# Dagger basics Основы кинжала

Внедрение зависимостей вручную или локаторы сервисов в приложении для Android могут быть проблематичными в зависимости от размера вашего проекта. Вы можете ограничить сложность своего проекта по мере его масштабирования, используя [Dagger](https://dagger.dev/) для управления зависимостями.

Dagger автоматически генерирует код, имитирующий код, который вы иначе написали бы от руки. Поскольку код создается во время компиляции, он отслеживается и более производительный, чем другие решения на основе отражения, такие как [Guice](https://en.wikipedia.org/wiki/Google_Guice) .

Используйте [Hilt](https://developer.android.com/training/dependency-injection/hilt-android) для внедрения зависимостей на Android. Hilt построен на основе Dagger и предоставляет стандартный способ включения внедрения зависимостей Dagger в приложение Android.

## Преимущества использования Dagger

Dagger освобождает вас от написания утомительного и подверженного ошибкам шаблонного кода:

* Генерация AppContainer кода (графа приложения), который вы вручную реализовали в ручном разделе DI.
* Создание фабрик для классов, доступных в графе приложения. Вот как внутренние зависимости удовлетворяются.
* Принятие решения о повторном использовании зависимости или создании нового экземпляра с помощью областей действия.
* Создание контейнеров для определенных потоков, как вы делали с потоком входа в систему в предыдущем разделе, с использованием подкомпонентов Dagger. Это улучшает производительность вашего приложения, освобождая объекты в памяти, когда они больше не нужны.

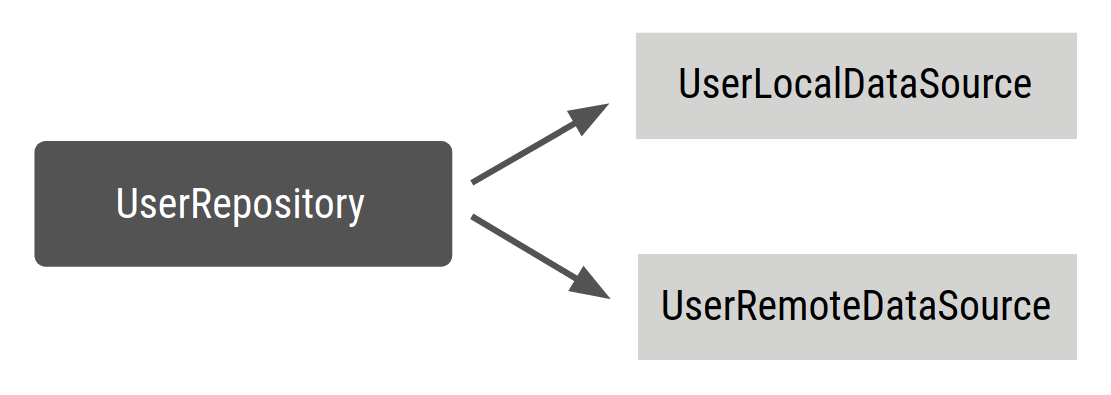
Dagger автоматически делает все это во время сборки, если вы объявляете зависимости класса и указываете, как удовлетворить их с помощью аннотаций. Dagger генерирует код, аналогичный тому, что вы написали бы вручную. Внутри Dagger создает граф объектов, на которые он может ссылаться, чтобы найти способ предоставить экземпляр класса. Для каждого класса в графе Dagger генерирует класс [фабричного типа,](https://en.wikipedia.org/wiki/Factory_method_pattern) который он использует внутри для получения экземпляров этого типа.

Во время сборки Dagger просматривает ваш код и:

* Строит и проверяет графики зависимостей, гарантируя, что:
  + Зависимости каждого объекта могут быть удовлетворены, поэтому исключений времени выполнения нет.
  + Циклов зависимостей не существует, поэтому нет бесконечных циклов.
* Создает классы, которые используются во время выполнения для создания реальных объектов и их зависимостей.

## **Простой пример использования в Dagger: создание фабрики**

Чтобы продемонстрировать, как вы можете работать с Dagger, давайте создадим простую [фабрику](https://en.wikipedia.org/wiki/Factory_method_pattern) для UserRepository класса, показанного на следующей диаграмме:



Определите UserRepository следующее:

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

Добавьте @Inject аннотацию к UserRepository конструктору, чтобы Dagger знал, как создать UserRepository:

// @Inject lets Dagger know how to create instances of this object

// @Inject позволяет Dagger узнать, как создавать экземпляры этого объектного класса

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

В приведенном выше фрагменте кода вы говорите Dagger:

1. Как создать UserRepository экземпляр с @Inject аннотированным конструктором.
2. Каковы его зависимости: UserLocalDataSource и UserRemoteDataSource.

Теперь Dagger знает, как создать экземпляр UserRepository, но не знает, как создавать его зависимости. Если вы аннотируете и другие классы, Dagger знает, как их создавать:

// @Inject lets Dagger know how to create instances of these objects

// @Inject позволяет Dagger знать, как создавать экземпляры этих объектов  
class UserLocalDataSource @Inject constructor() { ... }  
class UserRemoteDataSource @Inject constructor() { ... }

## Dagger components Компоненты кинжала

Dagger может создать график зависимостей в вашем проекте, который он может использовать, чтобы узнать, где он должен получить эти зависимости, когда они необходимы. Чтобы сделать это с помощью Dagger, вам нужно создать интерфейс и аннотировать его с помощью @Component. Dagger создает контейнер, как если бы вы делали инъекцию зависимостей вручную.

Внутри @Component интерфейса вы можете определить функции, которые возвращают экземпляры нужных вам классов (т.е. UserRepository). @Component сообщает Dagger, что нужно создать контейнер со всеми зависимостями, необходимыми для удовлетворения типов, которые он предоставляет. Это называется компонентом Dagger; он содержит граф, состоящий из объектов, которые Dagger знает, как предоставить, и их соответствующих зависимостей.

// @Component makes Dagger create a graph of dependencies

// @Component заставляет Dagger создать график зависимостей  
@Component  
interface ApplicationGraph {  
    // The return type  of functions inside the component interface is  
    // what can be provided from the container

// Тип возвращаемого значения функций внутри интерфейса компонента –

// то, что может быть предоставлено из репозитория контейнера  
    fun repository(): UserRepository  
}

При создании проекта, Dagger генерирует реализацию ApplicationGraph интерфейса для вас: DaggerApplicationGraph. С процессором аннотаций, Dagger создает граф зависимостей, который состоит из отношений между тремя классами (UserRepository, UserLocalDatasource и UserRemoteDataSource) только с одной точкой входа: получение UserRepository экземпляра. Вы можете использовать его следующим образом:  
 // Create an instance of the application graph

// Создание экземпляра графа приложения  
val applicationGraph: ApplicationGraph = DaggerApplicationGraph.create()  
// Grab an instance of UserRepository from the application graph

// Берем экземпляр UserRepository из графа приложения  
val userRepository: UserRepository = applicationGraph.repository()

Dagger создает новый экземпляр UserRepositoryкаждый раз, когда его запрашивают.  
   
val applicationGraph: ApplicationGraph = DaggerApplicationGraph.create()  
  
val userRepository: UserRepository = applicationGraph.repository()  
val userRepository2: UserRepository = applicationGraph.repository()  
  
assert(userRepository != userRepository2)

Иногда вам нужно иметь уникальный экземпляр зависимости в контейнере. Это может быть вам нужно по нескольким причинам:

1. Вы хотите, чтобы другие типы, которые имеют этот тип в качестве зависимости, совместно использовали один и тот же экземпляр, например, несколько ViewModel объектов в потоке входа в систему, использующие один и тот же LoginUserData.
2. Создание объекта дорого, и вы не хотите создавать новый экземпляр каждый раз, когда он объявляется как зависимость (например, парсер JSON).

В этом примере вы можете захотеть иметь уникальный экземпляр, UserRepository доступный на графике, чтобы каждый раз, когда вы запрашиваете a UserRepository, вы всегда получали один и тот же экземпляр. Это полезно в вашем примере, потому что в реальном приложении с более сложным графом приложения у вас может быть несколько ViewModel объектов в зависимости от, UserRepository и вы не хотите создавать новые экземпляры, UserLocalDataSource и UserRemoteDataSource каждый раз UserRepository нужно предоставлять.

При ручном внедрении зависимостей вы делаете это, передавая тот же экземпляр UserRepository конструкторам классов ViewModel; но в Dagger, поскольку вы не пишете этот код вручную, вы должны сообщить Dagger, что хотите использовать тот же экземпляр. Это можно сделать с помощью аннотаций области.

### **Scoping with Dagger -- Прицел с кинжалом**

Вы можете использовать аннотации области, чтобы ограничить время жизни объекта временем жизни его компонента. Это означает, что один и тот же экземпляр зависимости используется каждый раз, когда необходимо указать этот тип.

Чтобы иметь уникальный экземпляр, UserRepository когда вы запрашиваете репозиторий ApplicationGraph, используйте ту же аннотацию области для @Component интерфейса и UserRepository. Вы можете использовать @Singleton аннотацию, которая уже идет с javax.inject пакетом, который использует Dagger:

// Аннотации области в интерфейсе @Component информируют Dagger о том, что классы, аннотированные

// этой аннотацией (например, @Singleton), привязаны к жизненному циклу графа, поэтому

// один и тот же экземпляр этого типа предоставляется каждый раз, когда тип запрашивается.

// Scope annotations on a @Component interface informs Dagger that classes annotated  
// with this annotation (i.e. @Singleton) are bound to the life of the graph and so  
// the same instance of that type is provided every time the type is requested.  
@Singleton  
@Component  
interface ApplicationGraph {  
    fun repository(): UserRepository  
}  
  
// Scope this class to a component using @Singleton scope (i.e. ApplicationGraph)  
@Singleton  
class UserRepository @Inject constructor(  
    private val localDataSource: UserLocalDataSource,  
    private val remoteDataSource: UserRemoteDataSource  
) { ... }

Кроме того, вы можете создать и использовать аннотацию настраиваемой области. Вы можете создать аннотацию области следующим образом:

// Создает MyCustomScope

// Creates MyCustomScope  
@Scope  
@MustBeDocumented  
@Retention(value = AnnotationRetention.RUNTIME)  
annotation class MyCustomScope

Затем вы можете использовать его, как и раньше:

@MyCustomScope  
@Component  
interface ApplicationGraph {  
    fun repository(): UserRepository  
}  
  
@MyCustomScope  
class UserRepository @Inject constructor(  
    private val localDataSource: UserLocalDataSource,  
    private val service: UserService  
) { ... }

В обоих случаях объекту предоставляется та же область видимости, которая используется для аннотирования @Component интерфейса. Таким образом, каждый раз при вызове applicationGraph.repository() вы получаете один и тот же экземпляр UserRepository.

val applicationGraph: ApplicationGraph = DaggerApplicationGraph.create()  
  
val userRepository: UserRepository = applicationGraph.repository()  
val userRepository2: UserRepository = applicationGraph.repository()  
  
assert(userRepository == userRepository2)

## **Conclusion - Вывод**

Важно знать преимущества Dagger и основы его работы, прежде чем вы сможете использовать его в более сложных сценариях.

