**Dependency Injection/DI(Dagger2/Hilt/ButterKnif/Koin)**

**What is Dependency Injection?**

According to Dependency Injection principle, the class should not construct it's dependencies, instead it should get the dependencies from outside. We can do it with the help of a Constructor Parameter or a Setter Function.

As per DI- Di will help us to create & manage (create/re-create/re-uses/remove) the dependency object which we want in our program.

**Constructor/Field Injection:** If we are doing it with Constructor Parameter it is called "Constructor Injection". If we are doing it with Setter Function it is called, setter injection or field injection.

Classes often require references to other classes. For example, a Car class might need a reference to an Engine class. These required classes are called dependencies, and in this example the Car class is dependent on having an instance of the Engine class to run.

**Why Dependency Injection?**

* **Scalability**: Increase or add new feature in app
* **Maintainability**, as DI helps code to be loosely coupled maintaining the codes become easier.
* **Testability**, as the code is now loosely coupled, writing tests are easy.
* **Readability**, as DI helps to follow "Single Responsibility Principle

Dependency injection provides following benefit

* **Reusability of classes and decoupling of dependencies:** It's easier to swap out implementations of a dependency. Code reuse is improved because of inversion of control, and classes no longer control how their dependencies are created, but instead work with any configuration.
* **Ease of refactoring:** The dependencies become a verifiable part of the API surface, so they can be checked at object-creation time or at compile time rather than being hidden as implementation details.
* **Ease of testing:** A class doesn't manage its dependencies, so when you're testing it, you can pass in different implementations to test all of your different cases.

**Dagger 2**

Dagger is a fully static, compile-time dependency injection framework for both Java and Android. It is an adaptation of an earlier version created by Square and now maintained by Google.

Dagger automatically generates code that mimics the code you would otherwise have hand-written. Because the code is generated at compile time, it's traceable and more performant than other reflection-based solutions such as [Guice](https://en.wikipedia.org/wiki/Google_Guice).

Following are the basic annotations used in Dagger 2:

1. **@Module(**Provider**)** : This is used on the class that does the work of constructing objects that’ll be eventually provided as dependencies.  classes annotated with @Module is responsible for providing an object which can be injected.
2. **@Provides** : This is used on the methods inside the Module class that’ll return the object.
3. **@Inject(**Consumer**)** : This is used upon a constructor, field or a method and indicates that dependency has been requested. The @Inject annotation is used to defining a dependency
4. **@Component(**Connector**)** : The Module class doesn’t provide the dependency directly to the class that’s requesting it. For this, a Component interface is used that acts as a bridge between @Module and @Inject. A @Component annotated interface defines the connection between the provider and consumer.
5. **@Singleton** : This indicates that only a single instance of the dependency object would be created.

**HILT:**

[Hilt](https://developer.android.com/training/dependency-injection/hilt-android) is Jetpack's recommended library for dependency injection in Android. Hilt defines a standard way to do DI in your application by providing containers for every Android class in your project and managing their lifecycles automatically for you.

Hilt is built on top of the popular DI library [Dagger](https://developer.android.com/training/dependency-injection/dagger-basics) to benefit from the compile time correctness, runtime performance, scalability, and Android Studio support that Dagger provides.

Hilt is a dependency injection library for Android that reduces the boilerplate of doing manual dependency injection in your project.

**@HiltAndroidApp:**First, Enable Hilt in your app by annotating your **application class** with the **@HiltAndroidApp** to trigger Hilt’s code generation which will have the base class for our application and it acts as the application-level dependency container.

**@HiltAndroidApp**

class HiltApplication : Application() {

override fun onCreate() {

super.onCreate()

}

}

**Note:** Add/Register this Application class in Manifest file.

**@Singleton:** @Singleton annotation so that whenever we inject the dependency we inject the same single instance of Class/CryptocurrencyRepository were ever requested.

**@Module:** A Hilt module is a class that is annotated with ***@Module***. Like a Dagger module, it informs Hilt on how to provide instances of certain types. Unlike Dagger modules, you must annotate Hilt modules with ***@InstallIn*** to tell Hilt which Android class each module will be used or installed in.

Module class will inject dependency to other classes so, we need to annotate this class with **@Module** annotation which will make this class a module to inject dependency to other classes within its scope. Furthermore, we will also add one more annotation to it. i.e., **@InstallIn(XComponent::class)** this will make this class inject dependencies across the entire application.

**@Provide:** If you don’t directly own the class, you can tell Hilt how to provide instances of this type by creating a function inside a Hilt module and annotating that function with *@Provides*.

