Contents

[H.1 - Dependency Injection e Hilt 1](#_Toc136253685)

[1. DI in Android 1](#_Toc136253686)

[2. Hilt Project Setup 2](#_Toc136253687)

[build.gradle:project 2](#_Toc136253688)

[build.gradle:app 2](#_Toc136253689)

[Application override 2](#_Toc136253690)

[3. Entry Point 3](#_Toc136253691)

[4. Inject: constructor 4](#_Toc136253692)

[5. Inject: Module 4](#_Toc136253693)

[Gerarchia dei Componenti 7](#_Toc136253694)

[6. Instrumented Test Setup 8](#_Toc136253695)

[build.gradle:app 8](#_Toc136253696)

[TestManifest e TestActivity 8](#_Toc136253697)

[TestRunner e Extensions 9](#_Toc136253698)

[7. Instrumented Test 11](#_Toc136253699)

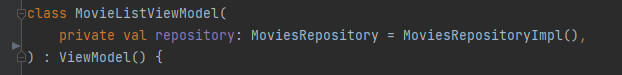
[Test Setup 11](#_Toc136253700)

[Ricreare Fragment dependencies 12](#_Toc136253701)

# H.1 - Dependency Injection e Hilt

## DI in Android

Diamo un’occhiata al costruttore di MovieListViewModel



Abbiamo fornito al viewModel tutte le sue dipendenze qui, non creando al suo interno istanze di altre classi. Questo pattern è definito **Dependency Injection**

Dependency Injection è un pattern di progettazione in cui un oggetto o una funzione riceve gli altri oggetti o funzioni da cui dipende

È una pratica fondamentale per ottenere un’app manutenibile e testabile.

Man mano che l’app cresce pero’, questo richiede un gran quantità di *boilerplate code* per instanziare una classe che, magari, fornisce solo poche funzionalità.

Vediamo le dipendenze del viewModel:

MovieListViewModel > MoviesRepository > MovieNetworkApi > Retrofit

Se dovessimo creare un’altra istanza del viewModel per riutilizzarlo in un'altra parte dell'app, dovremmo ricrearne anche tutte le dipendenze, magari cambiando l’implementazione di MoviesRepository, generando altro boilerplate. Questo codice cresce esponenzialmente con la crescita della nostra app.

Scrivere molto codice di setup non solo non è efficiente, ma è anche prono a errori umani.

Vediamo ora come utilizzare **HILT**, un framework di Dependency Injection che ci permetterà di automatizzare alcune di queste procedure.

Hilt richiede un importante setup, non solo delle dipendenze, ma anche nel dichiarare le injections. Teniamo sempre a mente pero’ che nel lungo termine, questo ci permetterà di risparmiare tempo ed effettuare test altrimenti impossibili.

## Hilt Project Setup

<https://developer.android.com/training/dependency-injection/hilt-android#setup>

E’ consigliabile seguire sempre il setup presente nella documentazione quando si effettua in un nuovo progetto, per evitare problemi derivanti dalle ultime modifiche al framework.

### build.gradle:project

plugins {  
  ...  
  id 'com.google.dagger.hilt.android' version "2.44" apply false  
}

### build.gradle:app

...  
plugins {

...  
  id 'kotlin-kapt'  
  id 'com.google.dagger.hilt.android'  
}  
  
android {  
  ...

compileOptions {

sourceCompatibility JavaVersion.VERSION\_17

targetCompatibility JavaVersion.VERSION\_17

}

kotlinOptions {

jvmTarget = '17'

}  
}  
  
dependencies {  
  ...

// Hilt  
  implementation "com.google.dagger:hilt-android:2.44"  
  kapt "com.google.dagger:hilt-compiler:2.44"

// // Inject of viewModel scoped to navGraph with: by hiltNavGraphViewModels

implementation 'androidx.hilt:hilt-navigation-fragment:1.0.0'  
}  
  
// Allow references to generated code  
kapt {  
  correctErrorTypes true  
}

È consigliato estrarre la versione (2.44 nell’esempio) in una variabile, cosi’ da coordinare gli import.

### Application override

**@HiltAndroidApp**  
class ExampleApplication : Application() { ... }

## Entry Point

Dopo aver impostato l’Application, dobbiamo dire a Hilt dove intervenire nel framework Android.

