(presenter) // Manually injecting
```



Esmaeel Moradi
@esmaeel-moradi700

- **Field Injection:**

Dependencies are injected directly into public fields. This method is less recommended in manual DI because it breaks encapsulation and hides dependencies within the class definition.

```
class ReportService {  
    // Should ideally be private, but for manual field injection it's often pu  
    lateinit var reportGenerator: ReportGenerator // Field Injection target  
}  
  
class ReportGenerator { /* ... */ }  
  
// Usage (e.g., in another class)  
// val reportService = ReportService()  
// reportService.reportGenerator = ReportGenerator() // Manually setting field
```

Advantages of Manual DI:

Deep understanding of the core "why" and "how" of DI. For very small projects, it might not have framework overhead.

Drawbacks of Manual DI: As projects grow to medium or large size, manually managing the creation and wiring of all dependencies (*UserRepository* needs *ApiService*, *ApiService* needs *OkHttpClient*, *OkHttpClient* needs specific Configuration, etc.) quickly becomes complex, tedious, and error-prone. A high volume of boilerplate code is needed for "wiring" dependencies.

These drawbacks of manual DI are what drive us towards DI frameworks.



Esmaeel Moradi

@esmaeel-moradi700

Dependency Injection Frameworks: Introduction.



DI frameworks are tools that automate the tedious process of creating, managing, and providing dependencies. By reading your dependency definitions (via annotations or code), they build the dependency graph of your application and inject the necessary dependencies into your classes at the appropriate time. Popular frameworks in the Android ecosystem include Dagger (and its simplified version Hilt) and Koin (which is a Service Locator framework based on Kotlin DSL, although often categorized with DI frameworks).

Hilt: Google's Recommended DI Solution for Android (Basics).

Hilt is a recommended library from the Jetpack suite that significantly simplifies the use of Dagger in Android applications. Hilt reduces much of the Dagger boilerplate through the use of annotations and conventions.

- **Why Hilt for Android?**
 - Designed specifically for standard Android components (Activity, Fragment, Service, Application, etc.).
 - Simplifies Scope management tied to Android component lifecycles.
 - Reduces Dagger complexity.



Esmaeel Moradi

@esmaeel-moradi700

Basic Concepts in Hilt:



- **@HiltAndroidApp**: Annotate your Application class with this to trigger Hilt's code generation process.
- **@AndroidEntryPoint**: Annotate Android components (Activity, Fragment, etc.) into which you want Hilt to inject dependencies.
- **@Inject**: Use this annotation to request dependency injection. Constructor Injection (@Inject constructor()) is the recommended and cleanest method. You can also use it for Field Injection, but prefer constructor injection whenever possible.
- **@Module**: Annotate classes that group the logic for providing dependencies that Hilt doesn't know how to create by default (e.g., interfaces or classes without @Inject annotated constructors).
- **@InstallIn**: This annotation, placed on a @Module, specifies which Hilt Component the Module belongs to and thus the scope of its provided dependencies (e.g., *SingletonComponent::class* for Application-level scope, *ActivityComponent::class* for Activity, *FragmentComponent::class* for Fragment).



Esmaeel Moradi

@esmaeel-moradi700



Simple Hilt Setup & Usage Example:

- build.gradle (Project level):

```
// build.gradle (Project Level - Groovy DSL)

plugins {
    // IMPORTANT: Check for the latest stable version!
    // Version 8.2.0 is likely outdated as of May 2025.
    id 'com.android.application' version '8.2.0' apply false
    // or your version

    // IMPORTANT: Check for the latest stable version!
    // Version 1.9.20 is likely outdated as of May 2025.
    id 'org.jetbrains.kotlin.android' version '1.9.20' apply false
    // or your version

    // Hilt plugin definition.
    // IMPORTANT: Check for the latest stable version!
    // Version 2.51 is likely outdated as of May 2025.
    id 'com.google.dagger.hilt.android' version '2.51' apply false
    // Hilt plugin
}

// Note: Plugins declared here with 'apply false' are typically
// applied and configured in the module-level build.gradle file(s).
```



Esmaeel Moradi
@esmaeeil-moradi700

build.gradle (Module level):



