Sobre o palestrante.

Carlos, 29 anos, desenvolvedor de Android desde 2014, sempre focando em Mobile Nativo em Java e Kotlin. Trabalhou nas empresas Livetouch, Cognizant, Bradesco e Ifood.

Tem seu blog no medium:
https://medium.com/@nicolaugalves/android-cwb-c4b4483b24e5

Assuntos de interesse: Clean Code, Unit Test, DI, Design Patterns, Architectural Patterns, Responsibility of layers, Functional programming, Object-Oriented Programming.

Injeção de Dependência no Mundo Android

Uma introdução ao assunto, buscando demonstrar
de uma maneira mais visual como funciona.

Injeção de dependência é um Design Pattern que tem como objetivo remover o acoplamento de classes. Controlar quando instanciar um novo objeto. Ajudando também nos testes unitários.

Por exemplo:

```
class Car {
    private val engine = Engine()

    fun start() {
        engine.start()
    }
}
```

https://github.com/nicconicco/googlecertificationkotlin2019/

Injeção manual:

```
class CarInjectedNative(private val engine: Engine) {

    fun start() {
        engine.start()
    }
    ...
}
```

https://github.com/nicconicco/googlecertificationkotlin2019/

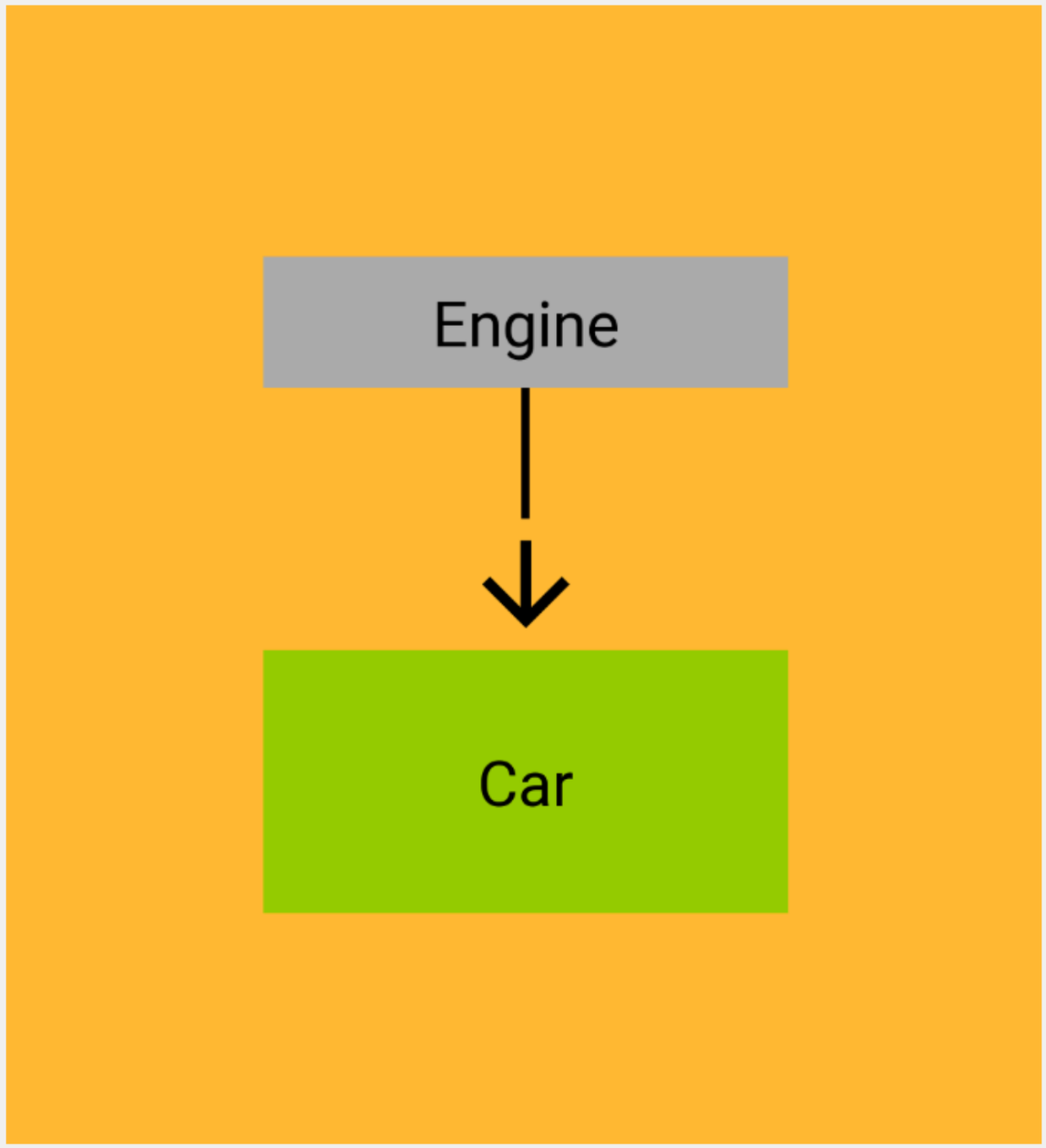
```kotlin
// Sem injeção nativa
val car = Car()

car.start()


// Com Injeção nativa
val engine = Engine()
val carInjected = CarInjectedNative(engine)

carInjected.start()
```

