# Dagger – Кинжал

Dagger - это полностью статический фреймворк для [внедрения зависимостей во](http://en.wikipedia.org/wiki/Dependency_injection) время компиляции для Java, Kotlin и Android. Это адаптация более [ранней версии,](https://github.com/square/dagger) созданной [Square](http://square.github.io/) и теперь поддерживаемой Google.

Dagger направлен на решение многих проблем разработки и производительности, которые мешали решениям, основанным на отражении. Более подробную информацию можно найти в [этом докладе](https://www.youtube.com/watch?v=oK_XtfXPkqw) ( [слайды](https://docs.google.com/presentation/d/1fby5VeGU9CN8zjw4lAb2QPPsKRxx6mSwCe9q7ECNSJQ/pub?start=false&loop=false&delayms=3000) ) [Грегори Кика](https://twitter.com/gk5885) .

**Где документы?**

* [Пользовательская документация](https://dagger.dev/users-guide)
* [Dagger API @ HEAD](https://dagger.dev/api/latest/)

**Где код?**

* <https://github.com/google/dagger>

**Есть вопрос?**

* Задайте его на Stack Overflow с тегом [dagger-2](http://stackoverflow.com/questions/tagged/dagger-2)
* Отправьте электронное письмо на адрес  
   [dagger-discuss@googlegroups.com](https://groups.google.com/forum/#!forum/dagger-discuss)

# Hilt Рукоять

Hilt предоставляет стандартный способ включения внедрения зависимостей Dagger в приложение Android.

*Цели Hilt:*

* Упростить инфраструктуру, связанную с Dagger, для приложений Android.
* Чтобы создать стандартный набор компонентов и областей для упрощения настройки, удобства чтения / понимания и совместного использования кода между приложениями.
* Обеспечить простой способ предоставления различных привязок к различным типам сборки (например, тестирование, отладка или выпуск).

**Обзор конструкции рукояти**

Hilt работает с помощью кода, генерирующего для вас код установки Dagger. Это убирает большую часть шаблонов использования Dagger и действительно просто оставляет аспекты определения того, как создавать объекты и куда их вставлять. Hilt сгенерирует компоненты Dagger и код для автоматического внедрения ваших классов Android (например, действий и фрагментов) для вас.

Hilt генерирует набор стандартных компонентов Android Dagger на основе вашего транзитивного пути к классам. Для этого необходимо пометить ваши модули Dagger аннотациями Hilt, чтобы сообщить Hilt, в какой компонент они должны входить. Получение объектов в классах вашей платформы Android выполняется с помощью другой аннотации Hilt, которая генерирует код внедрения Dagger в базовый класс, который вы расширяете. Для пользователей Gradle расширение этого класса выполняется с помощью [преобразования байт-кода](https://dagger.dev/hilt/gradle-setup.html#hilt-gradle-plugin) под капотом.

В ваших тестах Hilt также генерирует для вас компоненты Dagger, как и в производственной среде. В тестах есть и другие специальные утилиты, помогающие добавлять или заменять привязки тестов.

# Преимущества использования Hilt

Зачем использовать Hilt?

* Уменьшенный шаблон
* Независимые зависимости сборки
* Упрощенная конфигурация
* Улучшенное тестирование
* Стандартные компоненты

## Уменьшенный шаблон

Цель Hilt - позволить пользователям сосредоточиться на определениях и использовании привязки Dagger, не беспокоясь об остальной части настройки Dagger. Это означает сокрытие таких вещей, как определения компонентов с помощью списков модулей и интерфейсов, кода для создания и удержания компонентов в нужных точках жизненного цикла, интерфейсов и приведений для получения родительского компонента и т. Д.

Некоторая простота также исходит от Hilt, использующего монолитные компоненты (то есть использование одного компонента для всех действий, одного компонента для всех фрагментов и т. Д.). Hilt пытается создать по существу глобальное пространство имен привязки, чтобы было легко узнать, какое определение привязки используется, без необходимости отслеживать, из какого действия или фрагмента вы были внедрены. Подробнее об этом дизайнерском решении читайте [здесь](https://dagger.dev/hilt/monolithic.html) .

**Независимые зависимости сборки**

Наивное использование Dagger может вызвать проблемы сборки, если код напрямую ссылается на компонент Dagger. Эти проблемы возникают из-за того, что компонент Dagger имеет ссылки на все установленные модули. Это может привести к раздутым зависимостям, замедляющим сборку. Естественный способ решить эту проблему - это интерфейсы и небезопасное приведение типов. Это компромисс, потому что это может привести к ошибкам во время выполнения. Например, введение нового интерфейса инжектора позволяет избежать прямой зависимости от компонента, но затем, если вы забываете расширить интерфейс инжектора для вашего компонента, это приведет к исключению приведения.

С помощью кода, генерирующего интерфейсы, небезопасные преобразования и списки модулей / интерфейсов под капотом, Hilt делает эти небезопасные преобразования во время выполнения безопасными благодаря гарантиям генерации кода и обнаружения модулей / точек входа.

**Конфигурация**

Приложения часто имеют разные конфигурации сборок, например производственная сборка или сборка для разработки с разными функциями. Эти разные наборы функций часто означают другой набор модулей Dagger. В обычной сборке Dagger для другого набора модулей требуется отдельное дерево компонентов (отдельный компонент для каждой области), обычно с большим количеством повторяющихся частей. Поскольку Hilt устанавливает модули через зависимости сборки, а код генерирует компоненты, создание другой версии вашей сборки так же просто, как компиляция с добавленной или удаленной зависимостью.

**Тестирование**

Тестирование с Dagger может быть трудным из-за проблемы с конфигурацией, упомянутой выше. Hilt аналогичным образом упрощает замену тестовых модулей и привязок за счет генерации кода компонентов. В Hilt встроены специальные тестовые утилиты, которые упрощают управление модулями и привязку тестов, чтобы тесты могли использовать Dagger. Использование Dagger в тестах помогает уменьшить количество шаблонов в тестах и ​​делает тесты более надежными за счет создания экземпляров кода так же, как он создается в производственной среде.

**Стандартизация**

Hilt стандартизирует [иерархию компонентов](https://dagger.dev/hilt/components.html#component-heirarchy) . Это означает, что библиотеки, которые интегрируются с Hilt, могут легко добавлять или использовать привязки из этих известных компонентов. Это позволяет создавать более сложные библиотеки, которые можно легко и просто интегрировать в любое приложение Hilt.

# Настройка сборки Gradle

## Использование Hilt с Kotlin

Если вы используете Kotlin, примените [плагин kapt](https://kotlinlang.org/docs/reference/kapt.html) и объявите зависимость компилятора, используя kaptвместо annotationProcessor.

Дополнительно настройте kapt для исправления типов ошибок, задав [correctErrorTypes](https://kotlinlang.org/docs/reference/kapt.html#non-existent-type-correction) значение true.

dependencies **{**

implementation 'com.google.dagger:hilt-android:<VERSION>'

kapt 'com.google.dagger:hilt-android-compiler:<VERSION>'

*// For instrumentation tests*

androidTestImplementation 'com.google.dagger:hilt-android-testing:<VERSION>'

kaptAndroidTest 'com.google.dagger:hilt-android-compiler:<VERSION>'

*// For local unit tests*

testImplementation 'com.google.dagger:hilt-android-testing:<VERSION>'

kaptTest 'com.google.dagger:hilt-android-compiler:<VERSION>'

**}**

kapt **{**

correctErrorTypes **true**

**}**

## Плагин Hilt Gradle

Плагин Hilt Gradle выполняет преобразование байт-кода, чтобы упростить использование API. Плагин был создан для лучшего взаимодействия с разработчиками в среде IDE, поскольку сгенерированный класс может нарушить завершение кода для методов базового класса. Примеры в документации предполагают использование плагина. Чтобы настроить плагин Hilt Gradle, сначала объявите зависимость в корневом build.gradleфайле вашего проекта :

buildscript **{**

repositories **{**

*// other repositories...*

mavenCentral**()**

**}**

dependencies **{**

*// other plugins...*

classpath 'com.google.dagger:hilt-android-gradle-plugin:<version>'

**}**

**}**

затем в build.gradleваших модулях Android Gradle примените плагин:

apply plugin: 'com.android.application'

apply plugin: 'dagger.hilt.android.plugin'

android **{**

*// ...*

**}**

**Предупреждение:** плагин Hilt Gradle устанавливает аргументы обработчика аннотаций. Если вы используете другие библиотеки, которым требуются аргументы процессора аннотаций, убедитесь, что вы добавляете аргументы, а не переопределяете их. См [ниже](https://dagger.dev/hilt/gradle-setup#applying-other-processor-arguments) в качестве примера.

**Зачем использовать плагин?**

Одним из основных преимуществ плагина Gradle является то, что он упрощает @AndroidEntryPointи @HiltAndroidAppупрощает использование. Без подключаемого модуля Gradle базовый класс должен быть указан в аннотации, а аннотированный класс должен расширять сгенерированный класс:

Ява

Котлин

@HiltAndroidApp(**MultiDexApplication::class**)

**class** **MyApplication** : **Hilt\_MyApplication**()

С плагином Gradle аннотированный класс может напрямую расширять базовый класс:

Ява

Котлин

@HiltAndroidApp

**class** **MyApplication** : **MultiDexApplication**()

**Конфигурация локального теста**

По умолчанию плагин преобразует *инструментальные* тестовые классы (обычно находятся в androidTestисходной папке), но для преобразования *локальных* тестов *jvm* (обычно расположенных в testисходной папке) плагину требуется дополнительная конфигурация .

Чтобы включить преобразование @AndroidEntryPointклассов в локальных jvm-тестах, примените следующую конфигурацию в вашем модуле build.gradle:

hilt {

enableTransformForLocalTests = true

}

**Предупреждение:**enableTransformForLocalTests конфигурация работает только при запуске из командной строки, например ./gradlew test. Он не работает при запуске тестов с Android Studio (с помощью кнопки воспроизведения в тестовом методе или классе). См. Раздел [Выполнение тестов с Android Studio](https://dagger.dev/hilt/gradle-setup#running-with-android-studio) для получения дополнительной информации и возможных решений.

### Запуск тестов с Android Studio

### Применение других аргументов процессора

Плагин Hilt Gradle устанавливает аргументы процессора аннотаций. Если вы используете другие библиотеки, которым требуются аргументы процессора аннотаций, убедитесь, что вы добавляете аргументы, а не заменяете их.

Например, следующее, в частности, используется, +=чтобы избежать переопределения аргументов Hilt.

javaCompileOptions **{**

annotationProcessorOptions **{**

arguments **+=** **[**"foo" **:** "bar"**]**

**}**

**}**

Если они +отсутствуют и argumentsпереопределены, вероятно, Hilt не сможет скомпилировать с ошибками, подобными следующим:Expected @HiltAndroidApp to have a value. Did you forget to apply the Gradle Plugin?

# Краткое руководство

## Введение

Hilt упрощает добавление инъекции зависимостей в ваше приложение для Android. Это руководство поможет вам настроить существующее приложение для использования Hilt.

Чтобы узнать больше об основных концепциях компонентов Hilt, ознакомьтесь с [компонентами Hilt](https://dagger.dev/hilt/components.html) .

## Gradle против пользователей, не являющихся Gradle

Для пользователей [Gradle плагин Hilt Gradle](https://dagger.dev/hilt/gradle-setup.html#hilt-gradle-plugin) упрощает использование некоторых аннотаций Hilt, избегая ссылок на классы, созданные Hilt.

Без подключаемого модуля Gradle базовый класс должен быть указан в аннотации, а аннотированный класс должен расширять сгенерированный класс:

Ява

Котлин

@HiltAndroidApp(**MultiDexApplication::class**)

**class** **MyApplication** : **Hilt\_MyApplication**()

С плагином Gradle аннотированный класс может напрямую расширять базовый класс:

Ява

Котлин

@HiltAndroidApp

**class** **MyApplication** : **MultiDexApplication**()

Дальнейшие примеры предполагают использование плагина Hilt Gradle.

## Применение рукояти

Все приложения, использующие Hilt, должны содержать [Application](https://developer.android.com/reference/android/app/Application.html) класс, помеченный значком [@HiltAndroidApp](https://dagger.dev/api/latest/dagger/hilt/android/HiltAndroidApp.html). @HiltAndroidAppзапускает генерацию кода [компонентов Hilt,](https://dagger.dev/hilt/components.html) а также создает базовый класс для вашего приложения, который использует эти сгенерированные компоненты. Поскольку для генерации кода необходим доступ ко всем вашим модулям, цель, которая компилирует ваш Application класс, также должна иметь все ваши модули Dagger в своих транзитивных зависимостях.

Как и другие [точки входа в](https://dagger.dev/hilt/android-entry-point.html) Hilt [Android](https://dagger.dev/hilt/android-entry-point.html) , приложения также являются участниками. Это означает, что вы можете использовать введенные поля в приложении после super.onCreate()вызова.

Например, возьмите класс, MyApplicationкоторый расширяется MyBaseApplicationи имеет переменную-член Bar:

Ява

Котлин

**class** **MyApplication** : **MyBaseApplication**() {

@Inject **lateinit** **var** bar: **Bar**

**override** **fun** **onCreate**() {

**super**.**onCreate**()

**val** myComponent =

**DaggerMyComponent**

.**builder**()

**..**.

.**build**()

myComponent.**inject**(**this**)

}

}

После внедрения членов Hilt код выше становится следующим:

Ява

Котлин

@HiltAndroidApp

**class** **MyApplication** : **MyBaseApplication**() {

@Inject **lateinit** **var** bar: **Bar**

**override** **fun** **onCreate**() {

**super**.**onCreate**() *// Injection happens in super.onCreate()*

*// Use bar*

}

}

Для получения дополнительной информации см. [Приложение Hilt](https://dagger.dev/hilt/application.html) .

## @AndroidEntryPoint

После того, как вы включили внедрение членов в свой Application, вы можете начать включение внедрения членов в другие классы Android, используя [@AndroidEntryPoint](https://dagger.dev/api/latest/dagger/hilt/android/AndroidEntryPoint.html) аннотацию. Вы можете использовать @AndroidEntryPointследующие типы:

1. Деятельность
2. Фрагмент
3. Посмотреть
4. обслуживание
5. BroadcastReceiver

ViewModels не поддерживаются напрямую, но вместо этого поддерживаются [расширением Jetpack](https://developer.android.com/training/dependency-injection/hilt-jetpack) . В следующем примере показано, как добавить аннотацию к действию, но процесс такой же для других типов.

Чтобы включить внедрение членов в вашу деятельность, аннотируйте свой класс с помощью [@AndroidEntryPoint](https://dagger.dev/api/latest/dagger/hilt/android/AndroidEntryPoint.html).

Ява

Котлин

@AndroidEntryPoint

**class** **MyActivity** : **MyBaseActivity**() {

@Inject **lateinit** **var** bar: **Bar** *// Bindings in SingletonComponent or ActivityComponent*

**override** **fun** **onCreate**() {

*// Injection happens in super.onCreate().*

**super**.**onCreate**()

*// Do something with bar ...*

}

}

**Примечание. В** настоящее время Hilt поддерживает только действия, которые расширяют [ComponentActivity](https://developer.android.com/reference/androidx/activity/ComponentActivity)и фрагменты, которые расширяют библиотеку androidx [Fragment](https://developer.android.com/reference/androidx/fragment/app/Fragment), а не (теперь не рекомендуется) [Fragment](https://developer.android.com/reference/android/app/Fragment)на платформе Android.

Для получения дополнительной информации см. [@AndroidEntryPoint](https://dagger.dev/hilt/android-entry-point.html) .

## Модули рукояти

Модули Hilt - это стандартные модули Dagger, которые имеют дополнительную @InstallIn аннотацию, определяющую, в какие [компоненты Hilt](https://dagger.dev/hilt/components.html#hilt-components) следует установить модуль.

