Hilt - Android Dependency Injection

YouTube :

Hilt - Android Dependency Injection

Hilt and dependency injection - MAD Skills

문서 :

Hilt를 사용한 종속 항목 삽입

Android의 종속 항목 삽입

[우리의 룰]

주입할 클래스 별 알맞는 생명주기에 해당하는 컴포넌트를 사용하자

※ Component 별 사용 클래스

- @SingletonComponent
  :**Application-Level 에서 사용할 클래스**/ AppSettings, VCIComm, VehicleRepository, SQLiteHelper,EventBus, Utility 등
- @ActivityRetainedComponent
  :**Configurate Change(디바이스 화면 전환 등) 시에도 상태를 유지해야 하거나 ViewModel 에서 참조하는 클래스**/ UseCase 등
- @ActivityComponent
  :**Activity 에서 사용하는 클래스**/ Presenter, ViewModel, Model, Manager 등
- @FragmentComponent
  :**Fragment 에서 사용하는 클래스**/ Presenter, ViewModel, Model, Manager 등
- @ViewComponent,@ViewWithFragmentComponent
  : **View 에서 사용하는 클래스** / CustomView 등
- @ServiceComponent
  : **Service 에서 사용하는 클래스** / eReportService, GDSNUpdateService, NotificationService 등

※ 모듈 작성 기준

1. 주입하는 컴포넌트 개수가**단수이면 xxModule**,**복수라면 xxsModule**로 작성
2. 모듈 클래스를 만드는 기준은 주입 대상_패키지/di 패키지 안에서 생성
   - ex) app모듈의 provider패키지의 경우 :**com.gitauto.jupiter.provider.di**
3. Activity, Fragment, Service와 같은 안드로이드 컴포넌트는**@AndroidEntryPoint**로 해결
4. ViewModel의 경우**@HiltViewModel**어노테이션을 이용하여 모듈 필요없이 자동으로 주입하여 사용
5. **MVP Presenter**
   - Presenter의 경우 factory패턴을 통해 생성 받으므로, **xxx(모듈명)PresentersModule**로 클래스 구현
   - 다만, 기존처럼 Default Param 을 받는 것은 불가능하므로,**Parameter 를 생성 이후 시점으로 넘김(AssistedInject 사용)**
6. **UseCase-XXX(모듈명)UseCasesModule**로 클래스 구현
7. **Repository-XXX(모듈명)RepositoriesModule**로 클래스 구현
8. **Utility**
   - 기본적으로 공통으로 사용되는 Utillity의 경우 -**UtilitiesModule**
   - 특정 모듈에서만 사용되는 Utillity의 경우 -**XXX(모듈명)UtilitiesModule**

[DI 란?]

DI는 Dependency Injection의 줄임말로 **의존성 주입**이라는 뜻을 가지고 있다.

일반적으로 클래스에는 다른 클래스 참조가 필요한 상황이 자주 있다. 예를 들면 Car 라는클래스는Engine클래스 참조가 필요할 수 있다.

이처럼 **필요한 클래스를종속 항목**이라고 하며, 이 예에서Car클래스가 실행되기 위해서는Engine클래스의 인스턴스가 있어야 한다.

이제 여기서 우리는 **Engine클래스를 만들어 주는데 3가지 방법**이 있다.

1. Car 라는 클래스 안에서 직접 Engine 인스턴스를 생성한다.
2. 다른 곳에서 해당 인스턴스를 불러온다.
3. **매개변수로 외부에서 생성한 Engine 인스턴스를 제공받는다.**

이 세번째 방법이 바로 의존성 주입이다. 이로써 Car클래스는 Engine인스턴스를 생성하는 책임에서 벗어날 수 있다.

의존성 주입을 하는 이유는 여러 가지가 있다.

1. 코드 재사용 가능
2. 리팩터링 편의성
3. 테스트 편의성
4. 보일러 코드를 줄일 수 있다.
5. 클래스간 결합도를 느슨하게 한다.

[DI Framework 에는 뭐가 있을까?]

대표적으로

- **Dagger2**
- **Koin**
- **Hilt**

가 있으며 각각의 특징과 장단점이 있다.

