DI

classmate
Date 03/06/2024
Page

(I) MyApi.

```
interface MyApi {

    @GET("test")
    suspend fun doNetworkCall()

}.
```

(I) MyRepository.

```
interface MyRepository {

    suspend fun doNetworkCall()
}.
```

* We need to pass this MyApi into our repository as that is where our dependency injection starts.

The way we doin it is using the constructor.

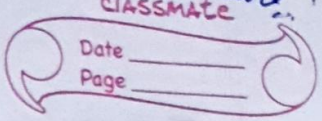
main

domain

repository

(I) MyRepository

→ Clean architecture.



main

data

repository

B MyRepositoryImpl.kt

→ Package.

~~RepositoryImpl~~ MyRepositoryImpl.kt

```
class MyRepositoryImpl : MyRepository {
    override suspend fun doNetworkCall() {

    }
}
```

we want to make our API call. (which comes from our MyAPI interface).

So, how do we now get this MyAPI into our repository ??

↳ Here Dependency Injection starts.

⇒ We need to pass this MyApi instance variable to the object which is going the repository implementation (here).

B MyRepositoryImpl.kt,

class MyRepositoryImpl(

private val api: MyApi

): MyRepository {

}

Q So how does dagger hilt actually know that we want to use it here and actually want to kind of delegate passing this api to this repository to dagger Hilt?

→ We now need to define.
We do that in so-called modules

Multiple modules in our app
↓
kind of containers for specific type of dependencies.

App Module: dependencies in that live as long as the application does., So, they are

effectively singletons.

Ex of of modules :-

- Broadcast module
- Auth module.

Modules with clear responsibilities or Modules for specific type of scopes.

main

di

@AppModule

Object

{ set of dependencies that we only need in a fragment, in an activity or a service, then we need to put these in the same module. }

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

}

In Dagger Hilt we have different so called components. Since the dependencies that we want to provide here, those are all singletons, like our API interface, our repository (repository lives as long as our App lives).

Because of that we want to install that into the singleton component :: class.

* This component decides how long will the dependencies that we provide in this module will actually live.

SingletonComponent → App.
ActivityComponent → Activity (they are actually inject into lives)

ViewModelComponent → ViewModel
ActivityRetainedComponent → Rotation (won't destroy).
(every activity is recreated)
ServiceComponent → Service

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

    @Provides
    @Singleton
    fun provideMyApi(): MyApi {

        return Retrofit.Builder()
            .baseUrl("https://test.com")
            .build()
            .create(MyApi::class.java)

    }
}
```

Now, from this point onwards, Dagger Hilt knows how to create this type of class: whenever, we kind of request an instance of MyApi. (like we do in MyRepository MyRepositoryImpl) then dagger hilt will look in its modules; if it can find such an instance & if it it will take that and pass it to the constructor behind the scene.

@Singleton → Scope → how many of these dependencies we actually have per component.
→ Single instance throughout the whole lifetime of our application.

> Take our repository and inject that into our ViewModel.

(B) MyViewModel

```
@HiltViewModel
class MyViewModel @Inject constructor(
    private val repository : MyRepository
) : ViewModel() {

}
```

$ Injecting dependencies into ~~them~~ ViewModels is a little bit tricky because as we might know, creating viewmodel needs a factory.

In Dagger Hilt its quite easy as compared to Dagger 2.

* @HiltViewModel →
  @Inject constructor → { Hey, Dagger, please inject all the dependencies that we have in our constructor & take a look into your modules and see if you can find these.

@object @AppModule

@Provides
@Singleton

fun provideMyRepository(): MyRepository {
    return MyRepositoryImpl()
}

We need an instance of MyAPI here created previously by Dagger Hilt in the App Module.

So,

fun provideMyRepository(api: MyApi): Repository {
    return MyRepositoryImpl(api)
}
}

⇓

Now UI layer

setContent {
    DaggerHiltCourseTheme {
        val viewModel = hiltViewModel<MyViewModel>()

to ⇒ scoped to current navigation graph.

classmate
Date
Page

@AndroidEntryPoint
↳ We inject dependency (ViewModel) in an android component class.

Android component class :-

→ Activity.
→ A fragment
→ A ~~service~~ service.

(B) MyApp.

@HiltAndroidApp
class MyApp : Application().
<-a

<application
android:name = ".MyApp" >

>.

{Application context.}

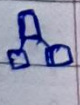
classmate
Date
Page

What happens when we have 2 dependencies of the same type? How does Dagger Hilt know which one it should inject?

Sol:- By using @Named Annotation:-

```
fun provideMyRepository(
    api: MyApi,
    app: Application,
    @Named("hello1") hello1: String
): MyRepository {

    return MyRepositoryImpl(api, app).
}
```



```
@Provides
@Singleton
@Named("hello1")
fun provideString1() = "Hello1"

@Provides
@Singleton
@Named("hello2")
fun provideString2() = "Hello2"
```

Improvement in MyRepositoryImpl injecting the provideMyRepository

> We want to inject implementation of this repository and right now we are just defining which implementation we actually want to use into the function.

```
fun provideMyRepository() : MyRepository {
    return MyRepositoryImpl(api, app)
}
```

⇒ There exists an easier way for providing such abstraction.
So if we either have an interface or an abstract class that we want to inject,

↓

Get rid of old fun (provide...)
Create new module in (di package)

↓

(E) Repository Module.

Using abstract classes for injection :-

@Module
@InstallIn(SingletonComponent::class)
abstract class RepositoryModule {

@Binds
@Singleton
abstract fun bindMyRepository(
myRepositoryImpl: MyRepositoryImpl
): MyRepository

}

Now it won't run fine. We have to do:

class MyRepositoryImpl @Inject constructor(
private val api: MyApi,