Когда компоненты Hilt сгенерированы, модули, помеченные значком, @InstallIn будут установлены в соответствующий компонент или подкомпонент через @Component#modulesили @Subcomponent#modulesсоответственно. Как и в Dagger, установка модуля в компонент позволяет получить доступ к этой привязке как к зависимости от других привязок в этом компоненте или любом дочернем компоненте (ах) ниже его в [иерархии компонентов](https://dagger.dev/hilt/components.html#component-hierarchy) . К ним также можно получить доступ из соответствующих @AndroidEntryPointклассов. Установка в компонент также позволяет привязать эту привязку к этому компоненту.

### С помощью @InstallIn

Модуль устанавливается в [Hilt Component](https://dagger.dev/hilt/components.html) , аннотируя модуль [@InstallIn](https://dagger.dev/api/latest/dagger/hilt/InstallIn.html) аннотацией. Эти аннотации требуются для всех модулей Dagger при использовании Hilt, но эту проверку можно дополнительно [отключить](https://dagger.dev/hilt/compiler-options.html#disable-install-in-check) .

**Примечание.** Если у модуля нет @InstallInаннотации, он не будет частью компонента и может привести к ошибкам компиляции.

Укажите, в какой компонент рукоятки следует установить модуль, передав соответствующий тип (ы) [компонента](https://dagger.dev/hilt/components.html) в @InstallInаннотацию. Например, чтобы установить модуль, чтобы все в приложении могло его использовать, используйте SingletonComponent:

Ява

Котлин

@Module

@InstallIn(**SingletonComponent::class**) *// Installs FooModule in the generate SingletonComponent.*

**object** **FooModule** {

@Provides

**fun** **provideBar**(): **Bar** {**..**.}

}

Для получения дополнительной информации см. [Модули рукояти](https://dagger.dev/hilt/modules.html) .

# Компоненты рукояти

**Примечание.** На следующей странице предполагается наличие базовых знаний о Dagger, включая компоненты, модули, области действия и привязки. (Дополнительную информацию см. В [руководстве пользователя Dagger](https://dagger.dev/dev-guide) .)

## Иерархия компонентов

В отличие от традиционного Dagger, пользователи Hilt никогда не определяют и не создают экземпляры компонентов Dagger напрямую. Вместо этого Hilt предлагает готовые компоненты, которые созданы для вас. Hilt поставляется со встроенным набором компонентов (и соответствующими аннотациями областей видимости), которые автоматически интегрируются в различные жизненные циклы приложения Android. На схеме ниже показана стандартная иерархия компонентов Hilt. Аннотации над каждым компонентом - это аннотация области видимости, используемая для привязки области видимости к времени жизни этого компонента. Стрелка под компонентом указывает на любые дочерние компоненты. Как обычно, привязка в дочернем компоненте может иметь зависимости от любой привязки в родительском компоненте.

**Примечание.** При определении области действия привязки в @InstallInмодуле область действия привязки должна соответствовать области [действия компонента](https://dagger.dev/hilt/components#component-lifetimes) . Например, привязка внутри @InstallIn(ActivityComponent.class)модуля может быть ограничена только областью @ActivityScoped.

## Внедрение компонентов компонентов

В этом [@AndroidEntryPoint](https://dagger.dev/hilt/android-entry-point.html)разделе показано, как внедрить ваши классы Android с помощью инъекции членов. Компоненты Hilt - это те, которые отвечают за внедрение своих привязок в ваши классы Android. Каждый компонент отвечает за внедрение разных типов классов Android. Это показано в таблице ниже:

| **Составная часть** | **Инжектор для** |
| --- | --- |
| **SingletonComponent** | Application |
| **ActivityRetainedComponent** | ViewModel(см. [Расширение модели просмотра](https://developer.android.com/training/dependency-injection/hilt-jetpack) ) |
| **ActivityComponent** | Activity |
| **FragmentComponent** | Fragment |
| **ViewComponent** | View |
| **ViewWithFragmentComponent** | View с участием @WithFragmentBindings |
| **ServiceComponent** | Service |

## Срок службы компонентов

Время жизни компонента важно, потому что оно связано со сроком жизни ваших привязок двумя важными способами:

1. Он ограничивает время существования привязок с заданной областью между моментом создания компонента и его уничтожением.
2. Он указывает, когда могут использоваться введенные членами значения (например, когда @Inject поля не равны нулю).

Время жизни компонентов обычно ограничивается созданием и уничтожением соответствующего экземпляра класса Android. В таблице ниже перечислены аннотации области и ограниченное время жизни для каждого компонента.

| **Составная часть** | **Объем** | **Создан в** | **Разрушен в** |
| --- | --- | --- | --- |
| **SingletonComponent** | @Singleton | Application#onCreate() | Application#onDestroy() |
| **ActivityRetainedComponent** | @ActivityRetainedScope | Activity#onCreate()[1](https://dagger.dev/hilt/components#fn:1) | Activity#onDestroy()[1](https://dagger.dev/hilt/components#fn:1) |
| **ActivityComponent** | @ActivityScoped | Activity#onCreate() | Activity#onDestroy() |
| **FragmentComponent** | @FragmentScoped | Fragment#onAttach() | Fragment#onDestroy() |
| **ViewComponent** | @ViewScoped | View#super() | View уничтожен |
| **ViewWithFragmentComponent** | @ViewScoped | View#super() | View уничтожен |
| **ServiceComponent** | @ServiceScoped | Service#onCreate() | Service#onDestroy() |

#### Привязки с ограничением по сравнению с привязками без области действия

По умолчанию все привязки в Dagger не имеют области действия. Это означает, что каждый раз при запросе привязки Dagger создает новый экземпляр привязки.

Однако Dagger также позволяет привязать привязку к определенному компоненту (см. Аннотации области в таблице выше). Привязка с областью действия будет создана только один раз для каждого экземпляра компонента, для которого она привязана, и все запросы для этой привязки будут использовать один и тот же экземпляр.

Пример:

Ява

Котлин

*// This binding is "unscoped".*

*// Each request for this binding will get a new instance.*

**class** **UnscopedBinding** @Inject **constructor**() {

}

*// This binding is "scoped".*

*// Each request from the same component instance for this binding will*

*// get the same instance. Since this is the fragment component, this means*

*// each request from the same fragment.*

@FragmentScoped

**class** **ScopedBinding** @Inject **constructor**() {

}

**Предупреждение.** Распространенное заблуждение состоит в том, что все экземпляры фрагментов будут использовать один и тот же экземпляр привязки с областью действия @FragmentScoped. Однако это не так. Каждый экземпляр фрагмента получает новый экземпляр компонента фрагмента и, следовательно, новый экземпляр всех его привязок с заданной областью.

#### Объем в модулях

В предыдущем разделе показано, как ограничить область привязки, объявленной с помощью @Inject конструктора, но привязка, объявленная в модуле, также может иметь область действия аналогичным образом.

Пример:

Ява

Котлин

@Module

@InstallIn(**FragmentComponent**.**class**)

**object** **FooModule** {

*// This binding is "unscoped".*

@Provides

**fun** **provideUnscopedBinding**() = **UnscopedBinding**()

*// This binding is "scoped".*

@Provides

@FragmentScoped

**fun** **provideScopedBinding**() = **ScopedBinding**()

}

**Предупреждение:** Распространенное заблуждение состоит в том, что все привязки, объявленные в модуле, будут привязаны к компоненту, в котором установлен модуль. Однако это неверно. Только объявления привязок, помеченные аннотацией области видимости, будут иметь область видимости.

#### Когда осматривать?

Определение области действия привязки связано как с размером сгенерированного кода, так и с его производительностью во время выполнения, поэтому используйте область видимости экономно. Общее правило для определения того, следует ли ограничивать область действия привязки, заключается в том, чтобы ограничивать область действия привязки только в том случае, если это требуется для правильности кода. Если вы считаете, что область привязки должна быть ограничена исключительно из соображений производительности, сначала убедитесь, что производительность является проблемой, и если ее можно использовать @Reusableвместо области компонента.

## Привязки компонентов по умолчанию

Каждый компонент Hilt поставляется с набором привязок по умолчанию, которые можно внедрить в качестве зависимостей в ваши собственные пользовательские привязки.

| **Составная часть** | **Привязки по умолчанию** |
| --- | --- |
| **SingletonComponent** | Application[2](https://dagger.dev/hilt/components#fn:2) |
| **ActivityRetainedComponent** | Application |
| **ActivityComponent** | Application, Activity |
| **FragmentComponent** | Application, Activity,Fragment |
| **ViewComponent** | Application, Activity,View |
| **ViewWithFragmentComponent** | Application, Activity, Fragment,View |
| **ServiceComponent** | Application, Service |

1. ActivityRetainedComponentживет при изменении конфигурации, поэтому он создается при первом onCreate и последнем onDestroy. [↩](https://dagger.dev/hilt/components#fnref:1) [↩ 2](https://dagger.dev/hilt/components#fnref:1:1)
2. ApplicationСвязывание доступно с использованием либо @ApplicationContext Contextили Application. [↩](https://dagger.dev/hilt/components#fnref:2)

Применение рукояти

**Примечание.** Примеры на этой странице предполагают использование подключаемого модуля Gradle. Если вы **не** используете плагин, прочтите эту [страницу](https://dagger.dev/hilt/gradle-setup.html#hilt-gradle-plugin) для получения подробной информации.

**Применение рукояти**

Все приложения, использующие Hilt, *должны* содержать [Application](https://developer.android.com/reference/android/app/Application.html) класс, помеченный значком [@HiltAndroidApp](https://dagger.dev/api/latest/dagger/hilt/android/HiltAndroidApp.html). @HiltAndroidAppзапускает генерацию кода [компонентов Hilt,](https://dagger.dev/hilt/components.html) а также создает базовый класс для вашего приложения, который использует эти сгенерированные компоненты. Поскольку для генерации кода необходим доступ ко всем вашим модулям, цель, которая компилирует ваш Application класс, также должна иметь все ваши модули Dagger в своих транзитивных зависимостях.

Как и другие [точки входа в](https://dagger.dev/hilt/android-entry-point.html) Hilt [Android](https://dagger.dev/hilt/android-entry-point.html) , приложения также являются участниками. Это означает, что вы можете использовать введенные поля в приложении после super.onCreate()вызова.

Например, возьмите класс, MyApplicationкоторый расширяется MyBaseApplicationи имеет переменную-член Bar:

Ява

Котлин

**class** **MyApplication** : **MyBaseApplication**() {

@Inject **lateinit** **var** bar: **Bar**

**override** **fun** **onCreate**() {

**super**.**onCreate**()

**val** myComponent =

**DaggerMyComponent**

.**builder**()

**..**.

.**build**()

myComponent.**inject**(**this**)

}

}

После внедрения членов Hilt код выше становится следующим:

Ява

Котлин

@HiltAndroidApp

**class** **MyApplication** : **MyBaseApplication**() {

@Inject **lateinit** **var** bar: **Bar**

**override** **fun** **onCreate**() {

**super**.**onCreate**() *// Injection happens in super.onCreate()*

*// Use bar*

}

}

Точки входа в Android

**Примечание.** Примеры на этой странице предполагают использование подключаемого модуля Gradle. Если вы **не** используете плагин, прочтите эту [страницу](https://dagger.dev/hilt/gradle-setup.html#hilt-gradle-plugin) для получения подробной информации.

**Типы Android**

После того, как вы включили внедрение членов в свой Application, вы можете начать включение внедрения членов в другие классы Android, используя [@AndroidEntryPoint](https://dagger.dev/api/latest/dagger/hilt/android/AndroidEntryPoint.html) аннотацию. Вы можете использовать @AndroidEntryPointследующие типы:

1. Деятельность
2. Фрагмент
3. Посмотреть
4. обслуживание
5. BroadcastReceiver [1](https://dagger.dev/hilt/android-entry-point#fn:1)

ViewModels не поддерживаются напрямую, но вместо этого поддерживаются [расширением Jetpack](https://developer.android.com/training/dependency-injection/hilt-jetpack) . ContentProviders не поддерживаются напрямую, так как onCreateони вызываются при запуске, но вы можете получить доступ к зависимостям через [точку входа](https://dagger.dev/hilt/entry-points.html) .

В следующем примере показано, как добавить аннотацию к действию, но процесс такой же для других типов.

Чтобы включить внедрение членов в вашу деятельность, аннотируйте свой класс с помощью [@AndroidEntryPoint](https://dagger.dev/api/latest/dagger/hilt/android/AndroidEntryPoint.html).

Ява

Котлин

@AndroidEntryPoint

**class** **MyActivity** : **MyBaseActivity**() {

*// Bindings in SingletonComponent or ActivityComponent*

@Inject **lateinit** **var** bar: **Bar**

**override** **fun** **onCreate**(savedInstanceState: **Bundle**?) {

*// Injection happens in super.onCreate().*

**super**.**onCreate**()

*// Do something with bar ...*

}

}

**Примечание. В** настоящее время Hilt поддерживает только действия, которые расширяют [ComponentActivity](https://developer.android.com/reference/androidx/activity/ComponentActivity)и фрагменты, которые расширяют библиотеку androidx [Fragment](https://developer.android.com/reference/androidx/fragment/app/Fragment), а не (теперь не рекомендуется) [Fragment](https://developer.android.com/reference/android/app/Fragment)на платформе Android.

**Сохраненные фрагменты**

Вызов setRetainInstance(true)в виде фрагмента по onCreateметоду будет держать фрагмент экземпляра через изменения конфигурации (вместо того , чтобы уничтожать и воссоздавать его).

Фрагмент Hilt *никогда не* следует сохранять, потому что он содержит ссылку на компонент (отвечающий за внедрение), и этот компонент содержит ссылки на предыдущий экземпляр Activity. Кроме того, привязки и поставщики с ограниченной областью видимости, которые вводятся во фрагмент, также могут вызывать утечки памяти, если фрагмент Hilt сохраняется. Чтобы предотвратить сохранение фрагментов рукояти, при изменении конфигурации будет выдано исключение времени выполнения, если обнаружен сохраненный фрагмент рукояти.

Фрагмент без рукояти *может* быть сохранен, даже если он прикреплен к действию рукояти. Однако если этот фрагмент содержит дочерний фрагмент Hilt, при изменении конфигурации будет выдано исключение времени выполнения.

**Примечание.** Хотя это не рекомендуется, фрагменты рукояти *можно* отсоединять и повторно присоединять к *одному* и тому *же* экземпляру действия. В этом случае фрагмент рукояти будет введен только при первом вызове onAttach. Обратите внимание, что это не то же самое, что сохранение фрагмента, потому что сохраненный фрагмент будет повторно присоединен к другому экземпляру действия. Опять же, это не рекомендуется, и зачастую проще просто создавать новый экземпляр фрагмента для каждого использования.

**Представления с привязками фрагментов**

По умолчанию в представление можно вводить только привязки SingletonComponentи ActivityComponent. Чтобы включить привязки фрагментов в вашем представлении, добавьте @WithFragmentBindingsаннотацию в свой класс.