	장점	단점
Dagger2	• 강력하고 빠른 의존성 주입 프레임워크로 많은 개발자들이 애용하고 있습니다. • 컴파일 타임에 DI 코드들을 생성해 런타임 성능이 좋습니다. • 자원 공유의 단순화가 가능합니다. • 라이브러리 크기가 작습니다.	• 러닝 커브가 높고 프로젝트 설정이 까다롭습니다. • 간단한 프로그램을 만들 때에는 배보다 배꼽이 더 클만큼 번거롭습니다. • 컴파일 시간이 오래걸립니다.
Koin	• 러닝커브가 낮아 쉽고 빠르게 적용이 가능합니다. • Kotlin 개발 환경에 도입하기 쉽습니다. • 별도의 어노테이션을 사용하지 않기 때문에 컴파일 시간이 단축됩니다. • ViewModel 주입을 쉽게 할 수 있는 별도의 라이브러리를 제공합니다.	• 런타임시 컴포넌트가 생성되어 있지 않다면 크래시가 발생합니다. • 런타임에 DI 코드를 생성해 런타임 성능이 좋지 않습니다. • 리플렉션을 이용해 성능상 좋지 않습니다. • koin.get() 함수와 같이 모듈간 의존성에 대해 신경을 쓰지 않고 인스턴스를 사용하는 경우, 추후 멀티모듈로 전향 시 어려움을 겪을 수 있습니다. • 모든 클래스가 서비스 로케이터에 종속됩니다. • 의존 관계 파악이 어렵습니다.
Hilt	• Dagger2 기반의 라이브러리이며 훨씬 사용하기 쉽고 표준화된 사용법을 제시합니다. • 프로젝트 설정이 간소화됨. • 쉬운 모듈 탐색과 통합 • 보일러 플레이트 코드 감소 • Android Studio의 지원 • AndroidX 라이브러리의 호환	• Hilt가 지원하는 안드로이드 클래스 외의 곳에서 사용하려면 부가적인 상용구를 추가해줘야 한다.

Dagger2 & Koin & Hilt 비교

	Dagger2	Koin	Hilt
적용 가능 언어	Java, Kotlin	Kotlin	Java, Kotlin
러닝 커브	높음	낮음	낮음
DI&에러 검출 시점	Compile Time	Run Time	Compile Time

[Hilt 란?]

Hilt는 DI를 쉽게 하기 위해 만들어진 안드로이드 라이브러리이다.Hilt는 프로젝트의 **모든 Android 클래스에 컨테이너를 제공하고 수명 주기를 자동으로 관리함으로써 애플리케이션에서 DI를 사용하는 표준 방법을 제공**한다.

Hilt는 Dagger가 제공하는 컴파일 시간 정확성, 런타임 성능, 확장성 및Android 스튜디오 지원의 이점을 누리기 위해 인기 있는 **DI 라이브러리Dagger를 기반으로 빌드**되었다.

Dagger와 관련하여 **Hilt의 목표**는 다음과 같습니다.

- Dagger 사용의 **단순화**
- 표준화된 컴포넌트 세트와 스코프로 **설정과 가독성/이해도 쉽게 만들기**
- 쉬운 방법으로 **다양한 빌드 타입에 대해 다른 바인딩 제공**

Android 운영체제는 많은 자체 프레임워크 클래스를 인스턴스화하므로 Android 앱에서 Dagger를 사용하려면 상당한 양의 상용구를 작성해야 한다.

Hilt는 Android 애플리케이션에서 Dagger 사용과 관련된 상용구 코드를 줄인다. Hilt는 자동으로 다음을 생성하고 제공한다.

- 달리 수동으로 생성해야 하는 Dagger와**Android 프레임워크 클래스를 통합하기 위한 구성요소**
- Hilt가 자동으로 생성하는 구성요소와 함께 사용할**범위 주석**
- `Application`또는`Activity`와 같은 Android 클래스를 나타내는**사전 정의된 결합**
- `@ApplicationContext`및`@ActivityContext`를 나타내는사전 **정의된 한정자**

Dagger 및 Hilt 코드는 동일한 코드베이스에 공존할 수 있다. 그러나 대부분의 경우 Android에서 Dagger의 모든 사용을 관리하려면 Hilt를 사용하는 것이 가장 좋다.