In questo caso è il Fragment, che userà il ViewModel, annotiamolo con @AndroidEntryPoint



Facendo cosi’ stiamo creando un **Container** con scopo Fragment, e preparando il fragment a ricevere le Injection.

Avendo definito MovieDetailFragment come entry point, dobbiamo annotare anche *tutte le classi che dipendono da lui*, in questo caso, l’activity:

A screen shot of a computer

Description automatically generated with low confidence

A screenshot of a computer

Description automatically generated with medium confidence

## Inject: constructor

Cominciamo dal basso a risolvere le dipendenze: RetrofitClient

A screen shot of a computer

Description automatically generated with low confidence

Aggiungendo @Inject al constructor() (ora dichiarato verbosamente) questa classe:

* Potrà ricevere Injection per le sue dipendenze
* Potrà essere iniettata come dipendenza

RetrofitClient non ha dipendenze, quindi possiamo lasciarlo vuoto

( In realtà è dipendente da Gson(), una classe al di fuori del nostro controllo, ma per ora ignoralo )

## Inject: Module

Il nostro Repo ha bisogno di uno specifico valore esposto da RetrofitClient:

A picture containing text, font, screenshot

Description automatically generated

Rimuovi il default dal costruttore - forniremo questa dipendenza tramite iniezione:

A screen shot of a computer program

Description automatically generated with low confidence

Sorge quindi un problema: MovieNetworkAPI è una Interfaccia, che non puo’ essere instanziata ma solo implementata, non possiamo dunque usare @Inject constructor()

Andiamo quindi a vedere come dichiarare un **Modulo** dove definiremo quale implementazione di MovieNetworkAPI verrà ricevuta da MoviesRepositoryImpl.

Crea un nuovo Object chiamato DataModule dentro un nuovo package DI

A screenshot of a computer

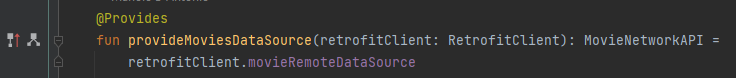
Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with low confidence

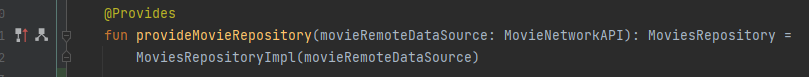
* @Module lo definisce come Hilt Module
* @InstallIn definisce lo scope, ovvero il lifecycle di questo modulo. In questo caso, scegliamo il viewModel, che è dove sarà consumato il Repository – il nostro obbiettivo.

Definiamo l’inject di MovieNetworkApi necessario al nostro repository:

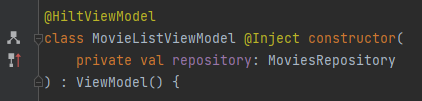


* @Provides definisce a Hilt una funzione che fornisce un’implementazione
*  indica che questo costruttore ottiene la sua dipendenza tramite iniezione
*  indica che questa funzione inietta il suo risultato a qualcuno che ne è dipendente

Crea un’altra funzione che fa la stessa cosa per il repository, dato che anche lui estende una interfaccia:



Infine, modifica MovieListViewModel per ricevere l’iniezione

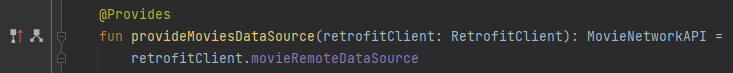


* @HiltViewModel crea un **hilt** **container** che è **scoped** al viewModel

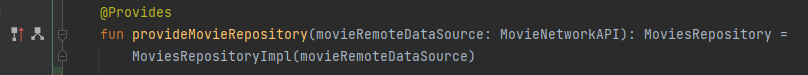
Ricapitolando:



Fornisce a..



Fornisce a..



Fornisce a…

A screen shot of a computer

Description automatically generated with low confidence

Cosa abbiamo ottenuto con questa procedura:

Ogni elemento di questa catena puo’ essere riutilizzato e iniettato in altre implementazioni.

Ogni elemento puo’ essere testato iniettando facilmente *test doubles*

Ora ogni volta che richiameremo MovieListViewModel tramite *byViewModels* o simili, le sue dipendenze verranno create e legate al suo lifecycle, quindi distrutte se esso viene distrutto.