На [следующей странице](https://developer.android.com/training/dependency-injection/dagger-android) вы узнаете, как добавить Dagger в приложение для Android.

# Использование Dagger в приложениях для Android

На странице с [основными сведениями](https://developer.android.com/training/dependency-injection/dagger-basics) о Dagger объясняется, как Dagger может помочь вам автоматизировать внедрение зависимостей в ваше приложение. С Dagger вам не нужно писать утомительный и подверженный ошибкам шаблонный код.

**Примечание.**Используйте [Hilt](https://developer.android.com/training/dependency-injection/hilt-android) для внедрения зависимостей на Android. Hilt построен на основе Dagger и предоставляет стандартный способ включения внедрения зависимостей Dagger в приложение Android.

**Сводка передовых практик**

**Примечание.**Если вы уже знакомы с Dagger, ознакомьтесь с этими рекомендациями. Если нет, прочтите содержание этой страницы и вернитесь к этому позже.

* По возможности используйте внедрение конструктора с помощью @Injectдля добавления типов в граф Dagger. Когда это не так:
  + Используйте, @Bindsчтобы указать Dagger, какую реализацию должен иметь интерфейс.
  + Используйте, @Providesчтобы сообщить Dagger, как предоставлять классы, не принадлежащие вашему проекту.
* Вы должны объявлять модули только один раз в компоненте.
* Назовите аннотации области видимости в зависимости от времени существования, в котором аннотация используется. Примеры включают в себя @ApplicationScope, @LoggedUserScopeи @ActivityScope.

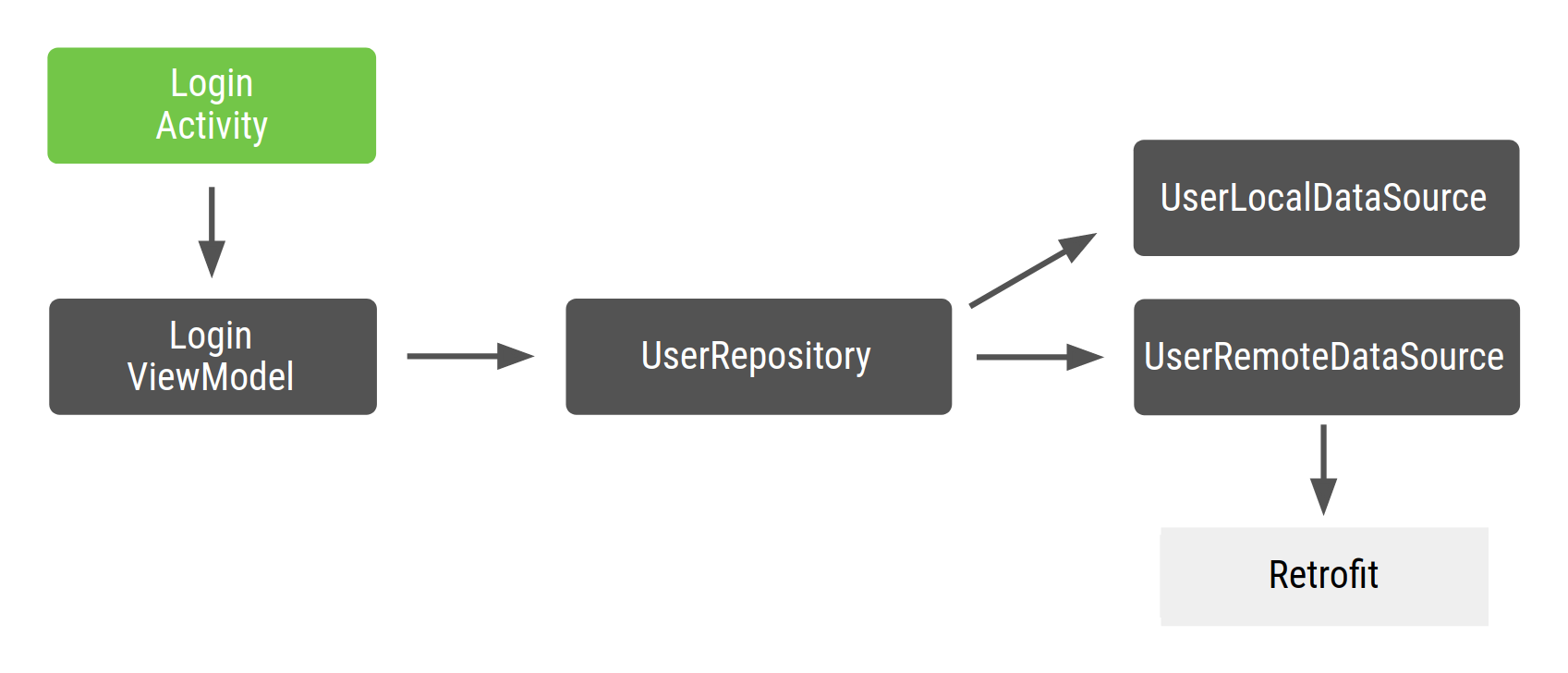
**Adding dependencies - Добавление зависимостей**

Чтобы использовать Dagger в своем проекте, добавьте эти зависимости в свое приложение в свой build.gradle файл. Вы можете найти последнюю версию Dagger [в этом проекте GitHub](https://github.com/google/dagger/releases) .

apply plugin: 'kotlin-kapt'  
dependencies {  
    implementation 'com.google.dagger:dagger:2.x'  
    kapt 'com.google.dagger:dagger-compiler:2.x'  
}

## **Dagger in Android - Кинжал в Android**

Рассмотрим пример приложения для Android с графиком зависимостей из рисунка 1.

****

**Рисунок 1. График зависимости примера кода**

В Android вы обычно создаете граф Dagger, который живет в вашем классе приложения, потому что вы хотите, чтобы экземпляр графа находился в памяти, пока приложение работает. Таким образом, график привязан к жизненному циклу приложения. В некоторых случаях может потребоваться, чтобы контекст приложения был доступен на графике. Для этого вам также потребуется, чтобы граф находился в классе приложения. Одним из преимуществ этого подхода является то, что граф доступен для других классов платформы Android. Кроме того, он упрощает тестирование, позволяя использовать в тестах собственный класс приложения.

Поскольку интерфейс, который генерирует график, снабжен аннотациями @Component, вы можете назвать его ApplicationComponent или ApplicationGraph. Обычно вы храните экземпляр этого компонента в своем пользовательском классе приложения и вызываете его каждый раз, когда вам нужен график приложения, как показано в следующем фрагменте кода:

// Определение графа приложения

// Definition of the Application graph  
@Component  
interface ApplicationComponent { ... }  
// appComponent живет в классе Application, чтобы поделиться своим классом жизненного цикла  
// appComponent lives in the Application class to share its lifecycle  
class MyApplication: Application() {

// Ссылка на граф приложения, который используется во всем приложении  
    // Reference to the application graph that is used across the whole app  
    val appComponent = DaggerApplicationComponent.create()  
}

Поскольку некоторые классы платформы Android, такие как действия и фрагменты, создаются системой, Dagger не может создавать их за вас. В частности, для действий любой код инициализации должен входить в onCreate() метод. Это означает, что вы не можете использовать @Inject аннотацию в конструкторе класса (внедрение конструктора), как вы это делали в предыдущих примерах. Вместо этого вы должны использовать инъекцию поля.

**Примечание.**На странице с [основными сведениями о Dagger описано,](https://developer.android.com/training/dependency-injection/dagger-basics) как использовать **@Inject** аннотацию Dagger в конструкторах. Эта аннотация сообщает Dagger, как создавать экземпляры класса.  
   
Вместо создания зависимостей, требуемых действием в onCreate() методе, вы хотите, чтобы Dagger заполнял эти зависимости за вас. Для внедрения поля вместо этого вы применяете @Inject аннотацию к полям, которые хотите получить из графа Dagger.

class LoginActivity: Activity() {

// Вы хотите, чтобы Dagger предоставил экземпляр LoginViewModel из графа  
// You want Dagger to provide an instance of LoginViewModel from the graph  
    @Inject lateinit var loginViewModel: LoginViewModel  
}

Для простоты LoginViewModelэто не модель представления [компонентов архитектуры Android](https://developer.android.com/topic/libraries/architecture/viewmodel) ; это просто обычный класс, который действует как ViewModel. Для получения дополнительной информации о том, как внедрить эти классы, ознакомьтесь с кодом в официальной реализации [Android Blueprints Dagger](https://github.com/android/architecture-samples/tree/dev-dagger) в ветке [dev-dagger](https://github.com/android/architecture-samples/tree/dev-dagger) .

Одно из соображений относительно Dagger заключается в том, что введенные поля не могут быть закрытыми. Они должны иметь видимость, по крайней мере, для частного пакета, как в предыдущем коде.

### **Примечание.**Внедрение поля следует использовать только в классах фреймворка Android, где нельзя использовать внедрение конструктора. **Injecting activities -**   **Инъекционная деятельность**

Dagger должен знать, что LoginActivityдолжен получить доступ к графу, чтобы предоставить то, что ViewModelему требуется. На странице с [основными сведениями](https://developer.android.com/training/dependency-injection/dagger-basics) о [Dagger](https://developer.android.com/training/dependency-injection/dagger-basics) вы использовали @Componentинтерфейс для получения объектов из графа, выставляя функции с типом возвращаемого значения, которое вы хотите получить от графа. В этом случае вам нужно сообщить Dagger об объекте ( LoginActivity в данном случае), который требует внедрения зависимости. Для этого вы предоставляете функцию, которая принимает в качестве параметра объект, запрашивающий инъекцию.

@Component  
 interface ApplicationComponent {

// This tells Dagger that LoginActivity requests injection so the graph needs to

// satisfy all the dependencies of the fields that LoginActivity is requesting.

