Isolated Context with Compose

In case of applications like SDK & White Labels, you may need to isolate your application from a consumer application that will use your features.

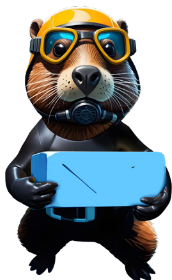
To isolate your context you need to declare a specific variable for you context, with `koinApplication`. Like here:

```
// Hold in an Object, Dedicated holder…
val IsolatedContext = koinApplication {
    // Context Config
}.koin
```

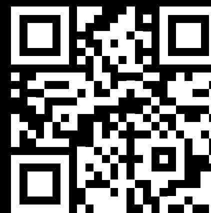
Now use this context to wrap your Composable content, using `KoinIsolatedContext` and your Koin isolated context variable:

```
@Composable
fun IsolatedApp() {

    KoinIsolatedContext(IsolatedContext) {
        // App Content ...
    }
}
```



Insert-koin.io



Koin Compose - Koin 3.5

Cheat Sheet

Now Available
Koin 3.5 Long-Term Support: offering professional support, updates, bug fixes, and security patches for 18+ months.

Koin is developed by Kotzilla and open-source contributors

kotzilla.io

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

Set Up Compose and Koin

Your project configuration can be done with `koin-androidx-compose` packages for an Android application, or `koin-compose` package if you are using it for a Kotlin Multiplatform project.

```
// Koin for Jetpack Compose
io.insert-koin:koin-androidx-compose
// Koin for Jetpack Compose Navigation
io.insert-koin:koin-androidx-compose-navigation
// Koin for Compose Multiplatform
io.insert-koin:koin-androidx-compose
```

Look at insert-koin.io website for the latest versions.

You can either use the `Koin DSL` or `Koin Annotations` to configure your application. The following API lets you inject your dependencies into Composable functions

Injecting inside a Composable

A dedicated API for Compose is available, to let you benefit from the system features. To inject a dependency inside a Composable function, you can use inject with `koinInject` as parameter:

```
@Composable
fun MyComposable(
    myFactory : MyFactory = koinInject()
) {

    // ...
}
```

or in the function body:

```
@Composable
fun MyComposable(
) {

    val myFactory = koinInject<MyFactory>()
}
```

Recomposition

By default the use of `koinInject` allows your dependency to benefit from the Compose cache, and won't trigger any recomposition when using it.
If you want to allow recomposition while resolving a dependency, you need to trigger it by a dynamic parameter by using `parametersOf`

```
@Composable
fun ClickCounter(
    clicks: Int,
    onClick: () -> Unit
) {
    // Dynamic parameter injection into MyFactory instance
    val myFactory = koinInject<MyFactory> { parametersOf("$clicks") }
}
}
```

Injecting a ViewModel

Inject a ViewModel component inside a Composable is also very easy. Just use `koinViewModel` function in your Composable parameter or body.

```
@Composable
fun MyComposable(
    myVM : MyViewModel = koinViewModel()
) {
    // ...
}
```

Note that a ViewModel instance is bound to underlying lifecycle component (Activity, Fragment …), not to the Compose cache system.

ViewModel or State Holder?

What to choose between a ViewModel or a factory

Component State ~ State Holder

-

Screen Level State ~ ViewModel

State hoisting: This is a pattern where you lift the state up to a higher-level Composable and pass it down to child Composables as parameters. This allows you to manage the state in a more centralized manner. This implies to use a `factory` instance to be injected to your Composable, to hold your state.

ViewModel: You can use the ViewModel architecture component to manage state in Jetpack Compose applications. ViewModels provide a way to store and manage UI-related data in a lifecycle-aware manner.

Running Koin Application from Compose

A Koin application context can be started directly from a Composable function, with `KoinApplication` component like this:

```
@Composable
fun MyComposable() {

    KoinApplication(application = { /* Koin Config ... */ }) {

        // Content running under Koin context
    }
}
```

This allows your Composable function to behave like an application starting point, and pass your Koin configuration. Note that if you are already using `startKoin` you don't need to use KoinApplication component.

Preview Composable with Koin Content

A good usage of `KoinApplication` is to help preview content that is relying on Koin container. Combine it with `@Preview` annotation, this will help you create a Koin context for this Composable. Having a dedicated "test module" for it is also a good practice:

```
@Preview
@Composablefun MyPreviewComposable() {

    KoinApplication(application = { modules(testModule) }) {

        // Content running under Koin context
        MyComposable(...)
    }
}
```

Dynamic Modules Load

Compose and Koin can together help you load Koin modules thanks to the `rememberKoinModules` function. This will trigger Koin module load at first creation of your Composable.

```
@Composable
fun App() {

    rememberKoinModules(modules = { listOf(appModule) })

            // content ...
}
```

Koin modules unload is can be also triggered when this Composable, making this component as a dynamic entry point for your app. Use the `unloadModules` parameter to set the behavior. The `unloadOnForgotten` and `unloadOnAbandoned` can allow to let you drop modules on the right moment.