**@Module**

**@InstallIn(SingletonComponent::class)**

object NetworkModule {

**@Provides**

fun provideRetrofit(okHttpClient: OkHttpClient): Retrofit {

return Retrofit.Builder()

.baseUrl("https://howtodoandroid.com/")

.addConverterFactory(GsonConverterFactory.create())

.client(okHttpClient)

.build()

}

**@Provides**

fun provideApiClient(retrofit: Retrofit): ApiService {

return retrofit.create(ApiService::class.java)

}

**@Module**

**@InstallIn(SingletonComponent::class)**

class AppModule {

**@Provides**

**@Singleton**

fun provideCryptocurrencyRepository():CryptocurrencyRepository=CryptocurrencyRepositoryImpl()

}

**Note:**

1. **@Module** annotation which will make this class a module to inject dependency to other class within it's scope.

2. **@InstallIn(SingletonComponent::class)** this will make this class to inject dependencies across the entire application.

3. we have used **@Provides with @Singleton** annotation so that whenever we inject the dependency we inject the same single instance of Class/CryptocurrencyRepository were ever requested.

**@Binds:** Inject interface instances with @Binds => **I**f you have an interface, then you cannot constructor-inject it. Instead, provide Hilt with the binding information by creating an abstract function annotated with *@Binds* inside a Hilt module.

**@Module**

**@InstallIn(ViewModelComponent::class)**

interface RepositoriesModule {

**@Binds**

fun mainRepository(mainRepositoryImpl: MainRepositoryImpl) : MainRepository

}

**@InstallIn(Component::class):**

**like @InstallIn(SingletonComponent::class): , @InstallIn(ActivityComponent::class):, @InstallIn(FragmentComponent::class): ,**

For each Android class in which you can perform field injection, there’s an associated Hilt component that you can refer to in the *@InstallIn* annotation. Each Hilt component is responsible for injecting its bindings into the corresponding Android class.

**@HiltViewModel :** A Hilt View Model is a Jetpack ViewModel that is a constructor injected by Hilt. To enable injection of a ViewModel by Hilt use the*@HiltViewModel* annotation:

**@AndroidEntryPoint:** Hilt will only provide dependencies to the classes(Application, Activity, Fragment, Services, BrodcastReciver, View, ViewModel ) annotated with @AndroidEntryPoint

Once you have enabled members injection in your Application, you can start enabling members injection in your other Android classes using the @AndroidEntryPoint annotation.

**@Enject**: use for Enject dependent classes or tell the Hilt give the object of Enjected calss

**@AndroidEntryPoint**

class MainActivity : AppCompatActivity() {

@Inject lateinit var movieAdapter: MovieAdapter

@Inject lateinit var repository: AuthRepository

}

**@qualifiers:**

Predefined qualifiers in Hilt(@ApplicationContext, @ActivityContext

**Hilt components:**

|  |
| --- |
|  |

|  |  |
| --- | --- |
| **Component** | **Injector for** |
| **SingletonComponent** | **Application** |
| **ViewModelComponent** | **ViewModel** |
| **ActivityComponent** | **Activity** |
| **FragmentComponent** | **Fragment** |
| **ViewComponent** | **View** |
| **ViewWithFragmentComponent** | **View with @WithFragmentBindings** |
| **ServiceComponent** | **Service** |

**Component lifetimes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Component** | **Scope** | **Created at** | **Destroyed at** |
| **SingletonComponent** | **@Singleton** | **Application#onCreate()** | **Application#onDestroy()** |
| **ActivityRetainedComponent** | **@ActivityRetainedScoped** | **Activity#onCreate()1** | **Activity#onDestroy()1** |
| **ViewModelComponent** | **@ViewModelScoped** | **ViewModel created** | **ViewModel destroyed** |
| **ActivityComponent** | **@ActivityScoped** | **Activity#onCreate()** | **Activity#onDestroy()** |
| **FragmentComponent** | **@FragmentScoped** | **Fragment#onAttach()** | **Fragment#onDestroy()** |
| **ViewComponent** | **@ViewScoped** | **View#super()** | **View destroyed** |
| **ViewWithFragmentComponent** | **@ViewScoped** | **View#super()** | **View destroyed** |
| **ServiceComponent** | **@ServiceScoped** | **Service#onCreate()** | **Service#onDestroy()** |

**Complete Example of Hilt DI -1**

1. **Application class: HiltApplication.kt @HiltAndroidApp**

@HiltAndroidApp

class HiltApplication : Application() {

override fun onCreate() {

super.onCreate()

}

}

1. **Retrofit Class: NetworkModule.kt : @Module**

@Module

@InstallIn(SingletonComponent::class)

object NetworkModule {

@Singleton

@Provides

fun provideOkHttp() : OkHttpClient{

return OkHttpClient.Builder()