// Это сообщает Dagger, что LoginActivity запрашивает внедрение, поэтому граф должен   
// удовлетворить все зависимости полей, которые запрашивает LoginActivity.

fun inject(activity: LoginActivity)

}  
 Эта функция сообщает Dagger, что LoginActivity хочет получить доступ к графику и запрашивает инъекцию. Dagger должен удовлетворять все требуемые зависимости LoginActivity( LoginViewModel со своими собственными зависимостями). Если у вас есть несколько классов, которые запрашивают инъекцию, вы должны специально объявить их все в компоненте с указанием их точного типа. Например, если у вас есть LoginActivityи вы RegistrationActivityзапрашиваете инъекцию, у вас будет два inject()метода вместо общего, охватывающего оба случая. Универсальный inject()метод не сообщает Dagger, что необходимо предоставить. Функции в интерфейсе могут иметь любое имя, но их вызов, inject()когда они получают объект для внедрения в качестве параметра, является соглашением в Dagger.

Чтобы внедрить объект в действие, вы должны использовать appComponentкласс, определенный в вашем классе приложения, и вызвать inject()метод, передав экземпляр действия, запрашивающего внедрение.

При использовании действий вставьте Dagger в onCreate()метод действия перед вызовом, super.onCreate()чтобы избежать проблем с восстановлением фрагмента. На этапе восстановления super.onCreate()действие присоединяет фрагменты, которым может потребоваться доступ к привязкам действий.

При использовании фрагментов вставьте Dagger в метод фрагмента onAttach() . В этом случае это можно сделать до или после звонка super.onAttach().

class LoginActivity: Activity() {  
// You want Dagger to provide an instance of LoginViewModel from the graph

// Вы хотите, чтобы Dagger предоставил экземпляр LoginViewModel из графа  
    @Inject lateinit var loginViewModel: LoginViewModel  
    override fun onCreate(savedInstanceState: Bundle?) {  
        // Make Dagger instantiate @Inject fields in LoginActivity  
 // Заставляет Dagger создавать экземпляры полей  
        (applicationContext as MyApplication).appComponent.inject(this)  
        // Now loginViewModel is available  
 // Теперь доступна loginViewModel  
        super.onCreate(savedInstanceState)  
    }  
}  
  
// @Inject tells Dagger how to create instances of LoginViewModel

// @Inject сообщает Dagger, как создавать экземпляры класса  
class LoginViewModel @Inject constructor(  
    private val userRepository: UserRepository  
) { ... }

Давайте расскажем Dagger, как предоставить остальные зависимости для построения графа:

class UserRepository @Inject constructor(  
    private val localDataSource: UserLocalDataSource,  
    private val remoteDataSource: UserRemoteDataSource  
) { ... }  
  
class UserLocalDataSource @Inject constructor() { ... }  
class UserRemoteDataSource @Inject constructor(  
    private val loginService: LoginRetrofitService  
) { ... }

**Dagger modules - Модули кинжала**

В этом примере вы используете сетевую библиотеку [Retrofit](https://square.github.io/retrofit/) . UserRemoteDataSourceимеет зависимость от LoginRetrofitService. Однако способ создания экземпляра LoginRetrofitServiceотличается от того, что вы делали до сих пор. Это не создание экземпляра класса; это результат вызова Retrofit.Builder()и передачи различных параметров для настройки службы входа в систему.

Помимо @Injectаннотации, есть еще один способ сообщить Dagger, как предоставить экземпляр класса: информацию внутри модулей Dagger. Модуль Dagger - это класс, помеченный как @Module. Там вы можете определить зависимости с @Providesаннотацией.

// @Module informs Dagger that this class is a Dagger Module

// @Module сообщает Dagger, что этот класс является модулем Dagger.  
@Module  
class NetworkModule {  
  
    // @Provides tell Dagger how to create instances of the type that this function  
    // returns (i.e. LoginRetrofitService).  
    // Function parameters are the dependencies of this type.

// @Provides сообщает Dagger, как создавать экземпляры типа, который

// возвращает эта функция (например, LoginRetrofitService).

// Параметры функции являются зависимостями этого типа.  
    @Provides  
    fun provideLoginRetrofitService(): LoginRetrofitService {

// Whenever Dagger needs to provide an instance of type LoginRetrofitService,  
// this code (the one inside the @Provides method) is run.

// Каждый раз, когда Dagger необходимо предоставить экземпляр типа LoginRetrofitService,

// этот код (тот, что находится внутри метода @Provides) запускается  
        return Retrofit.Builder()  
                .baseUrl("https://example.com")  
                .build()  
                .create(LoginService::class.java)  
    }  
}

**Примечание:**вы можете использовать **@Provides**аннотацию в модулях Dagger, чтобы сообщить Dagger, как предоставлять классы, которыми ваш проект не владеет (например, экземпляр **Retrofit**).  
  
Для реализации интерфейсов лучше всего использовать [**@Binds**](https://dagger.dev/faq.html#binds), как вы можете видеть в [кодовой таблице](https://codelabs.developers.google.com/codelabs/android-dagger/) « [Использование кинжала в приложении Android»](https://codelabs.developers.google.com/codelabs/android-dagger/) .

Модули — это способ семантически инкапсулировать информацию о том, как предоставлять объекты. Как видите, вы вызвали класс, NetworkModuleчтобы сгруппировать логику предоставления объектов, связанных с сетью. Если приложение расширяется, вы также можете добавить, как предоставить [OkHttpClient](https://square.github.io/okhttp/)здесь или как настроить [Gson](https://github.com/google/gson) или [Moshi](https://github.com/square/moshi) .

Зависимости @Providesметода — это параметры этого метода. Для предыдущего метода LoginRetrofitServiceможет быть предоставлено без зависимостей, потому что у метода нет параметров. Если вы объявили OkHttpClientв качестве параметра, Dagger должен был бы предоставить OkHttpClientэкземпляр из графа, чтобы удовлетворить зависимости LoginRetrofitService. Например:

@Module  
class NetworkModule {  
    // Hypothetical dependency on LoginRetrofitService  
// Гипотетическая зависимость от LoginRetrofitService  
    @Provides  
    fun provideLoginRetrofitService(  
        okHttpClient: OkHttpClient  
    ): LoginRetrofitService { ... }  
}

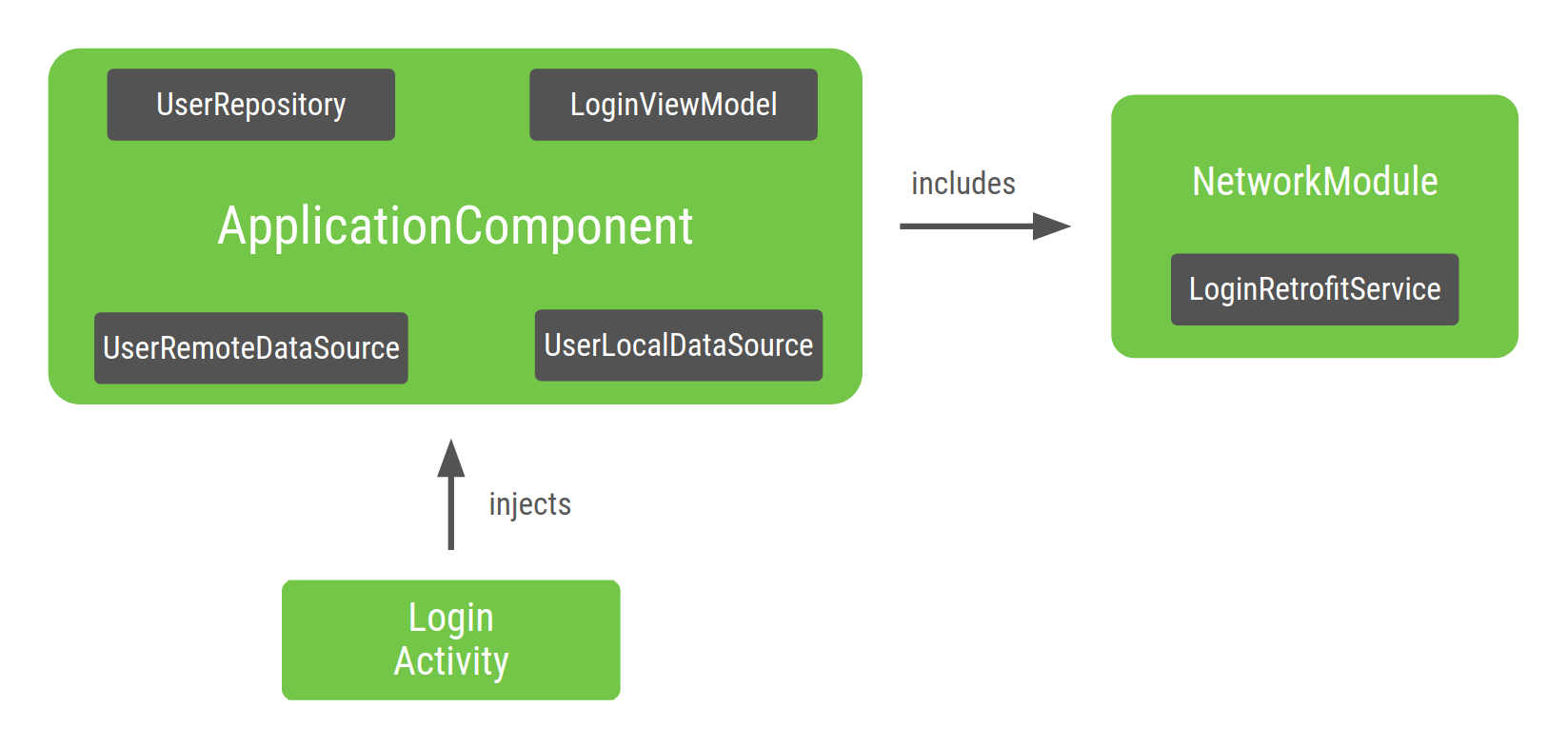
Чтобы граф Dagger знал об этом модуле, вам необходимо добавить его в @Component интерфейс следующим образом:

// The "modules" attribute in the @Component annotation tells Dagger what Modules  
// to include when building the graph  
// Атрибут "modules" в аннотации @Component сообщает Dagger, какие модули

// включать при построении графа @Component  
@Component(modules = [NetworkModule::class])  
interface ApplicationComponent {  
    ...  
}

Рекомендуемый способ добавления типов к графу Dagger - использование внедрения конструктора (то есть с @Injectаннотацией к конструктору класса). Иногда это невозможно, и вам приходится использовать модули Dagger. Один из примеров — это когда вы хотите, чтобы Dagger использовал результат вычисления, чтобы определить, как создать экземпляр объекта. Каждый раз, когда ему нужно предоставить экземпляр этого типа, Dagger запускает код внутри @Providesметода.