Ява

Котлин

@AndroidEntryPoint

@WithFragmentBindings

**class** **MyView** : **MyBaseView** {

*// Bindings in SingletonComponent, ActivityComponent,*

*// FragmentComponent, and ViewComponent*

@Inject **lateinit** **var** bar: **Bar**

**constructor**(context: **Context**) : **super**(context)

**constructor**(context: **Context**, attrs: **AttributeSet**?) : **super**(context, attrs)

**init** {

*// Do something with bar ...*

}

**override** **fun** **onFinishInflate**() {

**super**.**onFinishInflate**();

*// Find & assign child views from the inflated hierarchy.*

}

}

1. В отличие от других поддерживаемых классов Android, BroadcastReceiversони не имеют собственного компонента Dagger, а просто вводятся из SingletonComponent. [↩](https://dagger.dev/hilt/android-entry-point#fnref:1)

# Модули

  [НАСТРОЙКА GRADLE](https://dagger.dev/hilt/gradle-setup)

 [БЫСТРЫЙ СТАРТ](https://dagger.dev/hilt/quick-start)

 ОСНОВНЫЕ API

 [→ КОМПОНЕНТЫ](https://dagger.dev/hilt/components)

 [→ ПРИМЕНЕНИЕ РУКОЯТИ](https://dagger.dev/hilt/application)

 [→ ТОЧКИ ВХОДА ДЛЯ ANDROID](https://dagger.dev/hilt/android-entry-point)

 [→ МОДУЛИ](https://dagger.dev/hilt/modules)

 [→ ПУНКТЫ ВЪЕЗДА](https://dagger.dev/hilt/entry-points)

 [→ ПОЛЬЗОВАТЕЛЬСКИЕ КОМПОНЕНТЫ](https://dagger.dev/hilt/custom-components)

 ТЕСТИРОВАНИЕ

 [→ ОБЗОР ТЕСТИРОВАНИЯ](https://dagger.dev/hilt/testing)

 [→ РОБОЭЛЕКТРИЧЕСКИЕ ИСПЫТАНИЯ](https://dagger.dev/hilt/robolectric-testing)

 [→ КИПИА](https://dagger.dev/hilt/instrumentation-testing)

 МИГРАЦИЯ

 [→ ПУТЕВОДИТЕЛЬ](https://dagger.dev/hilt/migration-guide)

 [→ ДОПОЛНИТЕЛЬНАЯ ИНЪЕКЦИЯ](https://dagger.dev/hilt/optional-inject)

 [→ ПСЕВДОНИМЫ ОБЛАСТЕЙ ВИДИМОСТИ](https://dagger.dev/hilt/scope-aliases)

 [ПАРАМЕТРЫ КОМПИЛЯТОРА](https://dagger.dev/hilt/compiler-options)

 [СОЗДАНИЕ РАСШИРЕНИЙ](https://dagger.dev/hilt/creating-extensions)

 ПРОЕКТНЫЕ РЕШЕНИЯ

 [→ ОБЗОР ДИЗАЙНА](https://dagger.dev/hilt/design-overview)

 [→ ФИЛОСОФИЯ ТЕСТИРОВАНИЯ](https://dagger.dev/hilt/testing-philosophy)

 [→ МОНОЛИТНЫЕ КОМПОНЕНТЫ](https://dagger.dev/hilt/monolithic)

 [→ ПОДКОМПОНЕНТЫ](https://dagger.dev/hilt/subcomponents-vs-deps)

## Модули рукояти

Модули Hilt - это стандартные модули Dagger, которые имеют дополнительную [@InstallIn](https://dagger.dev/api/latest/dagger/hilt/InstallIn.html) аннотацию, определяющую, в какие [компоненты Hilt](https://dagger.dev/hilt/components.html#hilt-components) следует установить модуль.

Когда компоненты Hilt сгенерированы, модули, помеченные значком, @InstallIn будут установлены в соответствующий компонент или подкомпонент через @Component#modulesили @Subcomponent#modulesсоответственно. Как и в Dagger, установка модуля в компонент позволяет получить доступ к этой привязке как к зависимости от других привязок в этом компоненте или любом дочернем компоненте (ах) ниже его в [иерархии компонентов](https://dagger.dev/hilt/components.html#component-hierarchy) . К ним также можно получить доступ из соответствующих @AndroidEntryPointклассов. Установка в компонент также позволяет привязать эту привязку к этому компоненту.

### С помощью @InstallIn

Модуль устанавливается в [Hilt Component](https://dagger.dev/hilt/components.html) , аннотируя модуль [@InstallIn](https://dagger.dev/api/latest/dagger/hilt/InstallIn.html) аннотацией. Эти аннотации требуются для всех модулей Dagger при использовании Hilt, но эту проверку можно дополнительно [отключить](https://dagger.dev/hilt/compiler-options.html#disable-install-in-check) .

**Примечание.** Если у модуля нет @InstallInаннотации, он не будет частью компонента и может привести к ошибкам компиляции.

Укажите, в какой компонент рукоятки следует установить модуль, передав соответствующий тип (ы) [компонента](https://dagger.dev/hilt/components.html) в @InstallInаннотацию. Например, чтобы установить модуль, чтобы все в приложении могло его использовать, используйте SingletonComponent:

Ява

Котлин

@Module

@InstallIn(**SingletonComponent::class**) *// Installs FooModule in the generate SingletonComponent.*

**object** **FooModule** {

@Provides

**fun** **provideBar**(): **Bar** {**..**.}

}

Каждый компонент поставляется с аннотацией области действия, которую можно использовать для запоминания привязки к времени жизни компонента. Например, чтобы задать область привязки к SingletonComponentкомпоненту, используйте @Singletonаннотацию:

Ява

Котлин

@Module

@InstallIn(**SingletonComponent::class**)

**object** class **FooModule** {

*// @Singleton providers are only called once per SingletonComponent instance.*

@Provides

@Singleton

**fun** **provideBar**(): **Bar** {**..**.}

}

Кроме того, каждый компонент имеет привязки, доступные ему по умолчанию. ( Полный список см. В разделе [Компоненты Hilt](https://dagger.dev/hilt/components.html#component-bindings) .) Например, SingletonComponentкомпонент обеспечивает Application привязку:

Ява

Котлин

@Module

@InstallIn(**SingletonComponent::class**)

**object** class **FooModule** {

*// @InstallIn(SingletonComponent.class) module providers have access to*

*// the Application binding.*

@Provides

**fun** **provideBar**(app: **Application**): **Bar** {**..**.}

}

### Установка модуля из нескольких компонентов

Модуль может быть установлен в нескольких компонентах. Например, может быть , у вас есть привязка в ViewComponentи ViewWithFragmentComponentи не хотят , чтобы дублировать модули. @InstallIn({ViewComponent.class, ViewWithFragmentComponent.class})установит модуль в оба компонента.

При установке модуля из нескольких компонентов следует соблюдать три правила:

* Поставщики могут быть ограничены только в том случае, если все компоненты поддерживают одну и ту же аннотацию области. Например, привязка, предоставленная в ViewComponentи ViewWithFragmentComponentможет быть @ViewScopedсвязана с тем, что они оба поддерживают эту аннотацию области. Привязка предусмотрена Fragmentи Serviceне может быть охвачена ни одной из стандартных областей.
* Провайдеры могут вводить привязки только в том случае, если все компоненты имеют доступ к этим привязкам. Например, привязка в ViewComponentи ViewWithFragmentComponentможет вводить a View, тогда как что-то связано FragmentComponentи ServiceComponentне может вводить ни Fragmentили Service.
* Дочерний и родительский компоненты не должны устанавливать один и тот же модуль. (Просто установите модуль в предке, и ребенок получит доступ к этим привязкам).

## Варианты сборки приложения

Большинство приложений Android захотят использовать разные модули и привязки в зависимости от варианта сборки приложения (например, производство, отладка, тестирование и т. Д.).

В Hilt, если цель сборки вашего двоичного файла транзитивно зависит от модуля, то этот модуль будет установлен в соответствующий компонент для вашего приложения. Это делает настройку такой же простой, как определение другой цели сборки и добавление различных глубин в ваше двоичное определение.

## Bazel: Организация ваших файлов BUILD

Поскольку Bazel имеет тенденцию поощрять разделение на более мелкие целевые объекты сборки, часто для тестов лучше просто избегать зависимости от модулей, которые вы собираетесь заменить в тестах, а не [удалять](https://dagger.dev/hilt/testing.html#uninstall-modules) их. Это связано с тем, что он уменьшает зависимости сборки вашего теста, что может привести к общему сокращению времени сборки.

При организации целевого объекта BUILD для модуля вы должны подумать, должен ли этот модуль заменяться в тестах или других конфигурациях вашего приложения. Если его никогда не следует заменять, не стесняйтесь включать модуль в другие источники кода.

Однако, если он должен быть заменяемым, вам следует создать отдельную цель для своего модуля. Затем эту цель можно вставить в корень вашего приложения, чтобы каждый тестовый корень (или другой корень конфигурации) мог решить, использовать ли ваш модуль или нет.

Есть два способа организовать цели BUILD в отношении модулей в зависимости от ситуации:

* Просто включите модули в вашу обычную цель сборки. Это будет означать, что пользователи, зависящие от вашей библиотеки, всегда получат ваше определение.
* Для тех привязок, которые вы хотите заменить в тестах, разделите свои модули на цель, называемую «модуль», которая предназначена для использования на android\_binaryуровне.

Рекомендуется выбирать первый метод по умолчанию, а второй использовать только для привязок, которые необходимо заменить в тестах. Однако ожидается, что многие библиотеки будут использовать оба метода.

# Пункты въезда

**Что такое точка входа?**

Точка входа - это граница, на которой вы можете получить объекты, предоставляемые Dagger, из кода, который не может использовать Dagger для внедрения своих зависимостей. Это точка, в которой код впервые попадает в граф объектов, управляемых Dagger.

Если вы уже знакомы с компонентами Dagger, точка входа - это просто интерфейс, который будет расширять созданный Hilt компонент.

**Когда вам нужна точка входа?**

Вам понадобится точка входа при взаимодействии с библиотеками, отличными от Dagger, или компонентами Android, которые еще не поддерживаются в Hilt и которым необходимо получить доступ к объектам Dagger.

В целом, однако, большинство точек входа будут находиться в местах, созданных экземплярами Android, таких как действия, фрагменты и т. Д. [@AndroidEntryPoint](https://dagger.dev/hilt/android-entry-point.html)Это специализированный инструмент для обработки определения точек входа и доступа к точкам входа (среди прочего) для этих классов. Поскольку это уже обрабатывается специально для этих классов Android, в следующих документах мы предполагаем, что точка входа требуется в каком-то другом типе класса.

**Как использовать точку входа?**

**Создать EntryPoint**

Чтобы создать точку входа, определите интерфейс с методом доступа для каждого необходимого типа привязки (включая его квалификатор) и пометьте интерфейс [@EntryPoint](https://dagger.dev/api/latest/dagger/hilt/EntryPoint.html) аннотацией. Затем добавьте, @InstallInчтобы указать компонент, в котором нужно установить точку входа.

Ява

Котлин

@EntryPoint

@InstallIn(**SingletonComponent::class**)

**interface** **FooBarInterface** {

@Foo **fun** **getBar**(): **Bar**

}

**Доступ к EntryPoint**

Чтобы получить доступ к точке входа, используйте EntryPointsкласс, передающий в качестве параметра экземпляр компонента или @AndroidEntryPointобъект, который действует как держатель компонента. Убедитесь, что переданный вами компонент соответствует @InstallInаннотации на [@EntryPoint](https://dagger.dev/api/latest/dagger/hilt/EntryPoint.html) интерфейсе, который вы передаете.

Используя интерфейс точки входа, который мы определили выше:

Ява

Котлин

**val** bar = **EntryPoints**.**get**(applicationContext, **FooBarInterface::class**.java).**getBar**()

Кроме того, методы в [EntryPointAccessors](https://dagger.dev/api/latest/dagger/hilt/android/EntryPointAccessors.html) более подходящие и безопасные для получения точек входа из стандартных компонентов Android.

**Лучшая практика: где определить интерфейс точки входа?**

Если реализация класса, созданного из библиотеки, отличной от Hilt, и требуется класс Foo из Dagger, следует ли определять интерфейс точки входа с помощью класса using или с помощью Foo?

В общем, ответ заключается в том, что точка входа должна быть определена с помощью класса using, поскольку именно этот класс является причиной необходимости интерфейса точки входа, а не Foo. Если этому классу позже потребуются дополнительные зависимости, для их получения в интерфейс точки входа можно легко добавить дополнительные методы. По сути, интерфейс точки входа действует вместо @Injectконструктора этого класса. Если вместо этого точка входа была определена с помощью Foo, то другие люди могут быть сбиты с толку относительно того, следует ли им вводить Fooили использовать интерфейс точки входа. Это также приведет к добавлению дополнительных интерфейсов точек входа, если в будущем потребуются другие зависимости.

**Лучшая практика**

Ява

Котлин

*// No @Inject because this isn't instantiated in a Dagger context public*

**class** **MyClass** : **NonHiltLibraryClass**() {

@EntryPoint

@InstallIn(**SingletonComponent::class**)

**interface** **MyClassInterface** {

**fun** **getFoo**(): **Foo**

**fun** **getBar**(): **Bar**

}

**fun** **doSomething**(context: **Context**) {

**val** myClassInterface =

**EntryPoints**.**get**(applicationContext, **MyClassInterface::class**.java)

**val** foo = myClassInterface.**getFoo**()

**val** bar = myClassInterface.**getBar**()

}

}

**Плохая практика**

Ява

Котлин

@Module

@InstallIn(**SingletonComponent::class**)

**object** **FooModule** {

@Provides

**fun** **provideFoo**(): **Foo** {

**return** **Foo**()

}

@EntryPoint

@InstallIn(**SingletonComponent::class**)

**interface** **FooInterface** {

**fun** **getFoo**(): **Foo**

}

}

# Пользовательские компоненты

**Нужен ли нестандартный компонент?**

В Hilt есть готовые компоненты для Android, которыми вы управляете. Однако могут возникать ситуации, когда стандартные компоненты Hilt не соответствуют срокам службы объекта или потребностям конкретной функции. В этих случаях вам может понадобиться специальный компонент. Однако перед созданием настраиваемого компонента подумайте, действительно ли он вам нужен, поскольку не каждое место, где вы можете логически добавить настраиваемый компонент, заслуживает его.

Например, рассмотрим фоновую задачу. У задачи есть достаточно четко определенное время жизни, которое может иметь смысл для области действия. Кроме того, если для этой задачи был объект запроса, привязка его к Dagger может сэкономить некоторую работу, передав его в качестве параметра. Однако для большинства фоновых задач компонент действительно не нужен, а только добавляет сложности, когда простая передача пары объектов в стек вызовов проще и достаточно. Прежде чем приступить к добавлению пользовательского компонента, обратите внимание на следующие недостатки.

Добавление пользовательского компонента имеет следующие недостатки:

* Каждый компонент / объем добавляет когнитивные накладные расходы.
* Они могут усложнить граф комбинаторикой (например, если компонент ViewComponentконцептуально является дочерним по отношению к компоненту, вероятно, необходимо добавить два компонента для ViewComponentи ViewWithFragmentComponent).
* Компоненты могут иметь только одного родителя. Иерархия компонентов не может образовывать ромб. Создание большего количества компонентов увеличивает вероятность попадания в ситуацию, когда необходима алмазная зависимость. К сожалению, у этой проблемы с алмазами нет хорошего решения, и ее бывает сложно предсказать и избежать.
* Пользовательские компоненты работают против стандартизации. Чем больше пользовательских компонентов используется, тем сложнее разделяемые библиотеки.

Имея это в виду, вот несколько критериев, которые вы должны использовать для решения, нужен ли пользовательский компонент:

* Компонент имеет связанное с ним четко определенное время жизни.
* Концепция компонента хорошо понятна и широко применима. Компоненты рукоятки являются глобальными для приложения, поэтому концепции должны применяться везде. Глобальное понимание также помогает бороться с некоторыми проблемами, связанными с когнитивными перегрузками.
* Подумайте, достаточно ли компонента без рукояти (обычный кинжал). Для компонентов с ограниченным назначением иногда лучше использовать компонент, отличный от Hilt. Например, рассмотрим производственный компонент, представляющий одну фоновую задачу. Компоненты рукоятки превосходны в ситуациях, когда код должен быть добавлен из, возможно, несвязного / модульного кода. Если ваш компонент на самом деле не предназначен для расширения, он может не подходить для пользовательского компонента Hilt.

**Ограничения пользовательских компонентов**

Определения пользовательских компонентов в настоящее время имеют некоторые ограничения:

* Компоненты должны быть прямым или косвенным дочерним элементом SingletonComponent.
* Компоненты нельзя вставлять между стандартными компонентами. Например, нельзя добавить компонент между ActivityComponentсимволами FragmentComponent.

**Добавление пользовательского компонента Hilt**

Чтобы создать собственный компонент Hilt, создайте класс с аннотациями [@DefineComponent](https://dagger.dev/api/latest/dagger/hilt/DefineComponent.html). Это будет класс, используемый в @InstallInаннотациях.

