

Tema 3.4 - Inyección de Dependencias (ID) con Hilt

Descargar estos apuntes [pdf](#) o [html](#)

Índice

- [Introducción](#)
- ▼ [Hilt](#)
 - [Configuración](#)
 - ▼ [Preparando los componentes de Android para la inyección de dependencias](#)
 - [Cómo insertar objetos ViewModel con Hilt en Compose](#)
 - [Exponer dependencias y preparar módulos con Hilt](#)

Introducción

- Documentación oficial de Android: [Inyección de dependencias](#)

La **inyección de dependencias (DI)** es una técnica muy utilizada en programación y adecuada para el desarrollo de Android. Si sigues los principios de la DI, sentarás las bases para una buena arquitectura de apps.

Implementar la inyección de dependencias te proporciona las siguientes ventajas:

- Reutilización de código
- Facilidad de refactorización.
- Facilidad de prueba o test

Una de las razones de por qué necesitamos inyección de dependencias. Es porque durante el desarrollo de un proyecto, es importante aplicar el **Principio de Responsabilidad Única (SRP)**, uno de los 5 principios **SOLID**. Este principio establece, por explicarlo de forma simple, que nuestras entidades o clases deben hacer una sola cosa. Esto hace que las clases sean más legibles, fáciles de modificar y de probar.

Sin embargo, este principio puede llevar a la **creación de muchos módulos con muchas dependencias entre ellos**. Gestionar estas dependencias puede ser un problema ya que no se pueden ver los colaboradores de las entidades desde fuera y los módulos no pueden ser probados de forma aislada. Para evitar esto, es importante que cada módulo **exponga sus dependencias**. También, está íntimamente relacionado con el **principio de inversión de dependencias (DIP)** que vimos en primero también y que de forma resumida establece los módulos (clases) no deben crear sus dependencias, sino que deben ser provistas desde fuera.

Puesto que los módulos exponen sus dependencias a través de sus constructores y se necesita un componente adicional para proporcionarles instancias de memoria de estas dependencias. Crear una pieza que proporcione todas las dependencias puede ser un trabajo difícil, por lo que existen los inyectores de dependencias para facilitar este proceso.

En el caso de **aplicaciones grandes**, tomar todas las dependencias y conectarlas correctamente puede requerir una gran cantidad de código estándar. En una arquitectura de varias capas, para crear un objeto en una capa superior, debes proporcionar todas las dependencias de las capas que se encuentran debajo de ella. Por ejemplo, para construir un automóvil real, es posible que necesites un motor, una transmisión, un chasis y otras piezas; a su vez, el motor necesita cilindros y bujías.

Cuando no puedes construir dependencias antes de pasarlas (por ejemplo, si usas inicializaciones diferidas o solicitas permisos para objetos en los flujos de tu app), necesitas escribir y conservar un contenedor personalizado (**o un grafo de dependencias**) que administre las dependencias en la memoria desde el principio.


Hay **bibliotecas** que resuelven este problema automatizando el proceso de creación y provisión de dependencias. Se dividen en **dos categorías**:

1. Soluciones basadas en reflexiones que conectan las dependencias durante el **tiempo de ejecución**.
2. Soluciones estáticas que generan el código para conectar las dependencias durante el **tiempo de compilación** (Las más usadas).

Dagger es una biblioteca de inserción de dependencias popular para **Java**, **Kotlin** y Android que mantiene Google. Dagger facilita el uso de la DI en tu app mediante la creación y administración del **grafo de dependencias**. Proporciona dependencias totalmente estáticas y **en tiempo de compilación** que abordan muchos de los problemas de desarrollo y **rendimiento** de las soluciones basadas en reflexiones.

Por tanto, casi todos los frameworks modernos proveen algún tipo de librería de inyección de dependencias. En el ecosistema de Java existen diferentes opciones como: **Dagger2**, **Koin** para Kotlin e incluso **JCDI** en **Jakarta EE**.

En Android, la librería más popular es **Dagger2**. Sin embargo, **Dagger2** es una librería muy compleja y requiere de mucho código boilerplate para configurarla. Por eso, Google ha desarrollado **Hilt**, una librería de inyección de dependencias que se integra con el framework nativo de Android y que **elimina mucho código boilerplate** necesario en **Dagger2**.

 **Resumen:** En resumen, la inyección de dependencias no es más que una forma automática de proveer dependencias a los módulos que las requieran. Para requerirlas tienen que exponerlas por constructor.

Podemos pensar en el inyector de dependencias como un chef en una cocina. Los módulos o clases serían los platos y las dependencias los ingredientes. Cada vez que un plato requiera un ingrediente, el chef irá a la despensa y lo añadirá a la receta.

La gran mayoría de inyectores de dependencias funcionan de la misma forma:

- Los módulos exponen sus dependencias a través de sus constructores.
- Un inyector de dependencias se encarga de proveer instancias de memoria de las dependencias a los módulos que las requieran.
- Existen diferentes librerías de ID en Android de dependencias: **Hilt**, **Koin**, **Dagger2**...