[Annotations]

@HiltAndroidApp : Hilt 컴포넌트의 코드 생성과 컴포넌트를 사용하는 기본 클래스 생성

@AndroidEntryPoint : DI 컨테이너를 추가하여 멤버 주입을 가능하게 해준다

@HiltViewModel : ViewModel 주입을 위한 것으로 ViewModel 클래스에 작성

@Inject : 컴포넌트로부터 의존성 객체 주입을 요청

@XXComponent : 인스턴스 생명주기

@XXScope : 컴포넌트 바인딩 범위

@Module : 컴포넌트에 의존성을 제공하는 역할

@InstallIn : Hilt 의 표준 컴포넌트들 중 어떤 컴포넌트에 모듈을 설치할지 결정

@Binds : 인터페이스 인스턴스 주입

@Provides : 클래스 인스턴스 주입(외부 라이브러리 포함)

@EntryPoint : Hilt 가 지원하지 않는 클래스에서 의존성이 필요한 경우 객체를 얻을 수 있는 방법

[Gradle Setup]

Hilt를 프로젝트에 적용하기 위해서는 아래의 셋업 과정이 필수적으로 요구된다.

project/build.gradle

```
 buildscript {
    ...
    dependencies {
        ...
        classpath 'com.google.dagger:hilt-android-gradle-plugin:2.42'
    }
}
```

app/build.gradle

```
...
apply plugin: 'kotlin-kapt'
apply plugin: 'dagger.hilt.android.plugin'

android {
  ...
  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
}

dependencies {
    implementation "com.google.dagger:hilt-android:2.42"
    kapt "com.google.dagger:hilt-android-compiler:2.42"
}
```

+) Hilt는**AndroidX 라이브러리와 호환**되어 있다.

[Hilt Application]

Hilt를 사용하기 위해서는**Application 클래스를 반드시 @HiltAndroidApp 과 함께 만들어주고 Manifest에 android:name 세팅**을 해주어야 한다.

Hilt Application

```
@HiltAndroidApp
class MainApplication : Application()
```

@HiltAndroidApp은Hilt 관련 컴포넌트, 모듈 등 모든 코드 생성을 시작하는 어노테이션으로**DI 환경을 빌딩하는데Base가 되고 컨테이너 역할을**하게 된다.

컴파일 타임 시 표준 Hilt 컴포넌트 빌딩에 필요한 클래스들을 초기화를 해주기 때문에**Hilt를 사용하는 앱은 반드시@HiltAndroidApp을 가진Application 클래스를 manifest app에 포함**시켜야 한다.

추가로@HiltAndroidApp는 Application객체의 수명 주기에 연결된앱의 최상단 부모 컴포넌트이므로이와 관련한 수명주기와 ApplicationContext 등 같은 종속 항목들을 하위(서브) 컴포넌트들에게 제공할 수 있다.

(추가 참고 :**컴포넌트들은 계층으로 이루어져 있으며 하위(서브) 컴포넌트는 상위 컴포넌트의 의존성에 접근할 수 있다.)**

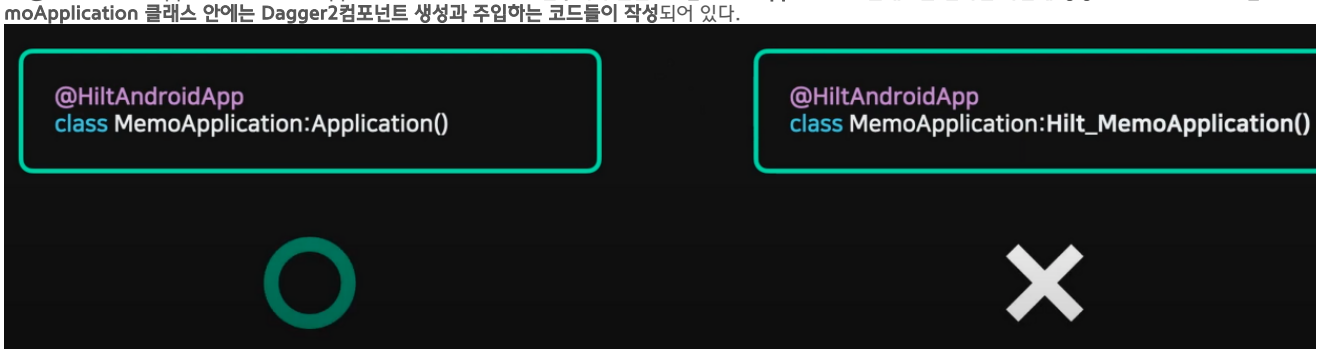
[Component 생성]