Родитель вашего компонента должен быть определен в значении @DefineComponentаннотации. Ваш @DefineComponentкласс также может быть аннотирован аннотацией области видимости, чтобы позволить объектам области видимости этого компонента.

Например:

Ява

Котлин

@DefineComponent(parent = **SingletonComponent::class**)

**interface** **MyCustomComponent**

Также должен быть определен интерфейс построителя. Если этот построитель отсутствует, компонент не будет создан, так как не будет возможности построить компонент. Этот интерфейс будет внедряться из родительского компонента и будет интерфейсом для создания новых экземпляров вашего компонента. Поскольку это пользовательские компоненты, после создания экземпляров вашей задачей будет удерживать или выпускать экземпляры компонентов в подходящее время.

Интерфейсы Builder определяются путем пометки интерфейса значком @DefineComponent.Builder. Строители должны иметь метод, возвращающий @DefineComponentтип. У них также могут быть дополнительные методы (например, @BindsInstanceметоды), которые может иметь обычный [конструктор компонентов](https://dagger.dev/api/latest/dagger/Component.Builder.html) Dagger .

Например:

Ява

Котлин

@DefineComponent.**Builder**

**interface** **MyCustomComponentBuilder** {

**fun** **fooSeedData**(@BindsInstance **Foo** foo): **MyCustomComponentBuilder**

**fun** **build**(): **MyCustomComponent**

}

Хотя @DefineComponent.Builderкласс может быть вложен в класс @DefineComponent, обычно лучше использовать его как отдельный класс. Его можно выделить в другой класс, если это транзитивная зависимость @HiltAndroidAppприложения или @HiltAndroidTestтеста. Поскольку на @DefineComponentкласс во многих местах ссылаются через @InstallIn, может быть лучше разделить построитель, чтобы зависимости в построителе не становились транзитивными зависимостями каждого модуля, установленного в компоненте.

По той же причине, чтобы избежать чрезмерных зависимостей, методы в @DefineComponentинтерфейсе не разрешены . Вместо этого к объектам Dagger следует обращаться через [точки входа](https://dagger.dev/hilt/entry-points.html) .

Ява

Котлин

@EntryPoint

@InstallIn(**MyCustomComponent::class**)

**interface** **MyCustomEntryPoint** {

**fun** **getBar**(): **Bar**

}

**class** **CustomComponentManager** @Inject **constructor**(

componentBuilder: **MyCustomComponentBuilder**) {

**fun** **doSomething**(foo: **Foo**) {

**val** component = componentBuilder.**fooSeedData**(foo).**build**();

**val** bar = **EntryPoints**.**get**(component, **MyCustomEntryPoint::class**.java).**getBar**()

*// Don't forget to hold on to the component instance if you need to!*

}

# Тестирование

## Введение

**Примечание:** В настоящее время, Эфес поддерживает только Android приборов и тесты Robolectric (хотя, см [здесь](https://dagger.dev/hilt/gradle-setup.html#running-with-android-studio) для ограничения при выполнении тестов Robolectric с помощью Android Studio). Кроме того, Hilt нельзя использовать в обычных тестах JVM, но это не мешает вам писать эти тесты, как обычно.

Hilt упрощает тестирование, привнося в ваши тесты Android возможности внедрения зависимостей. Hilt позволяет вашим тестам легко получать доступ к привязкам Dagger, предоставлять новые привязки или даже заменять привязки. Каждый тест имеет собственный набор компонентов Hilt, так что вы можете легко настраивать привязки на уровне теста.

Многие API-интерфейсы и функции тестирования, описанные в этой документации, основаны на неустановленной философии того, что делает хороший тест. Более подробную информацию о философии тестирования Hilt можно найти [здесь](https://dagger.dev/hilt/testing-philosophy.html) .

## Испытательная установка

**Примечание.** Для пользователей Gradle обязательно сначала добавьте зависимости тестовой сборки Hilt, как описано в руководстве по [настройке Gradle](https://dagger.dev/hilt/gradle-setup.html#hilt-test-dependencies) .

Чтобы использовать Hilt в тесте:

1. Аннотируйте тест с помощью [@HiltAndroidTest](https://dagger.dev/api/latest/dagger/hilt/android/testing/HiltAndroidTest.html),
2. Добавьте [HiltAndroidRule](https://dagger.dev/api/latest/dagger/hilt/android/testing/HiltAndroidRule.html)тестовое правило,
3. Используйте [HiltTestApplication](https://dagger.dev/api/latest/dagger/hilt/android/testing/HiltTestApplication.html)для своего класса приложения Android.

Например:

Ява

Котлин

@HiltAndroidTest

**class** **FooTest** {

@**get**:**Rule** **val** hiltRule = **HiltAndroidRule**(**this**)

**..**.

}

Обратите внимание, что установка класса приложения для теста (шаг 3 выше) зависит от того, является ли тест роботизированным или инструментальным. Более подробное руководство по настройке тестового приложения для конкретной тестовой среды см. В разделе [Робоэлектрическое тестирование](https://dagger.dev/hilt/robolectric-testing.html) или [Инструментальное тестирование](https://dagger.dev/hilt/instrumentation-testing.html) . Остальная часть этого документа относится как к робототехническим испытаниям, так и к испытаниям контрольно-измерительных приборов.

Если для вашего теста требуется настраиваемый класс приложения, см. Раздел о [настраиваемом тестовом приложении](https://dagger.dev/hilt/testing#custom-test-application) .

Если для вашего теста требуется несколько правил тестирования, см. Раздел о [порядке правила Hilt,](https://dagger.dev/hilt/testing#hilt-rule-order) чтобы определить правильное размещение правила Hilt.

## Доступ к привязкам

Тесту часто требуется запрашивать привязки от компонентов Hilt. В этом разделе описывается, как запрашивать привязки от каждого из различных компонентов.

### Доступ к привязкам SingletonComponent

SingletonComponentСвязывания могут быть введено непосредственно в тест с использованием @Injectаннотированного поля. Впрыск не происходит до звонка HiltAndroidRule#inject().

Ява

Котлин

@HiltAndroidTest

**class** **FooTest** {

@**get**:**Rule** **val** hiltRule = **HiltAndroidRule**(**this**)

@Inject **lateinit** **var** foo: **Foo**

@Test

**fun** **testFoo**() {

**assertThat**(foo).**isNull**()

hiltRule.**inject**()

**assertThat**(foo).**isNotNull**()

}

}

### Доступ к привязкам ActivityComponent

Для запроса ActivityComponentпривязки требуется экземпляр Hilt Activity. Один из способов сделать это - определить вложенное действие в вашем тесте, которое содержит @Injectполе для нужной вам привязки. Затем создайте экземпляр своего тестового действия, чтобы получить привязку.

Ява

Котлин

@HiltAndroidTest

**class** **FooTest** {

@AndroidEntryPoint

**class** **TestActivity** : **AppCompatActivity**() {

@Inject **lateinit** **var** foo: **Foo**

}

**..**.

**val** foo = testActivity.foo

}

В качестве альтернативы, если у вас уже есть экземпляр действия Hilt, доступный в вашем тесте, вы можете получить любую ActivityComponentпривязку, используя файл [EntryPoint](https://dagger.dev/hilt/entry-points.html).

Ява

Котлин

@HiltAndroidTest

**class** **FooTest** {

@EntryPoint

@InstallIn(**ActivityComponent::class**)

**interface** **FooEntryPoint** {

**fun** **getFoo**() : **Foo**

}

**..**.

**val** foo = **EntryPoints**.**get**(activity, **FooEntryPoint::class**.java).**getFoo**()

}

### Доступ к привязкам FragmentComponent

FragmentComponentСвязывание можно получить аналогичным образом , чтобы [ActivityComponentсвязывания](https://dagger.dev/hilt/testing#accessing-activitycomponent-bindings) . Основное отличие состоит в том, что для доступа к FragmentComponentпривязке требуются экземпляры рукояти Activityи рукояти Fragment.

Ява

Котлин

@HiltAndroidTest

**class** **FooTest** {

@AndroidEntryPoint

**class** **TestFragment** : **Fragment**() {

@Inject **lateinit** **var** foo: **Foo**

}

**..**.

**val** foo = testFragment.foo

}

В качестве альтернативы, если у вас уже есть экземпляр фрагмента Hilt, доступный в вашем тесте, вы можете получить любую FragmentComponentпривязку, используя файл [EntryPoint](https://dagger.dev/hilt/entry-points.html).

Ява

Котлин

@HiltAndroidTest

**class** **FooTest** {

@EntryPoint

@InstallIn(**FragmentComponent::class**)

**interface** **FooEntryPoint** {

**fun** **getFoo**() : **Foo**

}

**..**.

**val** foo = **EntryPoints**.**get**(fragment, **FooEntryPoint::class**.java).**getFoo**()

}

**Предупреждение** : Hilt в настоящее время не поддерживает, [FragmentScenario](https://developer.android.com/reference/androidx/fragment/app/testing/FragmentScenario) потому что нет способа указать класс активности, а Hilt требует, чтобы фрагмент Hilt содержался в действии Hilt. Один из способов решения этой проблемы - запустить действие Hilt и затем прикрепить фрагмент.

## Добавление привязок

Тест может потребовать предоставления дополнительных привязок Dagger, которые не включены в производственную сборку приложения. Кроме того, тестам может потребоваться предоставить одну и ту же привязку с разными значениями для каждого теста. В этом разделе описывается, как подготовить привязки для теста с помощью Hilt.

### Вложенные модули

Обычно @InstallInмодули устанавливаются в компоненты Hilt каждого теста. Однако, если привязку необходимо установить только в конкретном тесте, это можно сделать, вложив @InstallInмодуль в тестовый класс.

Ява

Котлин

@HiltAndroidTest

**class** **FooTest** {

*// Nested modules are only installed in the Hilt components of the outer test.*

@Module

@InstallIn(**SingletonComponent::class**)

**object** **FakeBarModule** {

@Provides **fun** **provideBar**() = **Bar**()

}

**..**.

}

Таким образом, если есть другой тест, который должен предоставить такую ​​же привязку с другой реализацией, он может сделать это без дублирования конфликта привязок.

Помимо статических вложенных @InstallInмодулей, Hilt также поддерживает внутренние (нестатические) @InstallInмодули в тестах. Использование внутреннего модуля позволяет @Providesметодам ссылаться на элементы тестового экземпляра.

**Примечание.** Hilt не поддерживает @InstallInмодули с параметрами конструктора.

### @BindValue

Для простых привязок, особенно тех, которые также должны быть доступны в тестовых методах, Hilt предоставляет удобную аннотацию, чтобы избежать шаблонного создания модуля и метода, обычно требуемых для предоставления привязки.

[@BindValue](https://dagger.dev/api/latest/dagger/hilt/android/testing/BindValue.html)- это аннотация, которая позволяет вам легко связывать поля в вашем тесте с графом Dagger. Чтобы использовать его, просто аннотируйте поле, @BindValue и оно будет привязано к объявленному типу поля с любыми квалификаторами, которые присутствуют в поле.

Ява

Котлин

@HiltAndroidTest

**class** **FooTest** {

**..**.

@BindValue

@JvmField

**val** fakeBar: **Bar** = new **FakeBar**()

}

Обратите внимание, что @BindValueне поддерживает использование аннотаций области, поскольку область привязки привязана к полю и контролируется тестом. Значение поля запрашивается всякий раз, когда оно запрашивается, поэтому оно может быть изменено по мере необходимости для вашего теста. Если вы хотите, чтобы привязка была одноэлементной, просто убедитесь, что поле устанавливается только один раз для каждого тестового примера, например, путем установки значения поля либо из инициализатора поля, либо из @Beforeметода теста.

Точно так же, рукоять также имеет удобство аннотацию для multibindings с [@BindValueIntoSet](https://dagger.dev/api/latest/dagger/hilt/android/testing/BindValueIntoSet.html), [@BindElementsIntoSet](https://dagger.dev/api/latest/dagger/hilt/android/testing/BindElementsIntoSet.html)и [@BindValueIntoMap](https://dagger.dev/api/latest/dagger/hilt/android/testing/BindValueIntoMap.html)для поддержки [@IntoSet](https://dagger.dev/api/latest/dagger/multibindings/IntoSet.html), [@ElementsIntoSet](https://dagger.dev/api/latest/dagger/multibindings/ElementsIntoSet.html)и [@IntoMap](https://dagger.dev/api/latest/dagger/multibindings/IntoMap.html)соответственно. (Обратите внимание, что это @BindValueIntoMapтребует, чтобы поле также было аннотировано ключевой аннотацией карты.)

### Предостережения

Будьте осторожны при использовании [@BindValue](https://dagger.dev/hilt/testing#bind-value)или [нестатические внутренние модули](https://dagger.dev/hilt/testing#nested-modules) с ActivityScenarioRule. ActivityScenarioRuleсоздает действие перед вызовом @Beforeметода, поэтому, если @BindValueполе инициализировано в @Before(или позже), действие может внедрить привязку в ее унифицированном состоянии. Чтобы этого избежать, попробуйте инициализировать @BindValueполе в инициализаторе поля.

## Замена крепления

### @UninstallModules

Распространенный вариант использования в тестах - это необходимость заменить производственную привязку тестовой привязкой, например, подделкой или макетом. В тесте Hilt производственную привязку можно заменить, сначала удалив производственный модуль, в котором он содержится [@UninstallModules](https://dagger.dev/api/latest/dagger/hilt/android/testing/UninstallModules.html), а затем подготовив новую привязку из теста.

Ява

Котлин

@UninstallModules(**ProdFooModule::class**)

@HiltAndroidTest

**class** **FooTest** {

**..**.

@BindValue

@JvmField

**val** fakeFoo: **Foo** = new **FakeFoo**()

}

Обратите внимание, что @UninstallModulesэто эквивалентно удалению @InstallIn аннотации из этого модуля в отношении данного теста. Hilt не поддерживает удаление отдельных привязок напрямую, но фактически это возможно, если включить только одну привязку в данный модуль.

## Пользовательское тестовое приложение

Каждый тест Hilt должен использовать тестовое приложение Hilt в качестве класса приложения Android. Hilt поставляется с тестовым приложением по умолчанию, [HiltTestApplication](https://dagger.dev/api/latest/dagger/hilt/android/testing/HiltTestApplication.html)которое расширяется [MultiDexApplication](https://developer.android.com/studio/build/multidex); однако бывают случаи, когда в тесте может потребоваться использовать другой базовый класс.

### @CustomTestApplication

Если вашему тесту требуется настраиваемый базовый класс, [@CustomTestApplication](https://dagger.dev/api/latest/dagger/hilt/android/testing/CustomTestApplication.html)его можно использовать для создания тестового приложения Hilt, которое расширяет данный базовый класс.

Для использования @CustomTestApplicationпросто аннотируйте класс или интерфейс @CustomTestApplicationи укажите базовый класс в значении аннотации:

Ява

Котлин

*// Generates MyCustom\_Application.class*

@CustomTestApplication(**MyBaseApplication::class**)

**interface** **MyCustom**

В приведенном выше примере Hilt сгенерирует приложение с именем MyCustom\_Applicationextends MyBaseApplication. Как правило, имя созданного приложения будет именем аннотированного класса, к которому добавлено \_Application. Если аннотированный класс является вложенным, имя также будет включать имя внешнего класса, разделенное подчеркиванием. Обратите внимание, что аннотированный класс не имеет значения, кроме имени созданного приложения.

### Лучшие практики

Рекомендуется избегать использования, @CustomTestApplicationа вместо этого использовать HiltTestApplicationв своих тестах. В целом, независимость ваших Activity, Fragment и т. Д. От родительского элемента, в котором они содержатся, упрощает их составление и повторное использование в будущем.

Однако, если вам необходимо использовать настраиваемое базовое приложение, следует помнить о некоторых тонких различиях с производственным жизненным циклом.