Вот как сейчас выглядит граф Dagger в примере:



**Рисунок 2.** Изображение графа с LoginActivityвведением Dagger

Точкой входа в график является LoginActivity. Поскольку LoginActivityвнедряет LoginViewModel, Dagger строит граф, который знает, как предоставить экземпляр LoginViewModelи рекурсивно своих зависимостей. Dagger знает, как это сделать, из-за @Injectаннотации конструктора классов.

Внутри ApplicationComponentсозданного Dagger есть фабричный метод для получения экземпляров всех классов, которые он знает, как предоставить. В этом примере Dagger делегирует NetworkModuleвключенный в, ApplicationComponentчтобы получить экземпляр LoginRetrofitService.

### **Dagger scopes -** **Прицелы кинжалов**

Области были упомянуты на странице с [основными сведениями](https://developer.android.com/training/dependency-injection/dagger-basics) о [Dagger](https://developer.android.com/training/dependency-injection/dagger-basics) как способ иметь уникальный экземпляр типа в компоненте. Это то, что подразумевается под *привязкой типа к жизненному циклу компонента* .

Поскольку вы, возможно, захотите использовать UserRepositoryдругие функции приложения и не захотите создавать новый объект каждый раз, когда он вам понадобится, вы можете обозначить его как уникальный экземпляр для всего приложения. То же самое для LoginRetrofitService: его создание может быть дорогостоящим, и вы также хотите, чтобы уникальный экземпляр этого объекта был повторно использован. Создание экземпляра UserRemoteDataSourceне так уж и дорого, поэтому нет необходимости ограничивать его жизненным циклом компонента.

[@Singleton](https://docs.oracle.com/javaee/7/api/javax/inject/Singleton.html)это единственная аннотация области, которая поставляется с javax.injectпакетом. Вы можете использовать его для аннотирования ApplicationComponent объектов, которые хотите повторно использовать во всем приложении.

**Примечание:**вы можете использовать **любую** аннотацию области, чтобы иметь уникальный экземпляр типа в компоненте, если компонент и тип аннотированы им. **@Singleton**поставляется с библиотекой Dagger и обычно используется для аннотирования компонента приложения, но вы можете создать собственный компонент с другим именем (например, **@ApplicationScope**).

@Singleton  
@Component(modules = [NetworkModule::class])  
interface ApplicationComponent {  
    fun inject(activity: LoginActivity)  
}  
  
@Singleton  
class UserRepository @Inject constructor(  
    private val localDataSource: UserLocalDataSource,  
    private val remoteDataSource: UserRemoteDataSource  
) { ... }  
  
@Module  
class NetworkModule {  
    // Way to scope types inside a Dagger Module  
// Способ определения типов в Dagger Module  
    @Singleton  
    @Provides  
    fun provideLoginRetrofitService(): LoginRetrofitService { ... }  
}

**Внимание!**Модули, использующие аннотацию области действия, могут использоваться только в компонентах, которые имеют аннотацию той же области.

Будьте осторожны, чтобы не вызвать утечки памяти при применении областей видимости к объектам. Пока компонент с заданной областью находится в памяти, созданный объект тоже находится в памяти. Поскольку ApplicationComponentсоздается при запуске приложения (в классе приложения), он уничтожается, когда приложение уничтожается. Таким образом, уникальный экземпляр UserRepositoryвсегда остается в памяти до тех пор, пока приложение не будет уничтожено.

**Примечание.**Добавьте аннотации области видимости в классы при использовании внедрения конструктора (с **@Inject**) и добавьте их в **@Provides**методы при использовании модулей Dagger.

**Dagger subcomponents - Подкомпоненты кинжала**

Если ваш поток входа (управляемый одним LoginActivity) состоит из нескольких фрагментов, вам следует повторно использовать один и тот же экземпляр LoginViewModelво всех фрагментах. @Singletonне может добавлять аннотации LoginViewModelдля повторного использования экземпляра по следующим причинам:

1. Экземпляр LoginViewModelбудет сохраняться в памяти после завершения потока.
2. Вам нужен отдельный экземпляр LoginViewModelдля каждого потока входа. Например, если пользователь выходит из системы, вам нужен другой экземпляр LoginViewModel, а не тот, который был при первом входе пользователя в систему.

Чтобы охватить LoginViewModelжизненный цикл, LoginActivityвам необходимо создать новый компонент (новый подграф) для потока входа в систему и новую область.

Давайте создадим график, специфичный для процесса входа в систему.

@Component  
interface LoginComponent {}

Теперь он LoginActivityдолжен получать инъекции, LoginComponentпотому что он имеет конфигурацию для входа в систему. Это снимает LoginActivityс ApplicationComponentкласса ответственность за инъекцию .

@Component  
interface LoginComponent {  
    fun inject(activity: LoginActivity)  
}

LoginComponentдолжен иметь доступ к объектам из, ApplicationComponent потому что LoginViewModelзависит от UserRepository. Способ сообщить Dagger, что вы хотите, чтобы новый компонент использовал часть другого компонента, - это *подкомпоненты Dagger* . Новый компонент должен быть подкомпонентом компонента, содержащего общие ресурсы.

*Подкомпоненты* - это компоненты, которые наследуют и расширяют граф объектов родительского компонента. Таким образом, все объекты, представленные в родительском компоненте, также предоставляются в субкомпоненте. Таким образом, объект из подкомпонента может зависеть от объекта, предоставленного родительским компонентом.

Чтобы создать экземпляры подкомпонентов, вам понадобится экземпляр родительского компонента. Следовательно, объекты, предоставленные родительским компонентом субкомпоненту, по-прежнему ограничены родительским компонентом.

В этом примере вы должны определить LoginComponentкак подкомпонент ApplicationComponent. Чтобы сделать это, аннотировать LoginComponentс @Subcomponent:

// @Subcomponent annotation informs Dagger this interface is a Dagger Subcomponent  
// аннотация @Subcomponent сообщает Dagger, что этот интерфейс является подкомпонентом Dagger.  
@Subcomponent  
interface LoginComponent {  
  
    // This tells Dagger that LoginActivity requests injection from LoginComponent  
    // so that this subcomponent graph needs to satisfy all the dependencies of the  
    // fields that LoginActivity is injecting  
// Это сообщает Dagger, что LoginActivity запрашивает инъекцию от LoginComponent   
//, чтобы этот граф подкомпонентов должен удовлетворять всем зависимостям полей   
// которые LoginActivity инъекционный  
    fun inject(loginActivity: LoginActivity)  
}

Вы также должны определить внутри фабрику подкомпонентов, LoginComponentчтобы она ApplicationComponentзнала, как создавать экземпляры LoginComponent.

@Subcomponent  
interface LoginComponent {  
  
    // Factory that is used to create instances of this subcomponent

// Фабрика, которая используется для создания экземпляров этого подкомпонента  
    @Subcomponent.Factory  
    interface Factory {  
        fun create(): LoginComponent  
    }  
  
    fun inject(loginActivity: LoginActivity)  
}

Чтобы сообщить Dagger, что LoginComponentэто подкомпонент ApplicationComponent, вы должны указать это:

1. Создание нового модуля Dagger (например SubcomponentsModule) передача класса подкомпонента subcomponentsатрибуту аннотации.

// «Подкомпоненты» атрибут в @Module аннотации говорит Dagger , что   
// Подкомпоненты являются дети Компоненты этого модуль включен в.  
// The "subcomponents" attribute in the @Module annotation tells Dagger what  
// Subcomponents are children of the Component this module is included in.  
@Module(subcomponents = LoginComponent::class)  
class SubcomponentsModule {}

1. Добавление нового модуля (т.е. SubcomponentsModule) в ApplicationComponent:

// Включая SubcomponentsModule, сообщаем ApplicationComponent, что // LoginComponent является его подкомпонентом.  
// Including SubcomponentsModule, tell ApplicationComponent that  
// LoginComponent is its subcomponent.  
@Singleton  
@Component(modules = [NetworkModule::class, SubcomponentsModule::class])  
interface ApplicationComponent {  
}

Обратите внимание, что ApplicationComponentбольше не нужно вводить LoginActivity, потому что теперь эта ответственность принадлежит LoginComponent, поэтому вы можете удалить inject()метод из ApplicationComponent.

Потребители ApplicationComponentдолжны знать, как создавать экземпляры LoginComponent. Родительский компонент должен добавить метод в свой интерфейс, чтобы потребители могли создавать экземпляры подкомпонента из экземпляра родительского компонента:

1. Раскройте фабрику, которая создает экземпляры LoginComponentв интерфейсе:

@Singleton  
@Component(modules = [NetworkModule::class, SubcomponentsModule::class])  
interface ApplicationComponent {  
// This function exposes the LoginComponent Factory out of the graph so consumers  
// can use it to obtain new instances of LoginComponent  
// Эта функция выводит FactoryComponent Factory за пределы графика, чтобы потребители // могли использовать ее для получения новых экземпляров  
fun loginComponent(): LoginComponent.Factory  
}

**Assigning scopes to subcomponents -   
Назначение областей субкомпонентам**

Если вы создаете проект, вы можете создавать экземпляры обоих ApplicationComponent и LoginComponent. ApplicationComponentпривязан к жизненному циклу приложения, потому что вы хотите использовать один и тот же экземпляр графа, пока приложение находится в памяти.

Каков жизненный цикл LoginComponent? Одна из причин, по которой вам нужно, LoginComponentзаключается в том, что вам нужно было использовать один и тот же экземпляр для LoginViewModelфрагментов, связанных с входом. Но также вам нужны разные экземпляры, LoginViewModelкогда есть новый поток входа. LoginActivity является правильным временем жизни для LoginComponent: для каждого нового действия вам нужен новый экземпляр LoginComponentи фрагменты, которые могут использовать этот экземпляр LoginComponent.