Questo avviene perché il modulo ha lo scope @InstallIn(ViewModelComponent::class), che è la classe dei **container creati dall’annotazione @HiltViewModel**. Il modulo sarà quindi installato in ogni container @HiltViewModel, e disponibile ad ogni viewModel che ha questa annotazione.

### Gerarchia dei Componenti

Quando un modulo o altri elementi vengono installati in un componente, questi saranno disponibili non solo ad esso, ma anche a tutti i suoi discendenti:

A picture containing text, screenshot, line, font

Description automatically generated

## Instrumented Test Setup

Per effettuare Unit Test con Hilt di solito non è necessaria alcuna modifica.

Per gli instrumented test dobbiamo invece collegare hilt al test runner e creare una activity di test.

### build.gradle:app

// Hilt For instrumented tests.  
androidTestImplementation "com.google.dagger:hilt-android-testing:$hilt\_version"  
// ...with Kotlin.  
kaptAndroidTest "com.google.dagger:hilt-android-compiler:$hilt\_version"

### TestManifest e TestActivity

Usando *New > Directory*, crea una cartella **debug** dentro /app/src

Dentro questa cartella:

* Crea una sottocartella /java
* crea un nuovo AndroidManifest.xml

A screen shot of a computer program

Description automatically generated with low confidence

Contenuto Android Manifest.xml :  
  
<?xml version="1.0" encoding="utf-8"?>  
  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="com.example.moviemaster">   
  
 <application>  
 <activity  
 android:name=".HiltTestActivity"  
 android:exported="false" />  
 </application>  
  
</manifest>

Fai attenzione che il campo *package* sia compilato correttamente per il tuo progetto.

Dentro la cartella *java*, crea un nuovo package con *new > package*. Dagli lo stesso nome del tuo package dentro *main*

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Dentro moviemaster [debug], crea questa classe:

import androidx.appcompat.app.AppCompatActivity  
import dagger.hilt.android.AndroidEntryPoint  
  
@AndroidEntryPoint  
class HiltTestActivity : AppCompatActivity()

A screenshot of a computer

Description automatically generated

### TestRunner e Extensions

Aggiungi un package *di* dentro androidTest



Crea questa classe dentro il package *di*

import android.app.Application  
import android.content.Context  
import androidx.test.runner.AndroidJUnitRunner  
import dagger.hilt.android.testing.HiltTestApplication  
  
class HiltTestRunner : AndroidJUnitRunner() {  
 override fun newApplication(cl: ClassLoader?, name: String?, context: Context?): Application {  
 return super.newApplication(cl, HiltTestApplication::class.*java*.*name*, context)  
 }  
}

*build.gradle:app*, dentro *defaultConfig*, sostituisci il test runner con quello che hai appena creato

testInstrumentationRunner "com.example.moviemaster.di.HiltTestRunner"

Crea questa classe *HiltExt* dentro il package *di*

import android.content.ComponentName  
import android.content.Intent  
import android.os.Bundle  
import androidx.annotation.StyleRes  
import androidx.core.util.Preconditions  
import androidx.fragment.app.Fragment  
import androidx.navigation.NavHostController  
import androidx.navigation.Navigation  
import androidx.test.core.app.ActivityScenario  
import androidx.test.core.app.ApplicationProvider  
import com.example.moviemaster.HiltTestActivity

import androidx.fragment.testing.manifest.R  
  
*/\*\*  
 \* launchFragmentInContainer from the androidx.fragment:fragment-testing library  
 \* is NOT possible to use right now as it uses a hardcoded Activity under the hood  
 \* (i.e. [EmptyFragmentActivity]) which is not annotated with @AndroidEntryPoint.  
 \*  
 \* As a workaround, use this function that is equivalent. It requires you to add  
 \* [HiltTestActivity] in the debug folder and include it in the debug AndroidManifest.xml file  
 \* as can be found in this project.  
 \*/*inline fun <reified T : Fragment> launchFragmentInHiltContainer(  
 fragmentArgs: Bundle? = null,  
 @StyleRes themeResId: Int = R.style.*FragmentScenarioEmptyFragmentActivityTheme*,  
 navHostController: NavHostController? = null,  
 crossinline action: Fragment.() -> Unit = **{}**): ActivityScenario<HiltTestActivity> {  
 val startActivityIntent = Intent.makeMainActivity(  
 ComponentName(  
 ApplicationProvider.getApplicationContext(),  
 HiltTestActivity::class.*java* )  
 ).putExtra(  
 "androidx.fragment.app.testing.FragmentScenario.EmptyFragmentActivity.THEME\_EXTRAS\_BUNDLE\_KEY",  
 themeResId  
 )  
  