.build()

}

@Singleton

@Provides

@Named("loggingInterceptor")

fun provideLoggingInterceptor(): HttpLoggingInterceptor {

return HttpLoggingInterceptor().apply {

this.level = HttpLoggingInterceptor.Level.BODY

}

}

@Provides

fun provideRetrofit(okHttpClient: OkHttpClient): Retrofit {

return Retrofit.Builder()

.baseUrl("https://howtodoandroid.com/")

.addConverterFactory(GsonConverterFactory.create())

.client(okHttpClient)

.build()

}

@Provides

fun provideApiClient(retrofit: Retrofit): ApiService {

return retrofit.create(ApiService::class.java)

}

}

1. **Repository Class : RepositoriesModule.kt : @Module**

@Module

@InstallIn(ViewModelComponent::class)

interface RepositoriesModule {

@Binds

fun mainRepository(mainRepositoryImpl: MainRepositoryImpl) : MainRepository

}

1. **ViewModel Class : MainViewModel.kt @HiltViewModel**

Having placed the **@HiltViewModel** above our ViewModel, we can now inject dependencies that are in either SingletonComponent or ViewModelComponent by using the **@Inject** annotation on the constructor or above fields or methods.

@HiltViewModel

class MainViewModel @Inject constructor(@ActivityContext context: ActivityContext,private val mainRepository: MainRepository): ViewModel() {

val movieList = MutableLiveData<List<Movie>>()

val progressBarStatus = MutableLiveData<Boolean>()

fun fetchAllMovies() {

progressBarStatus.value = true

CoroutineScope(Dispatchers.IO).launch {

val response = mainRepository.getAllMovies()

if (response.isSuccessful) {

movieList.postValue(response.body())

}

}

progressBarStatus.value = false

}

}

1. **Activity Class : MainActivity.kt @@AndroidEntryPoint**

@AndroidEntryPoint

class MainActivity : AppCompatActivity() {

@Inject lateinit var movieAdapter: MovieAdapter

private val viewModel : MainViewModel by viewModels()

lateinit var binding: ActivityMainBinding

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

binding = ActivityMainBinding.inflate(layoutInflater)

setContentView(binding.root)

binding.recyclerview.adapter = movieAdapter

viewModel.movieList.observe(this, Observer {

movieAdapter.setMovies(it)

})

viewModel.progressBarStatus.observe(this, Observer {

if (it) {

binding.progressDialog.visibility = View.VISIBLE

} else {

binding.progressDialog.visibility = View.GONE

}

})

viewModel.fetchAllMovies()

}

}

**Complete Example of Hilt DI -1**

1. **Application Class | @HiltAndroidApp**

**@HiltAndroidApp**

class NoteApplication : Application(){

}

1. **Data Class: for handle Request & response**

**data class UserRequest**(val email: String, val password: String, val username: String)

**data class UserResponse**( val token: String, val user: User)

**data class User**(val createdAt: String, val email: String, val id: String, val updatedAt: String, val username: String)

1. **Sealed class: For handle API response**

**sealed class NetworkResult<T>(val data: T? = null, val message: String? = null) {**

class Success<T>(data: T) : NetworkResult<T>(data)

class Error<T>(message: String?, data: T? = null) : NetworkResult<T>(data, message)

class Loading<T> : NetworkResult<T>()

}

1. **User / API Interface calss**

**interface UserAPI** {

@POST("/users/signup")

suspend fun signup(@Body userRequest: UserRequest) : Response<UserResponse>

@POST("/users/signin")

suspend fun signin(@Body userRequest: UserRequest) : Response<UserResponse>

}

1. **SharedPreferences. for save & Get The session Token**

**class TokenManager @Inject constructor**(@ApplicationContext context: Context) {

private var prefs: SharedPreferences =

context.getSharedPreferences(PREFS\_TOKEN\_FILE, Context.MODE\_PRIVATE)

fun saveToken(token: String) {

val editor = prefs.edit()

editor.putString(USER\_TOKEN, token)

editor.apply()

}

fun getToken(): String? {

return prefs.getString(USER\_TOKEN, null)

}

}

1. **AuthInterceptor : Interceptor**

**class AuthInterceptor @Inject constructor()** : Interceptor {

**@Inject**

lateinit var tokenManager: TokenManager

override fun intercept(chain: Interceptor.Chain): Response {

val request = chain.request().newBuilder()

val token = tokenManager.getToken()

request.addHeader("Authorization", "Bearer $token")

return chain.proceed(request.build())