Поскольку LoginComponentон привязан к LoginActivityжизненному циклу, вы должны сохранить ссылку на компонент в действии точно так же, как вы сохранили ссылку на applicationComponentв классе приложения. Таким образом, фрагменты могут получить к нему доступ.

class LoginActivity: Activity() {  
    // Reference to the Login graph Ссылка на график      
    lateinit var loginComponent: LoginComponent  
    ...  
}

Обратите внимание, что переменная loginComponentне аннотирована, @Inject потому что вы не ожидаете, что эта переменная будет предоставлена ​​Dagger.

Вы можете использовать, ApplicationComponentчтобы получить ссылку, LoginComponent а затем вставить LoginActivityследующим образом:

class LoginActivity: Activity() {  
    // Reference to the Login graph  
    lateinit var loginComponent: LoginComponent  
  
    // Fields that need to be injected by the login graph  
    @Inject lateinit var loginViewModel: LoginViewModel  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        // Creation of the login graph using the application graph  
        loginComponent = (applicationContext as MyDaggerApplication)  
                            .appComponent.loginComponent().create()  
  
        // Make Dagger instantiate @Inject fields in LoginActivity  
        loginComponent.inject(this)  
  
        // Now loginViewModel is available  
  
        super.onCreate(savedInstanceState)  
    }  
}

class LoginActivity : Activity () { // Ссылка на граф     входа в систему lateinit var loginComponent : LoginComponent // Поля, которые необходимо ввести графом входа в систему @Inject lateinit var loginViewModel : LoginViewModel override fun onCreate ( savedInstanceState : Bundle ?) { // Создание графа входа в систему с использованием графа приложения         loginComponent = ( applicationContext как MyDaggerApplication )   
                           . appComponent . loginComponent (). create () // Заставляем Dagger создавать экземпляры полей @Inject в LoginActivity         loginComponent . inject ( this ) // Теперь доступна loginViewModel super . onCreate ( savedInstanceState ) } }

LoginComponentсоздается в onCreate()методе действия, и он будет неявно уничтожен при уничтожении действия.

При каждом запросе объект LoginComponentдолжен всегда предоставлять один и тот же экземпляр LoginViewModel. Вы можете убедиться в этом, создав настраиваемую область аннотации и аннотируя как ее, так LoginComponentи LoginViewModelее. Обратите внимание, что вы не можете использовать @Singletonаннотацию, потому что она уже использовалась родительским компонентом, и это сделало бы объект синглтоном приложения (уникальный экземпляр для всего приложения). Вам нужно создать другую область аннотации.

**Примечание**. Правила определения объема следующие:

* Когда тип помечен аннотацией области действия, он может использоваться только компонентами, снабженными аннотацией той же области действия.
* Когда компонент помечен аннотацией области, он может предоставлять только типы с этой аннотацией или типы, которые не имеют аннотации.
* Подкомпонент не может использовать аннотацию области, используемую одним из его родительских компонентов.

Компоненты также включают в себя подкомпоненты в этом контексте.

В этом случае вы могли бы назвать эту область видимости, @LoginScopeно это не очень хорошая практика. Имя аннотации области не должно быть явным для той цели, которую она выполняет. Вместо этого ему следует называть имя в зависимости от его срока службы, поскольку аннотации могут повторно использоваться родственными компонентами, такими как RegistrationComponent и SettingsComponent. Вот почему вы должны называть его @ActivityScopeвместо @LoginScope.

// Definition of a custom scope called ActivityScope  
@Scope  
@Retention(value = AnnotationRetention.RUNTIME)  
annotation class ActivityScope  
  
// Classes annotated with @ActivityScope are scoped to the graph and the same  
// instance of that type is provided every time the type is requested.  
@ActivityScope  
@Subcomponent  
interface LoginComponent { ... }  
  
// A unique instance of LoginViewModel is provided in Components  
// annotated with @ActivityScope  
@ActivityScope  
class LoginViewModel @Inject constructor(  
    private val userRepository: UserRepository  
) { ... }

// Определение настраиваемой области действия под названием ActivityScope @Scope @Retention ( value = AnnotationRetention . RUNTIME ) класс аннотации ActivityScope // Классы, аннотированные с помощью @ActivityScope, привязаны к графику, и один и тот же экземпляр // этого типа предоставляется каждый раз, когда тип запрашивается. @ActivityScope @Subcomponent interface LoginComponent { ... } // Уникальный экземпляр LoginViewModel предоставляется в Components // с аннотацией @ActivityScope @ActivityScope class LoginViewModel @Inject constructor (  
   private val userRepository : UserRepository ) { ... }

Теперь, если у вас есть два необходимых фрагмента LoginViewModel, им обоим предоставляется один и тот же экземпляр. Например, если у вас есть a LoginUsernameFragmentи a, LoginPasswordFragmentони должны быть введены LoginComponent:

@ActivityScope  
@Subcomponent  
interface LoginComponent {  
  
    @Subcomponent.Factory  
    interface Factory {  
        fun create(): LoginComponent  
    }  
  
    // All LoginActivity, LoginUsernameFragment and LoginPasswordFragment  
    // request injection from LoginComponent. The graph needs to satisfy  
    // all the dependencies of the fields those classes are injecting  
    fun inject(loginActivity: LoginActivity)  
    fun inject(usernameFragment: LoginUsernameFragment)  
    fun inject(passwordFragment: LoginPasswordFragment)  
}

@ActivityScope @Subcomponent interface LoginComponent { @Subcomponent . Заводской интерфейс Factory { fun create (): LoginComponent } // Все LoginActivity, LoginUsernameFragment и LoginPasswordFragment // запрашивают инъекцию от LoginComponent. Потребности график , чтобы удовлетворить // все зависимости полей эти классы инъекционные весело инъекционные ( loginActivity : LoginActivity ) весело инъекционные ( usernameFragment : LoginUsernameFragment ) весело  
инъекционные ( passwordFragment : LoginPasswordFragment ) }

Компоненты обращаются к экземпляру компонента, который живет в LoginActivityобъекте. Пример кода для LoginUserNameFragmentпоявляется в следующем фрагменте кода:

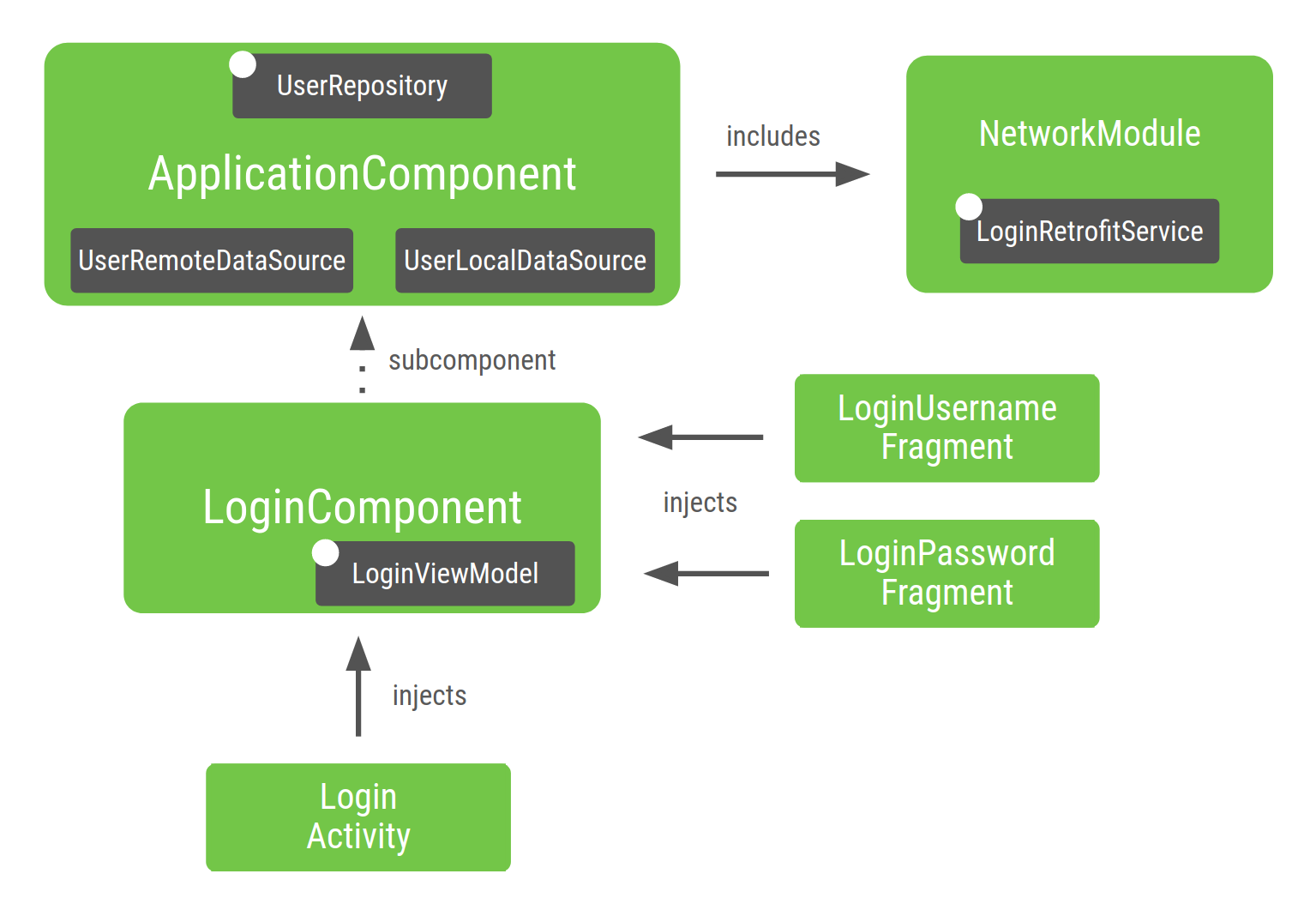
class LoginUsernameFragment: Fragment() {  
  
    // Fields that need to be injected by the login graph  
    @Inject lateinit var loginViewModel: LoginViewModel  
  
    override fun onAttach(context: Context) {  
        super.onAttach(context)  
  
        // Obtaining the login graph from LoginActivity and instantiate  
        // the @Inject fields with objects from the graph  
        (activity as LoginActivity).loginComponent.inject(this)  
  
        // Now you can access loginViewModel here and onCreateView too  
        // (shared instance with the Activity and the other Fragment)  
    }  
}

class LoginUsernameFragment : Fragment () { // Поля, которые должны быть введены графом входа в систему @Inject lateinit var loginViewModel : LoginViewModel override fun onAttach ( context : Context ) { super . onAttach ( context ) // Получение графика входа в систему из LoginActivity и создание экземпляра // полей @Inject с объектами из графика ( активность как LoginActivity ). loginComponent . вводить   
( this ) // Теперь вы можете получить доступ к loginViewModel здесь и onCreateView тоже // (общий экземпляр с Activity и другим фрагментом) } }

И то же самое для LoginPasswordFragment:

class LoginPasswordFragment: Fragment() {  
  
    // Fields that need to be injected by the login graph  
    @Inject lateinit var loginViewModel: LoginViewModel  
  
    override fun onAttach(context: Context) {  
        super.onAttach(context)  
  
        (activity as LoginActivity).loginComponent.inject(this)  
  
        // Now you can access loginViewModel here and onCreateView too  
        // (shared instance with the Activity and the other Fragment)  
    }  
}

class LoginPasswordFragment : Fragment () { // Поля, которые должны быть введены графом входа в систему @Inject lateinit var loginViewModel : LoginViewModel override fun onAttach ( context : Context ) { super . onAttach ( context ) ( активность как LoginActivity ). loginComponent . inject ( this ) // Теперь вы можете получить доступ к loginViewModel здесь и onCreateView тоже   
// (общий экземпляр с Activity и другим фрагментом) } }  
    На рисунке 3 показано, как выглядит график Dagger с новым подкомпонентом. Классы с белой точкой ( UserRepository, LoginRetrofitServiceи LoginViewModel) - это те классы , у которых есть уникальная область действия экземпляра, относящаяся к их соответствующим компонентам.