Hilt

Hilt es una librería de inyección de dependencias **desarrollada y recomendada por Google** en incluida en el **framework nativo de Android**.

La principal característica de Hilt es su **facilidad de uso** y la cantidad de código boilerplate que elimina con respecto a la integración de **Dagger2**. Por tanto, el objetivo de Hilt es **hacer todo este proceso muy sencillo e integrado con el framework nativo de Android**.

Configuración

Importante

En el momento de escribir este tutorial, Hilt está haciendo un cambio importante, para usar el procesador de anotaciones **Ksp** en lugar del procesador de anotaciones **Kapt**. Este cambio aún está en fase inicial y no por ello aún no se refleja en la [documentación oficial de Android](#), pero sí en la de la página [Hilt](#).

Por tanto:

- Versión de Hilt \leq **2.44** → Solo **Kapt**
- Versión de Hilt \geq **2.48** → **Ksp** o **Kapt**

Nosotros vamos a usar la versión 2.48

La forma de incluir Hilt en nuestro proyecto de forma actualizada la puedes encontrar a [Aquí](#) .

En el `build.gradle.kts` raíz del proyecto añadiremos el siguiente plugin:

```
plugins {  
    ...  
    // Plugin para que Gradle use Ksp  
    id("com.google.devtools.ksp") version "1.9.0-1.0.12" apply false  
    // Plugin para que Gradle use Hilt con Ksp  
    id("com.google.dagger.hilt.android") version "2.48" apply false  
}
```

En el `build.gradle.kts` del **módulo de la aplicación** (app) añadiremos:

```

plugins {
    ...
    // Plugin para que Gradle use Ksp
    id("com.google.devtools.ksp")
    // Plugin para que Gradle use Hilt
    id ("com.google.dagger.hilt.android")
}

android {
    ...
    // Debemos usar el SDK de la versión 17 o superior
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_17
        targetCompatibility = JavaVersion.VERSION_17
    }
    kotlinOptions {
        jvmTarget = "17"
    }

    // Mínimo el compilador de compose 1.5.0
    composeOptions {
        kotlinCompilerExtensionVersion = "1.5.0"
    }
}

dependencies {
    ...
    implementation("com.google.dagger:hilt-android:2.44")
    implementation("androidx.hilt:hilt-navigation-compose:1.0.0")
    ksp("com.google.dagger:hilt-compiler:2.48")
}

```

Tras ello pulsamos 'Sync Now' para sincronizar los cambios en Gradle.

Preparando los componentes de Android para la inyección de dependencias

En el paquete raíz creamos una clase que extienda de **Application** y le añadimos la anotación **@HiltAndroidApp** para indicar que es la clase principal de la aplicación:

```

// Indicamos que es la clase principal de la aplicación
// Esta clase se ejecutará antes que cualquier otra clase de la aplicación
// y se encargará de proveer las dependencias a los módulos que las requieran
@HiltAndroidApp
class MiApplication : Application() {
    ...
}

```

Recuerda a continuación, en el `AndroidManifest.xml` añadir la clase que acabamos de crear como la clase principal de la aplicación si no la habías hecho ya:

```
<application
    android:name=".MiApplication"
    android:allowBackup="true"
    ...>

    ...

</application>
```

Aunque no es necesario, es recomendable añadir la anotación `@AndroidEntryPoint` a la `MainActivity` para que Hilt pueda inyectar dependencias en ella:

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

Cómo insertar objetos ViewModel con Hilt en Compose

🚩 **Nota:** Recuerda que para usar Hilt en Compose estamos usando la dependencia `implementation "androidx.hilt:hilt-navigation-compose:1.0.0"` en el `build.gradle.kts` del módulo de la aplicación.

- Documentación oficial: [Hilt en Compose](#)

```
// Inyectamos el contexto de la aplicación
@HiltViewModel
class MiViewModel @Inject constructor() : ViewModel() {
    ...
}
```

TODO: AQUIIIIIIII

Todos los `ViewModels` de Hilt los proporciona el `ViewModelComponent` que sigue el mismo ciclo de vida que un `ViewModel` y, por lo tanto, **puede conservar los cambios de configuración**. Para definir el alcance de una dependencia a un `ViewModel`, usa la anotación `@ViewModelScoped`.

Un tipo `@ViewModelScoped` hará que se proporcione una sola instancia del tipo con alcance en todas las dependencias insertadas en `ViewModel`. Otras instancias de un `ViewModel` que soliciten la instancia con alcance recibirán una instancia diferente.