Component 생성

```
//  Dagger2
class MemoApplication: Application() {
        override fun onCreate() {
            super.onCreate()
                DaggerMemoComponent.builder()
                .build()
                        .inject(this)
    }
}

// Hilt
@HiltAndroidApp
class MemoApplication: Application() {
        override fun onCreate() {
            super.onCreate() //   super.onCreate()  bytecode
                //
    }
}
```

어떻게 위와 같이 마법이 일어날까?

1. @HiltAndroidApp 이 붙은 MemoApplication 클래스는 **Hilt 접두어가 붙은 Hilt_MemoApplication 클래스를 컴파일 타임에 생성**해낸다. 그리고 **Hilt_MemoApplication 클래스 안에는 Dagger2컴포넌트 생성과 주입하는 코드들이 작성**되어 있다.



※ 오른쪽과 같이 변경하지 않아도 된다.

2. 그리고 MemoApplication 은 Hilt_MemoApplication 을 상속받는 구조가 되면 해결이 될 것 같지만,

실제로는 Hilt_MemoApplication 을 상속받지 않아도, 컴파일시 Gradle 에서 MemoApplication 바이트 코드를 Hilt_MemoApplication 을 상속하는 MemoApplication 바이트 코드로 변경(조작)한다.

3. 혹시나 바이트 코드 변환을 원하지 않으면, Gradle 옵션을 disabled 시키면 된다. 이때는 Hilt_MemoApplication 을 상속하도록 개발자가 직접 코드를 변경해 주면 된다.

[AndroidEntryPoint]

@AndroidEntryPoint는@HiltAndroidApp설정 후 사용 가능하며**@AndroidEntryPoint**어노테이션이 추가된 안드로이드 클래스에 DI 컨테이너를 추가 해준다.

안드로이드 클래스 중**@AndroidEntryPoint는**

- Activity
- Fragment
- View
- Service
- BroadcastReceiver

를 지원한다.

앞서**Application 클래스에 @HiltAndroidApp**으로 애플리케이션 수준인 최상단 컴포넌트를 사용할 수 있게 되었으니 그 하위(서브) Android 클래스들에**@AndroidEntryPoint**를 설정함으로써**Application(상위) -> 안드로이드 클래스(하위) 종속 항목(dependecies)를 제공**할 수 있게 해준다.

추가로 프래그먼트에**@AndroidEntryPoint를 설정**하려면 상위 개념에 해당하는 액티비티에도**@AndroidEntryPoint가 설정**되어 있어야 한다.

AndroidEntryPoint

```
@AndroidEntryPoint
class GalleryFragment : Fragment() {
```

Dagger2 비교

Hilt	Dagger2
@HiltAndroidApp	@Component
@AndroidEntryPoint	@SubComponent

[ViewModel Injection]

Hilt가 적용된**ViewModel 이**다. Dagger2와 비교해서**@Inject constructor**는 똑같이 사용하는데 ViewModel 전용의**@HiltViewModel**이라는게 생겼다.

Dagger2 에서는 ViewModel하고 엮을때 되게 복잡했었는데 간단해졌다고 느꼈다. @Inject construtor 와 @HiltViewModel 을 이어서 살펴보겠다.

HiltViewModel

```
@HiltViewModel
class PlantDetailViewModel @Inject constructor(
    savedStateHandle: SavedStateHandle,
    plantRepository: PlantRepository,
    private val gardenPlantingRepository: GardenPlantingRepository,
) : ViewModel() {
```

@Inject constructor이란?

