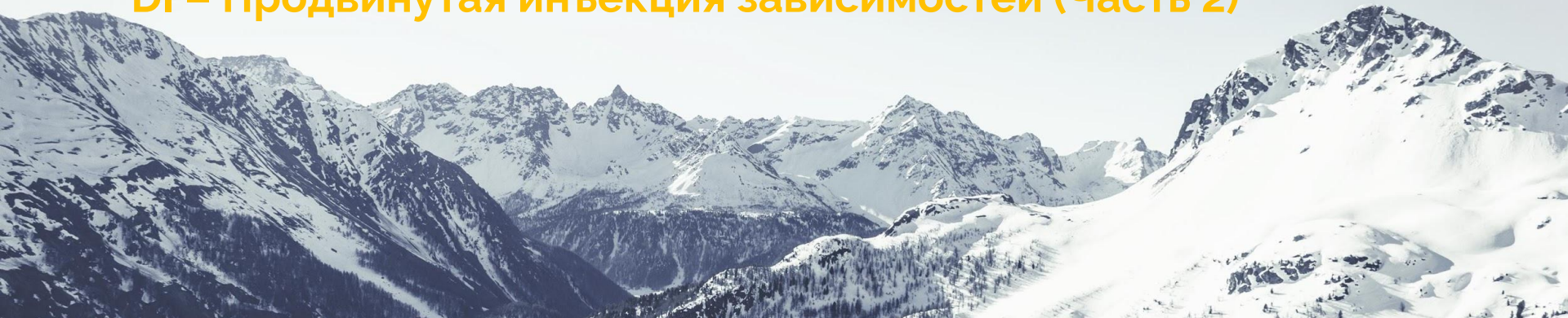
<TeachMeSkills/>

курс

Android разработчик

DI – Продвинутая инъекция зависимостей (Часть 2)



Агенда занятия:

@BindsInstance

@Qualifier

@Scope

@SubComponent

- @BindsInstance
  @Qualifier
  @Scope
  @SubComponent

@BindsInstance

Позволяет передавать объект (например, Context, параметры запуска, данные пользователя) в граф зависимостей на момент создания компонента, а не через модули.

Почему это удобно?

Это единственный способ безопасно инициализировать зависимости, значения которых доступны только во время сборки компонента (например, Context, аргументы Activity, параметры конфигурации).

Позволяет не создавать лишние модули для передачи простых значений.

@BindsInstance

```
@Component
interface AppComponent {
    fun inject(app: MyApp)
}

// Создание компонента с передачей context:
val component = DaggerAppComponent.builder()
    .context(applicationContext) // <-- @BindsInstance
    .build()



@Component.Builder
interface Builder {
    @BindsInstance
    fun context(context: Context): Builder

    fun build(): AppComponent
}
```

```
class SomeRepository @Inject constructor(
    private val context: Context // <-- будет подставлен через @BindsInstance
)
```

@Qualifier

Используется, когда в графе есть несколько реализаций одного типа (например, две строки, два репозитория), и нужно явно указать, какую именно внедрять.

Почему это важно?

Без Qualifier Dagger не поймет, какой из одинаковых типов подставлять, и выдаст ошибку “multiple bindings”.

@Qualifier

```
@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class ApiUrl

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class AuthUrl

@Module
class NetworkModule {
    @Provides @ApiUrl
    fun provideApiUrl(): String = "https://api.example.com"

    @Provides @AuthUrl
    fun provideAuthUrl(): String = "https://auth.example.com"
}

class ApiService @Inject constructor(
    @ApiUrl val apiUrl: String,
    @AuthUrl val authUrl: String
)
```

@Qualifier

Аннотация **@Named** — это стандартный Qualifier в Dagger/Dagger2, который помогает различать зависимости одного и того же типа.

Когда в вашем DI-графе есть несколько реализаций одного типа (например, две строки, два репозитория, два интеджера и т.д.), Dagger не сможет понять, какую именно зависимость вы хотите внедрить.
В этом случае используется аннотация **@Qualifier**, и самой простой ее реализацией является аннотация **@Named**.

@Scope

Позволяет ограничить жизненный цикл зависимости определённым “контекстом” (например, Singleton, Activity, Fragment).
Гарантирует, что в пределах одного компонента будет одна и та же инстанция зависимости.

Почему это важно?

Позволяет создавать синглтоны и другие “разделяемые” объекты в рамках компонента.
Защищает от случайного создания лишних экземпляров.

@Scope

```
@Scope
@Retention(AnnotationRetention.RUNTIME)
annotation class AppScope

@AppScope
@Component(modules = [AppModule::class])
interface AppComponent {
    fun getRepository(): Repository
}

@AppScope
class Repository @Inject constructor()
```

```
val appComponent = DaggerAppComponent.create()
val repo1 = appComponent.getRepository()
val repo2 = appComponent.getRepository()
// repo1 и repo2 — это один и тот же объект!
```

@Subcomponent

Позволяет создавать вложенные графы зависимостей, например, для Activity, Fragment, Feature-модуля.
Subcomponent может наследовать зависимости родительского компонента и добавлять свои.