Si se debe compartir una sola instancia entre varios ViewModels, se debe definir el alcance con `@ActivityRetainedScoped` o `@Singleton`.

```
// Posteriormente cuando queramos usar el ViewModel haremos ...
val miViewModel : MiViewModel = hiltViewModel()

// o también
val miViewModel = hiltViewModel<MiViewModel>()

// También podemos obtener un ViewModel con alcance o Scope e una ruta de navegación de Compo
// Veremos más adelante cómo funciona la navegación en Compose y cómo hacer este tipo de inye
// de forma más concreta.
val miViewModel = hiltViewModel<ViewModel>(navBackStackEntry)
```

Exponer dependencias y preparar módulos con Hilt

Realmente ya lo hemos hecho al usar el viewmodel, y es a través de la anotación `@Inject`. Con esta anotación indicamos que queremos inyectar una dependencia en el constructor de una clase. En otras palabras, le estamos diciendo a Hilt que queremos que nos provea una instancia de memoria de una dependencia que necesitamos. Nos será útil para indicar que necesitamos instancias de otros módulos tales como repositorios, servicios, etc.

Pero también deberemos preparar los módulos que queramos inyectar con Hilt. Para ello, debemos añadir la anotación `@Module` a la clase que queramos inyectar. En otras palabras, deberemos anotar aquellas clases que queramos que Hilt provea instancias de memoria de ellas.

Por ejemplo, un colaborador típico o agregación que genera una dependencia en el viewmodel es el repositorio o repositorios de datos a los que accedemos.

```
@HiltViewModel
class MiViewModel @Inject constructor() : ViewModel() {

    // Dependencia que queremos inyectar
    // En este caso creamos una instancia de MiRepositorio cada vez que se crea un MiViewModel
    private val miRepositorio = MiRepositorio()
    ...
}
```

Realizaremos una inversión de dependencias **exponiendo** en el constructor la instancia del objeto agregado o colaboradores que queremos que nos inyecte Hilt.

```

@HiltViewModel
class MiViewModel @Inject constructor(
    private val miRepositorio : MiRepository
) : ViewModel() {
    ...
}

```

Ahora deberemos **preparar el módulo** o la clase **MiRepositorio** para poder ser inyectado. Para ello, definiremos una clase que definirá los métodos proveedores de las instancias de las clases que queremos inyectar.

Esta clase deberá tener la anotación **@Module** y la anotación **@InstallIn** que indicará el alcance o Scope de la clase que se provee. En este caso, la clase se proveerá en el **SingletonComponent** que es el alcance por defecto.

La clase que define los proveedores la podemos llamar **AppModule** y la añadiremos el paquete **di** al paquete raíz de nuestro proyecto ya que centralizará todos proveedores de dependencias de la aplicación.

```

package com.pmdm.miapplication.di

@Module
@InstallIn(SingletonComponent::class)
class AppModule {

    // Crearemos un método que suele empezar con el prefijo 'provide'
    // y que devolverá una instancia de MiRepositorio
    @Provides
    @Singleton
    fun provideMiRepositorio() : MiRepository = MiRepository()

    // Crearemos un módulo para cada clase que queramos inyectar
}

```

Anotaciones:

- **@Provides** : Indica que el método provee una instancia de la clase que se indica en el tipo de retorno del método.
- **@Singleton** : Indica que la instancia de la clase que se provee es única y que se reutilizará en todas las clases que la necesiten.
- **@InstallIn** : Indica el alcance o Scope de la clase que se provee. En este caso, la clase se proveerá en el **SingletonComponent** que es el alcance más alto que existe.

Puede darse el caso de que se produzca una '*cadena*' de dependencias a inyectar. Por ejemplo, puede ser que nuestro repositorio necesite de una DAO (Data Access Object) para acceder a la fuente de datos. Expondremos la inyección de la dependencia en el constructor a Hilt.

```
class MiRepository @Inject constructor(  
    private val dao: ModeloDaoMock  
) {  
    ...  
}
```

definiremos el método proveedor de las instancias de **ModeloDaoMock**

```
@Module  
@InstallIn(SingletonComponent::class)  
class AppModule {  
  
    @Provides  
    @Singleton  
    fun provideModeloDaoMock() : ModeloDaoMock = ModeloDaoMock()  
  
    // El modeloDaoMock es inyectado por provideModeloDaoMock() en el proveedor  
    // y por tanto resuelto automáticamente por Hilt.  
    @Provides  
    @Singleton  
    fun provideMiRepository(  
        modeloDaoMock : ModeloDaoMock  
    ) : MiRepository = MiRepository(modeloDaoMock)  
}
```

Otras anotaciones que podemos usar en los métodos '*proveedores*' son:

- **@Binds** : Indica que la clase que se provee es una interfaz y que se debe enlazar con una clase concreta que implemente dicha interfaz. Se usa en lugar de **@Provides** .
- **@IntoMap** : Indica que la clase que se provee es un mapa y que se debe añadir a un mapa concreto. Se usa en lugar de **@Provides** .
- **@IntoSet** : Indica que la clase que se provee es un conjunto y que se debe añadir a un conjunto concreto. Se usa en lugar de **@Provides** .

<https://developer.android.com/codelabs/android-hilt?hl=es-419#6>