Одно отличие состоит в том, что инструментальные тесты используют один и тот же экземпляр приложения для каждого теста и тестового примера. Таким образом, при использовании настраиваемого тестового приложения легко случайно просочиться через тестовые случаи. Вместо этого лучше избегать хранения в вашем приложении любого тестового или зависимого состояния тестового случая.

Еще одно отличие состоит в том, что компонент Hilt в тестовом приложении не создается в super#onCreate. Это ограничение в основном связано с тем, что некоторые функции Hilt (например [@BindValue](https://dagger.dev/hilt/testing#bind-value)) полагаются на тестовый экземпляр, который недоступен в тестах до тех пор, пока не Application#onCreateбудет вызван. Таким образом, в отличие от производственных приложений, пользовательские базовые приложения не должны вызывать компонент во время Application#onCreate. Это включает в себя внедрение участников в приложение. Чтобы предотвратить эту проблему, Hilt не допускает внедрения в базовое приложение.

## Порядок правил рукояти

Если в вашем тесте используется несколько правил тестирования, убедитесь, что они HiltAndroidRuleвыполняются перед любыми другими правилами тестирования, которым требуется доступ к компоненту Hilt. Например, [ActivityScenarioRule](https://developer.android.com/reference/androidx/test/ext/junit/rules/ActivityScenarioRule) вызовы Activity#onCreate, которые (для действий Hilt) требуют, чтобы компонент Hilt выполнял инъекцию. Таким образом, ActivityScenarioRuleдолжен запускаться после, HiltAndroidRuleчтобы убедиться, что компонент был правильно инициализирован.

**Примечание.** Если вы используете JUnit <4.13 [RuleChain](https://junit.org/junit4/javadoc/4.12/org/junit/rules/RuleChain.html) , вместо этого укажите порядок.

Ява

Котлин

@HiltAndroidTest

**class** **FooTest** {

*// Ensures that the Hilt component is initialized before running the ActivityScenarioRule*

@**get**:**Rule**(order = 0)

**val** hiltRule = **HiltAndroidRule**(**this**)

@**get**:**Rule**(order = 1)

**val** scenarioRule = **ActivityScenarioRule**(**MyActivity::class**.java)

}

# Робоэлектрические испытания

**Предупреждение:** см. [Здесь](https://dagger.dev/hilt/gradle-setup.html#running-with-android-studio) ограничения при запуске робоэлектрических тестов через Android Studio при использовании [плагина Hilt Gradle](https://dagger.dev/hilt/gradle-setup.html#hilt-gradle-plugin) .

**Настройка тестового приложения**

[API-интерфейсы](https://dagger.dev/hilt/testing.html) Hilt для [тестирования не](https://dagger.dev/hilt/testing.html) зависят от конкретной среды тестирования; однако инструкции по настройке класса приложения в вашем тесте будут зависеть от того, используете ли вы инструментальные тесты Robolectric или Android.

Для Робоэлектрических тестов приложение может быть настроено как локально, так [@Config](http://robolectric.org/javadoc/latest/org/robolectric/annotation/Config.html)и глобально robolectric.properties. Для тестов Hilt приложение должно быть [HiltTestApplication](https://dagger.dev/api/latest/dagger/hilt/android/testing/HiltTestApplication.html)либо одним из [пользовательских тестовых приложений](https://dagger.dev/hilt/testing.html#custom-test-application) Hilt .

**Примечание.** Эта установка не является специфической для Hilt. См. [Официальную документацию Robolectric](http://robolectric.org/configuring/) для более подробной информации.

**Использование @Config**

Класс приложения Hilt можно установить локально с помощью @Config. Чтобы настроить приложение, просто аннотируйте тест (или метод теста) с помощью @Configи установите значение аннотации для желаемого класса приложения.

Ява

Котлин

@HiltAndroidTest

@Config(application = **HiltTestApplication::class**)

**class** **FooTest** {**..**.}

**Использование robolectric.properties**

Класс приложения Hilt можно установить глобально с помощью robolectric.propertiesфайла. Чтобы настроить приложение, просто создайте robolectric.propertiesфайл в соответствующем resourcesпакете и задайте класс тестового приложения Hilt.

application**=**dagger**.**hilt**.**android**.**testing**.**HiltTestApplication

Этот подход может быть полезен, когда тест необходимо запустить как в инструментальной среде Robolectric, так и в инструментальной среде Android, поскольку @Configаннотация не может использоваться с инструментальными тестами Android.

Инструментальное тестирование

**Настройка тестового приложения**

[API-интерфейсы](https://dagger.dev/hilt/testing.html) Hilt для [тестирования не](https://dagger.dev/hilt/testing.html) зависят от конкретной среды тестирования; однако инструкции по настройке класса приложения в вашем тесте будут зависеть от того, используете ли вы инструментальные тесты Robolectric или Android.

Для инструментальных тестов Android приложение можно настроить с помощью настраиваемого средства запуска тестов, которое расширяется [AndroidJUnitRunner](https://developer.android.com/reference/androidx/test/runner/AndroidJUnitRunner). Чтобы настроить приложение с помощью бегуна, просто переопределите newApplicationметод и передайте имя класса приложения. Для тестов Hilt приложение должно быть [HiltTestApplication](https://dagger.dev/api/latest/dagger/hilt/android/testing/HiltTestApplication.html)либо одним из [пользовательских тестовых приложений](https://dagger.dev/hilt/testing.html#custom-test-application) Hilt .

Ява

Котлин

**package** my.pkg

**class** **MyTestRunner**: **AndroidJUnitRunner**() {

**override** **fun** **newApplication**(

cl: **ClassLoader**,

appName: **String**,

context: **Context**) : **Application** {

**return** **super**.**newApplication**(

cl, **HiltTestApplication::class**.java.**getName**(), context)

}

}

Кроме того, для данного модуля Gradle testInstrumentationRunnerнеобходимо настроить в build.gradleфайле:

android **{**

defaultConfig **{**

testInstrumentationRunner "my.pkg.MyTestRunner"

**}**

**}**

Переход на Hilt

Переход на Hilt может сильно различаться по сложности в зависимости от состояния вашей кодовой базы и того, каким практикам или шаблонам следует ваша кодовая база. На этой странице представлены советы по некоторым типичным проблемам, с которыми может возникнуть миграция приложений. На этой странице предполагается, что вы уже в целом разбираетесь в основных API Hilt. Если это не так, сначала ознакомьтесь с нашим [кратким](https://dagger.dev/hilt/quick-start.html) руководством по Hilt. Эта страница также предполагает общее понимание Dagger, что должно быть так, поскольку эта страница полезна только для тех, кто переносит кодовую базу, которая уже использует Dagger. Если ваша кодовая база не использует Dagger, добавьте Hilt в свое приложение, просмотрев руководство по [быстрому запуску, так](https://dagger.dev/hilt/quick-start.html) как это руководство касается только миграции из настроек без Hilt Dagger.

Совет по рефакторингу : всякий раз, когда вы изменяете код класса, убедитесь, что неиспользуемые или уже не существующие импорты удалены из файла.

# Содержание

* [0. Спланируйте миграцию](https://dagger.dev/hilt/migration-guide#0-plan-your-migration)
  + [Сравнить иерархии компонентов](https://dagger.dev/hilt/migration-guide#compare-component-hierarchies)
  + [Знайте, когда Hilt вводит классы](https://dagger.dev/hilt/migration-guide#be-aware-of-when-hilt-injects-classes)
  + [Обзор миграции](https://dagger.dev/hilt/migration-guide#migration-overview)
* [1. Перенести Application](https://dagger.dev/hilt/migration-guide#1-migrate-the-application)
  + [Перенос Component](https://dagger.dev/hilt/migration-guide#migrating-a-component)
    - [а. Обработка модулей](https://dagger.dev/hilt/migration-guide#a-handle-the-modules)
    - [б. Обработка любых расширенных интерфейсов или методов](https://dagger.dev/hilt/migration-guide#b-handle-any-extended-interfaces-or-methods)
      * [Перемещение всего с @EntryPoint](https://dagger.dev/hilt/migration-guide#moving-everything-with-entrypoint)
      * [Инъекционные методы](https://dagger.dev/hilt/migration-guide#inject-methods)
      * [Доступ к интерфейсам](https://dagger.dev/hilt/migration-guide#accessing-the-interfaces)
    - [c. Области применения](https://dagger.dev/hilt/migration-guide#c-scopes)
    - [d. Обработка аргументов компонента](https://dagger.dev/hilt/migration-guide#d-handling-component-arguments)
    - [е. Очистка агрегаторов](https://dagger.dev/hilt/migration-guide#e-cleaning-up-aggregators)
  + [Добавление рукояти к Application](https://dagger.dev/hilt/migration-guide#adding-hilt-to-the-application)
  + [dagger.android приложение](https://dagger.dev/hilt/migration-guide#daggerandroid-application)
  + [Проверьте свою сборку](https://dagger.dev/hilt/migration-guide#check-your-build)
* [2. Перенос действий и фрагментов (и других классов)](https://dagger.dev/hilt/migration-guide#2-migrate-activities-and-fragments-and-other-classes)
  + [Помните о различиях с монолитными компонентами](https://dagger.dev/hilt/migration-guide#be-aware-of-differences-with-monolithic-components)
    - [Конфликтующие привязки](https://dagger.dev/hilt/migration-guide#conflicting-bindings)
    - [В зависимости от конкретного вида деятельности](https://dagger.dev/hilt/migration-guide#depending-on-the-specific-activity-type)
  + [Сохраненные фрагменты](https://dagger.dev/hilt/migration-guide#retained-fragments)
  + [Добавление рукояти к действию / фрагменту](https://dagger.dev/hilt/migration-guide#adding-hilt-to-the-activityfragment)
  + [Кинжал](https://dagger.dev/hilt/migration-guide#dagger)
  + [кинжал. андроид](https://dagger.dev/hilt/migration-guide#daggerandroid)
    - [Простой пример dagger.android](https://dagger.dev/hilt/migration-guide#a-simple-daggerandroid-example)
  + [Проверьте свою сборку](https://dagger.dev/hilt/migration-guide#check-your-build-1)
* [3. Другие компоненты Android.](https://dagger.dev/hilt/migration-guide#3-other-android-components)
* [Что делать с…?](https://dagger.dev/hilt/migration-guide#what-to-do-with--)
  + [Отборочные](https://dagger.dev/hilt/migration-guide#qualifiers)
  + [Компонент аргументы](https://dagger.dev/hilt/migration-guide#component-arguments)
  + [Пользовательские компоненты](https://dagger.dev/hilt/migration-guide#custom-components)
    - [Зависимости компонентов](https://dagger.dev/hilt/migration-guide#component-dependencies)
    - [Подкомпоненты](https://dagger.dev/hilt/migration-guide#subcomponents)
  + [Зависимости компонентов для компонентов, которые отображаются на компоненты Hilt](https://dagger.dev/hilt/migration-guide#component-dependencies-for-components-that-map-to-hilt-components)

## 0. Спланируйте миграцию

При переходе на Hilt вы захотите разбить свою работу на шаги. В этом руководстве должен быть изложен общий подход, который должен работать в большинстве случаев, но каждая миграция будет отличаться. Рекомендуемый подход - начать с компонента Applicationor @Singletonи постепенно расти оттуда. После Applicationи @Singleton, перенесите действия, а затем фрагменты. Обычно это можно сделать в виде инкрементной миграции. Даже если у вас относительно небольшая кодовая база, постепенная миграция даст вам возможность встраиваться между шагами, чтобы проверить ваш прогресс.

### Сравнить иерархии компонентов

Первое, что нужно сделать, это сравнить вашу текущую иерархию компонентов с иерархией в [Hilt](https://dagger.dev/hilt/components.html) . Вы захотите решить, какие компоненты соответствуют какому компоненту Hilt. Надеюсь, они должны быть относительно простыми, но если нет четкого сопоставления, вы можете сохранить пользовательские компоненты как ручные компоненты Dagger. Эти компоненты могут быть дочерними по отношению к компонентам Hilt. Однако Hilt не позволяет вставлять компоненты в иерархию (например, изменять родительский элемент компонента Hilt). Смотрите пользовательские компоненты [раздел](https://dagger.dev/hilt/migration-guide#custom-components) в руководстве ниже. **Остальная часть этого руководства предполагает миграцию, при которой все компоненты отображаются непосредственно на компоненты Hilt.**

Кроме того, если ваш код использует зависимости компонентов, вам также следует сначала прочитать [раздел о](https://dagger.dev/hilt/migration-guide#component-dependencies-for-components-that-map-to-hilt-components) зависимостях компонентов ниже. **Остальная часть этого руководства предполагает использование подкомпонентов.**

Если вы используете dagger.android @ContributesAndroidInjectorи не уверены в своей иерархии компонентов, то ваша иерархия должна примерно соответствовать компонентам Hilt.

### Знайте, когда Hilt вводит классы

Вы можете узнать, когда Hilt внедряет классы для каждого класса Android, [здесь](https://dagger.dev/hilt/components.html#component-lifetimes) . Мы надеемся, что они должны быть похожи на то, где ваш код в настоящее время внедряется, но если нет, имейте в виду, что это вызовет какие-либо различия в вашем коде.

### Обзор миграции

По окончании миграции код следует изменить следующим образом:

* Все @Component/ @Subcomponent(или при использовании dagger.android @ContributesAndroidInjector) использования должны быть удалены.
* Все @Moduleклассы должны быть помечены @InstallIn.
* Все классы Application/ Activity/ Fragment/ View/ Service/ BroadcastReceiverдолжны быть аннотированы @AndroidEntryPoint.
* Любой код, создающий экземпляры или распространяющий компоненты (например, интерфейсы в вашей деятельности для раскрытия компонента), должен быть удален.
* Все dagger.androidссылки следует удалить.

## 1. Перенести Application

Первое , что изменение будет перенести Applicationи @Singleton компонент сгенерированной Эфесе SingletonComponent. Для этого мы сначала хотим убедиться, что все, что установлено в вашем текущем компоненте, установлено в Hilt SingletonComponent.

### Перенос Component

Чтобы перенести приложение, нам необходимо перенести все, что есть в уже существующем @Singletonкомпоненте, в SingletonComponent.

#### а. Обработка модулей

Во-первых, мы должны установить все модули в SingletonComponent. Это можно сделать, пометив каждый модуль, установленный в данный момент в вашем компоненте, с помощью @InstallIn(SingletonComponent.class). Если модулей много, вместо того, чтобы менять их все сейчас, вы можете создать и установить один @Moduleкласс агрегатора, который включает в себя все текущие модули. Однако это всего лишь временное решение, поскольку для того, чтобы в полной мере воспользоваться функциями Hilt, такими как [удаление модулей](https://dagger.dev/hilt/testing.html#uninstall-modules) , вам в будущем потребуется разбить модуль агрегатора.

Ява

Котлин

*// Starting with this component*

@Component(modules = [

**FooModule::class**,

**BarModule::class**,

**..**.

])

**interface** **MySingletonComponent** {

}

*// Becomes the following classes*

@InstallIn(**SingletonComponent::class**)

@Module(includes = [

**FooModule::class**,

**BarModule::class**,

**..**.

])

**interface** **AggregatorModule** {}

**Предупреждение:** модули, не помеченные значком @InstallIn, не используются Hilt. Hilt по умолчанию выдает ошибку при обнаружении неаннотированных модулей, но эту ошибку можно [отключить](https://dagger.dev/hilt/compiler-options.html#disable-install-in-check) .

#### б. Обработка любых расширенных интерфейсов или методов

Аналогичный процесс можно использовать для любых интерфейсов, которые расширяет ваш текущий компонент [@EntryPoint](https://dagger.dev/hilt/entry-points.html).

Интерфейсы компонентов обычно используются либо для добавления методов внедрения, либо для получения доступа к таким типам, как привязки или подкомпоненты. В Hilt многие из них не понадобятся после завершения миграции, потому что Hilt сгенерирует их для вас или они будут заменены инструментами Hilt. Что касается миграции, в этом разделе описывается, как сохранить текущее поведение, чтобы код продолжал работать. Однако вам следует изучить все эти методы и оценить, нужны ли они по-прежнему, когда миграция продолжается.