```
// build.gradle (Module Level)

plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'com.google.dagger.hilt.android' // Apply Hilt plugin
    // id 'kotlin-kapt' // If using Java or older Kotlin
}

android { /* ... */ } // Your Android configurations

dependencies {
    // Dependencies
    implementation 'androidx.core:core-ktx:1.13.1'
        // or latest
    implementation 'androidx.appcompat:appcompat:1.6.1'
        // or latest
    // ... other dependencies

    // Hilt
    implementation 'com.google.dagger:hilt-android:2.51' // Hilt core
    annotationProcessor 'com.google.dagger:hilt-compiler:2.51'
        // Hilt annotation processor

    // kapt 'com.google.dagger:hilt-compiler:2.51'
        // If using kapt

    // Testing
    // testImplementation 'com.google.dagger:hilt-android-testing:2.51'
        // For testing
    // kaptTest 'com.google.dagger:hilt-compiler:2.51'
        // Use kaptTest
    // androidTestImplementation 'com.google.dagger:hilt-android-testing:2.51'
        // For Android testing
    // kaptAndroidTest 'com.google.dagger:hilt-compiler:2.51'
        // Use kaptAndroidTest
}

// Other module configs here.
```



Esmaeel Moradi

@esmaeel-moradi700



Application Class:

```
import android.app.Application
import dagger.hilt.android.HiltAndroidApp

@HiltAndroidApp // Annotate your Application class
class MyApplication : Application() {
    // Hilt will generate necessary code here
}
```

A Simple Dependency:

```
import javax.inject.Inject

class MyDependency @Inject constructor() { // Hilt knows how to create this
    fun doSomething() = "Dependency is doing something"
}
```



Esmaeel Moradi
@esmaeeil-moradi700

Injecting in an Activity:



```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import dagger.hilt.android.AndroidEntryPoint
import javax.inject.Inject // Standard JSR 330 Inject annotation
import android.util.Log

@AndroidEntryPoint // Annotate the Android component
class MainActivity : AppCompatActivity() {

    @Inject // Hilt will inject MyDependency here
    lateinit var myDependency: MyDependency

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Now you can use myDependency
        Log.d("HiltExample", myDependency.doSomething())
    }
}
```



Esmaeel Moradi
@esmaeeil-moradi700

A Simple Module

(if you need to provide something Hilt can't @Inject constructor):



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

interface AnalyticsService { // Example: An interface
    fun trackEvent(eventName: String)
}

// This implementation could also be @Inject constructor(),
// but showing a module to provide it via interface
class FirebaseAnalyticsService : AnalyticsService {
    override fun trackEvent(eventName: String) {
        // Log event to Firebase...
    }
}

@Module // Group dependency provision logic
@InstallIn(SingletonComponent::class) // Install in Application-level component
object AnalyticsModule {

    @Provides // Tells Hilt how to provide AnalyticsService
    @Singleton // Scope to Application lifecycle
    fun provideAnalyticsService(): AnalyticsService {
        // Return the instance Hilt should provide
        return FirebaseAnalyticsService() // Or another implementation
    }
}
```



Esmaeel Moradi
@esmaeeil-moradi700

Dagger: The Underlying Powerhouse (Basics).



Dagger is a code-generating DI framework that validates your dependency graph at compile time. Hilt handles many of the complexities of connecting Dagger to Android components, but understanding Dagger's basic structure can be helpful.

- **Basic Concepts in Raw Dagger (for Hilt context):**
- *@Component*: An interface representing the dependency graph. You define methods to get required instances from this graph.
- *@Subcomponent*: (Similar to Hilt's internal structure) Represents a sub-graph.
- *@Module*: (Similar to Hilt) A class holding @Provides methods.
- *@Provides*: A method within a @Module that tells Dagger how to construct a Dependency.
- *@Inject*: (Similar to Hilt) Request for Dependency injection.