생성자 삽입 방법으로 클래스의 인스턴스를 제공하는 방법을 Hilt에 알려주게 된다. 제공하는 방법을 알려주니 당연히 해당 인스턴스를 제공하는 방법도 알아야하고 제공해주는 녀석도 있을 것이다.

@HiltViewModel 이란?

먼저 @HiltViewModel을 타고 들어가면 다음과 같다.

@HiltViewModel 어노테이션이 붙은 ViewModel은 HiltViewModelFactory에 의해 생성되고 @AndroidEntryPoint 어노테이션이 붙은 액티비티와 프래그먼트에서 기본 디폴트로 회수해오는게 가능해지게 한다.

또한 @HiltViewModel에서 @Inject 어노테이션이 붙은 생성자는 생성자 파라미터가 Hilt에 의해 주입받을 것이라고 정의내리는 종속성을 갖게 해준다.

@AndroidEntryPoint 어노테이션이 있는 액티비티나 프래그먼트에서 이 Hilt가 적용된 ViewModel 인스턴스를 얻으려면**ViewModelProvider**나 kt-extensions 인**by viewmodels()**을 사용하면 된다.

by viewModels()

```
@AndroidEntryPoint
class PlantDetailFragment : Fragment() {
        ...
        private val plantDetailViewModel: PlantDetailViewModel by viewModels()
        ...
}
```

참고 :Hilt로 ViewModel 객체 삽입

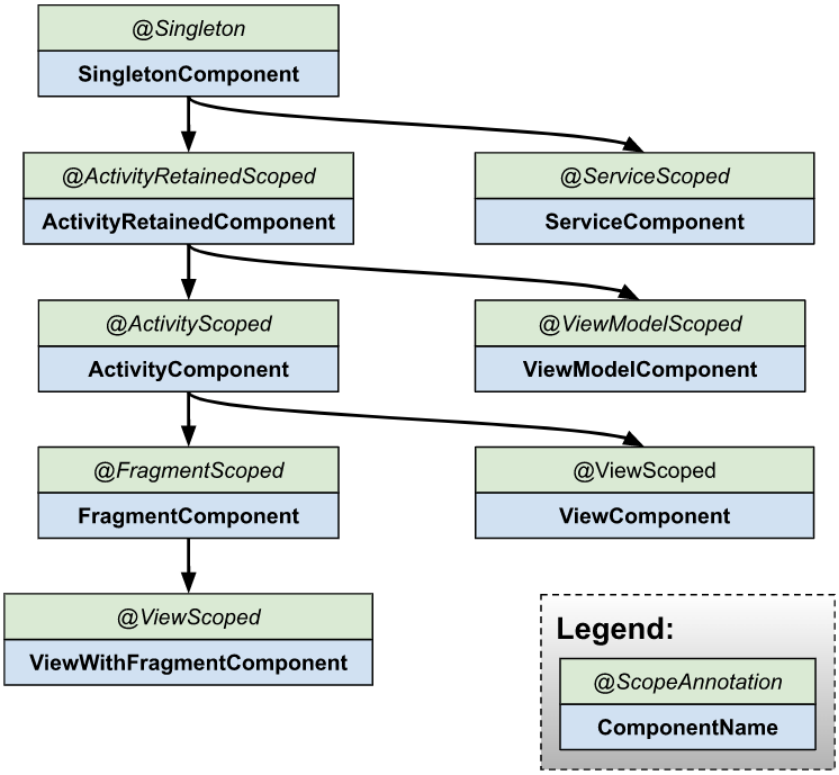
[Component hierarchy]

component는 물건을 만드는 방법을 알고 있는 **공장(Factory)**과 같다. 그러나 **공장에서 생성하는 것들의 인스턴스를 유지**할 수 있기 때문에 어떤 면에서는 **컨테이너**이기도 하다.