https://github.com/nicconicco/googlecertificationkotlin2019/



https://github.com/nicconicco/googlecertificationkotlin2019/

Recomenda-se criar interfaces da sua classe na passagem do construtor, pois assim fica mais fácil depois realizar os testes.

```kotlin
interface ModelCar {
    fun getModel() : String
}

class Car(
      private val model: ModelCar
) {
    fun showModelCar() =
model.getModel()
}
```

```kotlin
class CarStep2Test {
    private lateinit var car: Car

    @Test
    fun Car_ShowFerrariModel() {
        car = Car(Ferrari())

        assertEquals(car.showModelCar(), "Meu modelo é uma
Ferrari”)

    }

    @Test
    fun Car_ShowMercedes() {
        car = Car(Mercedes())

        assertEquals(car.showModelCar(), "Meu modelo é uma
Mercedes”)
    }
}

class Ferrari : ModelCar {
    override fun getModel(): String {
        return "Meu modelo é uma Ferrari"
    }
}

class Mercedes : ModelCar {
    override fun getModel(): String {
        return "Meu modelo é uma Mercedes"
    }
}
```

https://github.com/nicconicco/googlecertificationkotlin2019/

Em nosso dia a dia com arquiteturas diferentes, temos ao menos em padrões de projetos maiores sempre estes personagens presentes.

1. View
2. ViewModel
3. UseCase
4. Repository
5. Datasource
6. Worker

Se tentarmos adicionar eles no construtor do OnCreate da Activity, não vai funcionar.

```
class LoginDaggerActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?, loginViewModel: LoginViewModel) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_dagger)
    }
}
```

'onCreate' overrides nothing

Para isso, para não tocarmos nos nossos métodos do ciclo de vida do Android, surge o Dagger, a nossa caixa mágica que fará as injeções de dependências em nosso projeto.

Dagger!

