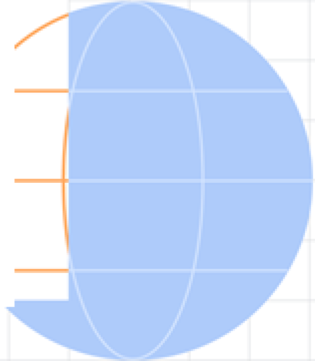
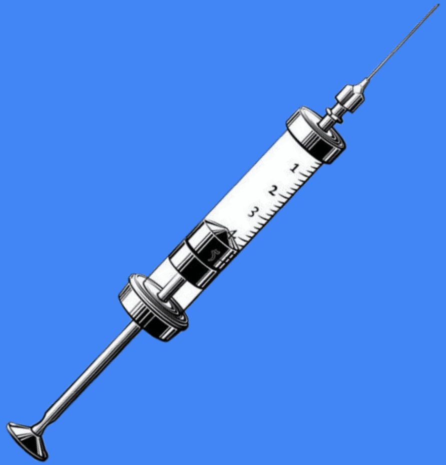
GDG



Dependency injection

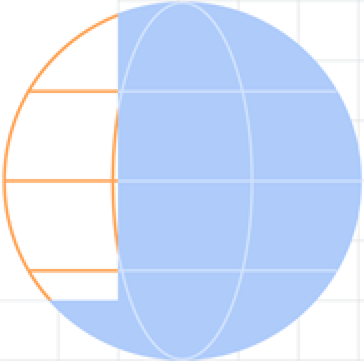
Don't let dependencies drag you down. Inject them up!



Content

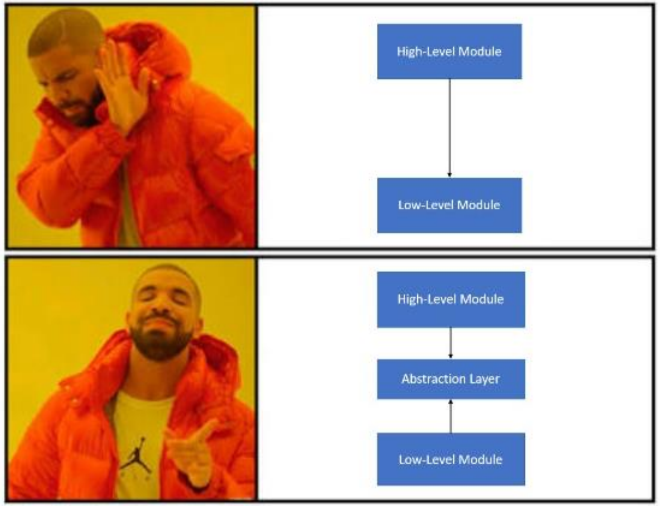
- Intro
- Manual DI implementation
- Intro to Koin
- Koin implementation
- Best practices & common pitfalls
- Alternatives
- Conclusion

Intro



Intro

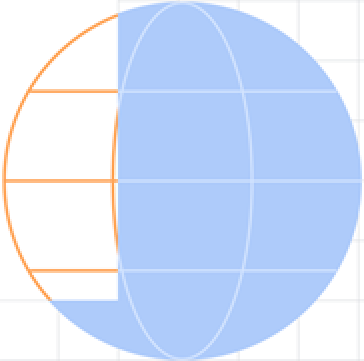
- Dependency – **direct relationship between two classes**; i.e. one relies on another for its functionality
- Dependency injection – implementation technique for **populating instance variables of a class**
- Based onto **Dependency Inversion Principle** (SOLID) – high-level modules shouldn't depend on low-level modules, both **should depend on abstractions**