}

}

1. Module Class : NetworkModule

**@InstallIn(SingletonComponent::class)**

**@Module**

**class NetworkModule {**

**@Singleton**

**@Provides**

fun providesRetrofit(): Retrofit.Builder {

return Retrofit.Builder().baseUrl(Constants.BASE\_URL)

.addConverterFactory(GsonConverterFactory.create())

}

**@Singleton**

**@Provides**

fun provideOkHttpClient(interceptor: AuthInterceptor): OkHttpClient {

return OkHttpClient.Builder().addInterceptor(interceptor).build()

}

**@Singleton**

**@Provides**

fun providesUserAPI(retrofitBuilder: Retrofit.Builder): UserAPI {

return retrofitBuilder.build().create(UserAPI::class.java)

}

**@Singleton**

**@Provides**

fun providesNoteAPI(retrofitBuilder: Retrofit.Builder, okHttpClient: OkHttpClient): NoteAPI {

return retrofitBuilder.client(okHttpClient).build().create(NoteAPI::class.java)

}

}

1. Repository

**class UserRepository @Inject constructor(private val userAPI: UserAPI) {**

private val \_userResponseLiveData = MutableLiveData<NetworkResult<UserResponse>>()

val userResponseLiveData: LiveData<NetworkResult<UserResponse>>

get() = \_userResponseLiveData

suspend fun registerUser(userRequest: UserRequest) {

\_userResponseLiveData.postValue(NetworkResult.Loading())

val response = userAPI.signup(userRequest)

handleResponse(response)

}

suspend fun loginUser(userRequest: UserRequest) {

\_userResponseLiveData.postValue(NetworkResult.Loading())

val response =userAPI.signin(userRequest)

handleResponse(response)

}

private fun handleResponse(response: Response<UserResponse>) {

if (response.*isSuccessful* && response.body() != null) {

\_userResponseLiveData.postValue(NetworkResult.Success(response.body()!!))

}

else if(response.errorBody()!=null){

val errorObj = JSONObject(response.errorBody()!!.charStream().*readText*())

\_userResponseLiveData.postValue(NetworkResult.Error(errorObj.getString("message")))

}

else{

\_userResponseLiveData.postValue(NetworkResult.Error("Something Went Wrong"))

}

}

}

9. Activity/Fragment

**@AndroidEntryPoint**

class LoginFragment : Fragment() {

private val authViewModel by *activityViewModels*<AuthViewModel>()

@Inject

lateinit var tokenManager: TokenManager

override fun onCreateView(

}

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

al userRequest = getUserRequest()

authViewModel.loginUser(userRequest)

val userRequest = getUserRequest()

authViewModel.registerUser(userRequest)

bindObservers()

}

private fun getUserRequest(): UserRequest {

return binding.*run* **{**

UserRequest(

txtEmail.*text*.toString(),

txtPassword.*text*.toString(),

""

)

**}**

}

private fun bindObservers() {

authViewModel.userResponseLiveData.observe(*viewLifecycleOwner*, *Observer* **{**

binding.progressBar.*isVisible* = false

when (**it**) {

is NetworkResult.Success -> {

tokenManager.saveToken(**it**.data!!.token)

*findNavController*().navigate(R.id.*action\_loginFragment\_to\_mainFragment*)

}

is NetworkResult.Error -> {

showValidationErrors(**it**.message.*toString*())

}

is NetworkResult.Loading ->{

binding.progressBar.*isVisible* = true

}

}

**}**)

}

**KOIN**:

Koin is a Kotlin based DI framework & it is fairly simple, lightweight, easy to use and does not have much boilerplate code as compare to Dagger-2 & Hilt.

**Koin Model:** Koin uses modules to create dependencies. Create a new Kotlin file with the name ‘module’ and add the following code.

val appModule = module {

single { Course() }

single { DataBase() }

factory { Friend() }

factory { Student(get(), get()) }

}

val RetrofitModule = module {

single { APICall() }

single { DataBase() }

factory { UserData() }

factory { Marks(get(), get()) }

}

1. The **module** function as the name suggests is used to create a module.

2. **single** is used to create a singleton instance

3. **factory** is used when we want a new instance of the class every time we call it

4. You can create any number of module and module name will your choice

**Starting Koin:** Starting Koin is fairly simple. Create an application class with the name of your choice and add the following code to start Koin.

startKoin(this, listOf(appModule, RetrofitModule))

1. you can pass list of module in **listOf(appModule)**

**Enject Koin:** You can enject the koin in very simple way in activity or fragment

val student: Student by inject()

val student: Student by inject()

**ButterKnife**

We use this library for injecting views into your Android Components. It is a lightweight library, and it uses annotation processing. It can save you from using findViewById(). And it will do the ffindViewById() thing for you automatically