**Рис. 3.** Представление графика, построенного для примера приложения Android.

Давайте разберем части графика:

1. NetworkModule(И , следовательно LoginRetrofitService) входит в ApplicationComponentпотому , что вы указали его в компоненте.
2. UserRepositoryостается внутри, ApplicationComponentпотому что область его действия ограничена ApplicationComponent. Если проект растет, вы хотите использовать один и тот же экземпляр для разных функций (например, для регистрации).

Поскольку UserRepositoryявляется частью ApplicationComponent, его зависимости (т. Е. UserLocalDataSourceИ UserRemoteDataSource) тоже должны быть в этом компоненте, чтобы иметь возможность предоставлять экземпляры UserRepository.

1. LoginViewModelвключен, LoginComponentпотому что он требуется только для классов, введенных LoginComponent. LoginViewModelне включен, ApplicationComponentпотому что не зависит от ApplicationComponentпотребностей LoginViewModel.

Точно так же, если бы не было областью действия , UserRepositoryчтобы ApplicationComponent, Dagger автоматически включили бы UserRepositoryи его зависимости как часть , LoginComponentпотому что в настоящее время является единственным местом , UserRepositoryиспользуется.

Помимо привязки объектов к разному жизненному циклу, **создание подкомпонентов является хорошей практикой для инкапсуляции разных частей вашего приложения друг от друга** .

Структурирование вашего приложения для создания различных подграфов Dagger в зависимости от потока вашего приложения помогает создать **более производительное и масштабируемое приложение** с точки зрения памяти и времени запуска.

**Примечание.**Если вам нужно, чтобы контейнер пережил изменения конфигурации, такие как ротация устройства, следуйте руководству « [Сохранение состояний пользовательского интерфейса»](https://developer.android.com/topic/libraries/architecture/saving-states) . Возможно, вы захотите обрабатывать изменения конфигурации так же, как и завершение процесса; в противном случае ваше приложение может потерять состояние на устройствах низкого уровня.

**Best practices when building a Dagger graph**

**Лучшие практики при построении графа Dagger**

При построении графа Dagger для вашего приложения:

* Когда вы создаете компонент, вы должны учитывать, какой элемент отвечает за время жизни этого компонента. В этом случае класс приложения ведал ApplicationComponentи LoginActivityотвечает за LoginComponent.
* Используйте область видимости только тогда, когда это имеет смысл. Чрезмерное использование области действия может отрицательно сказаться на производительности вашего приложения во время выполнения: объект находится в памяти, пока компонент находится в памяти, а получение объекта с заданной областью обходится дороже. Когда Dagger предоставляет объект, он использует DoubleCheckблокировку вместо поставщика фабричного типа.

**Testing a project that uses Dagger**

**Тестирование проекта, использующего Dagger**

Одним из преимуществ использования фреймворков внедрения зависимостей, таких как Dagger, является то, что они упрощают тестирование кода.

**Unit tests - Модульные тесты**

Вам не нужно использовать Dagger для *модульных тестов* . При тестировании класса, который использует внедрение конструктора, вам не нужно использовать Dagger для создания экземпляра этого класса. Вы можете напрямую вызвать его конструктор, передавая поддельные или фиктивные зависимости напрямую, как если бы они не были аннотированы.

Например, при тестировании LoginViewModel:

@ActivityScope  
class LoginViewModel @Inject constructor(  
    private val userRepository: UserRepository  
) { ... }  
  
class LoginViewModelTest {  
  
    @Test  
    fun `Happy path`() {  
        // You don't need Dagger to create an instance of LoginViewModel  
        // You can pass a fake or mock UserRepository  
        val viewModel = LoginViewModel(fakeUserRepository)  
        assertEquals(...)  
    }  
}

@ActivityScope класс LoginViewModel @Inject constructor ( private val userRepository : UserRepository ) { ... } класс LoginViewModelTest  
{  
    @Test  
    fun `Happy path`() {  
        // You don't need Dagger to create an instance of LoginViewModel  
        // You can pass a fake or mock UserRepository  
        val viewModel = LoginViewModel(fakeUserRepository)  
        assertEquals(...)  
    }  
}

**End-to-end tests - Сквозные тесты**

Для *интеграционных тестов* рекомендуется создать TestApplicationComponentфайл, предназначенный для тестирования. *В производстве и тестировании используется другая конфигурация компонентов* .

Это требует более [предварительной разработки модулей](https://dagger.dev/testing.html#organize-modules-for-testability) в вашем приложении. Компонент тестирования расширяет производственный компонент и устанавливает другой набор модулей.

// TestApplicationComponent extends from ApplicationComponent to have them both  
// with the same interface methods. You need to include the modules of the  
// component here as well, and you can replace the ones you want to override.  
// This sample uses FakeNetworkModule instead of NetworkModule  
@Singleton  
@Component(modules = [FakeNetworkModule::class, SubcomponentsModule::class])  
interface TestApplicationComponent : ApplicationComponent {  
}

// TestApplicationComponent расширяется от ApplicationComponent, чтобы у них обоих // были одинаковые методы интерфейса. Вам также необходимо включить сюда модули // компонента, и вы можете заменить те, которые хотите переопределить. // В этом примере используется FakeNetworkModule вместо NetworkModule @Singleton @Component ( modules = [ FakeNetworkModule :: class , SubcomponentsModule :: class ]) interface TestApplicationComponent : ApplicationComponent { }

FakeNetworkModuleесть поддельная реализация оригинала NetworkModule. Там вы можете предоставить поддельные экземпляры или макеты того, что хотите заменить.

// In the FakeNetworkModule, pass a fake implementation of LoginRetrofitService  
// that you can use in your tests.  
@Module  
class FakeNetworkModule {  
    @Provides  
    fun provideLoginRetrofitService(): LoginRetrofitService {  
        return FakeLoginService()  
    }  
}

// В FakeNetworkModule передайте поддельную реализацию LoginRetrofitService //, которую вы можете использовать в своих тестах. @Module class FakeNetworkModule { @Provides fun provideLoginRetrofitService (): LoginRetrofitService { return FakeLoginService () } }

В вашей интеграции или от конца до конца испытаний, вы бы использовать , TestApplicationчто создает TestApplicationComponentвместо ApplicationComponent.

// Your test application needs an instance of the test graph  
class MyTestApplication: MyApplication() {  
    override val appComponent = DaggerTestApplicationComponent.create()  
}

// Ваш тест приложения необходим экземпляр теста граф класса MyTestApplication : MyApplication () { переопределение вал appComponent = DaggerTestApplicationComponent . create () }

Затем это тестовое приложение используется в пользовательском интерфейсе, TestRunnerкоторый вы будете использовать для запуска инструментальных тестов. Для получения дополнительной информации об этом ознакомьтесь с [использованием Dagger в кодовой таблице приложения Android](https://codelabs.developers.google.com/codelabs/android-dagger/) .

**Working with Dagger modules**

**Работа с модулями Dagger**

Модули Dagger — это способ инкапсулировать, как предоставлять объекты семантическим способом. Вы можете включать модули в компоненты, но вы также можете включать модули внутри других модулей. Это мощный инструмент, но им легко злоупотребить.

После того, как модуль был добавлен к компоненту или другому модулю, он уже находится в графе Dagger; Dagger может предоставить эти объекты в этом компоненте. Перед добавлением модуля проверьте, является ли этот модуль частью графа Dagger, проверив, добавлен ли он уже в компонент, или скомпилировав проект и проверив, может ли Dagger найти необходимые зависимости для этого модуля.

Хорошая практика диктует, что модули следует объявлять в компоненте только один раз (за пределами конкретных расширенных вариантов использования Dagger).

Допустим, у вас настроен график таким образом. ApplicationComponent включает в себя Module1и Module2и Module1включает в себя ModuleX.

@Component(modules = [Module1::class, Module2::class])  
interface ApplicationComponent { ... }  
  
@Module(includes = [ModuleX::class])  
class Module1 { ... }  
  
@Module  
class Module2 { ... }

@Component ( modules = [ Module1 :: class , Module2 :: class ]) интерфейс ApplicationComponent { ... } @Module ( includes = [ ModuleX :: class ]) class Module1 { ... } @Module class Module2 { .. . }

Если теперь Module2зависит от классов, предоставленных ModuleX. **Плохая практика** в том числе ModuleXв Module2потому , что ModuleXвходит дважды в графике , как показано в следующем фрагменте кода:

// Bad practice: ModuleX is declared multiple times in this Dagger graph

// Плохая практика: ModuleX объявляется несколько раз в этом графе Dagger  
@Component(modules = [Module1::class, Module2::class])  
interface ApplicationComponent { ... }  
  
@Module(includes = [ModuleX::class])  
class Module1 { ... }  
  
@Module(includes = [ModuleX::class])  
class Module2 { ... }

Вместо этого вы должны сделать одно из следующего:

1. Выполните рефакторинг модулей и извлеките общий модуль из компонента.
2. Создайте новый модуль с объектами, которые используются обоими модулями, и извлеките его в компонент.