Intro

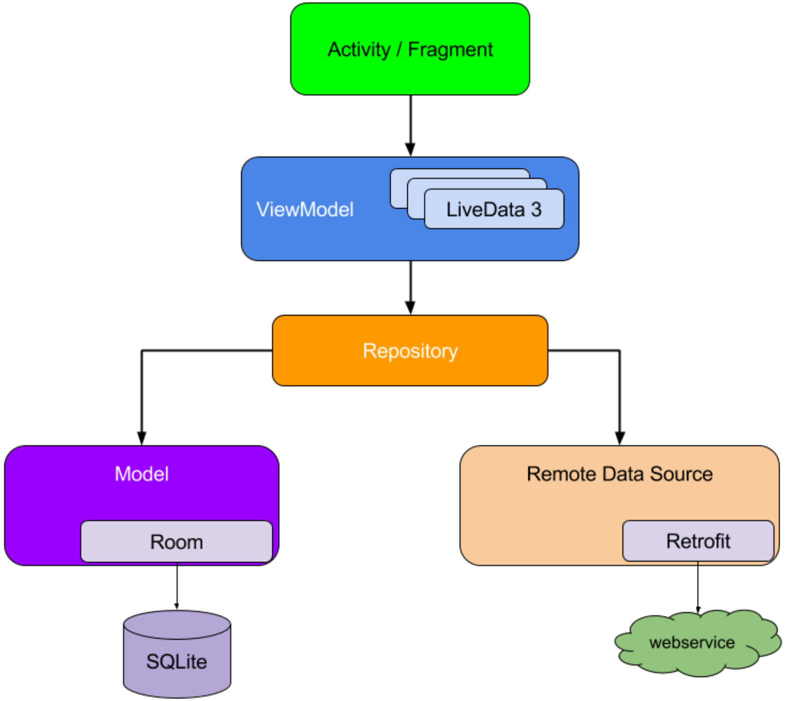
- 4 roles of DI:
  - **Service** – class that carries some functionality
  - **Client** – class that requests something from a service
  - **Interface** – abstract component implemented by a service for use by one or more clients
  - **Injector** – component that introduces a service to a client; i.e. creating and inserting a service into a client
- Types of injection:
  - **Constructor injection** (clear glance on required dependencies)
  - Public property/field injection
  - Method injection

