Kotlin Multiplatform (KMP) Cheat Sheet

Koin is the Kotlin Multiplatform (KMP) integration framework.

You can write code once and deploy it on multiple platforms, with KMP as the main cross-platform technology, and Koin as the Dependency Injection framework.

Koin offers a fully Kotlin-centric approach, leveraging the language's features and syntax to provide a simpler and more intuitive DI experience.

Multiplatform Application Setup

Take a look at the Kotlin KMP getting started guide to help start your new Kotlin Multiplatform application

For Koin setup instructions, please refer to the Koin Setup page.

Add the `koin-core` or `koin-test` Gradle dependency to your KMP project.

Koin Annotations & KMP

To integrate Koin Annotations into your KMP project, begin by following the KSP guide from Google.

After integrating the Google KSP plugin, add the `koin-annotations` & `koin--ksp-compiler` dependencies to your KMP setup. Detailed steps can be found in this tutorial.

Koin Dependency Injection in Shared Code

Koin is a pure Kotlin framework. It interacts naturally with your Kotlin shared code project the same way as your JVM or Android project.

Define Koin modules for the needed classes to inject into your application.

We recommend utilizing `constructor injection` whenever possible.
Alternatively, the `KoinComponent` interface can assist in dependency injection through properties.

```kotlin
fun commonModule() = module {
    single { createJson() }
    single { createHttpClient(get(), get()) }
    // ...
}
```

You can use `Constructor DSL` as well, with keywords such as `singleOf`, `factoryOf`

Implementing Native Dependency Modules

It's also possible to define a Koin module, for a specific platform. Use the `actual` Kotlin keyword to define it:

```kotlin
// in common code
expect fun platformModule(): Module
```

And implement your module, following your platform-specific needs:

JVM

```kotlin
// JVM implementation
actual fun platformModule() = module {
    single {
        val driver = JdbcSqliteDriver(...)
        PeopleInSpaceDatabase(driver)
    }
    single { Java.create() }
}
```

iOS

```kotlin
// iOS implementation
actual fun platformModule() = module {
    single {
        val driver = NativeSqliteDriver(...)
        PeopleInSpaceDatabase(driver)
    }
    single { Darwin.create() }
}
```

Starting Common & Native Modules

After defining your common Koin modules and implementing the necessary native modules, initiate dependency initialization by invoking your native module from the shared code:

```kotlin
// in common shared code
fun initKoin() =
    startKoin {
        modules(commonModule(), platformModule())
    }
```

Kotlin Multiplatform (KMP) Cheat Sheet

Injecting Dependency - Wrapper Strategy

One way to begin injecting Koin dependencies into your iOS code is by creating a designated function to retrieve your dependency. In this example, we'll define `getKMMViewModel` to be used in your Swift code:

```kotlin
// in Common code
object KotlinDependencies : KoinComponent {
    fun getKMMViewModel() = getKoin().get<KMMViewModel>()
}
```

Here's how you can implement this in Swift:

```swift
// use KotlinDependencies in Swift
class ObservableViewModel: ObservableObject {
    private var vm: KMMViewModel?

    func activate() {
        let vm = KotlinDependencies.shared.getKMMViewModel()
    }
}
```

Injecting Dependency - Koin Component Strategy

An alternative method involves using the `KoinComponent` interface to inject properties into a class that will be manually instantiated:

```kotlin
// Define a Common Component
class KMMRepository : KoinComponent {

    // Inject properties
    private val KMMRemoteApi: KMMRemoteApi by inject()
}
```

Utilizing Koin in the Android/JVM realm allows for direct benefits such as constructor injection or the use of the `inject` extensions, which enables Koin to retrieve the necessary dependencies:

```kotlin
// in Android ViewModel
class KMMViewModel(
    private val KMMRepository: KMMRepository
) : ViewModel()
```

In Swift iOS, we can use it in a native component:

```swift
// In Swift
class KMMViewModel {

    private let repository: KMMRepository
    init(repository: KMMRepository) {
        self.repository = repository
    }
}
```

Injecting with ViewModels and KMM-ViewModel

You can declare a ViewModel, usage across Android and iOS with KMM-ViewModel.

For this, incorporate the `KMMViewModel()` class for your ViewModel:

```kotlin
open class ViewModelShared : KMMViewModel(), KoinComponent {
    private val repository: Repository by inject()
}
```

Then, declare it in your Android module and use it in the "classical" way, with `by viewModel` :

```kotlin
// android
val appModule = module {
    viewModel { ViewModelShared() }
}

// in Android
val viewModel by viewModel<ViewModelShared>()
```

To achieve this in iOS, simply implement the code below:

```swift
// iOS
struct ContentView : View {
    @StateViewModel var viewModel = ViewModelShared()

}
```

Injecting iOS dependencies in Koin

You may need to pass data from the iOS platform to the Koin dependencies. The best is always to rely on KMP Libraries, to have KMP API ready to be used for your code.

Sample code for providing `NSUserDefaults` to the KMP Settings library is shown below:

```kotlin
// Init Koin for iOS
fun initKoinIos(
    // iOS settings
    ud: NSUserDefaults
): KoinApplication = initKoin(

    module {
        single<Settings> { NSUserDefaultsSettings(ud) }
    }
)
```

To call the Kotlin function from iOS, ensure that you use the appropriate native object parameter:

```kotlin
func startKoin() {
    let userDefaults = UserDefaults(suiteName: "KAMPSTARTER_SETTINGS")!

    let koinApplication = KoinIOSKt.doInitKoinIos(
        userDefaults: userDefaults
    )
}
```