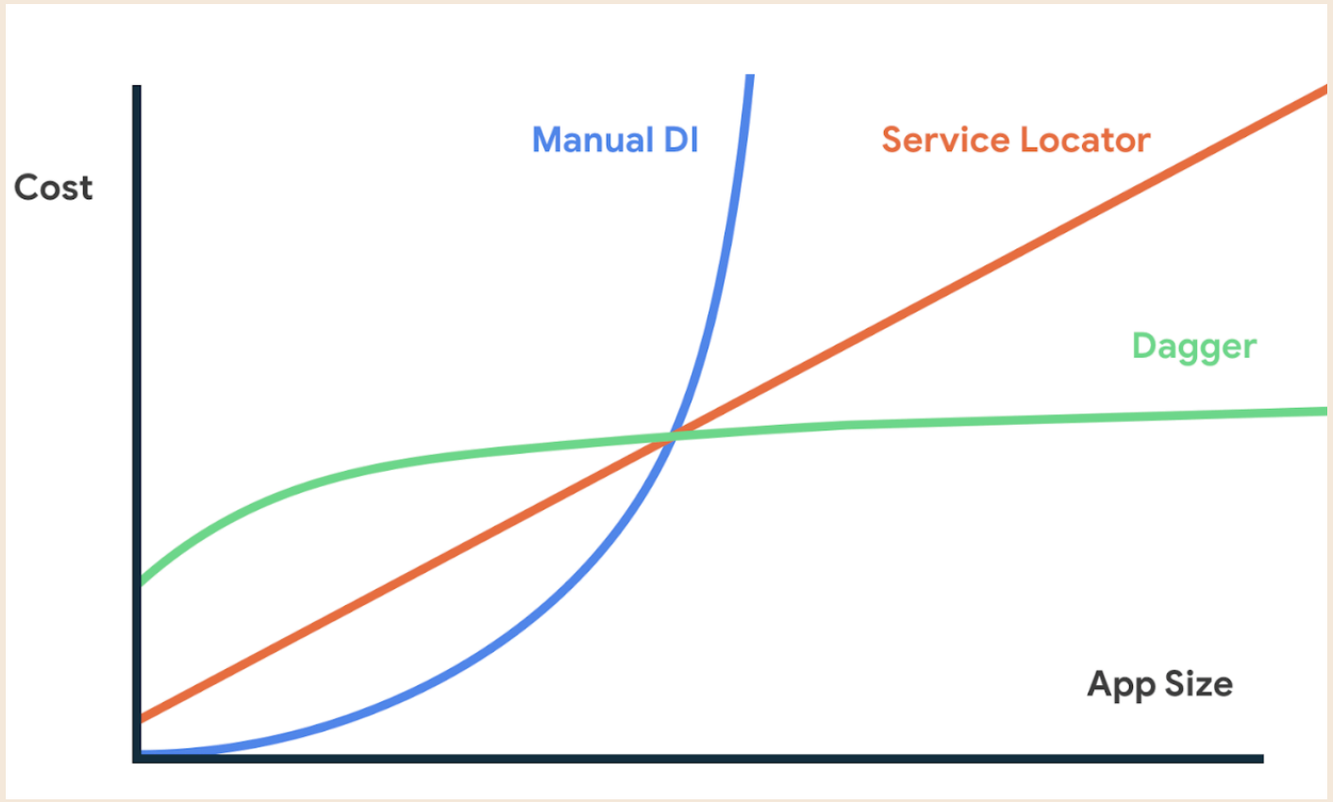
Vindo do Square, Dagger1, Dagger2 e agora Dagger.Android é a biblioteca oficial do Google até o momento desta apresentação.

https://medium.com/androiddevelopers/dependency-injection-guidance-on-android-ads-2019-b0b56d774bc2

Project Size - 3 - 4 - 8

Project Size	Small	Medium	Large
Tool to Use	Manual DI Service Locator Dagger	Dagger	Dagger

Learning curve



Configurando nosso app.build.gradle:

```
apply plugin: 'kotlin-kapt'

// Dagger
implementation 'com.google.dagger:dagger:2.24'
kapt ‘com.google.dagger:dagger-compiler:2.24'
```

Adicionando ao manifest o nosso Application:

```
<application
    android:name=".GoogleCertificationKotlinApplication"
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="GoogleCertificationKotlin"
    android:roundIcon="@mipmap/ic_launcher_round"
```

```
open class GoogleCertificationKotlinApplication : Application() {
    // Instance of the AppComponent that will be used by all the Activities in the project
    val appComponent: AppComponent by lazy {
        initializeComponent()
    }

    open fun initializeComponent(): AppComponent = DaggerAppComponent.factory().create(applicationContext)
}
```