Отсутствие рефакторинга таким образом приводит к тому, что множество модулей включают друг друга без четкого представления об организации и затрудняют понимание того, откуда происходит каждая зависимость.

**Хорошая практика (вариант 1)** : ModuleX объявляется один раз в графе Dagger.

@Component(modules = [Module1::class, Module2::class, ModuleX::class])  
interface ApplicationComponent { ... }  
  
@Module  
class Module1 { ... }  
  
@Module  
class Module2 { ... }

**Хорошая практика (вариант 2)** : общие зависимости из Module1и Module2 в ModuleXизвлекаются в новый модуль с именем, ModuleXCommonкоторый включен в компонент. Затем создаются два других модуля с именами ModuleXWithModule1Dependenciesи ModuleXWithModule2Dependenciesзависимостями, специфичными для каждого модуля. Все модули объявляются один раз в графе Dagger.

@Component(modules = [Module1::class, Module2::class, ModuleXCommon::class])  
interface ApplicationComponent { ... }  
  
@Module  
class ModuleXCommon { ... }  
  
@Module  
class ModuleXWithModule1SpecificDependencies { ... }  
  
@Module  
class ModuleXWithModule2SpecificDependencies { ... }  
  
@Module(includes = [ModuleXWithModule1SpecificDependencies::class])  
class Module1 { ... }  
  
@Module(includes = [ModuleXWithModule2SpecificDependencies::class])  
class Module2 { ... }

**Conclusion - Вывод**

Если вы еще этого не сделали, просмотрите [раздел с рекомендациями](https://developer.android.com/training/dependency-injection/dagger-android#best-practices) . Чтобы узнать, как использовать Dagger в приложении для Android, см. [Таблицу кодов использования Dagger в приложении Android](https://codelabs.developers.google.com/codelabs/android-dagger/) .

### https://developer.android.com/codelabs/android-dagger#0     **Using Dagger in multi-module apps**

# Использование Dagger в многомодульных приложениях

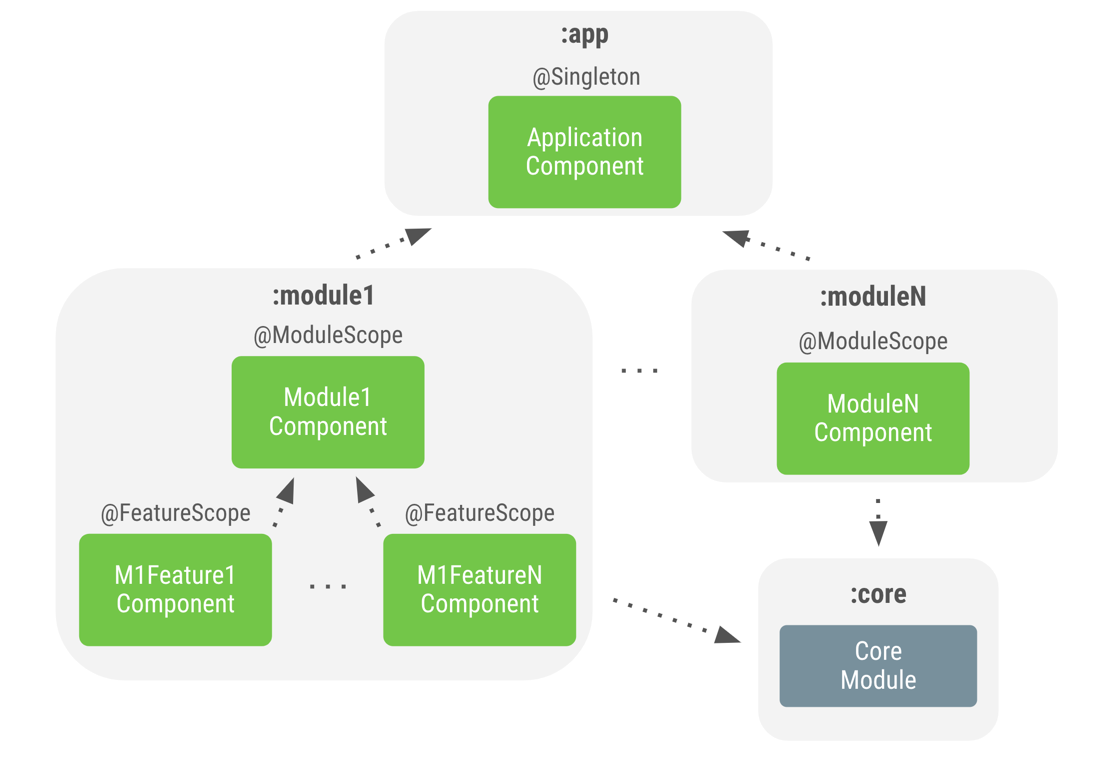
**Примечание: на**этой странице ссылки на модули относятся к модулям Gradle, а не к модулям Dagger.

Проект с несколькими модулями Gradle известен как многомодульный проект. В многомодульном проекте, который поставляется как один APK без функциональных модулей, обычно используется appмодуль, который может зависеть от большинства модулей вашего проекта, и модуль baseor core, от которого обычно зависят остальные модули. appМодуль обычно содержит свой Applicationкласс, в то время как baseмодуль содержит все общие классы общих для всех модулей в проекте.

appМодуль является хорошим местом , чтобы объявить компонент приложения (например, ApplicationComponentв изображении ниже) , которые могут обеспечить объекты, другие компоненты могут понадобиться, а также одиночек вашего приложения. Например, классы, такие как OkHttpClientпарсеры JSON, средства доступа для вашей базы данных или SharedPreferencesобъекты, которые могут быть определены в coreмодуле, будут предоставлены ApplicationComponentопределенными в appмодуле.

В appмодуле могут быть и другие компоненты с более коротким сроком службы. Примером может служить UserComponentконфигурация для конкретного пользователя (например, a UserSession) после входа в систему.

В различных модулях вашего проекта вы можете определить по крайней мере один подкомпонент, логика которого будет специфичной для этого модуля, как показано на рисунке 1.



**Рисунок 1.** Пример графа Dagger в многомодульном проекте.

Например, в loginмодуле у вас может быть LoginComponent область действия с настраиваемой @ModuleScopeаннотацией, которая может предоставлять объекты, общие для этой функции, такие как LoginRepository. Внутри этого модуля вы также можете иметь другие компоненты, зависящие от a, LoginComponentс другой настраиваемой областью, например @FeatureScopeдля a LoginActivityComponentили a, TermsAndConditionsComponentгде вы можете охватить более специфичную для функции логику, такую ​​как ViewModelобъекты.

Для других модулей, таких как Registration, у вас будет аналогичная настройка.

Общее правило для многомодульного проекта - модули одного уровня не должны зависеть друг от друга. Если это так, подумайте, должна ли эта общая логика (зависимости между ними) быть частью родительского модуля. Если это так, выполните рефакторинг, чтобы переместить классы в родительский модуль; в противном случае создайте новый модуль, который расширяет родительский модуль, и оба исходных модуля расширяют новый модуль.

Лучше всего создавать компонент в модуле в следующих случаях:

* Вам нужно выполнить инъекцию поля, как в случае с LoginActivityComponent.
* Вам нужно ограничивать объекты, как в случае с LoginComponent.

Если ни один из этих классов не применим и вам нужно сообщить Dagger, как предоставлять объекты из этого модуля, создайте и откройте модуль Dagger с помощью методов @Providesили, @Bindsесли внедрение конструкции для этих классов невозможно.

## Реализация с подкомпонентами Dagger

На странице документации " [Использование Dagger в приложениях Android"](https://developer.android.com/training/dependency-injection/dagger-android#dagger-subcomponents) рассказывается, как создавать и использовать подкомпоненты. Однако вы не можете использовать тот же код, потому что функциональные модули не знают о appмодуле. В качестве примера, если вы думаете о типичном потоке входа в систему и коде, который мы имеем на предыдущей странице, он больше не компилируется:

[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

class LoginActivity : Activity () { ... override fun onCreate ( savedInstanceState : Bundle ?) { // Создание графа входа в систему с использованием графа приложения     loginComponent = ( applicationContext как MyDaggerApplication ) . appComponent . loginComponent (). create () // Заставляем Dagger создавать экземпляры полей @Inject в LoginActivity     loginComponent . inject ( это ) ... }   
    
  
    
      
   
                          
  
      
  
      
    
}

Причина в том, что loginмодуль не знает MyApplicationни appComponent. Чтобы это работало, вам необходимо определить интерфейс в функциональном модуле, который предоставляет объект, FeatureComponentкоторый MyApplicationнеобходимо реализовать.

В следующем примере вы можете определить LoginComponentProviderинтерфейс, который предоставляет LoginComponentв loginмодуле для потока входа в систему:

[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

интерфейс LoginComponentProvider { fun provideLoginComponent (): LoginComponent }   
   

Теперь он LoginActivityбудет использовать этот интерфейс вместо фрагмента кода, определенного выше:

[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

class LoginActivity : Activity () { ... переопределить развлечение onCreate ( savedInstanceState : Bundle ?) {     loginComponent = ( applicationContext as LoginComponentProvider ) . provideLoginComponent ()     loginComponent . inject ( это ) ... } }   
    
  
    
   
                          
  
  
      
 

Теперь MyApplicationнеобходимо реализовать этот интерфейс и реализовать необходимые методы:

