

Tema 3.4 - Inyección Dependencias (DI)

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

Índice

- [Introducción](#)
- ▼ [Hilt](#)
 - [Procesador de anotaciones KSP](#)
 - [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](#)
 - [ViewModels con Hilt](#)
 - [Preparando Módulos](#)
 - [Inyectando el contexto de la aplicación](#)
 - [Resumen configuración Hilt](#)

Introducción

Enlaces de interés:

- Documentación oficial de Android: [Inyección de dependencias](#)
- Vídeo tutorial Divulgativo (Castellano): [DevExperto](#)
- Codelab Google: [Inyección de dependencias](#)
- Lista de reproducción: [Andorid Developers](#)
- Vídeo tutorial (Inglés): [Philipp Lackner](#)

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 como **Hilt**, **Koin**, Kotlin-inject, 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**.


Como **única pega** podemos decir que, al estar construida sobre la librería de Java **Dagger2**, **Hilt** funciona solo con *Kotlin/JVM* y por tanto **módulos comunes de proyectos de KMP** (Kotlin Multiplatform) donde se use *Kotlin/native* o *Kotlin/wasm* en ese caso la alternativa sería **Koin**.

Procesador de anotaciones KSP

Puesto que **Hilt utiliza anotaciones**, para poder usarlo en un proyecto de Android, necesitamos añadir el **procesador de anotaciones KSP** (Kotlin Symbol Processing) a nuestro proyecto. **KSP** es un procesador de anotaciones de Kotlin creado por **Google** que reemplaza a '*antiguo*' **KAPT** (Kotlin Annotation Processing Tool) usado hasta la versión 2.48 de Hilt ya que **KSP** mejora mucho los tiempos de compilación.

- Puedes encontrar más información sobre por qué usar **Ksp** [Aquí](#) 

Configuración

- La forma de incluir Hilt en nuestro proyecto de forma actualizada la puedes encontrar a [Aquí](#)  o en la [documentación oficial de Android en Inglés](#).

En el `libs.versions.toml` añadiremos las versiones de las librerías y plugins que vamos a usar:

```

[versions]
kotlin = "2.0.20"
...
// Fíjate que la versión de KSP debe coincidir con la de Kotlin 2.0.20
ksp = "2.0.20-1.0.25"
hilt = "2.52"
hiltNavigationCompose = "1.2.0"

[libraries]
dagger-hilt-android
= { group = "com.google.dagger", name = "hilt-android", version.ref = "hilt" }
dagger-hilt-android-compiler
= { group = "com.google.dagger", name = "hilt-android-compiler", version.ref = "hilt" }
androidx-hilt-navigation-compose
= { group = "androidx.hilt", name = "hilt-navigation-compose", version.ref = "hiltNavigationCompose" }

[plugins]
devtools-ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
com-google-dagger = { id = "com.google.dagger.hilt.android", version.ref = "hilt" }

```

En el `build.gradle.kts` raíz del proyecto añadiremos los plugins de **KSP** y **Hilt** definidos en el `libs.versions.toml` :

```

plugins {
    ...
    alias(libs.plugins.devtools.ksp) apply false
    alias(libs.plugins.com.google.dagger) apply false
}

```

En el `build.gradle.kts` del **módulo de la aplicación** (app) definiremos los plugins y las dependencias de **Hilt**:

```

plugins {
    ...
    alias(libs.plugins.devtools.ksp)
    alias(libs.plugins.com.google.dagger)
}

dependencies {
    ...
    implementation(libs.dagger.hilt.android)
    implementation(libs.androidx.hilt.navigation.compose)
    ksp(libs.dagger.hilt.android.compiler)
    kspAndroidTest(libs.dagger.hilt.android.compiler)
}

```

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

Preparando los componentes de Android para la inyección de dependencias

- Puedes descargar una chuleta con todas las anotaciones que vamos a ver [en este enlace](#).

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:

- Documentación oficial: [HiltAndroidApp](#)

```
// 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>
```

Es recomendable añadir la anotación `@AndroidEntryPoint` a la MainActivity para que Hilt pueda inyectar dependencias en ella:

- Documentación oficial: [AndroidEntryPoint](#)

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

Cómo insertar objetos ViewModel con Hilt en Compose



Nota

Fíjate que hemos añadido la dependencia `androidx.hilt:hilt-navigation-compose` en el `build.gradle.kts` del módulo de la aplicación. Ya que tal y como nos indicaba el [cheatsheet](#) del ViewModel que vimos en el tema anterior. Esta dependencia nos permitirá inyectar `ViewModels` en las funciones `@Composable`.

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

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

Para acceder al `ViewModel` dentro de una `Activity` seguiremos usando el delegado `by viewModels()`

Para usar un `ViewModel` con Hilt dentro de una función `@Composable` usaremos ahora `hiltViewModel()` ...

```
@Composable
fun MiScreen(miVm: MiViewModel = hiltViewModel()) { ... }
```

También es posible usarlo en la navegación como veremos más adelante con ...

```
val miVm = hiltViewModel<MiViewModel>(navBackStackEntry)
```

de esta manera el `ViewModelStoreOwner` será una determinada ruta de navegación y no la `Activity`.