Manual DI implementation

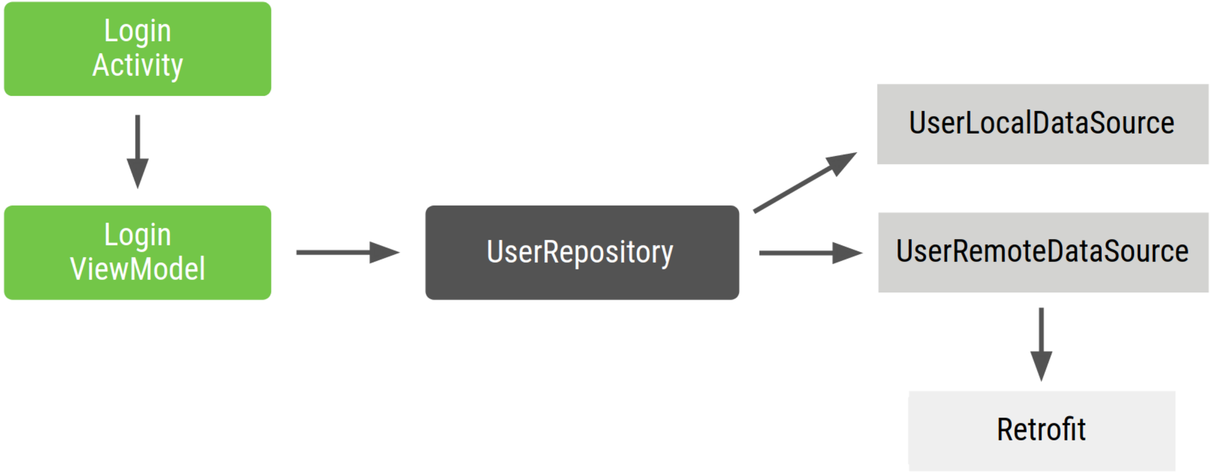


Manual DI implementation

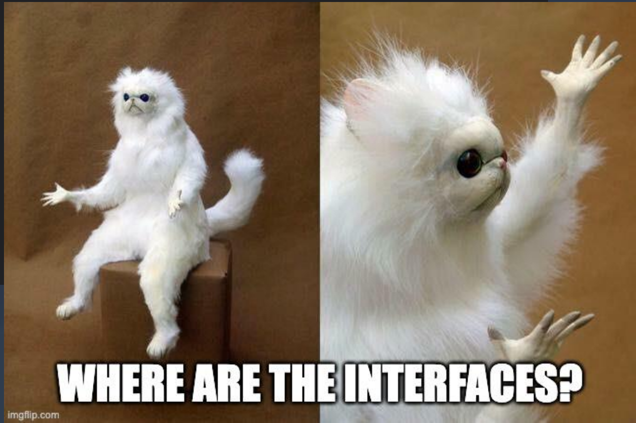
Manual DI implementation

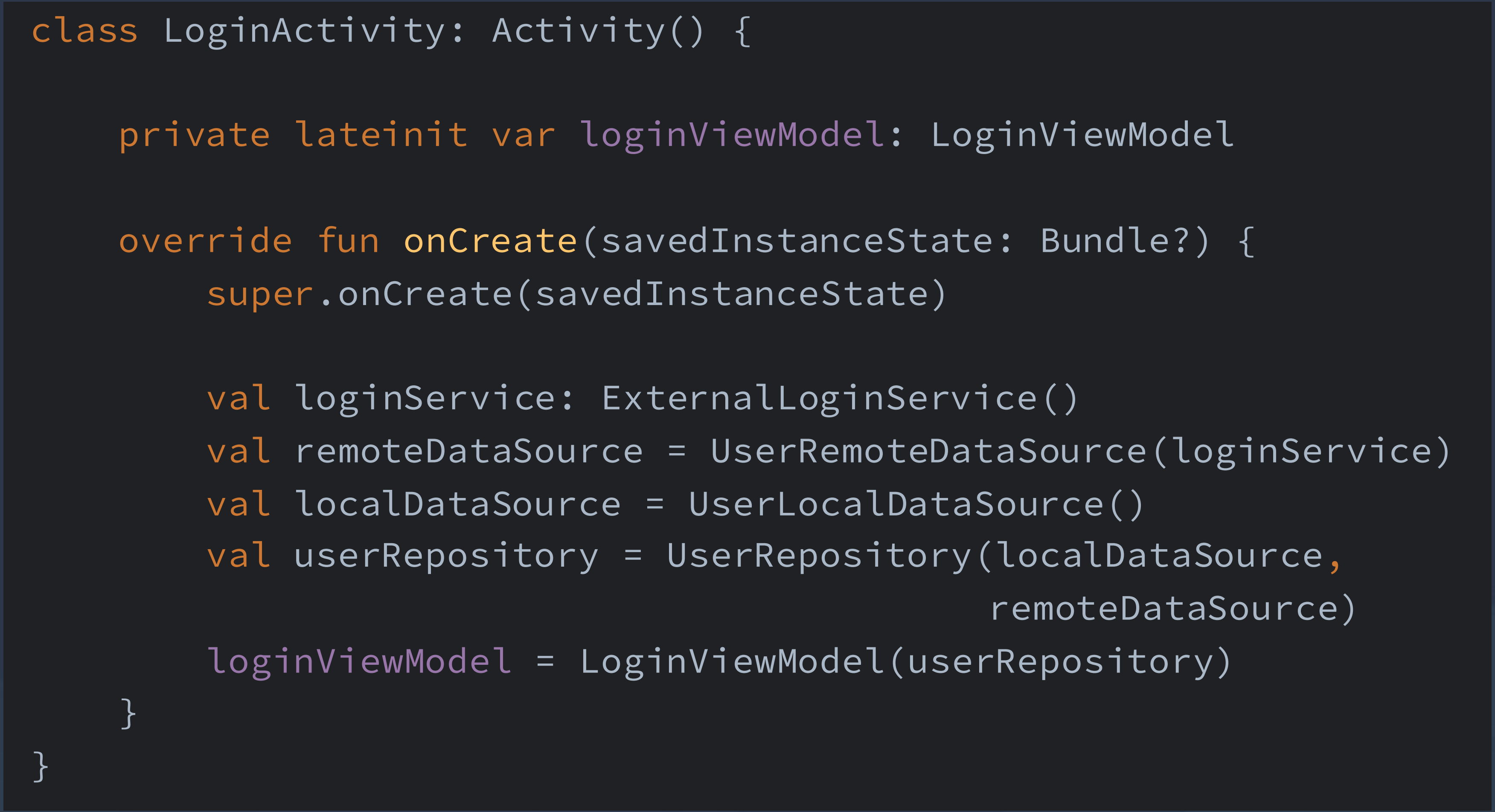


Manual DI implementation



```kotlin
class LoginViewModel(
    private val userRepository: UserRepository
) { ... }

class UserRepository(
    private val localDataSource: UserLocalDataSource,
    private val remoteDataSource: UserRemoteDataSource
) { ... }

class UserLocalDataSource { ... }
class UserRemoteDataSource(
    private val loginService: ExternalLoginService
) { ... }
```





```
class LoginActivity: Activity() {

    private lateinit var loginViewModel: LoginViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val loginService: ExternalLoginService()
        val remoteDataSource = UserRemoteDataSource(loginService)
        val localDataSource = UserLocalDataSource()
        val userRepository = UserRepository(localDataSource,
                                            remoteDataSource)
        loginViewModel = LoginViewModel(userRepository)
    }
}
```