기존의 Dagger2는 개발자가 직접 필요한 component들을 작성하고 상속 관계를 정의했다면, **Hilt에서는 Android 환경에서 표준적으로 사용되는 component들을 기본적으로 제공하고 있다.**

또한 Hilt 내부적으로 제공하는 **component들의 전반적인 라이프 사이클 또한 자동으로 관리**해주기 때문에 사용자가 **초기 DI 환경을 구축하는데 드는 비용을 최소화**하고 있다.

다음은 Hilt에서 제공하는 **표준 component hierarchy**이다.



※ 하위 컴포넌트는 상위 컴포넌트가 가지고 있는 의존성에 대해 접근 가능하다 (직계 수직 관계에서만 가능)

Hilt에서 표준적으로 제공하는 **Component, 관련 Scope, 생성 및 파괴 시점**은 아래와 같다.

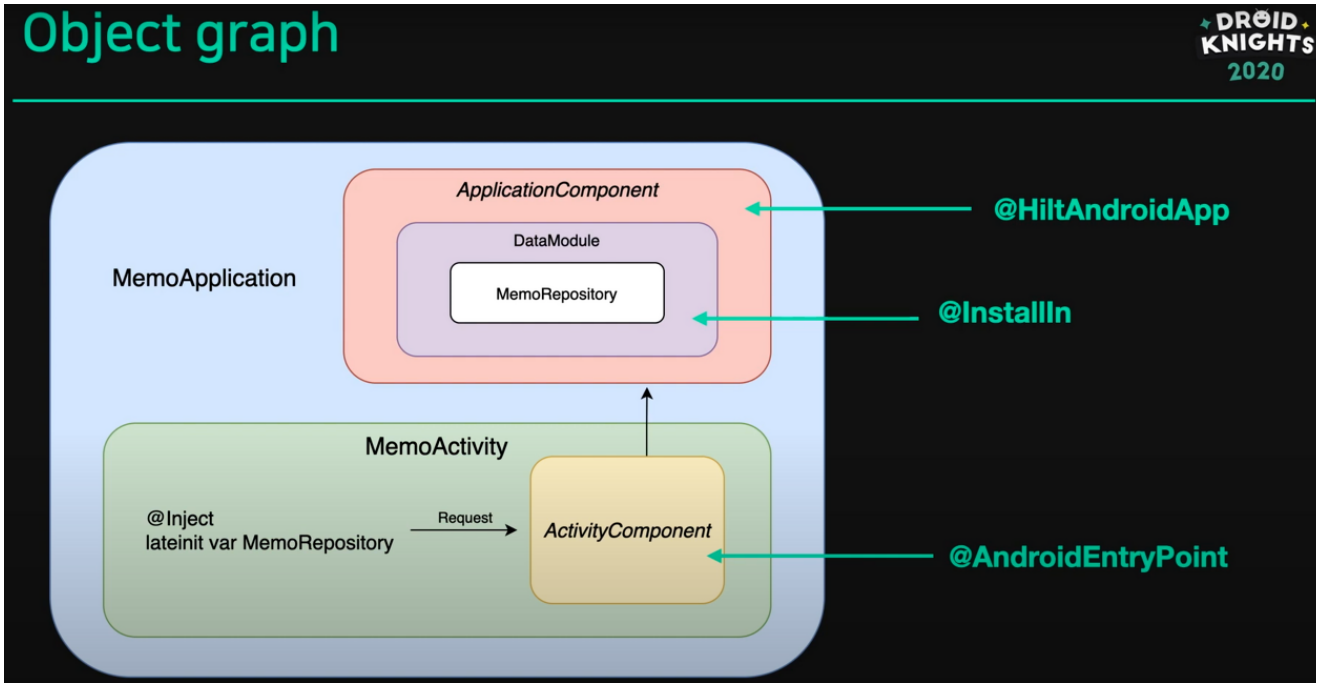
Compoent	Scope	Created at	Destroyed at	LifeCycle
SingletonComponent	@Singleton	Application#onCreate()	Application#onDestroy()	Application
ActivityRetainedComponent	@ActivityRetainedScoped	Activity#onCreate()	Activity#onDestroy()	Activity, cofiguration change
ActivityComponent	@ActivityScoped	Activity#onCreate()	Activity#onDestroy()	Activity
FragmentComponent	@FragmentScoped	Fragment#onAttach()	Fragment#onDestroy()	Fragment
ViewComponent	@ViewScoped	View#super()	View destroyed	View
ViewWithFragmentComponent	@ViewScoped	View#super()	View destroyed	View(Fragment)
ServiceComponent	@ServiceScoped	Service#onCreate()	Service#onDestroy()	Service