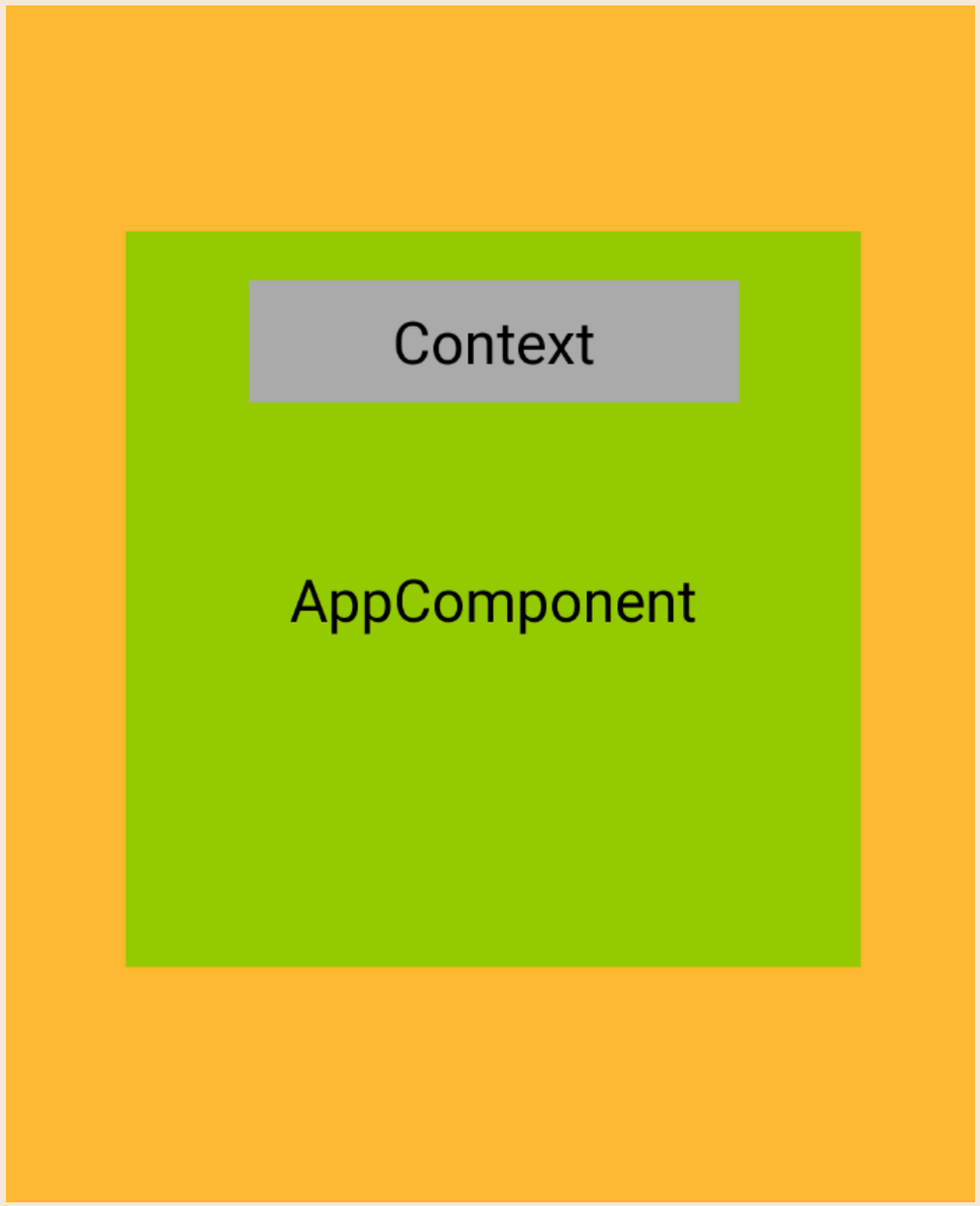
https://github.com/nicconicco/googlecertificationkotlin2019/

```kotlin
@Singleton
// Definition of a Dagger component that adds info from the different modules to the
graph

@Component()
interface AppComponent {

    // Factory to create instances of the AppComponent
    @Component.Factory
    interface Factory {
        // With @BindsInstance, the Context passed in will be available in the graph
        fun create(@BindsInstance context: Context): AppComponent
    }
}
```

https://github.com/nicconicco/googlecertificationkotlin2019/



https://github.com/nicconicco/googlecertificationkotlin2019/

```
// This module tells a Component which are its subcomponents
@Module(
    subcomponents = [
        LoginComponent::class
    ]
)
class AppSubcomponents
```