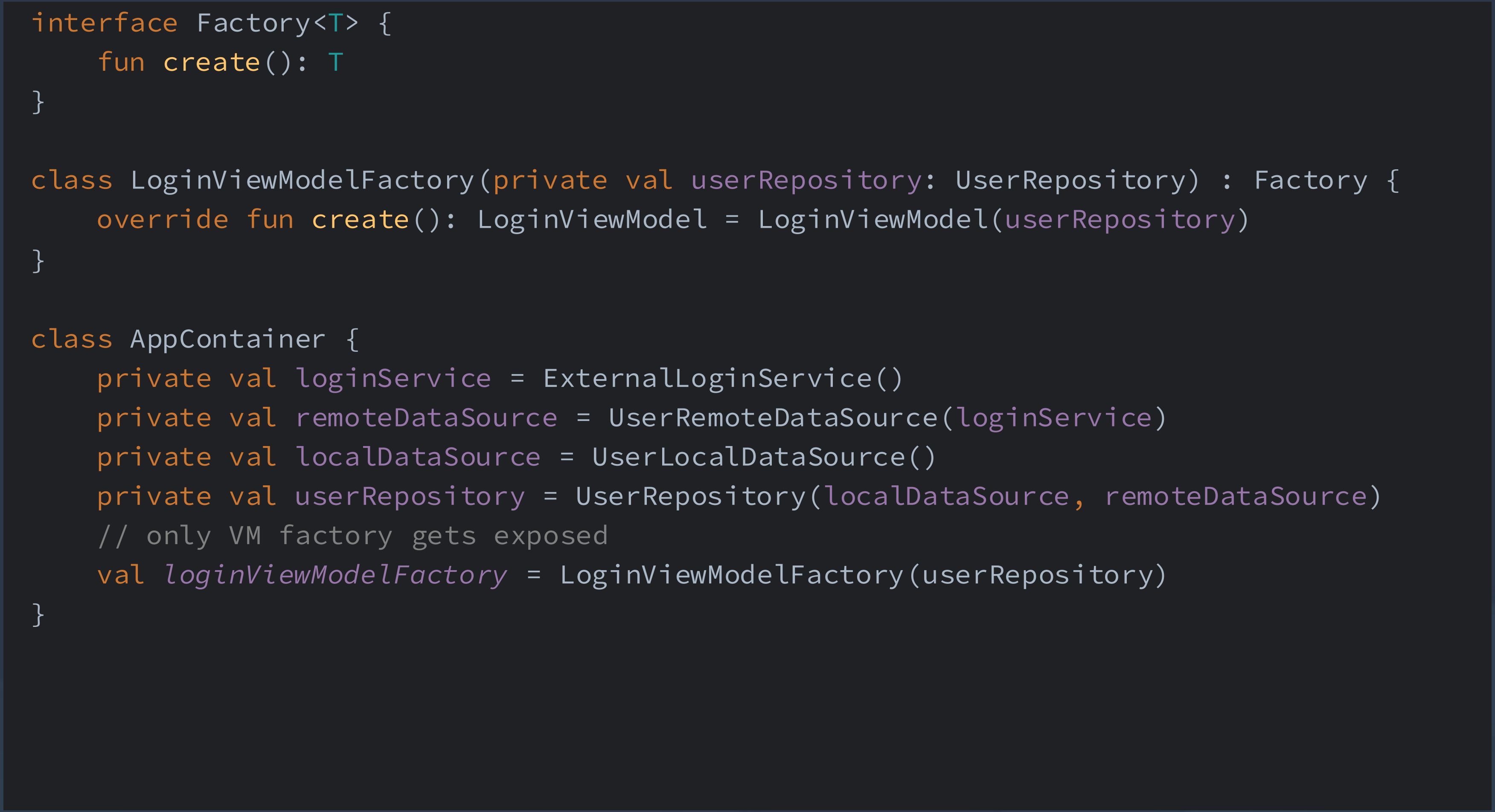
Why this sucks?