이렇게**@AndroidEntryPoint**로 설정된 클래스는**개별적인 Hilt 컴포넌트를 만들고 컴포넌트의 계층 구조(Component hierarchy) 에 설명된 대로 각 상위 클래스 에서 종속 항목을 제공받을**수 있게 된다.

또한모듈에서 사용되는 Scope 어노테이션은 반드시 InstallIn 에 명시된 Component 와 쌍을 이뤄야 한다.(ActivityComponent 에는 ActivityScoped, FragmentComponent 에는 FragmentScoped 사용하기)

위 사진처럼 표준화된 컴포넌트 세트와 스코프를 제공함을 볼 수 있다.

```
// Hilt
@HiltAndroidApp //
class MemoApp: Application()

@AndroidEntryPoint // Activity   Inject
class MemoActivity: AppcompatActivity() {

    @Inject
    lateinit var repository: MemoRepository

    override fun onCreate(savedInstanceState: Bundle) {
        super.onCreate(savedInstanceState)
        repository.load("8922")
    }
}

//
class MemoRepository @Inject constructor(
    private val db: MemoDatabase
) {
    fun load(id: String) { ... }
}
@Module
@InstallIn(SingletonComponent::class)
object DataModule {

    @Provides
        @Singleton
    fun provideMemoDB(@ApplicationContext context: Context) =
            Room.databaseBuilder(context, MemoDatabase::class.java, "Memo.db").build()

}
```



1. @HiltAndroidApp 을 통해 SingletonComponent 가 먼저 생성된다

2. @InstallIn 을 통해 SingletonComponent 안에 DataModule 이 설치된다

3. @AndroidEntryPoint 를 통해 SingletonComponent 의 하위 컴포넌트인 ActivityComponent 가 생성되고, ActivityComponent 를 사용하여 MemoRepository 를 주입 받는다.

[Hilt Modules]

기존의 Dagger2에서는 새로운 module을 생성하면, 사용자가 정의한 component에 해당 module 클래스를 직접 include 해주는 방법이었습니다.

반면, Hilt는 표준적으로 제공하는 component 들이 이미 존재하기 때문에`@InstallIn`어노테이션을 사용하여 표준 component에 module들을 install 할 수 있습니다.

Hilt에서 제공하는 기본적인 규칙은 모든 module에`@InstallIn`어노테이션을 사용하여 어떤 component에 install 할지 반드시 정해주어야 합니다.

또한 Hilt 모듈은 @Binds와 @Provides로 Hilt binding 정보를 제공해줄 수 있습니다.

@Binds와 @provides의 눈에 보이는 큰 차이점은

@Binds는 인터페이스 인스턴스(Interface instance)를 제공하는 것이고

@Provides는 클래스의 인스턴스(소유하지 않은 클래스)를 제공하는 것입니다.

@Binds

@Binds는 Hilt에 아래 정보를 알려줍니다.

함수 Return type : 인스턴스(Instance)로 제공되는 인터페이스
함수 매개변수(Parameter) : 실제 제공하는 클래스(구현)

```
@Module
@InstallIn(SingletonComponent::class)
abstract class ExampleRepositoryModule {

        @Binds
        @Singleton
        abstract fun bindExampleRepository(
                exampleRepositoryImpl: ExampleRepositoryImpl
        ): ExampleRepository
}
```

@Provides

@Provides는 의존성을 삽입하려는 클래스가 외부 라이브러리에서 제공되어 클래스를 소유하지 않은 경우(Retrofit이나 Room database)나 빌드 패턴 등의 인스턴스를 생성해야하는 경우 사용됩니다.