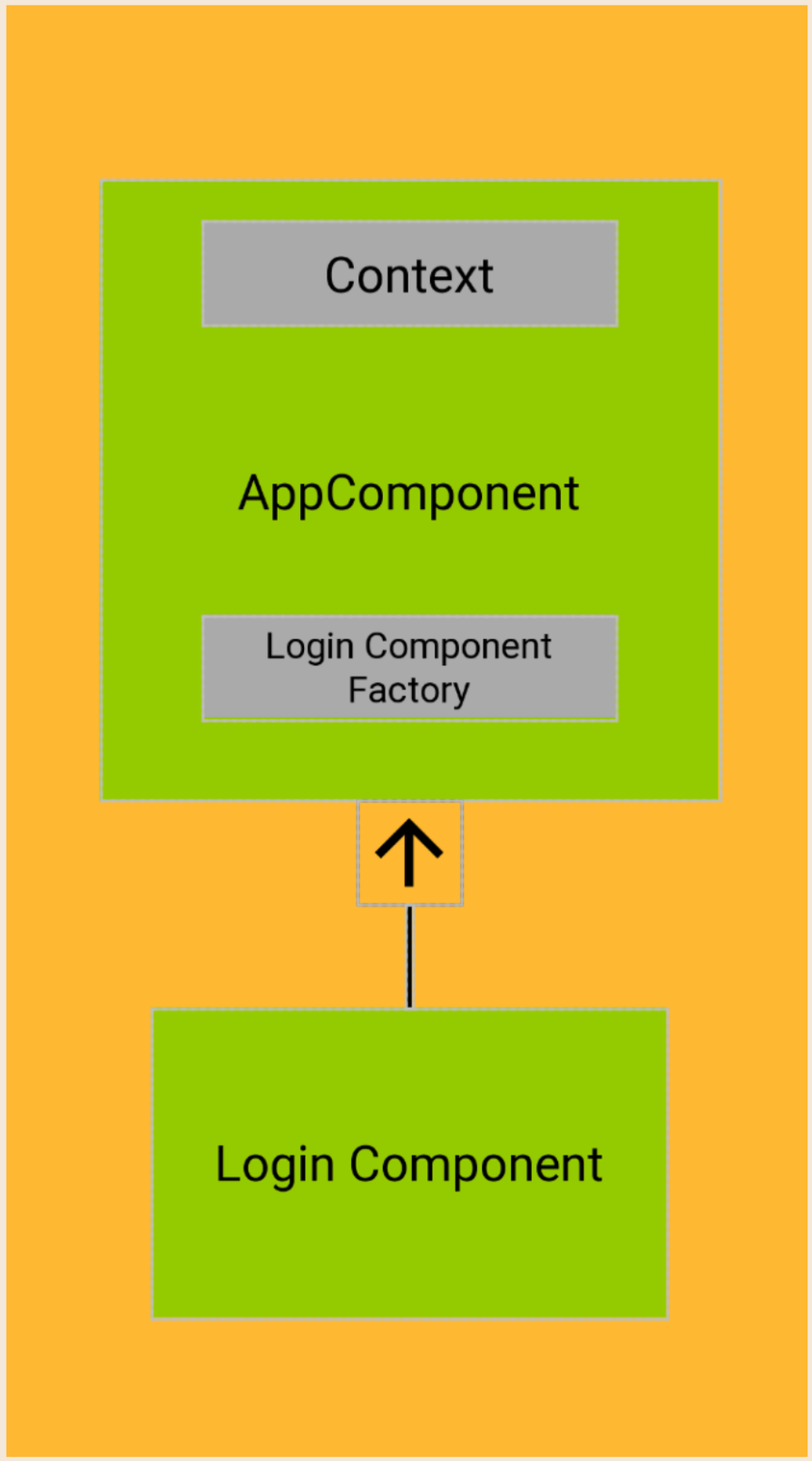
```
// Definition of a Dagger subcomponent
@Subcomponent
interface LoginComponent {
    // Factory to create instances of LoginComponent
    @Subcomponent.Factory
    interface Factory {
           fun create(): LoginComponent
    }
    // Classes that can be injected by this Component
    fun inject(activity: LoginDaggerActivity)
}
```