Esmaeel Moradi

@esmaeel-moradi700

• Very Simple Conceptual Dagger Structure Example

(without full Setup details):

As you can see, raw Dagger requires explicit definition of Components and sometimes injection methods, whereas Hilt automates these parts for Android components. →

```
// Simple conceptual Dagger structure

import dagger.Component
import dagger.Module
import dagger.Provides
import javax.inject.Inject
import javax.inject.Singleton // Dagger's Singleton

interface AnalyticsService { /* ... */ }

// Break FirebaseAnalyticsService definition more aggressively
class FirebaseAnalyticsService
    @Inject constructor()
    : AnalyticsService {
    /* ... */
}

// Dagger 2 requires an annotation processor (Kapt or KSP)
// to generate the Component implementation.
class UserRepository @Inject constructor(
    private val apiService: ApiService
    // Constructor parameter on a new line for better readability
) { /* ... */ }

class ApiService @Inject constructor() { /* ... */ }

@Module // Dagger Module
object AppModule {
    @Provides // Provider method
    @Singleton // Dagger Scope annotation
    fun provideAnalyticsService(): AnalyticsService {
        return FirebaseAnalyticsService()
    }
    // Add other providers as needed
}

@Singleton // Dagger Scope annotation for the Component
@Component(modules = [AppModule::class]) // Dagger Component (the dependency graph)
interface AppComponent {
    // Method to get a top-level dependency from the graph
    fun getUserRepository(): UserRepository

    // Methods for Field Injection (for classes not constructed by Dagger)
    // fun inject(activity: MainActivity)
}

// Usage (Manual Component creation - Hilt automates this for Android)
// In your Application class or Composition Root
// val appComponent = DaggerAppComponent.create()
// DaggerAppComponent is generated by Kapt/KSP
// val userRepository = appComponent.getUserRepository() // Getting dependency
// appComponent.inject(this) // Field injecting into an Activity if needed
```



Esmaeel Moradi

@esmaeel-moradi700

Comparing Manual DI, Hilt Basics, and Dagger Basics:

Feature	Manual DI	Hilt (Basics)	Dagger (Basics)
Concept Understanding	Excellent (you implement it yourself)	Good (with annotation helpers)	Good (requires understanding Component/Module structure)
Boilerplate Code	High (increases with project size)	Low (especially for Android components)	Medium to High (requires Component/Subcomponent definition)
Graph Management	Very High (manual wiring)	Low (Hilt handles wiring)	Medium (requires connecting Components/Modules)
Error Validation	Runtime	Compile-time - Excellent	Compile-time - Excellent
Scoping	Manual and complex	Simple (with <code>@InstallIn</code> and built-in Components)	Manual and requires custom Scope definitions
Initial Setup	Simple (just write code!)	Medium (Gradle plugin, Application annotation)	Medium to High (Gradle plugin, initial Component definition)
Flexibility	Medium (depends on implementation)	High (for common Android scenarios)	Maximum
Suitable For	Very small projects, prototyping, learning	Most medium to large Android applications	Complex DI needs, libraries



Esmaeel Moradi
@esmaeel-moradi700

Advantages of Using a Framework (Hilt/Dagger) in Real Projects Compared to Manual DI:

- **Automation:** Frameworks automate the process of creating, connecting, and managing the lifecycle of dependencies.
- **Compile-Time Validation:** The most significant advantage! Frameworks like Dagger/Hilt verify your dependency graph at compile time. You catch errors before running the app.
- **Reduced Boilerplate:** The amount of code for wiring dependencies is significantly reduced.
- **Standardization:** Using a framework standardizes the DI pattern across your entire project.
- **Simplified Scoping:** Frameworks provide easier tools for managing the lifespans of dependencies.



Esmaeel Moradi

@esmaeel-moradi700

Conclusion:



Building a Strong Foundation for Testable and Maintainable Code.

Dependency Injection is not just a **"fad"** in the development world; it's a **fundamental design principle** that helps you write code that is easier to test, simpler to maintain, and flexible for future changes. Understanding the "why" of DI and how to implement it manually is a crucial first step. But for real-world projects, using a powerful DI framework like Hilt (as Google's recommended tool for Android due to its simplicity and good integration) or Dagger is indispensable. Hilt is an excellent starting point.

This article only covered the basics of DI. In the next article (Day 18), we will delve into more advanced DI concepts, custom scopes, different types of Bindings, Qualifiers, injection strategies in testing, and more, to complete your mastery of DI.



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