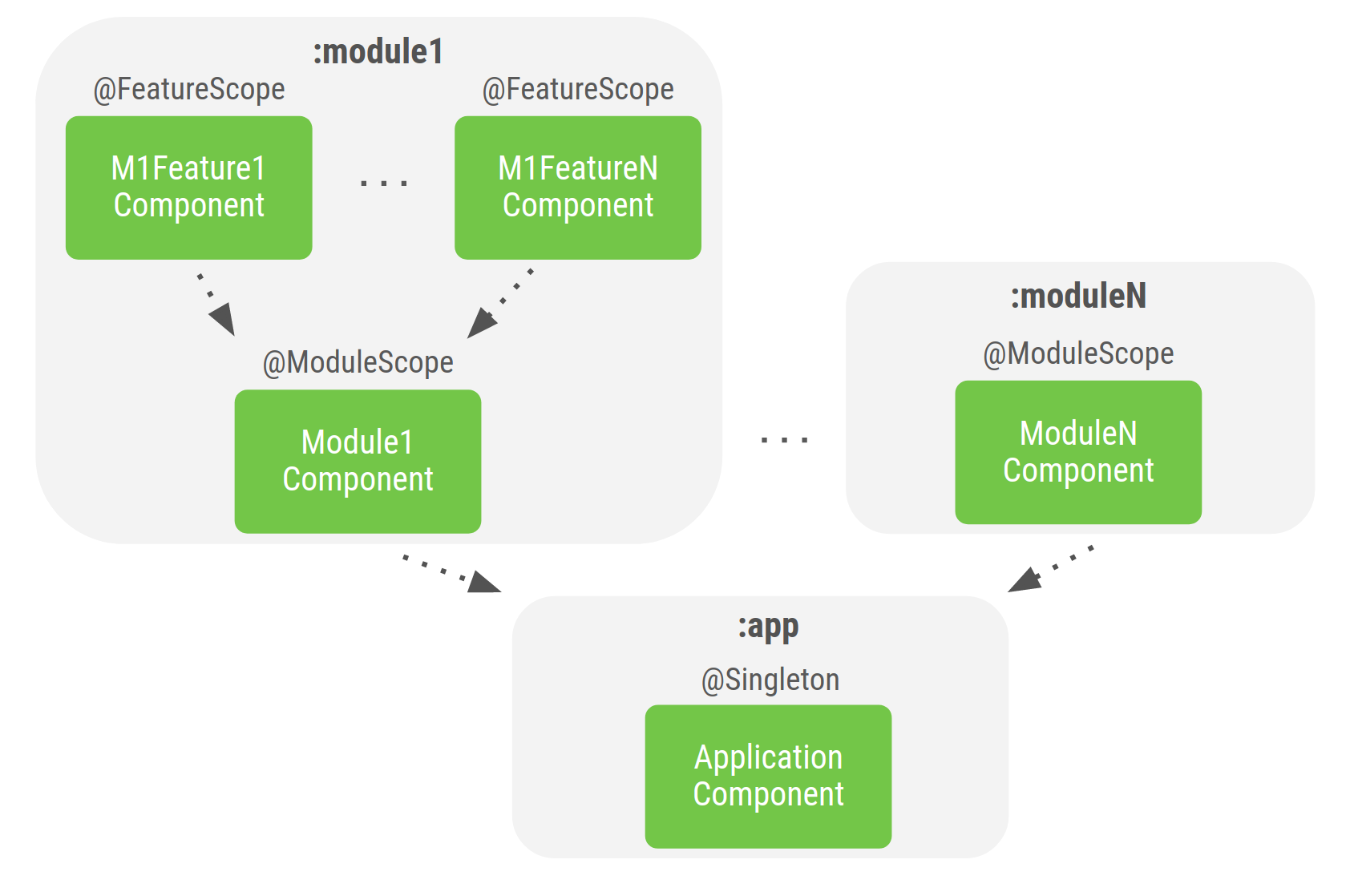
[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

class MyApplication : Application (), LoginComponentProvider { // Ссылка на граф приложения, который используется во всем приложении val appComponent = DaggerApplicationComponent . create () переопределить удовольствие provideLoginComponent (): LoginComponent { return appComponent . loginComponent (). create () } }   
    
    
  
    
      
 

Вот как вы можете использовать подкомпоненты Dagger в многомодульном проекте. С функциональными модулями решение отличается из-за того, как модули зависят друг от друга.

## Зависимости компонентов с функциональными модулями

С [функциональными модулями](https://developer.android.com/studio/projects/dynamic-delivery#customize_delivery) способ, которым модули обычно зависят друг от друга, инвертируется. Вместо appмодуля, включающего функциональные модули, функциональные модули зависят от appмодуля. См. Рисунок 2, где представлена ​​структура модулей.



**Рисунок 2.** Пример графа Dagger в проекте с функциональными модулями

В Dagger компоненты должны знать о своих подкомпонентах. Эта информация включена в модуль Dagger, добавленный к родительскому компоненту (например, SubcomponentsModuleмодуль в разделе [Использование Dagger в приложениях Android](https://developer.android.com/training/dependency-injection/dagger-android#dagger-subcomponents) ).

К сожалению, при обратной зависимости между приложением и функциональным модулем подкомпонент не виден из appмодуля, потому что он не находится в пути сборки. Например, LoginComponentопределенный в loginфункциональном модуле не может быть подкомпонентом ApplicationComponentопределенного в appмодуле.

В Dagger есть механизм, называемый **зависимостями компонентов** , который можно использовать для решения этой проблемы. Вместо того, чтобы дочерний компонент был подкомпонентом родительского компонента, дочерний компонент зависит от родительского компонента. При этом нет родительско-дочерних отношений; теперь **компоненты** зависят от других, чтобы получить определенные **зависимости** . Компоненты должны предоставлять типы из графа, чтобы зависимые компоненты могли их использовать.

**Примечание:**эта проблема возникает всякий раз, когда вы хотите создать подкомпонент **ApplicationComponent**. Если вам нужно создать обычный модуль gradle, который зависит от функционального модуля и должен создать компонент, который зависит от компонента, определенного в этом функциональном модуле, вы можете использовать подкомпоненты как обычно.

Например: функциональный модуль с именем loginхочет создать объект, LoginComponentкоторый зависит от того, что AppComponentдоступно в appмодуле Gradle.

Ниже приведены определения классов и тех, AppComponentкоторые являются частью appмодуля Gradle:

[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

// зависимости UserRepository в класс UserLocalDataSource @Inject конструктор () { ... } класс UserRemoteDataSource @Inject конструктор () { ... } // UserRepository в области видимости AppComponent @Singleton класса UserRepository @Inject конструктор ( частный валь localDataSource : UserLocalDataSource , частный val remoteDataSource : UserRemoteDataSource ) { ... } @Singleton  
   
   
  
  
  
   
      
      
   
  
  
@Component interface AppComponent { ... }

В вашем loginGradle модуль , который включает в себя appGradle модуль, у вас есть , LoginActivityчто нужен LoginViewModelэкземпляр , который будет введен:

[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

// LoginViewModel зависит от UserRepository, область действия которого ограничена классом AppComponent. LoginViewModel @Inject constructor ( private val userRepository : UserRepository ) { ... }  
   
   

LoginViewModelимеет зависимость от того, UserRepositoryчто доступно и ограничено AppComponent. Давайте создадим, LoginComponentкоторый зависит от того, AppComponentчтобы вводить LoginActivity:

[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

// Использование атрибутов зависимостей в компоненте аннотации указать // зависимости этого компонента @Component ( зависимости = [ AppComponent :: класс ]) интерфейс LoginComponent { забавной Inject ( активность : LoginActivity ) }  
  
   
   
   

LoginComponentуказывает зависимость AppComponent, добавляя ее к параметру dependencies аннотации компонента. Поскольку LoginActivityбудет внедрен Dagger, добавьте inject()метод в интерфейс.

**Примечание:**помечено значком, а не [значком, как на странице](https://developer.android.com/training/dependency-injection/dagger-android#dagger-subcomponents) « [Использование кинжала на странице приложения Android»](https://developer.android.com/training/dependency-injection/dagger-android#dagger-subcomponents) . **LoginComponent@Component@Subcomponent**

При создании необходимо передать LoginComponentэкземпляр AppComponent. Для этого используйте фабрику компонентов:

[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

@Component ( dependencies = [ AppComponent :: class ]) интерфейс LoginComponent { @Component . Фабрика интерфейс Factory { // принимает экземпляр AppComponent при создании // экземпляр LoginComponent веселья создать ( appComponent : AppComponent ): LoginComponent } весело инъекционные ( активность : LoginActivity ) }   
   
  
      
      
          
          
          
      
  
   

Теперь LoginActivityможно создать экземпляр LoginComponentи вызвать inject()метод.

[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

class LoginActivity : Activity () { // Вы хотите, чтобы Dagger предоставил экземпляр LoginViewModel из графа входа @Inject lateinit var loginViewModel : LoginViewModel override fun onCreate ( savedInstanceState : Bundle ?) { // Получает appComponent из MyApplication, доступный в базовом Gradle модуль val appComponent = ( applicationContext как MyApplication ). appComponent // Создает новый экземпляр LoginComponent   
  
      
      
  
      
          
          
  
          
        // Внедряет компонент для заполнения полей @Inject DaggerLoginComponent . фабрика (). создать ( appComponent ). залить ( это ) супер . onCreate ( savedInstanceState ) // Теперь вы можете получить доступ к loginViewModel } }  
          
  
          
  
          
   

LoginViewModelзависит от UserRepository; и LoginComponentчтобы иметь возможность получить к нему доступ AppComponent, AppComponentнеобходимо открыть его в своем интерфейсе:

[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

@Singleton @Component interface AppComponent { fun userRepository (): UserRepository }  
  
   
   

Правила области действия с зависимыми компонентами работают так же, как и с подкомпонентами. Поскольку LoginComponentиспользуется экземпляр AppComponent, они не могут использовать одну и ту же аннотацию области.

Если вы хотите , чтобы рамки , LoginViewModelчтобы LoginComponent, вы могли бы сделать это , как вы делали ранее с использованием пользовательских @ActivityScopeаннотаций.

[КОТЛИН](https://developer.android.com/training/dependency-injection/dagger-multi-module#kotlin)[ЯВА](https://developer.android.com/training/dependency-injection/dagger-multi-module#java)

@ActivityScope @Component ( dependencies = [ AppComponent :: class ]) interface LoginComponent { ... } @ActivityScope class LoginViewModel @Inject constructor ( private val userRepository : UserRepository ) { ... }  
   
   
  
  
   
   

**Примечание.**Не раскрытие всех типов, которые требуются зависимому компоненту от компонента, приведет к ошибке времени компиляции Dagger, поскольку он не может предоставить определенные типы для зависимого компонента.

## Лучшие практики

* ApplicationComponentВсегда должен находиться в appмодуле.
* Создавайте компоненты Dagger в модулях, если вам нужно выполнить внедрение поля в этом модуле или вам нужно ограничить объекты для определенного потока вашего приложения.
* Для модулей Gradle, которые предназначены для использования в качестве утилит или помощников и не нуждаются в построении графа (вот почему вам понадобится компонент Dagger), создайте и предоставьте общедоступные модули Dagger с помощью методов @Provides и @Binds тех классов, которые не Не поддерживает внедрение конструктора.
* Чтобы использовать Dagger в приложении Android с функциональными модулями, используйте зависимости компонентов, чтобы иметь возможность доступа к зависимостям, предоставляемым ApplicationComponentопределенными в appмодуле.

### **…..**