##### Перемещение всего с [@EntryPoint](https://dagger.dev/hilt/entry-points.html)

Аннотируйте любой интерфейс, который расширяет ваш компонент, [@EntryPoint](https://dagger.dev/hilt/entry-points.html)и @InstallIn(SingletonComponent.class). Если интерфейсов много, создайте единый интерфейс агрегатора, чтобы собрать их все, как модули. Любой метод, определенный непосредственно в интерфейсе компонента, может быть перемещен либо в интерфейс агрегатора, либо в один из расширений агрегатора.

Пример:

Ява

Котлин

*// Starting with this component*

@Component

@Singleton

**interface** **MySingletonComponent** : **FooInjector**, **BarInjector** {

**fun** **inject**(myApplication: **MyApplication**)

**fun** **getFoo**() : **Foo**

}

*// Becomes the following class*

@InstallIn(**SingletonComponent::class**)

@EntryPoint

**interface** **AggregatorEntryPoint** : **FooInjector**, **BarInjector** {

*// This is moved as an example, but further below we will see that inject*

*// methods for the Application can just be removed.*

**fun** **inject**(myApplication: **MyApplication**)

**fun** **getFoo**() : **Foo**

}

##### Инъекционные методы

Hilt обрабатывает инъекцию вашего Applicationкласса под капотом, поэтому, если у вас есть какие-либо методы инъекции для Application, их можно удалить. Методы внедрения для других типов Android также должны быть в конечном итоге удалены, поскольку позже они будут перенесены для использования @AndroidEntryPoint.

Ява

Котлин

@Component

@Singleton

**interface** **MySingletonComponent** {

*// Hilt takes care of Application injection for you, so this can be deleted.*

**fun** **inject**(myApplication: **MyApplication**)

*// This can be deleted once FooActivity is migrated to use @AndroidEntryPoint*

**fun** **inject**(fooActivity: **FooActivity**)

}

##### Доступ к интерфейсам

В вашем коде, вероятно, есть метод, в котором вы вернули компонент либо напрямую, либо как один из типов интерфейса, чтобы другой код мог получить доступ к методам внедрения или методам доступа. Чтобы этот код продолжал работать во время миграции, вы можете получить ссылку с помощью [EntryPoints](https://dagger.dev/hilt/entry-points.html#access-an-entrypoint)класса. По мере продолжения миграции вы сможете удалить эти методы и сделать так, чтобы вызывающий код [EntryPoints](https://dagger.dev/hilt/entry-points.html)напрямую использовал Hilt API.

Ява

Котлин

*// If you started with code like this:*

**class** **MyApplication** : **Application**() {

**fun** **component**(): **MySingletonComponent** {

**return** component

}

}

*// After adding the aggregator entry point, it will look like the following:*

@InstallIn(**SingletonComponent::class**)

@EntryPoint

**interface** **AggregatorEntryPoint** : **LegacyInterface**, **..**. {

}

@HiltAndroidApp

**class** **MyApplication** : **Application**() {

*// The return type changed the the AggregatorEntryPoint, but that should be*

*// okay as this implements all the interfaces the old component used to.*

**fun** **component**(): **AggregatorEntryPoint** {

*// Use EntryPoints to get an instance of the AggregatorEntryPoint.*

**return** **EntryPoints**.**get**(**this**, **AggregatorEntryPoint::class**.java)

}

}

#### c. Области применения

При переносе компонента в Hilt вам также потребуется перенести привязки для использования аннотаций области действия Hilt. В случае с SingletonComponentэто @Singleton. Вы можете узнать , какие аннотации соответствуют какому компоненту [здесь](https://dagger.dev/hilt/components.html#component-lifetimes) . Если вы не используете @Singletonи у вас есть собственная аннотация области видимости, вы можете сообщить Hilt, что ваша аннотация эквивалентна аннотации области видимости Hilt, используя [псевдонимы](https://dagger.dev/hilt/scope-aliases.html) области [действия](https://dagger.dev/hilt/scope-aliases.html) . Это позволит вам перенести и удалить аннотацию области действия на досуге позже в процессе.

#### d. Обработка аргументов компонента

Компоненты Hilt не могут принимать аргументы компонентов, поскольку инициализация компонента скрыта от пользователей. Обычно это используется для добавления экземпляра приложения (или для других компонентов экземпляра активности / фрагмента) в граф Dagger. В этих случаях вам следует переключиться на использование предопределенных привязок в Hilt, перечисленных [здесь](https://dagger.dev/hilt/components.html#component-bindings) .

Если у вашего компонента есть какие-либо другие аргументы либо через экземпляры модуля, переданные построителю, либо @BindsInstanceпрочтите этот [раздел](https://dagger.dev/hilt/migration-guide#component-arguments) об их обработке. Как только вы справитесь с ними, вы можете просто удалить свой @Component.Builderинтерфейс, так как он не будет использоваться.

#### е. Очистка агрегаторов

Если вы использовали модуль агрегатора или точку входа, вам в конечном итоге придется вернуться и удалить модуль агрегатора и класс точки входа. Вы можете сделать это, индивидуально аннотируя все включенные модули и реализованные интерфейсы с той же @InstallInаннотацией, которая используется в агрегаторе.

Ява

Котлин

@InstallIn(**SingletonComponent::class**)

@Module(includes = [**FooModule::class**, **..**.])

**interface** **AggregatorModule** {

}

*// Remove FooModule from the list above and annotate it directly*

@InstallIn(**SingletonComponent::class**)

@Module

**interface** **FooModule** {

}

### Добавление рукояти к Application

Теперь вы можете просто аннотировать свой Applicationс помощью, @HiltAndroidAppкак описано в нашем [кратком](https://dagger.dev/hilt/quick-start.html) руководстве. Кроме того, в нем не должно быть кода, связанного со сборкой или хранением экземпляра вашего компонента. Вы можете удалить свой @Componentкласс и @Component.Builderкласс, если вы еще этого не сделали.

### dagger.android приложение

Если ваш код Applicationрасширяется DaggerApplicationили реализуется HasAndroidInjector, сохраните этот код до тех пор, пока все ваши действия / фрагменты dagger.android также не будут перенесены. Скорее всего, это будет один из последних шагов вашей миграции. Эти части dagger.android нужны для того, чтобы убедиться, что получение зависимостей работает (например, когда приложение Activityпытается внедрить себя). Разница в том, что теперь они удовлетворяются рукояткой, SingletonComponentа не компонентом, удаленным на вышеуказанных шагах.

Например, перенесенный dagger.android, Applicationкоторый поддерживает как действия Hilt, так и действия dagger.android, может выглядеть следующим образом:

Ява

Котлин

@HiltAndroidApp

**class** **MyApplication** : **HasAndroidInjector** {

@Inject **lateinit** **var** dispatchingAndroidInjector: **DispatchingAndroidInjector**<**Object**>

**override** **fun** **androidInjector**() = dispatchingAndroidInjector

}

Или, если вы использовали DaggerApplicationраньше, вы можете сделать следующее. [@EntryPoint](https://dagger.dev/hilt/entry-points.html)Класс , чтобы сделать компонент Dagger реализации AndroidInjector<MyApplication>. Вероятно, это то, что делал ваш предыдущий компонент Dagger раньше.

Ява

Котлин

@HiltAndroidApp

**class** **MyApplication** : **DaggerApplication**() {

@EntryPoint

@InstallIn(**SingletonComponent::class**)

**interface** **ApplicationInjector** : **AndroidInjector**<**MyApplication**>

**override** **fun** **applicationInjector**(): **AndroidInjector**<**MyApplication**> {

**return** **EntryPoints**.**get**(**this**, **ApplicationInjector::class**.java)

}

}

Когда вы перенесли все остальные использования dagger.android и готовы удалить этот код, просто расширьте Applicationи удалите переопределенные методы и DispatchingAndroidInjectorклассы.

### Проверьте свою сборку

На этом этапе вы должны иметь возможность остановить и успешно построить / запустить приложение. Ваше приложение успешно использует Hilt для платформы SingletonComponent.

## 2. Перенос действий и фрагментов (и других классов)

Теперь, когда приложение поддерживает Hilt, вы можете начать перенос своих действий, а затем фрагментов в Hilt. При переносе приложения можно сочетать @AndroidEntryPointдействия и бездействие @AndroidEntryPoint. То же самое верно и для фрагментов внутри действия. Единственное ограничение при смешивании Hilt и не-Hilt кода - это родительский код. К приложениям Hilt необходимо прикрепить действия с рукоятью. Фрагменты рукояти должны быть прикреплены к действиям рукояти. Мы рекомендуем выполнить все действия перед выполнением любого из фрагментов, но если это проблематично, есть инструмент, который поможет ослабить это ограничение с помощью [необязательной инъекции](https://dagger.dev/hilt/optional-inject.html) .

Миграция действий и фрагментов будет очень похожа на компонент приложения с точки зрения механики. Вы должны взять все модули из текущего компонента и установить их в соответствующий компонент с @InstallInмодулем. Точно так же возьмите все расширенные интерфейсы текущего компонента и установите их в соответствующий компонент с @InstallIn [точкой входа](https://dagger.dev/hilt/entry-points.html) . Вернитесь к [этому разделу](https://dagger.dev/hilt/migration-guide#migrating-a-component) выше для получения подробностей, но также прочтите ниже о некоторых дополнительных соображениях, которые необходимо учитывать для действий и фрагментов.

**Примечание.** Если вы используете dagger.android's @ContributesAndroidInjector, то, следуя [этому разделу](https://dagger.dev/hilt/migration-guide#migrating-a-component) по миграции компонента, модули @ContributesAndroidInjectorявляются модулями, которые вам необходимо перенести. У вас нет интерфейсов для миграции [@EntryPoint](https://dagger.dev/hilt/entry-points.html).

### Помните о различиях с монолитными компонентами

Одно из дизайнерских решений Hilt - использовать один компонент для всех действий и один компонент для всех фрагментов. Если вам интересно, вы можете прочитать о причинах [здесь](https://dagger.dev/hilt/monolithic.html) . Причина, по которой это важно, заключается в том, что если у вас был отдельный компонент для каждого действия (как по умолчанию в dagger.android), вы будете объединять компоненты в один компонент при переходе на Hilt. В зависимости от вашей кодовой базы вы можете столкнуться с проблемами.

Две наиболее частые проблемы:

#### Конфликтующие привязки

Это происходит, если вы по-разному определили один и тот же ключ привязки в двух действиях. Когда они объединяются, вы получаете дублирующую привязку. Это ограничение глобального пространства ключей привязки Hilt, и вам необходимо переопределить эту привязку, чтобы иметь одно определение. Обычно это не так уж плохо и достигается за счет логики внедренной активности. Примеры см. В разделе, посвященном [аргументам компонентов](https://dagger.dev/hilt/migration-guide#component-arguments) .

#### В зависимости от конкретного вида деятельности

Из-за объединенного компонента привязки для a FooActivityили BarActivity часто больше не имеют смысла, поскольку, когда компонент используется для BarActivity(или любого другого действия), FooActivityпривязка не может быть удовлетворена. Обычно код на самом деле не полагается на фактический дочерний тип действия и просто нуждается в действии или общем подтипе, например FragmentActivity. Код, использующий дочерний тип, необходимо реорганизовать, чтобы использовать более общий тип. Если вам нужен общий подтип, который не предоставляется Hilt автоматически, вы можете предоставить привязку с приведением (пример [здесь](https://dagger.dev/hilt/migration-guide#component-arguments) ), но будьте осторожны!

Пример замены использования общим подтипом:

Ява

Котлин

*// This class only uses the activity to get the FragmentManager. It can instead*

*// use the non-specific FragmentActivity class.*

**class** **Foo** @Inject **constructor**(**private** **val** activity: **FooActivity**) {

**fun** **doSomething**() {

activity.**getSupportFragmentManager**()**..**.

}

}

*// Changed to FragmentActivity when migrating to Hilt class Foo @Inject*

**class** **Foo** @Inject **constructor**(**private** **val** activity: **FragmentActivity**) {

**fun** **doSomething**() {

activity.**getSupportFragmentManager**()**..**.

}

}

### Сохраненные фрагменты

Эфес не поддерживает оставшиеся фрагменты. Вы можете найти больше информации о том, почему [здесь](https://dagger.dev/hilt/android-entry-point.html#retained-fragments) . Если у вас есть какие-либо сохраненные фрагменты, распространенный способ решить эту проблему - переместить любое сохраненное состояние в файл [ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel).

### Добавление рукояти к действию / фрагменту

Теперь вы можете просто аннотировать свой Activityили Fragmentс помощью, @AndroidEntryPointкак описано в нашем [кратком](https://dagger.dev/hilt/quick-start.html) руководстве. Базовые классы, даже если они выполняют внедрение поля, не нуждаются в аннотации (если только нет ситуации, когда они создаются непосредственно как самый дочерний класс).

Ява

Котлин

@AndroidEntryPoint

**class** **FooActivity** : **AppCompatActivity**() {

@Inject **lateinit** **var** foo: **Foo**

}

**Примечание.** Даже если вашему действию не требуется внедрение поля, если к нему прикреплены фрагменты, которые используются @AndroidEntryPoint, вы также должны перенести действие для использования @AndroidEntryPoint.

### Кинжал

Теперь вы можете удалить любой код инициализации компонента или интерфейсы внедрения, если они у вас есть.

### кинжал. андроид

Если вы используете @ContributesAndroidInjectorэтот класс, вы можете удалить его сейчас. Вы также можете удалить любые звонки на AndroidInjection/, AndroidSupportInjectionесли они у вас есть. Если ваш класс реализует HasAndroidInjectorи не является родителем каких-либо фрагментов или представлений, отличных от Hilt, вы можете удалить этот код сейчас.

Если ваша Activity или Fragment расширяются от DaggerAppCompatActivity, DaggerFragmentили подобных классов, их необходимо удалить и заменить на эквиваленты, отличные от Dagger (например, AppCompatActivityили обычные Fragment). Если у вас есть дочерние фрагменты или представления, которые все еще используют dagger.android, вам нужно будет реализовать их HasAndroidInjectorпутем внедрения DispatchingAndroidInjector(см. Пример ниже).

Когда вы перенесли всех детей с dagger.android, вернитесь позже, чтобы удалить HasAndroidInjectorкод.

#### Простой пример dagger.android

В следующем примере показано, как переносить действие, но при этом поддерживать фрагменты Hilt и dagger.android.

Начальное состояние:

Ява

Котлин

**class** **MyActivity** : **DaggerAppCompatActivity**() {

@Inject **lateinit** **var** foo: **Foo**

}

@Module

**interface** **MyActivityModule** {

*// If you have a scope annotation, see the section on scope aliases*

@ContributesAndroidInjector(modules = [ **FooModule::class**, **..**. ])

**fun** **bindMyActivity**(): **MyActivity**

}

Промежуточное состояние, позволяющее использовать фрагменты как рукояти, так и кинжала. Андроида:

Ява

Котлин

@AndroidEntryPoint

**class** **MyActivity** : **AppCompatActivity**(), **HasAndroidInjector** {

@Inject **lateinit** **var** foo: **Foo**

*// Remove the code below later when all the children have been migrated*

@Inject **lateinit** **var** androidInjector: **DispatchAndroidInjector**<**Object**>

**override** **fun** **androidInjector**() = androidInjector

}

*// If the list of modules is very short, you don’t need this aggregator*

*// module, just put the @InstallIn(ActivityComponent.class) annotation on*

*// all the modules in includes list like FooModule*

@Module(includes = [ **FooModule::class**, **..**.])

@InstallIn(**ActivityComponent::class**)

**interface** **MyActivityAggregatorModule**

Конечное состояние:

Ява

Котлин

@AndroidEntryPoint

**class** **MyActivity** : **AppCompatActivity**() {

@Inject **lateinit** **var** foo: **Foo**

}

*// Each activity module is annotated with @InstallIn(ActivityComponent::class)*

### Проверьте свою сборку

Вы должны иметь возможность останавливать и успешно строить / запускать свое приложение после миграции активности или фрагмента. После переноса каждого класса рекомендуется проверить, что вы на правильном пути.