No AppComponent:

```
@Component(modules = [AppSubComponents::class])

fun loginComponent() : LoginComponent.Factory
```

https://github.com/nicconicco/googlecertificationkotlin2019/



https://github.com/nicconicco/googlecertificationkotlin2019/

```kotlin
@Singleton
// Adicionando o modulo de controle do Login
@Component(modules = [AppSubcomponents::class, LoginModule::class])
```

```kotlin
@Module(
    includes = [
        LoginViewModelModule::class,
        LoginRepositoryModule::class,
        LoginDataSourceModule::class
    ]
)
abstract class LoginModule
```

```kotlin
@Module
internal abstract class LoginDataSourceModule {
    @Binds
    internal abstract fun bindLoyaltyDataSourceModule(remote: RemoteLoginDataSourceImp): RemoteLoginDataSource
}
```

```kotlin
@Module
internal abstract class LoginRepositoryModule {
    @Binds
    internal abstract fun bindLoginRepositoryModule(repository: LoginRepositoryImp): LoginRepository
}
```

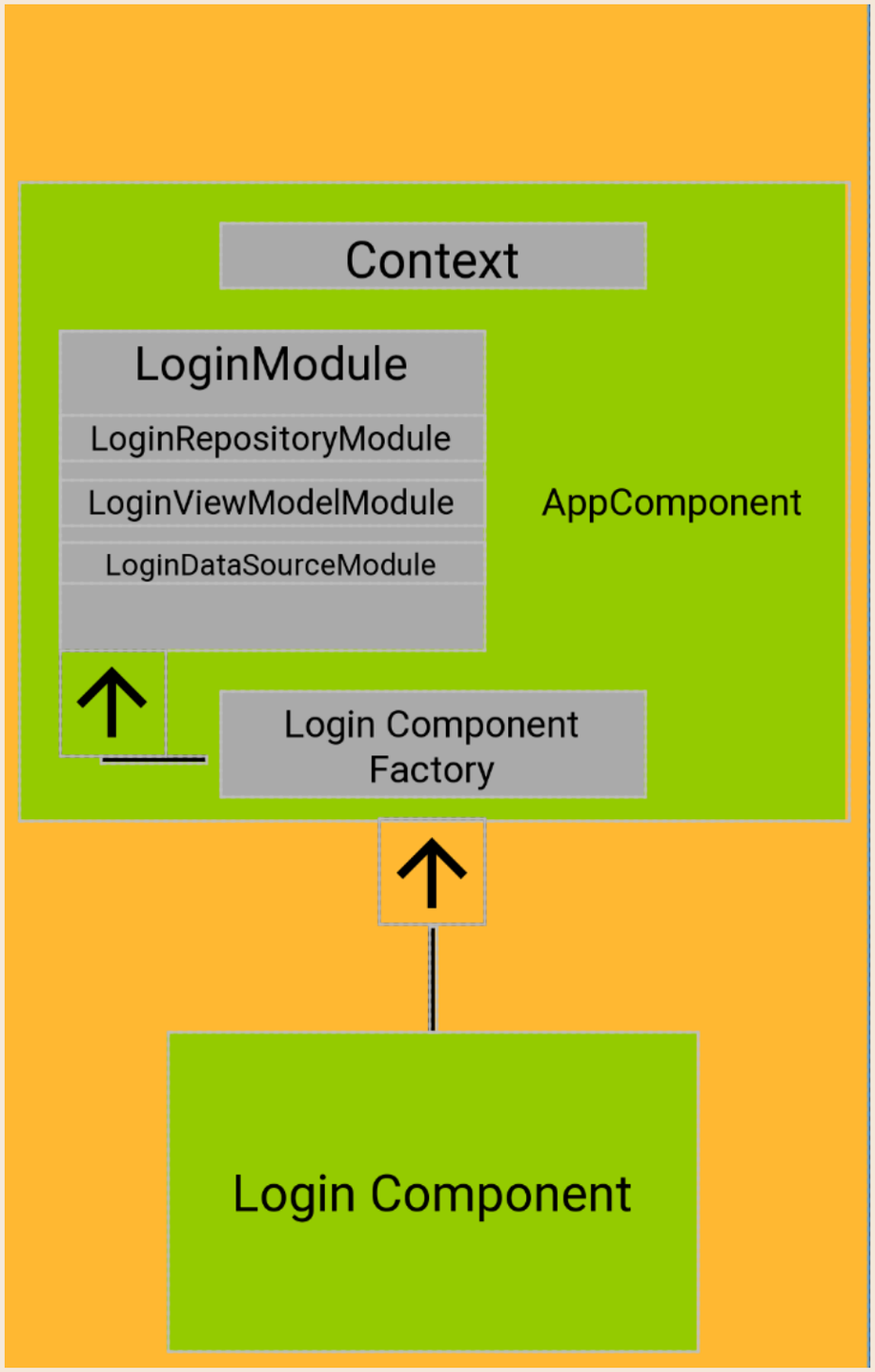
```kotlin
@Module
internal abstract class LoginViewModelModule {
    @Binds
    internal abstract fun bindLoginViewModelModule(viewModel: LoginViewModel): ViewModel
}
```

```kotlin
LoginDaggerActivity ->

@Inject
lateinit var loginViewModel: LoginViewModel
```

```kotlin
(application as GoogleCertificationKotlinApplication).appComponent.loginComponent().create()
.inject(this)
```