- **Lot of boilerplate code** – duplication of the same code pattern for each feature or screen
- **Dependencies have to be declared in order** – you can't instantiate a ViewModel object before Repository object
- **Difficult to reuse object** – to share Repository across multiple features you need to construct the Repository following the singleton pattern (deprecated) + makes testing more difficult because all tests share the same singleton instance

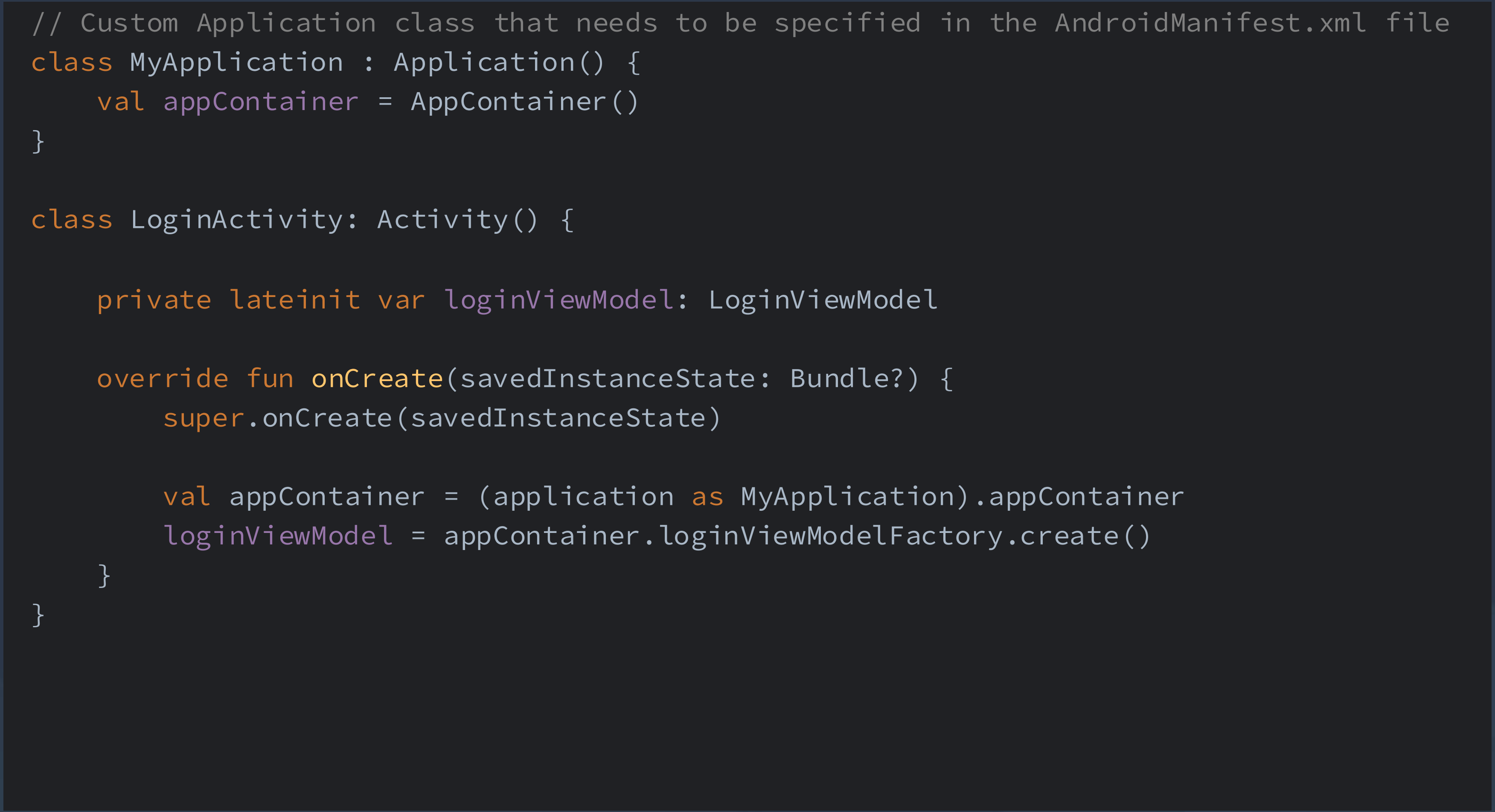
There are ways to improve it

- Manage dependencies with a **dependency container**
- Implement **factory pattern** for instantiation of multiple objects of the same type – example: you need a viewmodel in more places in the app

- Use an **DI framework** to simplify your development



```kotlin
interface Factory<T> {
    fun create(): T
}

class LoginViewModelFactory(private val userRepository: UserRepository) : Factory {
    override fun create(): LoginViewModel = LoginViewModel(userRepository)
}

class AppContainer {
    private val loginService = ExternalLoginService()
    private val remoteDataSource = UserRemoteDataSource(loginService)
    private val localDataSource = UserLocalDataSource()
    private val userRepository = UserRepository(localDataSource, remoteDataSource)
    // only VM factory gets exposed
    val loginViewModelFactory = LoginViewModelFactory(userRepository)
}
```



```kotlin
// Custom Application class that needs to be specified in the AndroidManifest.xml file
class MyApplication : Application() {
    val appContainer = AppContainer()
}

class LoginActivity: Activity() {

    private lateinit var loginViewModel: LoginViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val appContainer = (application as MyApplication).appContainer
        loginViewModel = appContainer.loginViewModelFactory.create()
    }
}
```