## 3. Другие компоненты Android.

View, Serviceи BroadcastReceiverтипы должны следовать той же формуле, что и выше, и быть готовыми к миграции сейчас. Как только вы все переместили, все готово!

Запомни:

* Вернитесь и очистите все оставшиеся HasAndroidInjectorобычаи.
* Очистите все оставшиеся модули агрегатора или интерфейсы точек входа. В общем, вам не нужно использовать @Module(includes=)с Hilt, поэтому, если вы это видите, вам нужно удалить его и просто добавить @InstallInаннотацию к включенному модулю.
* Перенесите любую старую аннотацию области и псевдоним области, если вы использовали эту функцию
* Перенесите все, что @Bindsвам нужно, чтобы привязки аргументов компонентов совпадали

## Что делать с…?

### Отборочные

Квалификаторы, которые у вас есть в вашем проекте, по-прежнему действительны, они будут использоваться Hilt так же, как и Dagger.

Если у вас есть свои собственные квалификаторы @ApplicationContextи @ActivityContextквалификаторы, чтобы различать разные Contextsв вашем приложении, вы можете добавить, @Binds чтобы сопоставить их вместе, а затем выбрать замену своего использования квалификаторами Hilt на досуге.

Ява

Котлин

@InstallIn(**SingletonComponent::class**)

@Module

**interface** **ApplicationContextModule** {

@Binds

@my.app.**ApplicationContext**

**fun** **bindAppContext**(

@dagger.hilt.android.qualifiers.**ApplicationContext** context: **Context**) :

**Context**

}

### Компонент аргументы

Поскольку создание экземпляра компонента скрыто при использовании Hilt, невозможно добавить в свои собственные аргументы компонента ни экземпляры модуля, ни @BindsInstanceвызовы. Если они есть в вашем компоненте, вам нужно будет реорганизовать код и отказаться от них. Hilt поставляется с набором привязок по умолчанию для каждого компонента, которые можно увидеть [здесь](https://dagger.dev/hilt/components.html#component-bindings) . В зависимости от аргументов вашего компонента вы можете захотеть, чтобы некоторые из них зависели от этих привязок по умолчанию. Иногда это требует небольшого изменения дизайна, но большинство случаев можно решить таким образом, используя следующие стратегии. Если это не так, возможно, вам придется подумать об использовании [специального компонента](https://dagger.dev/hilt/migration-guide#custom-components) .

Например, в простейшем случае иногда привязку вообще не нужно передавать, и это может быть обычный статический @Providesметод. В другом простом случае ваш аргумент может быть просто вариантом привязки по умолчанию, например настраиваемого типа BaseFragment. Хилт не может знать, что все фрагменты будут вашим экземпляром BaseFragment, поэтому, если вам нужен фактический тип, привязанный к вашему BaseFragment, вам нужно будет сделать это с помощью приведения.

Ява

Котлин

@Component.**Builder**

**interface** **Builder** {

@BindsInstance

**fun** **fragment**(fragment: **BaseFragment**): **Builder**

}

@InstallIn(**FragmentComponent::class**)

@Module

**object** **BaseFragmentModule** {

@Provides

**fun** **provideBaseFragment**(fragment: **Fragment**) : **BaseFragment** {

**return** fragment **as** **BaseFragment**

}

}

В других случаях вашим аргументом может быть что-то в одной из привязок по умолчанию, например, действие Intent.

Ява

Котлин

@Component.**Builder**

**interface** **Builder** {

@BindsInstance

**fun** **intent**(intent: **Intent**): **Builder**

}

@InstallIn(**ActivityComponent::class**)

@Module

**object** **IntentModule** {

@Provides

**fun** **provideIntent**(activity: **Activity**) : **Intent** {

**return** activity.**getIntent**()

}

}

Наконец, вам, возможно, придется переделать некоторые вещи, если они были по-разному настроены для разных компонентов активности или фрагментов. Например, вы можете использовать новый интерфейс для действия, чтобы предоставить объект.

Ява

Котлин

@Component.**Builder**

**interface** **Builder** {

@BindsInstance

**fun** **foo**(foo: **Foo**): **Builder** *// Foo is different per Activity*

}

*// Define an interface the activity can implement to provide a custom Foo*

**interface** **HasFoo** {

**fun** **getFoo**() : **Foo**

}

@InstallIn(**ActivityComponent::class**)

@Module

**object** **FooModule** {

@Provides

**fun** **provideFoo**(activity: **Activity**) : **Foo**? {

**if** (activity **is** **HasFoo**) {

**return** activity.**getFoo**()

}

**return** **null**

}

}

### Пользовательские компоненты

Если у вас есть другие компоненты, которые не соответствуют компонентам Hilt, вы должны сначала подумать, можно ли их упростить в компоненты Hilt. Если нет, вы можете оставить свои компоненты как ручные компоненты Dagger. Выберите раздел ниже в зависимости от того, хотите ли вы использовать зависимости компонентов или подкомпоненты.

#### Зависимости компонентов

Зависимости компонентов могут быть связаны с [@EntryPoint](https://dagger.dev/hilt/entry-points.html).

Например, если у вас была зависимость компонента от класса SingletonComponent, вы можете сохранить его работоспособность, выделив необходимые методы в интерфейс, аннотированный с помощью [@EntryPoint](https://dagger.dev/hilt/entry-points.html).

Ява

Котлин

*// Starting with this component dependency*

@Component

**interface** **MySingletonComponent** {

*// These bindings are exposed for MyCustomComponent*

**fun** **getFoo**(): **Foo**

**fun** **getBar**(): **Bar**

**fun** **getBaz**(): **Baz**

**..**.

}

@Component(dependencies = [**MySingletonComponent::class**])

**interface** **MyCustomComponent** {

@Component.**Builder**

**interface** **Builder** {

**fun** **appComponent**(appComponent: **MySingletonComponent**): **Builder**

**fun** **build**(): **MyCustomComponent**

}

}

*// It can be migrated to Hilt with the following classes*

@InstallIn(**SingletonComponent::class**)

@EntryPoint

**interface** **CustomComponentDependencies** {

**fun** **getFoo**(): **Foo**

**fun** **getBar**(): **Bar**

**fun** **getBaz**(): **Baz**

**..**.

}

@Component(dependencies = [**CustomComponentDependencies::class**])

**interface** **MyCustomComponent** {

@Component.**Builder**

**interface** **Builder** {

**fun** **appComponentDeps**(deps: **CustomComponentDependencies**): **Builder**

**fun** **build**(): **MyCustomComponent**

}

}

При создании пользовательского компонента вы можете получить экземпляр класса CustomComponentDependencies, используя EntryPoints.

Ява

Котлин

**DaggerMyCustomComponent**.**builder**()

.**appComponentDeps**(

**EntryPoints**.**get**(

applicationContext,

**CustomComponentDependencies::class**.java))

.**build**()

#### Подкомпоненты

Подкомпоненты могут быть добавлены как дочерние элементы любого компонента Hilt таким же образом, как если бы вы устанавливали обычный подкомпонент со встроенным конструктором подкомпонентов в Dagger. Просто установите подкомпонент в модуль с соответствующим @InstallInродительским компонентом.

Например, если у вас есть FooSubcomponentдочерний объект SingletonComponent, вы можете установить его, как показано в следующем примере:

Ява

Котлин

@InstallIn(**SingletonComponent::class**)

@Module(subcomponents = **FooSubcomponent::class**)

**interface** **FooModule** {}

### Зависимости компонентов для компонентов, которые отображаются на компоненты Hilt

Если в настоящее время вы используете зависимости компонентов и ваши компоненты относительно хорошо сопоставляются с компонентами Hilt, то при миграции вам также нужно будет помнить о различиях между зависимостями компонентов и подкомпонентами. Вы также можете посетить эту страницу, где описаны некоторые из причин, по которым Hilt решил использовать [подкомпоненты](https://dagger.dev/hilt/subcomponents-vs-deps.html) .

Основные отличия, о которых следует помнить, заключаются в том, что привязки автоматически наследуются от родителя. Это, вероятно, означает избавление от дополнительных методов для раскрытия привязок, а также работу с любыми повторяющимися привязками, которые могут возникнуть, если привязка определена как в родительском, так и в дочернем компонентах. Избавиться от этих дополнительных методов для раскрытия привязок необязательно, поскольку они технически не нарушат вашу сборку, но рекомендуется, поскольку они могут предотвратить обрезку мертвого кода. Их можно безопасно перенести, как описано в этом [разделе](https://dagger.dev/hilt/migration-guide#b-handle-any-extended-interfaces-or-methods) .

Вот пример открытых привязок:

Ява

Котлин

@Component

**interface** **MySingletonComponent** {

*// These bindings were likely exposed for component dependencies.*

*// Consider getting rid of them.*

**fun** **getFoo**(): **Foo**

**fun** **getBar**(): **Bar**

**fun** **getBaz**(): **Baz**

**..**.

}

Затем, когда вы выполните шаги, описанные выше для переноса компонентов, если ваш компонент имеет зависимость от компонента, эквивалентного родительскому элементу Hilt, просто удалите dep по мере удаления остальной части компонента.

Ява

Котлин

*// Just delete these deps as you follow the migration guide for migrating*

*// the rest of the component*

@Component(deps = [**MySingletonComponent::class**])

**interface** **MyActivityComponent** {

**..**.

}

# Дополнительная инъекция

## Зачем вам необязательная инъекция?

Фрагменты рукояти должны быть прикреплены к действиям Hilt, а действия Hilt должны быть прикреплены к приложениям Hilt. Хотя это естественное ограничение для чистой кодовой базы Hilt, это может быть проблемой во время миграции на Hilt, если у вас есть фрагмент или действие, которые используются в контексте, отличном от Hilt. Например, вы хотите перенести фрагмент в Hilt, но он используется слишком во многих местах для одновременной миграции. Без дополнительной инъекции вам придется сначала перенести в Hilt все действия, использующие этот фрагмент, иначе фрагмент выйдет из строя при поиске компонентов Hilt, когда он пытается внедрить себя. В зависимости от размера вашей кодовой базы это может быть большим мероприятием.

## Как использовать @OptionalInject

Если вы отметите [@AndroidEntryPoint](https://dagger.dev/hilt/android-entry-point.html)класс, [@OptionalInject](https://dagger.dev/api/latest/dagger/hilt/android/migration/OptionalInject.html) он будет пытаться внедрить только в том случае, если родитель использует Hilt и не требует этого. Использование этой аннотации также вызовет создание wasInjectedByHilt()метода для сгенерированного базового класса, который возвращает истину, если инъекция была успешной.

**Примечание:** поскольку API, созданный на основе базового класса, недоступен для пользователей [плагина gradle](https://dagger.dev/hilt/gradle-setup.html#hilt-gradle-plugin) , существует альтернативный API для доступа к этой функции с использованием статического вспомогательного метода в [OptionalInjectCheck](https://dagger.dev/api/latest/dagger/hilt/android/migration/OptionalInjectCheck.html).

Это дает вам возможность предоставлять зависимости другим способом (обычно в зависимости от того, каким способом вы получали зависимости до использования Hilt).

Например:

Ява

Котлин

@OptionalInject

@AndroidEntryPoint

**public** **final** **class** **MyFragment** **extends** **Fragment** **{**

@Inject **Foo** foo**;**

@Override **public** **void** **onAttach(Activity** activity**)** **{**

**super.**onAttach**(**activity**);** *// Injection will happen here, but only if the Activity used Hilt*

**if** **(!**wasInjectedByHilt**())** **{**

*// Get Dagger components the previous way and inject*

**}**

**}**

**}**

# Псевдонимы областей видимости

## Зачем вам нужен псевдоним области?

Псевдонимы области полезны во время миграции на Hilt, если у вас много кода, использующего предыдущую аннотацию области, и теперь вы хотите переключиться на одну из аннотаций области, предоставленных Hilt. В зависимости от размера вашей кодовой базы это может означать изменение аннотации области во многих местах. Добавляя псевдоним области, вы можете вносить изменения постепенно.

Использование псевдонима области видимости просто говорит Dagger и Hilt, что эти аннотации области действия должны обрабатываться одинаково.

## Как использовать @AliasOf

Если вы отметите аннотацию области с помощью [@AliasOf](https://dagger.dev/api/latest/dagger/hilt/migration/AliasOf.html), она сообщит Hilt, что аннотированная аннотация области должна обрабатываться так же, как и в значении @AliasOfаннотации. Значение аннотации должно быть аннотацией области видимости, используемой в @DefineComponentтипе, чтобы Hilt знал, что с ней делать.

Например, следующее использует предыдущую @MyActivityScopedаннотацию и делает ее эквивалентной Hilt @ActivityScoped. Теперь его будет легко постепенно заменить @MyActivityScopedверсией Hilt.

Ява

Котлин

@Scope

@AliasOf**(**dagger**.**hilt**.**android**.**scopes**.**ActivityScoped**.**class**)**

**public** @interface **MyActivityScoped** **{}**

# Параметры компилятора

## Отключение @InstallInчека

По умолчанию Hilt проверяет @Moduleклассы на наличие @InstallInаннотации и выдает ошибку, если она отсутствует. Это связано с тем, что если кто-то случайно забудет установить @InstallInмодуль, будет очень сложно отладить, что Hilt его не берет.

Однако эта проверка иногда может быть чрезмерно широкой, особенно в середине миграции. Чтобы отключить эту проверку, можно использовать этот флаг:

-Adagger.hilt.disableModulesHaveInstallInCheck=true.

Кроме того, проверку можно отключить на уровне отдельного модуля, пометив модуль с помощью [@DisableInstallInCheck](https://dagger.dev/api/latest/dagger/hilt/migration/DisableInstallInCheck.html).

# Создание расширений

## Создание модулей и точек входа

Hilt особенно хорошо подходит для расширений или библиотек, которые хотят интегрироваться с Hilt из-за стандартных компонентов и способа, которым модули и точки входа выбираются из пути к классам.

Однако расширения, которые генерируют @InstallInмодуль или точку входа, должны будут добавить некоторую дополнительную информацию к сгенерированным классам, чтобы Hilt мог их правильно обработать.

### @GeneratesRootInput

Поскольку Hilt неявно выбирает модули и точки входа из пути к классам, Hilt нужна дополнительная информация, чтобы знать, нужно ли ему ждать, пока ваше расширение сгенерирует код, прежде чем пытаться сгенерировать компоненты Dagger. Это делается путем аннотирования вашего класса аннотации, который запускает генерацию кода, с помощью [@GeneratesRootInput](https://dagger.dev/api/latest/dagger/hilt/GeneratesRootInput.html).

Например, если расширение генерирует модуль каждый раз, когда кто-то использует @GenerateMyModuleаннотацию, @GenerateMyModuleего необходимо аннотировать следующим образом:

Ява

Котлин

@GeneratesRootInput

**annotation** **class** **GenerateMyModule** {}

Обратите внимание, что без аннотации Hilt не обязательно пропустит ваши модули, потому что он все равно может подобрать их, если ожидает создания чего-то еще. Это, конечно, ненадежно.

### @OriginatingElement

Как описано на [странице тестирования](https://dagger.dev/hilt/testing.html#nested-modules) , вложенные модули в тестах изолированы от внешнего теста. Однако сгенерированные модули для теста не могут быть созданы как вложенный класс. Чтобы правильно поддерживать это, сгенерированный код должен быть аннотирован [@OriginatingElement](https://dagger.dev/api/latest/dagger/hilt/codegen/OriginatingElement.html)аннотацией с классом верхнего уровня в качестве значения. Обратите внимание, что это не всегда то же самое, что и включающий класс, поскольку может быть много уровней вложенности.

Например, предположим, что расширение запускается следующим кодом и генерирует модуль с именем FooTest\_FooModule.