 return ActivityScenario.launch<HiltTestActivity>(startActivityIntent).*apply* **{** onActivity **{** activity **->** val fragment: Fragment = activity.*supportFragmentManager*.*fragmentFactory*.instantiate(  
 Preconditions.checkNotNull(T::class.*java*.*classLoader*),  
 T::class.*java*.*name* )  
 fragment.*arguments* = fragmentArgs  
 fragment.*viewLifecycleOwnerLiveData*.observeForever **{** viewLifecycleOwner **->** if (viewLifecycleOwner != null) {  
 navHostController?.*let* **{** Navigation.setViewNavController(fragment.requireView(), **it**)  
 **}** }  
 **}** activity.*supportFragmentManager* .beginTransaction()  
 .add(android.R.id.*content*, fragment, "")  
 .commitNow()  
  
 fragment.action()  
 **}  
 }**}

Ecco come appare l’albero ora

A screenshot of a computer

Description automatically generated with medium confidence

## Instrumented Test

Testiamo MovieDetailFragment:

### Test Setup

Creiamo la classe di test:

A screen shot of a computer program

Description automatically generated with medium confidence

* @HiltAndroidTest permette a Hilt di Iniettare dipendenze
* hiltRule fa si che tutto sia iniettato prima di cominciare i test
* Entrambi sono necessari per testare con Hilt.

Creiamo lo scenario:

A screen shot of a computer program

Description automatically generated with low confidence

* Utilizziamo la extension per creare uno scenario che abbia come base l’activity che abbiamo creato dentro *debug*. E’ necessario perché, come abbiamo visto, Hilt necessita per ogni fragment annotato con @AndroidEntryPoint, che ci sia tale annotazione anche nell’activity che lo contiene.

### Ricreare Fragment dependencies

MovieDetail fragment ha le seguenti dipendenze

* E’ dipendente da MovieListViewModel
* Richiede un **id** al momento della creazione

MovieListViewModel

Ora cominciamo a sfuttare Hilt per diminuire il carico di boilerplate.

Copia FakeMoviesRepository che abbiamo creato per gli UnitTest, dentro androidTest\data

A screen shot of a computer

Description automatically generated with low confidence

Dentro la cartella DI, crea un nuovo modulo Hilt, *FakeDataModule*

A screen shot of a computer program

Description automatically generated with low confidence

FakeDataModule *replaces* DataModule nei tests.

Ora MovieViewModel, che dipende da MoviesRepository, sarà iniettato in tutti i test senza bisogno di dichiararlo.

Movie ID

MovieDetailFragment richiede un ID al momento della creazione, che nell’app è passato da MovieList tramite gli *args.* Dobbiamo fornire questo *bundle*, e lo faremo in maniera molto simile a come è nell’app*.*

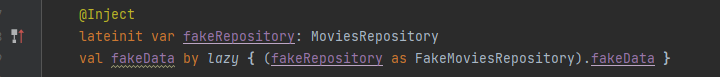
Ottieni un riferimento al Repo:

A picture containing text, font, screenshot, graphics

Description automatically generated

* Otteniamo questo repo tramite provideRepository che abbiamo scritto nel fakeDataModule

Bisogna castare MoviesRepository a FakeMoviesRepository se vogliamo ottenere i suoi membri interni, come fakeData. Scriviamo una val che ci permette di evitare il cast ad ogni invocazione:



Ora usa fakeData :

A screen shot of a computer program

Description automatically generated with medium confidence

* MovieDetailFragmentArgs è lo stesso utilizzato nella navigazione dell’app. toBundle genera il bundle che possiamo passare al fragment.
* Aggiungiamo anche lo stile dell’app, cosi’ il fragment sara renderizzato come nell'app.

Aggiungi un semplice test

A picture containing screenshot, text, font

Description automatically generated