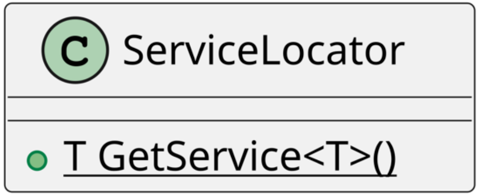
Drawbacks for this approach

- AppContainer quickly gets complicated when you want to include more functionality into your app
- Optimizing the AppContainer can be difficult and tiring – deleting or updating the whole dependency chain
- Resulting into **harder maintenance**, **reduced testability**, larger time consumption on **boilerplate code**, **error-prone**, **limited flexibility**, *depression and lack of motivation for developing*

Intro to Koin

Intro to Koin

- DI framework which uses Kotlin DSL (Domain Specific Language) with **declarative approach**
- Popular because of its **simplicity** and support for Kotlin Multiplatform
- Latest stable version: v4.0.1
- Similar to service locator pattern (registry of available services ready to be requested for usage) but there are differences:
    - Koin defines its own **modules** instead of a **single static registry**
    - Dependencies are **tightly coupled** to the locator, Koin uses **loose coupling** due to modules and constructor injection



C ServiceLocator

T GetService<T>()

Koin terminology

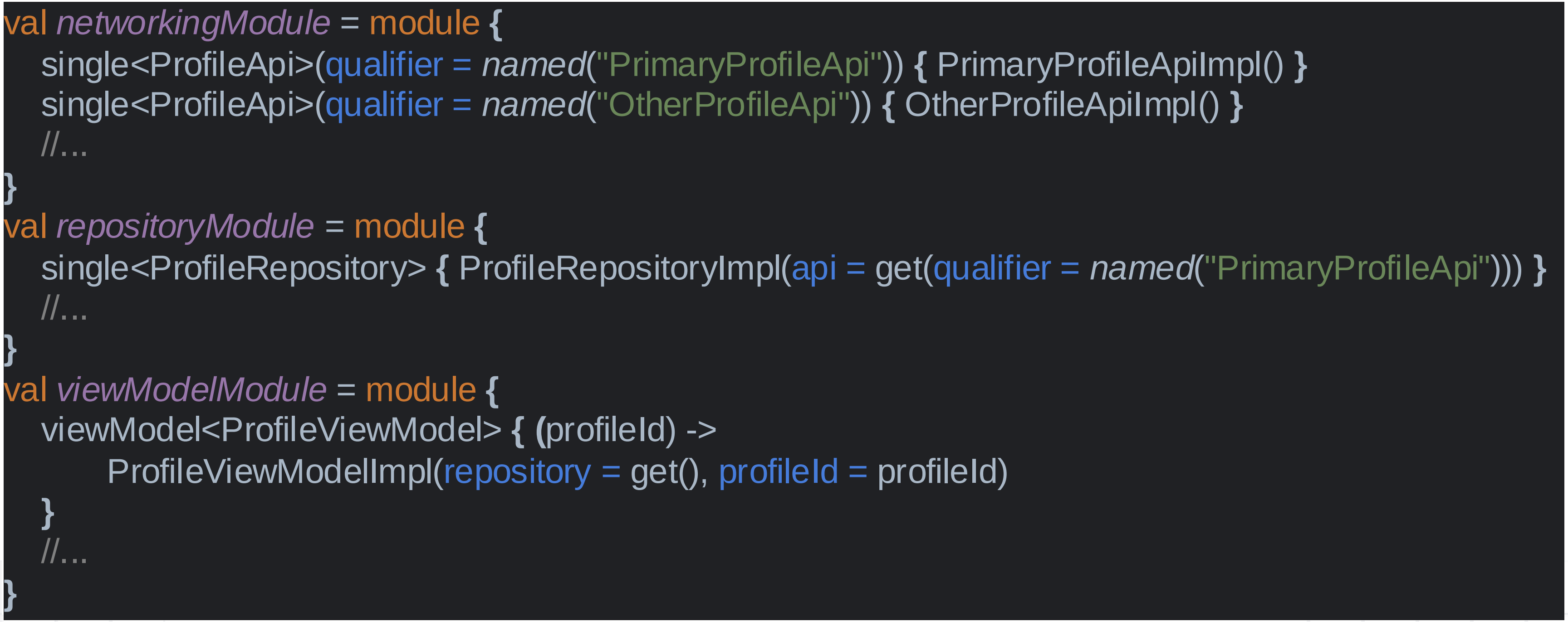
- Module - a **container** for defining dependencies and their relationships
- Definition - a **declaration of a dependency** within a module. Definitions specify how to create or retrieve an instance of a particular type
- Scope - a mechanism for **controlling the lifecycle** of dependencies
  - single – singleton dependency (one instance across the application)
  - factory - a factory dependency, meaning a new instance is created each time it's requested
  - viewModel - a ViewModelScope instance for managing the lifecycle of coroutines within a ViewModel
- get() – resolves dependencies **inside module**
- inject() – retrieving dependency instances **from a module**
- Qualifier – **name of the definition**, differentiation between definitions of the same type