Exponer dependencias y preparar módulos con Hilt

- [Entry Points](#)

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.

ViewModels con Hilt

- Documentación oficial: [Hilt ViewModels](#)

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
    // Creamos una instancia de MiRepositorio cada vez que se crea un MiViewModel
    private val miRepositorio = MiRepositorio() // 🐛

    ...
}
```

Realizaremos una **inversión de dependencia** (Principio SOLID) por la cual **exponemos** en el constructor la instancia del **objeto colaborador** que queremos que nos inyecte Hilt.

```
@HiltViewModel
class MiViewModel @Inject constructor(
    private val miRepositorio : MiRepository // Sigue siendo una propiedad privada
) : ViewModel() {

    ...
}
```

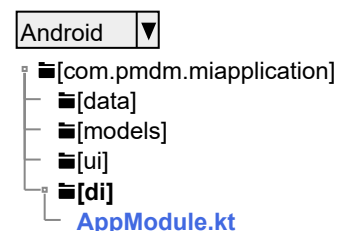
Preparando Modulos

- Documentación oficial: [Modules](#)

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. A esta clase le denominaremos **Componente de Hilt**

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.

Para seguir los convenios de nomenclatura propuesta en el curso al ver la arquitectura de una app de Android. La clase que define los proveedores la podemos llamar `AppModule` y la añadiremos el paquete `di` (*dependency injection*) 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. **Cuidado no siempre una instancia única es una buena opción.**
- **@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.

1. Expondremos la inyección de la dependencia en el **constructor del repositorio** a Hilt y de los objetos que se inyecten.

```

class MiRepository @Inject constructor(
    private val dao: ModeloDaoMock // Como se inyecta también tendremos que exponerlo
) { ... }

class ModeloDaoMock @Inject constructor(){ ... }

```

2. Definiremos el método proveedor de las instancias de `ModeloDaoMock` hemos marcado como `@Inject` en `MiRepository`

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

    @Provides
    @Singleton
    fun provideModeloDaoMock() : ModeloDaoMock = ModeloDaoMock()
}
```

3. Por último, `modeloDaoMock` es inyectado por `provideModeloDaoMock()` en el proveedor de `MiRepository` y por tanto resuelto automáticamente por Hilt.

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

    @Provides
    @Singleton
    fun provideModeloDaoMock() : ModeloDaoMock = ModeloDaoMock()

    @Provides
    @Singleton
    fun provideMiRepositorio(
        modeloDaoMock : ModeloDaoMock // Es inyectado por provideModeloDaoMock()
    ) : MiRepository = MiRepository(modeloDaoMock)
}
```

Importante

Aunque es recomendable definir los proveedores de dependencias. En aquellos módulos cuyo '**proveedor**' simplemente se instancie un constructor por defecto o que reciba objetos ya marcados con `@Inject` no es necesario definir un proveedor. Hilt lo hará automáticamente.

Por ejemplo, en el caso de `ModeloDaoMock` no sería necesario definir el proveedor `fun provideModeloDaoMock() : ModeloDaoMock = ModeloDaoMock()` ya que no tiene dependencias y se instanciará automáticamente. Pero si quisiéramos que se instanciara una única vez, podríamos aplicarle la anotación `@Singleton` en la definición.

```
@Singleton
class ModeloDaoMock @Inject constructor(){ ... }
```

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** .

Inyectando el contexto de la aplicación

Aquellos objetos que no se puedan inyectar por Hilt, como por ejemplo **SharedPreferences** , **Resources** , etc. se pueden inyectar en el **AppModule** usando la **inyección del contexto** de la aplicación con las anotaciones **@ApplicationContext** o **@ActivityContext** :

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

    @Provides
    @Singleton
    fun provideSharedPreferences(@ApplicationContext context: Context) : SharedPreferences {
        return context.getSharedPreferences("miApp", Context.MODE_PRIVATE)
    }
}
```

Resumen configuración Hilt

- [CheatSheet Anotaciones](#)

Resumen

1. Definir los **Entry Points** de la aplicación:
 - `@HiltAndroidApp` en la clase `Application`.
 - `@AndroidEntryPoint` en la clase `MainActivity`.
2. Exponer las dependencias en los constructores de las clases que queramos inyectar con `@Inject` de forma 'recursiva'.
3. Preparar aquellos módulos que hemos expuesto con la anotación `@Inject`. Por tanto, a la clase `AppModule` dentro del paquete `di`.
 - La anotaremos con:
 - `@Module`
 - `@InstallIn(SingletonComponent::class)`
 - `@Provides` en los métodos proveedores de las instancias de las clases que queramos inyectar que empiezan por `fun provide...`.