Почему это важно?
Позволяет разделять зависимости по областям видимости (например, app-wide и activity-wide).
Позволяет реализовать scoping (разные “жизни” для зависимостей).

@Subcomponent

```
@Scope
@Retention(AnnotationRetention.RUNTIME)
annotation class ActivityScope

@ActivityScope
@Subcomponent
interface ActivityComponent {
    fun inject(activity: MainActivity)
}

@AppScope
@Component
interface AppComponent {
    fun activityComponent(): ActivityComponent.Factory
}

// Внутри Activity:
val activityComponent = (application as MyApp)
    .appComponent
    .activityComponent()
    .create()
activityComponent.inject(this)
```

```
@Subcomponent(modules = [ActivityModule::class])
interface ActivityComponent {
    ...
    @Subcomponent.Factory
    interface Factory {
        fun create(): ActivityComponent
    }
}
```

Component Dependencies

Component dependencies — это способ "делиться" зависимостями между разными Dagger-компонентами. Один компонент может получить доступ к зависимостям, предоставляемым другим (родительским) компонентом, не включая его модули напрямую.

Component Dependencies

Когда использовать?

Когда нужно разделить граф зависимостей на части (например, app-wide и feature-wide).

Когда нельзя или неудобно использовать Subcomponent (например, если компоненты не вложены друг в друга по жизненному циклу, или находятся в разных модулях проекта).

Как это работает?

Один компонент (Child) объявляет зависимость от другого (Parent).

Dagger позволяет внедрять зависимости из ParentComponent в ChildComponent.

Component Dependencies

```
// Родительский компонент
@Component(modules = [AppModule::class])
interface AppComponent {
    fun provideLogger(): Logger
    // ... другие зависимости
}

// Дочерний компонент, который зависит от AppComponent
@Component(
    dependencies = [AppComponent::class],
    modules = [FeatureModule::class]
)
interface FeatureComponent {
    fun inject(activity: FeatureActivity)
}
```

```
val appComponent = DaggerAppComponent.create()
val featureComponent = DaggerFeatureComponent.builder()
    .appComponent(appComponent)
    .build()

featureComponent.inject(this)
```

Теперь все зависимости, объявленные в AppComponent, доступны в FeatureComponent.

Отличие от Subcomponent

Subcomponent всегда получается из метода родителя и наследует его scope.

Component dependencies — компоненты независимы, но один может использовать зависимости другого через интерфейс.

Multibinding

Multibinding — это возможность добавлять в граф Dagger несколько объектов одного типа и автоматически собирать их в коллекцию (Set или Map).
Очень удобно, если у вас, например, несколько обработчиков событий, несколько ViewModel, несколько стратегий, и вы не знаете их список заранее.

Когда использовать?
Если нужно внедрить в класс набор реализаций, которые могут меняться или расширяться (например, плагины, стратегии, обработчики).
Часто используется для “плагинов”, “фич”, “интерцепторов” и т.д.

Multibinding

Примеры смотрим в проекте

Практика

Задача 1. @BindsInstance

Передайте applicationContext в repository через Dagger2

Задача 2.

В одном модуле создайте два разных источника данных:

- RemoteDataSource и LocalDataSource (оба реализуют интерфейс DataSource).
- Используйте кастомные qualifier-ы (@Remote, @Local) или @Named("remote")/@Named("local") для их различения.
- Внедрите оба источника в класс Repository через конструктор и выведите, какой источник используется при вызове метода.

Задача 3.

Создайте @Scope для объекта, который должен жить в течение всего времени работы приложения (@AppScope).

Проверьте, что при запросе зависимости из компонента вы получаете один и тот же объект (например, с помощью сравнения ссылок или логирования).

Задача 4. @Subcomponent

- Реализуйте AppComponent и дочерний ActivityComponent как subcomponent.

- Пусть в ActivityComponent внедряется зависимость ActivityLogger, а в AppComponent — AppLogger.

- Покажите, что ActivityLogger не существует вне ActivityComponent, а AppLogger доступен в обоих.

Задача 5. Component dependencies

Реализуйте два независимых компонента:

- NetworkComponent (отвечает только за сеть)
- FeatureComponent, который зависит от первого через dependencies = [NetworkComponent::class].

В FeatureComponent внедрите сервис, который получает доступ к сети через NetworkComponent.

Домашнее задание

Задача 1.

В рамках домашнего задания из прошлого занятия “Инъекция зависимостей (Часть 1)” с настройкой Dagger2:

- Внедрите Context приложения через @BindsInstance в любой элемент.
- Создайте кастомный @Qualifier для двух разных API-клиентов (например, @WeatherApi и @NewsApi).
- Добавьте в граф несколько реализаций ViewModel через Set/Map multibinding

Q&A

Ваши вопросы

Спасибо

<TeachMeSkills/>