Koin implementation

Add Koin to project

```
dependencies {
    // ...
    val koinVersion = ”4.0.1”

    // Koin core features
    implementation("org.koin:koin-core:$koinVersion")

    // Koin Android features
    implementation("org.koin:koin-android:$koinVersion")
    implementation("org.koin:koin-androidx-viewmodel:$koinVersion")
    // ...
}
```

Define modules and dependencies

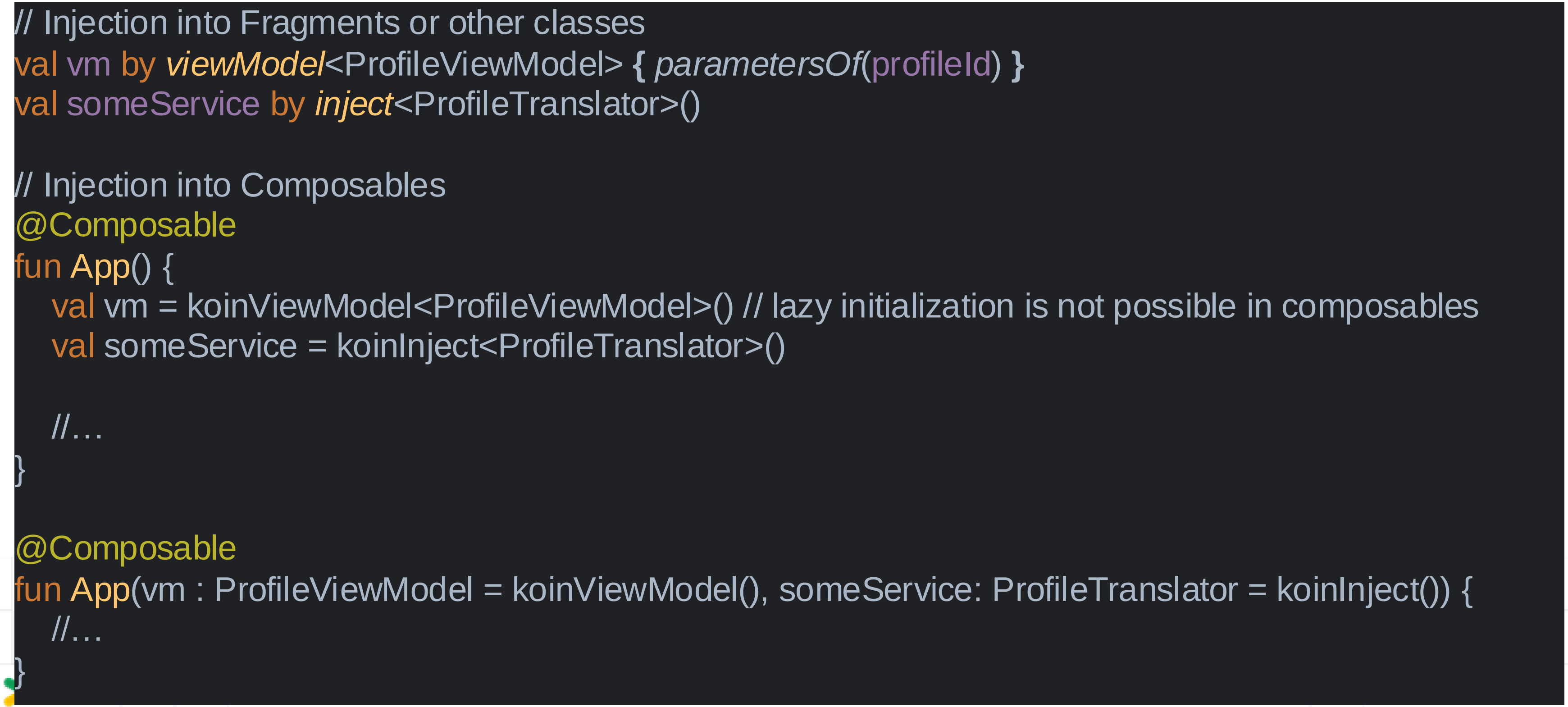


```
val networkingModule = module {
    single<ProfileApi>(qualifier = named("PrimaryProfileApi")) { PrimaryProfileApiImpl() }
    single<ProfileApi>(qualifier = named("OtherProfileApi")) { OtherProfileApiImpl() }
    //...
}
val repositoryModule = module {
    single<ProfileRepository> { ProfileRepositoryImpl(api = get(qualifier = named("PrimaryProfileApi"))) }
    //...
}
val viewModelModule = module {
    viewModel<ProfileViewModel> { (profileId) ->
            ProfileViewModelImpl(repository = get(), profileId = profileId)
    }
    //...
}
```