함수 Return type : 제공하는 인스턴스

함수 매개변수(Parameter) : 인스턴스의 종속 항목

함수 Body : 인스턴스를 제공하는 방법

아래 예시는 NetworkModule을 SingletonComponent에install하고,`SingletonComponent`에서 제공해주는`Application`class를 내부적으로 활용하고 있습니다.

Module

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

    @Provides
    @Singleton
    fun provideUnsplashService(): UnsplashService {
        return UnsplashService.create()
    }
}

---

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

    @Provides
    @Singleton
    fun provideAppDatabase(@ApplicationContext context: Context): AppDatabase {
        return AppDatabase.getInstance(context)
    }

    @Provides
    fun providePlantDao(appDatabase: AppDatabase): PlantDao {
        return appDatabase.plantDao()
    }

    @Provides
    fun provideGardenPlantingDao(appDatabase: AppDatabase): GardenPlantingDao {
        return appDatabase.gardenPlantingDao()
    }
}
```

SingletonComponent로 하는 이유를 생각해보면 DB, 서버 API 통신 객체는 여러 하나의 클래스에 종속되어 있는게 아닌 여러 Repository에서 사용할 수 있고, 어디서든 접근이 가능해야 하므로 Singleton 컴포넌트로 작성을 한다.

그러한 이유로 DatabaseModule과 NetworkModule 은 InstallIn(SingletonComponent::class)를 통해 싱글턴 모듈임을 나타내도록 한다.

blocked URL

*** 프래그먼트 컴포넌트가 액티비티 컴포넌트의 하위 컴포넌트라 Module을 같이 제공받을 수 있다.**

Hilt Module의 제약사항

@Module 클래스에@installIn이 없으면 컴파일 에러!

@InstallIn 검사 비활성화

```
android {
        defaultConfig {
                javaCompileOptions {
                        annotationProcessorOptions {
                                arguments += ["dagger.hilt.disableModulesHavenInstallInCheck":"true"]
                        }
                }
        }
}
```

Q. 참고로 Repository는 왜 싱글턴 객체여야 하는가?

A.Repository는 네트워크 작업 혹은 데이터베이스 작업같이 데이터를 주고 받기 위해 만들어진 View와 ViewModel과는 별개의 공간이다. 만약 싱글턴이 아닌 단순 클래스라고 가정하면, 매번 네트워크 작업 혹은 데이터베이스 작업이 일어날 시 새로운 클래스 객체를 생성한다는 것은 매우 비효율적이다.

만약 클래스 생성이 오래 걸린다고 가정하면, 네트워크 작업 및 데이터베이스 작업을 하기 위해 클래스 객체를 생성하는 것은 네트워크 처리, 데이터베이스 처리 시간에 더해져 매우 오래 걸릴 것이다. 그래서 싱글턴 객체로 선언을 하여 항상 어디서든 준비되어 있도록 한다.

Repository

```
@Singleton
class GardenPlantingRepository @Inject constructor(
    private val gardenPlantingDao: GardenPlantingDao
)
```

[번외]

1.**@EntryPoint**:Hilt가 지원하지 않는 클래스에서 의존성이 필요한 경우 사용한다. (예 : ContentProvider, DFM, Dagger를 사용하지 않는 3rd-party 라이브러리 등)

- @EntryPoint는 인터페이스에서만 사용

-@InstallIn 이 반드시 함께 있어야 함

- EntryPoints 클래스의 정적 메서드를 통해 그래프에 접근

EntryPoint

```
// EntryPoint
@EntryPoint
@InstallIn(SingletonComponent::class)
interface MemoEntryPoint {
        fun getRepository(): MemoRepository
}

// EntryPoint
class MemoProvider: ContentProvider() {
        override fun query(...) {
                val entryPoint = EntryPointAccessors.fromApplication(context, MemoEntryPoint::class.java)
                val repository = entryPoint.getRepository()
                ...
        }
}
```

2.**@DefineComponent**: 표준 Hilt 컴포넌트 이외에 새로운 컴포넌트를 만드는 것이다. 커스텀 컴포넌트를 사용하면 복잡하고 이해하기 어려워지기 때문에 꼭 필요한 경우만 사용한다.

3.**@HiltWorker**:WorkManager with Hilt :Hilt로 WorkManager 삽입