https://github.com/nicconicco/googlecertificationkotlin2019/



https://github.com/nicconicco/googlecertificationkotlin2019/

Vantagens:

Se criarmos desde o inicio de nosso projeto o controle das instâncias de nossas classes, fica mais fácil de garantir que não temos “muitas instancias da mesma coisa fazendo o mesmo trabalho que apenas uma conseguiria fazer”

- Código sendo reusado
- Fácil de refatorar
- Fácil de testar

Mais sobre,

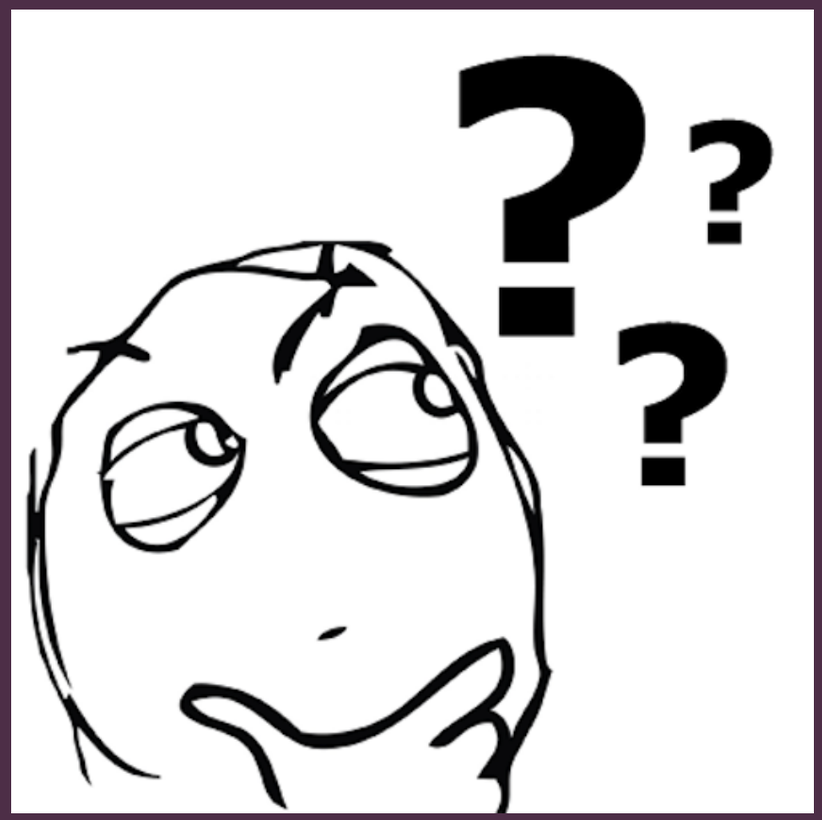
- Vai ser integrado com o Jetpack
- Facilidade de integração com o ViewModel
- Facilitar a vida no Kotlin
- Dagger Modulo
- Java / Kotlin

O google recomenda

"According to our introduction to Dependency Injection (DI), we believe you should always use DI principles in your applications.”

