DAGGER

- @Inject
- @Provides
- @Singleton
- @Module
- @Component

DEPENDENCIAS

```
implementation 'com.google.dagger:dagger:2.23.1'
kapt 'com.google.dagger:dagger-compiler:2.23.1'
```

@Inject

- Indica a Dagger dónde debe inyectar las dependencias
- Dos lugares principales:
  - Constructor
  - Property
- Si es por constructor, se incluye ese objeto como parte del grafo de Dagger automáticamente

@Inject - Constructor

```
class GetPopularMovies @Inject constructor(
    private val moviesRepository: MoviesRepository
) {
    suspend fun invoke(): List<Movie> = moviesRepository.getPopularMovies()
}
```

@Inject - Constructor

```
class GetPopularMovies @Inject constructor(
    private val moviesRepository: MoviesRepository
) {
    suspend fun invoke(): List<Movie> = moviesRepository.getPopularMovies()
}
```

- No podemos usarlo si tenemos interfaces
- Nos obliga a incluir Dagger en todos los módulos

@Inject - Propiedades

```
@Inject
lateinit var viewModel: MainViewModel
```

@Inject - Propiedades

```
@Inject
lateinit var viewModel: MainViewModel
```

- Para clases en las que no tenemos acceso al constructor
- Nos obliga a inyectar el objeto en el grafo manualmente

@Inject - Propiedades

```
@Inject
lateinit var viewModel: MainViewModel
```

- Para clases en las que no tenemos acceso al constructor
- Nos obliga a inyectar el objeto en el grafo manualmente
- En general es más sencillo si no usamos @Inject (luego lo vemos)

@Provides

- Para las dependencias que no podemos proveer con @Inject en el constructor
- Anotamos la función que provee dicha dependencia
- Los argumentos de esa función también serán proporcionados por Dagger

@Provides

```
@Provides
fun moviesRepositoryProvider(
    localDataSource: LocalDataSource,
    remoteDataSource: RemoteDataSource,
    regionRepository: RegionRepository,
    @Named("apiKey") apiKey: String
) = MoviesRepository(localDataSource, remoteDataSource, regionRepository, apiKey)
```

@Singleton

```
@Provides @Singleton
fun moviesRepositoryProvider(
    localDataSource: LocalDataSource,
    remoteDataSource: RemoteDataSource,
    regionRepository: RegionRepository,
    @Named("apiKey") apiKey: String
) = MoviesRepository(localDataSource, remoteDataSource, regionRepository, apiKey)
```

@Singleton

- Una implementación mucho más cómoda del patrón singleton
- Solo habrá una instancia en la App, pero esta puede ser modificada en tests por ejemplo
- La dependencia es explícita donde se use

@Module

- Un conjunto de dependencias comunes
- El módulo contiene una serie de métodos anotados con *@Provides*
- Se podrán combinar un conjunto de módulos para formar el grafo final

@Module

```
@Module
class DataModule {

    @Provides
    fun regionRepositoryProvider(locationDataSource: LocationDataSource,permissionChecker:
PermissionChecker
    ) = RegionRepository(locationDataSource, permissionChecker)

    @Provides
    fun moviesRepositoryProvider(localDataSource: LocalDataSource,remoteDataSource: RemoteDataSource,
        regionRepository: RegionRepository,@Named("apiKey") apiKey: String
    ) = MoviesRepository(localDataSource, remoteDataSource, regionRepository, apiKey)
}
```

@Component

- Identifica los módulos que componen el grafo
- Se pueden especificar las dependencias que expone
- También qué a qué clases les podemos añadir dependencias anotadas con *@Inject*
- Si es un component que solo se puede crear una vez, hay que marcarlo con *@Singleton*

@Component

```
@Singleton
@Component(modules = [AppModule::class, DataModule::class,
    UseCaseModule::class, ViewModelsModule::class])
interface MyMoviesComponent
```

@Component

```
component = DaggerMyMoviesComponent
    .builder()
    .appModule(AppModule(this))
    .dataModule(DataModule())
    .useCaseModule(UseCaseModule())
    .viewModelsModule(ViewModelsModule())
    .build()
```

@Component.Factory

```
@Singleton
@Component(modules = [...])
interface MyMoviesComponent {

    @Component.Factory
    interface Factory {
        fun create(@BindsInstance app: Application): MyMoviesComponent
    }
}
```

@Component.Factory

```
component = DaggerMyMoviesComponent
    .factory()
    .create(this)
```

@Inject vs exponer dependencias

```
@Singleton @Component(modules = [...])
interface MyMoviesComponent {

    fun inject(mainActivity: MainActivity)
    fun inject(detalActivity: DetailActivity)

    val mainViewModel: MainViewModel
    val detaiViewModel: DetailViewModel
}
```

@Inject vs exponer dependencias

```
@Singleton @Component(modules = [...])
interface MyMoviesComponent {

    fun inject(mainActivity: MainActivity)
    fun inject(detalActivity: DetailActivity)

    val mainViewModel: MainViewModel
    val detaiViewModel: DetailViewModel
}
```

@Inject vs exponer dependencias

```
@Inject
lateinit var viewModelProvider: Lazy<DetailViewModel>
private lateinit var viewModel: DetailViewModel

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_detail)

    app.component.inject(this)
    viewModel = getViewModel { viewModelProvider.get() }
}
```

@Inject vs exponer dependencias

```
private val viewModel by lazy {
    getViewModel { app.component.detailViewModel }
}
```

Dependencias en activities y fragments

- Puede haber dependencias que solo nos interesen durante el tiempo de vida de Activities o Fragments
- Mantenerlas en el grafo principal es un malgasto de memoria
- Podemos crearnos Components que se crean y viven durante el tiempo de vida de esa Activity
- Usaremos la anotación *@Subcomponent*

@Subcomponent - Creación

```
@Subcomponent(modules = [(MainActivityModule::class)])
interface MainActivityComponent
```

@Subcomponent - Creación

```
@Subcomponent(modules = [(MainActivityModule::class)])
interface MainActivityComponent {
    val mainViewModel: MainViewModel
}
```

@Subcomponent - El módulo

```
@Module
class MainActivityModule {

    @Provides
    fun mainViewModelProvider(getPopularMovies: GetPopularMovies) =
        MainViewModel(getPopularMovies)

    @Provides
    fun getPopularMoviesProvider(moviesRepository: MoviesRepository) =
        GetPopularMovies(moviesRepository)
}
```

@Subcomponent - Creación

```
private lateinit var component: MainActivityComponent

override fun onCreate(savedInstanceState: Bundle?) {
    ...
    component = app.component.plus(MainActivityModule())
}
```