koin
Cloud_Inject

Your new
application
architecture
superpowers

Koin Cloud_Inject is a platform for developers that allows you to :

- Anticipate your application architecture behaviour so you can squash issues before they occur.
- Foresee trends in your development roadmap so you can deploy new features safer and faster

Register to join the closed beta testing program here : register.kotzilla.io

koin by kotzilla

Koin 3.4

Cheat Sheet For Android

Koin is a Kotlin framework to help you build any kind of Kotlin application, from Android mobile to backend Ktor server applications, including Kotlin Multiplatform and Compose.

Gradle Setup

Check the Koin setup page for the latest version:
setup.insert-koin.io

Declaring your Components with Modules

Use the Koin DSL to describe your components inside modules:

```
class MyClassA()
class MyClassB(val classA : MyClassA)

val myModule = module {
    single { MyClassA() }
    single { MyClassB(get()) }
}
```

Or directly with constructors:

```
val myModule = module {
    singleOf(::MyClassA)
    singleOf(::MyClassB)
}
```

Retrieving dependencies with Koin API

To retrieve definitions in your Android classes with by inject() or the by viewModel():

```
class MyActivity : AppCompatActivity() {

    val presenter by inject<Presenter>()
    val myViewModel by viewModel<MyViewModel>()
}
```

Start Koin

Start and configure your Koin application with the following DSL:

Start DSL	Description
startKoin	Start a Koin instance
androidLogger	Use the Android logger for Koin
androidContext	Use the Android Context to be resolved in Koin
modules	Load a list of Koin modules

```
class MyApplication : Application(){
    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            androidContext(this@MyApplication)
            modules(appModule)
        }
    }
}
```

Koin DSL Keywords

In a Koin module, you can use the following keywords:

Function DSL	Constructor DSL	Description
includes	-	Includes other Koin modules
single	singleOf	Singleton instance definition (unique instance)
factory	factoryOf	Factory instance definition (instance recreated each time)
scope	-	Define a Scope for given type or qualifier
scoped	scopedOf	Define a scoped instance (live for the duration of the scope)
viewModel	viewModelOf	Android ViewModel instance
fragment	fragmentOf	Android Fragment instance
worker	workerOf	Android Worker for work Manager

Lazy Modules & Background Loading

You may define modules with lazyModule , to load them in the background later:

```
val myLazyModule = lazyModule {
        // definitions
}
```

In your Koin application use the lazyModules function to load your lazy modules in the background

```
startKoin {
        // background loading
    lazyModules(appModule)
}

KoinPlatform.getKoin().runOnKoinStarted { koin ->
        // run code after start is completed
}
```

Koin Annotations Cheat Sheet For Android

koin by kotzilla

www.kotzilla.io

@kotzilla_io

The goal of the Koin Annotations project is to help declare the Koin definition in a very fast and intuitive way and generate all underlying Koin DSL for you.

KSP & Gradle Setup

Check the Koin setup page for latest version: setup.insert-koin.io

Be sure to configure your project to use KSP generated sources.

Declaring your Components

Use Koin Annotations on your classes to declare it in Koin. Annotated classes will be injected into their first constructor:

```
@Single
class MyClassA()

@Single
class MyClassB(val classA : MyClassA)
```

Any interface is automatically detected as a bound type:

```
@Single
class MyClassB(val classA : MyClassA) : MyInterface
```

Injected dependency can be nullable:

```
@Single
class MyClassB(val classA : MyClassA?)
```

Lazy type and List types are detected (new in 1.2)

```
// Detect Lazy
@Single
class MyClassB(val classA : Lazy<MyClassA>)
```

```
// Detect List
@Single
class MyClassB(val allInterfaces : List<MyInterface>)
```

Injected parameter can declared just by tagging your parameter with @InjectedParam

```
@Single
class MyClass(@InjectedParam val id : String)
```

Declaring with a Kotlin function (new in 1.2)

Use Koin Annotations on your top-level function to declare it in Koin:

```
@Single
fun buildMyClassB(classA : MyClassA) = MyClassB(classA)
```

Modules

Annotate a class with @Module and @ComponentScan, to scan for annotated components in the given package:

```
@Module
@ComponentScan(«com.my.package»)
class MyModule
```

You can also declare components directly in a module class:

```
@Module
class MyModule {

    @Single
    fun myClassA() = MyClassA()

    @Single
    fun myClassB(classA : MyClassA) = MyClassB(classA)
}
```

Specify included modules directly on the @Module annotation with the includes attribute:

```
@Module(includes = [Module1::class])
class MyModule
```

Koin Annotations

The following annotations can be used in classes:

Annotation	Description
@Module	Declare class as a Koin module. Can also use includes parameter, to specify included modules
@ComponentScan	Scan Koin definitions for given package
@Single	Singleton instance definition (unique instance)
@Factory	Factory instance definition (instance recreated each time)
@Scope	Define a Scope for given type or qualifier
@Scoped	Define a scoped instance (live for the duration of the scope) for given type scope
@KoinViewModel	Android ViewModel instance
@KoinWorker	Android Worker WorkManager

You can inject the Android context easily: just use a Context parameter in your constructor.

Start Koin

Start and configure your Koin instance with the regular Start DSL. To use an annotated module class, just instantiate it and use the .module generated extension.

Don't forget to use the right import for generated sources:

```
// Use Koin Generation
import org.koin.ksp.generated.*

startKoin {
    androidLogger()
    androidContext(this@MyApplication)
    modules(MyModule().module)
}
```