Mas e ai ?

Só tem Dagger para resolver o meu problema?



Singleton não resolve?

Singleton é uma instância viva que vai ficar viva enquanto seu app esta funcionando. O que pode deixar ele devagar (Acúmulos de Singletons onde não precisa). Dependência de injeção temos o controle do que é uma Fábrica para certos componentes, quando precisamos renovar essas fábricas e quando queremos uma única instância viva dessa classe sim com Singleton Pattern.

E o ServiceLocator?

Service Locator, também um Design Pattern tem o problema de uma classe ficar “inchada" com muitos objetos vivos e no fim sua classe de modulo vai ver mais do que precisa. Além das constantes repetições de tipos diferentes vivas como no exemplo Ferrari, Mereces. DI é uma injeção externa, eu digo lá na config que para aquele caso vou precisar disso, disso e acabou.

Koin!

Koin é uma lib alternativa que esta bombando também no mercado. Ela é escrita em Kotlin e isso da essa sensação de ser melhor (Já que é na linguagem nativa). Ele não usa anotações de processo e nem algum tipo de reflexão.

```
internal val appModule = module(override = true) {
    //region DataSource
    factory<GetUserDataSource> { GetUserDataSourceImpl(get(), get()) }
    //endregion

    //region AppProvider
    single<DispatcherProvider> { AppDispatcherProvider() }
    //endregion

    //region UseCase
    factory<GetUserUseCase> {
        GetUserUseCaseImpl(
            GetUserRepositoryImpl(
                GetUserDataSourceImpl(
                    get(),
                    get()
                )
            )
        )
    }
    //endregion

    //region Repository
    factory<GetUserRepository>
{ GetUserRepositoryImpl(GetUserDataSourceImpl(get(), get())) }
    //endregion

. . .
```

https://github.com/nicconicco/AndroidCWB/

```
. . .
    //region database
    single {
        Room.databaseBuilder(
            get(),
            AndroidCWBRoom::class.java,
            "database"
        )
            .build()
    }

    single { get<AndroidCWBRoom>().getUserDAO() }
    //endregion

    //region FirebaseFirestoreUtils
    single<FirebaseFirestoreUtils> { FirebaseFirestoreUtilsImpl(FirebaseFirestore.getInstance()) }
    //endregion

    //region InternetUtils
    single <InternetUtils>{ InternetUtilsImpl(get()) }
    //endregion

    //region ViewModel
    viewModel {
        LoginViewModelImpl(
            get(),
            get(),
            GetUserUseCaseImpl(
                GetUserRepositoryImpl(GetUserDataSourceImpl(get(), get()))
            ),
            AppDispatcherProvider().ui(),
            AppDispatcherProvider().io(),
            get()
        )
    }
    //endregion
}
```

https://github.com/nicconicco/AndroidCWB/

```
class AndroidCWBApplication : Application() {
    override fun onCreate() {
        super.onCreate()
        injectDependencies()
    }

    private fun injectDependencies() {
        startKoin {
            if (BuildConfig.DEBUG) androidLogger() else EmptyLogger()
            androidContext(applicationContext)
            modules(appModule) // Da pra separar por modulos sim.
        }
    }
    . . .
}
```

https://github.com/nicconicco/AndroidCWB/

Na Activity:

```
val loginViewModel: LoginViewModelImpl by viewModel()
```

No Construtor do ViewModel:

```
class LoginViewModelImpl(
    application: Application,
    db: AndroidCWBRoom,
    interactors: GetUserUseCase,
    mainDispacher: CoroutineDispatcher,
    ioDispacher: CoroutineDispatcher,
    internetUtils: InternetUtils
)
```

E assim para os demais casos...

https://github.com/nicconicco/AndroidCWB/

Perguntas?

Obrigado!

Carlos Nicolau Galves
carlos.galves@ifood.com.br