Initialize Koin

```
class MyApplication : Application() {
    override fun onCreate() {
        super.onCreate()
        startKoin {
            androidContext(this@MyApplication)
            modules(networkingModule, repositoryModule, viewModelModule)
        }
    }
}

// Inside Manifest.xml
<manifest>
    <application
        android:name=".MyApplication” … />
</manifest>
```

Components injection

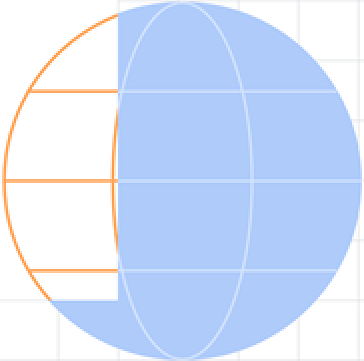


```kotlin
// Injection into Fragments or other classes
val vm by viewModel<ProfileViewModel> { parametersOf(profileId) }
val someService by inject<ProfileTranslator>()

// Injection into Composables
@Composable
fun App() {
    val vm = koinViewModel<ProfileViewModel>() // lazy initialization is not possible in composables
    val someService = koinInject<ProfileTranslator>()

    //…
}

@Composable
fun App(vm : ProfileViewModel = koinViewModel(), someService: ProfileTranslator = koinInject()) {
    //…
}
```

Practical example

Waiting for Android Studio to load up

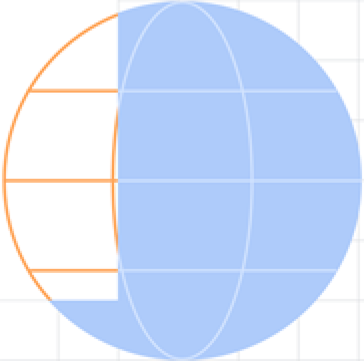
Best practices & common pitfalls



Best practices & common pitfalls

- Best practices:
  - **Separate responsibilities** – organize dependencies into layered modules (feature or functionality based)
  - Write clear **unidirectional dependencies** – avoid circular references
  - Use property injection with **lazy injection** technique where possible
- Common pitfalls:
  - Incorrect scoping - overuse of Singletons
  - Tight Coupling
  - Unclear Module Structure

Alternatives



Alternatives

- **Dagger** – relies on compile-time code generation, processes annotations during compilation to generate the DI code for object instantiation and injection

- **Hilt** – built on top of Dagger with more simplified and Android-oriented (pre-defined) annotations

Alternatives comparison

Feature	Koin	Dagger	Hilt
Approach	Runtime service locator	Compile-time code generation	Inherits from Dagger
Learning curve	Easy	Steep	Moderate
Boilerplate code	Minimal	High	Medium
Setup	Simple, declarative	Complex (requires component definitions)	Simplified setup than Dagger (predefined components and scopes)
Compile time	None	Increases (can be significant)	Increases (less than Dagger)
Debugging	Easier	Complex (code generation)	Easier than Dagger

Conclusion

Conclusion

- Dependency Injection promotes loose coupling, testability, and maintainability in your app
- Relying on interfaces makes your code clean 🧹
- Smaller or bigger project, Koin does the job 💪
- If you are not familiar with SOLID principles, check it out 👀

Literature

Google Developers

GDG

Thanks! :]

Questions?

Martin Zagoršćak
Android & KM dev @ Endava
@martin.zagorscak

Homework

Implement DI system into your existing projects by following best practices.

Bonus: Study how Hilt works and try it out.