Ява

Котлин

@HiltAndroidTest

**class** **FooTest** {

@GenerateMyModule

**val** foo: **Foo** = **Foo**()

**..**.

}

Затем сгенерированный FooTest\_FooModuleобъект необходимо аннотировать следующим образом:

Ява

Котлин

@OriginatingElement(**FooTest::class**)

@Module

@InstallIn(**SingletonComponent::class**)

**interface** **FooTest\_FooModule** {

**..**.

}

**ПРОЕКТНЫЕ РЕШЕНИЯ**

# Обзор дизайна

## Генерация компонентов и установка модуля / точки входа

Hilt генерирует компоненты, находя все модули и точки входа в транзитивном пути к классам. @InstallInАннотация на каждом модуль и ввод точке генерирует небольшой класс метаданных в определенном пакете. Специальный пакет проверяется при обработке, @HiltAndroidAppчтобы найти все агрегированные элементы, которые необходимо установить в компоненты. Та же стратегия используется для других вспомогательных классов, таких как @DefineComponentи @AliasOf.

Поскольку Android Applicationсоздается одновременно, сгенерированный файл Applicationимеет прямую ссылку на корневой компонент, который является файлом SingletonComponent.

Поскольку приложение HiltTestApplicationдолжно поддерживать несколько тестов, в отличие от производственного приложения, для поиска сгенерированных компонентов используется отражение. Это полезно, потому что позволяет отделить тестовое приложение от сборки с тестами, что позволяет Hilt предоставить удобное значение по умолчанию вместо того, чтобы требовать от каждого проекта кодирования генерации тестового приложения. Отражение не используется в производстве, потому что оно дает меньшую ценность, а отражение может иметь больше затрат.

Агрегирование всех модулей в пути к классам хорошо работает для тестов, потому что это означает, что тесты могут легко добавлять привязки, просто вкладывая классы в тестовый класс (или, что еще лучше, используя тот, @BindValueкоторый генерирует модуль). Точно так же обнаружение модуля также позволяет классам встраивать @Moduleклассы как внутренние классы. Это можно использовать, чтобы гарантировать, что класс не может использоваться без связанных привязок Dagger, и сделает его использование менее подверженным ошибкам (например, соединение класса с @BindsOptionalOfпотребляемым им или @Bindsс интерфейсом).

## @AndroidEntryPoint инъекция

@AndroidEntryPointработает путем создания базового класса, который пользовательский код расширяет прямо или косвенно через преобразование в [подключаемом модуле Gradle](https://dagger.dev/hilt/gradle-setup.html#hilt-gradle-plugin) . Этот базовый класс отвечает за получение родительского компонента (через интерфейсы Hilt на родительском), создание компонента, внедрение класса и предоставление компонента дочерним элементам через интерфейсы Hilt.

Например, чтобы внедрить действие, сгенерированный код, по сути, выполняет следующие действия (упрощенный для удобства чтения):

@Override **public** **void** **onCreate(Bundled** savedInstanceState**)** **{**

*// This gets the parent component from the Application (in reality there is*

*// actually the activity retained component as the parent).*

**Object** parentComponent **=**

**((GeneratedComponentManager)** getApplication**()).**generatedComponent**();**

*// This creates the activity component. This involves an unsafe cast*

*// to know the parent component has the methods to build the activity component.*

**Object** activityComponent **=** **((ActivityComponentBuilderEntryPoint)** parentComponent**)**

**.**activityComponentBuilder**()**

**.**activity**(this)**

**.**build**();**

*// This injects the activity. It also involves an unsafe cast to get access*

*// to the activity inject method. Like the other unsafe casts, these casts*

*// break build dependencies and are safe because they are code generated and*

*// guaranteed via the classpath discovery of modules/interfaces.*

**(**MyActivity\_GeneratedInjector**)** activityComponent**).**inject**(this).**

**}**

Генерация всего этого связующего кода упрощает и упрощает снятие зависимостей с помощью небезопасных преобразований. Кроме того, автоматическое обнаружение в сочетании с тем фактом, что интерфейсы генерируются с действием, которое их использует, делает так, что включение или удаление @AndroidEntryPointдобавляет / уносит с собой все его зависимости. Это позволяет приложениям, созданным с помощью Hilt, быть модульными.

В большинстве случаев родительский компонент легко получить, но в случае представлений и фрагментов это не так просто, потому что представления получают контекст активности. Для поддержки представлений с привязками фрагментов сгенерированный базовый класс для фрагментов переопределяет, getLayoutInflaterчтобы обернуть Contextв объект ContextWrapper, содержащий компонент Dagger, который должно получить представление.

Благодаря стандартизации всех этих проектных решений в Hilt, интеграция библиотек с действиями / фрагментами / представлениями должна быть намного проще.

# Философия тестирования рукояти

## Обзор

Эта страница призвана объяснить методы тестирования, на которых построен Hilt. Многие API и функциональность в Hilt (а также определенная нехватка функциональности) были созданы на основе неустановленной философии того, что делает хороший тест. Однако понятие хорошего теста не является общепринятым, поэтому этот документ призван разъяснить философию тестирования команды Hilt.

## Что тестировать

Hilt поощряет тестирование в максимально возможной степени с точки зрения внешнего пользователя. Взгляд внешнего пользователя может означать многое. Это может означать фактических пользователей вашего приложения или службы, но также может быть ограничено до пользователей вашего API или класса.

Ключевой момент в том, что тесты не должны кодировать детали реализации. Если полагаться на детали реализации, такие как проверка того, что был вызван внутренний метод, тест становится нестабильным. Если в результате рефакторинга изменяется имя внутреннего метода, хороший тест не нужно обновлять. Единственные изменения, которые должны нарушить существующие тесты, - это те, которые изменяют ваше поведение, видимое пользователем.

## Использование реальных зависимостей

Философия тестирования Hilt не предписывает строгих правил, например, у каждого класса должен быть свой собственный тест. Фактически, обычно такое правило нарушает вышеуказанный принцип тестирования с точки зрения пользователя. Тесты должны быть настолько малыми, насколько это необходимо, чтобы их было удобно писать и запускать (например, достаточно маленькими, чтобы быть быстрыми или не ресурсоемкими). При прочих равных, тесты в этом порядке должны предпочесть:

* Используйте реальный код для зависимости
* Используйте стандартную подделку, предоставленную библиотекой
* Используйте издевательство как последнее средство

Однако есть компромиссы. Использование реальных зависимостей / реальных DI в тестах может быть чрезмерно трудным по одной или обеим из следующих причин:

* Настройка и создание реальной зависимости / реального DI - это слишком много шаблонного или повторяющегося кода.
* Использование реальной зависимости приводит к снижению производительности (например, необходимость запуска внутреннего сервера).

Рукоять была создана для решения первой проблемы настройки (подробнее об этом ниже). Производительность может быть проблемой, но часто не является проблемой для большинства зависимостей. Вероятно, это проблема только при использовании зависимостей со значительным количеством операций ввода-вывода. Итак, если тест можно написать более удобно и надежно, используя больше реальных зависимостей без значительного снижения производительности, его следует писать с использованием этих зависимостей. Для тех классов, которые должны иметь серьезные отрицательные эффекты в тестах, Hilt предоставляет средства для отключения привязок.

Использование большего количества реальных зависимостей дает значительные преимущества:

* Реальные зависимости с большей вероятностью обнаружат реальные проблемы. Они не могут устареть, как шутки.
* В сочетании с описанным выше принципом тестирования с точки зрения пользователя вам, вероятно, потребуется написать меньше тестов для того же покрытия.
* Обрыв теста с большей вероятностью указывает на реальную проблему, чем на неверно настроенную подделку или имитацию (и, наоборот, прохождение теста с большей вероятностью означает, что код действительно работает).
* Использование более реальных зависимостей часто согласуется с вышеуказанным принципом тестирования с точки зрения пользователя, поскольку они, скорее всего, не смогут поменять местами ваши зависимости.

Если реальную зависимость использовать невозможно, стандартная подделка, предоставляемая библиотекой, обычно является следующим лучшим вариантом. Стандартизированная подделка лучше, чем имитация, потому что она с большей вероятностью будет синхронизироваться с производственным кодом, если она поддерживается авторами библиотеки и, таким образом, обеспечивает более надежное покрытие. По этим причинам моки обычно являются последним средством.

## Рукоять, DI и тестирование

Объяснив эти основы, мы теперь переходим к особенностям Hilt, DI и тестирования. В соответствии с философией использования реальных объектов, Hilt предлагает использовать в тестах внедрение зависимостей / Dagger. Это более реалистично, поскольку объекты создаются так же, как и в производственном коде. Это означает, что тесты не более хрупкие, чем был бы производственный код, и упрощают использование реальных объектов. Фактически, для типов, которые имеют @Injectконструкторы, на самом деле проще и меньше кода следовать этому совету и использовать реальный код, чем настраивать и связывать макет.

К сожалению, такое тестирование без Hilt традиционно было затруднено на практике из-за шаблонности и дополнительной работы по настройке Dagger в тестах. Тем не менее, Hilt создает шаблон для вас и имеет четкую историю для настройки различных конфигураций привязок для тестов, когда вам действительно нужна подделка или имитация. С Hilt эта проблема больше не должна препятствовать написанию тестов с Dagger и, следовательно, легко использовать реальные зависимости.

## Минусы альтернативы

Альтернатива не использовать Dagger в модульных тестах - довольно распространенный совет. Это, к сожалению, имеет существенные недостатки, хотя это вполне понятный совет, учитывая сложность использования Dagger в тестах без рукояти.

Например, допустим, у нас есть класс Foo, который мы хотим протестировать:

Ява

Котлин

**class** **Foo** @Inject **constructor**(bar: **Bar**) {

}

В этом случае, если не использовать Dagger, тест будет напрямую создавать экземпляр Foo, просто вызывая конструктор. На первый взгляд это кажется очень простым и разумным делом; однако все начинает распутываться, когда вы пытаетесь предоставить экземпляр Barв Fooконструктор.

### Прямая инстанциация поощряет моки

Исходя из ранее обсуждавшейся философии тестирования, мы должны предпочесть настоящий Barкласс. Однако как нам это сделать? На самом деле это всего лишь рекурсия получения реального Fooкласса для тестирования: вам нужно будет создать его экземпляр самостоятельно, а если он Barимеет собственные зависимости, то для этого потребуется аналогичное создание экземпляра. Чтобы не заходить слишком глубоко, вам, вероятно, придется начать использовать подделку или имитацию, не из-за влияния на скорость или производительность теста, а просто для того, чтобы избежать слишком хрупкого шаблона, который вызывает проблемы с обслуживанием. Это не повод использовать подделку или имитацию, но вы все равно вынуждены это делать.

Альтернативой, как обсуждалось выше, является использование стандартной подделки, которая может помочь сократить зависимости и снизить нагрузку на обслуживание прямого создания экземпляра. Однако и это не всегда так просто. Часто у хорошей подделки также есть необходимые зависимости. Например, FakeBarможет потребоваться принять a, FakeClockесли настоящий Barвзял a Clock. Это потому, что a FakeClockчасто является точкой координации между различными классами. (Представьте, что если бы у вас Fooбыла другая зависимость, Bazкоторая также использовала часы, вы бы хотели, FakeBazчтобы он использовал тот же FakeClockэкземпляр, чтобы все координировалось при опережении времени). Управление этими зависимостями может быстро выйти из-под контроля.

Обычно это приводит авторов тестов к насмешкам. Мок решает проблему утомительного следования этим цепочкам зависимостей, но имеет существенные недостатки в том, что он может легко устареть незаметно и сделать тест бесполезным с точки зрения общей цели поиска реальных ошибок. Поскольку никто не проверяет фиктивное поведение, кроме автора теста, это обычно означает, что по прошествии достаточного времени существует приличная вероятность того, что тест больше не проверяет полезный сценарий.

### Прямое создание экземпляра кодирует детали реализации в тесте

Прямое создание экземпляров также нарушает философию отказа от кодирования деталей реализации в тесте, поскольку вызов конструктора кодирует детали его зависимостей. Если бы Barэто был @Injectтип конструктора, то пользователю Fooнезачем было бы знать о существовании Barкласса, поскольку он мог бы легко быть деталью реализации от логики рефакторинга Fooв другой класс, частный для библиотеки.

Чтобы проиллюстрировать этот момент, рассмотрим наличие Fooдвух зависимостей, например Foo(Bar, Baz). В Dagger переключение порядка этих параметров в @Inject конструкторе не выполняется. Тем не менее, если бы мы тестировали Fooчерез прямое создание экземпляра, нам все равно пришлось бы обновить тест. Точно так же добавление использования нового @Inject класса или необязательной привязки также будет невидимым изменением для рабочих пользователей этого класса, но тест все равно нужно будет обновить.

## Резюме

Hilt был разработан для устранения недостатков использования Dagger в тестах, чтобы облегчить тестирование с реальными зависимостями. Тесты, написанные с использованием Hilt, будут лучше восприниматься, если будут следовать этим принципам.

# Монолитные компоненты

## Обзор

Hilt использует монолитную компонентную систему. Это означает, что для внедрения всех классов активности используется одно определение компонента деятельности. То же самое для фрагментов и других типов Android. Однако каждое действие имеет отдельный экземпляр компонента, но используется только определение класса. Это отличие от полилитической компонентной системы, в которой каждое действие имеет отдельное определение компонента. Полилитическая система - это режим по умолчанию при использовании dagger.android @ContributesAndroidInjector. На этой странице рассматриваются некоторые причины, по которым Hilt был разработан с использованием монолитных компонентов, а также компромиссы между двумя моделями.

## Пространство для ключей с единой привязкой

Одним из основных преимуществ использования монолитной системы, такой как в Hilt, является объединение пространства ключей привязки. Если вы находитесь во фрагменте, внедряющем Fooкласс, гораздо легче найти, откуда эта Fooпривязка, потому что она не может различаться в зависимости от активности, к которой прикреплен фрагмент. Полилитические компоненты дают вам больше гибкости для определения различных привязок для каждого действия, но это обычно приводит к еще большей запутанности, поскольку базы кода становятся больше и их труднее отслеживать.

Чтобы привязки оставались закрытыми только для кода, который должен их использовать, мы рекомендуем использовать аннотации квалификаторов, которые защищены ограничением видимости, или подключаемый модуль [SPI](https://dagger.dev/dev-guide/spi) для принудительного разделения кода.

## Простота настройки

Пространство для ключей с единой привязкой также значительно упрощает настройку. Это уменьшает количество мест, где может быть установлен модуль, что упрощает замену привязок для тестирования. Это также означает, что вам не нужно беспокоиться о распространении модулей для функций во все места, где эта функция используется. Это может быть действительно полезно для функций, которые используют разные области действия. В полилитическом мире функция, использующая объект с областью действия фрагмента и объект с областью действия, может потребовать от пользователя включения модулей во фрагмент, а затем во все действия, использующие этот фрагмент. Часто этот код конфигурации просто добавляет к шаблону и нарушает инкапсуляцию.

## Менее сгенерированный код

Использование монолитной системы также означает меньше генерируемого кода. Когда общий модуль используется во многих подкомпонентах (как это может быть в случае с общим вспомогательным классом активности), это означает, что сгенерированный код Dagger должен быть повторен для каждого подкомпонента. Хотя поначалу это может показаться не очень большим, он может быстро накапливаться во многих действиях и умножаться на множество фрагментов или представлений.

## fastInit и задержка запуска

Некоторые пользователи могут беспокоиться о том, как это повлияет на задержку запуска. Если вы используете [опцию](https://dagger.dev/compiler-options#fastinit-mode) компиляции [fastInit](https://dagger.dev/compiler-options#fastinit-mode) , монолитные компоненты не должны оказывать заметного влияния на задержку запуска. Это настройка по умолчанию для пользователей Hilt gradle, использующих [плагин,](https://dagger.dev/hilt/gradle-setup.html#hilt-gradle-plugin) и обычно это режим компиляции Dagger, используемый на Android.

# Подкомпоненты и зависимости компонентов

## Обзор

Hilt основан на использовании подкомпонентов Dagger, а не на зависимостях компонентов. На этой странице объясняются некоторые причины, по которым Hilt был разработан таким образом.

## Пространство для ключей с единой привязкой

Подкомпоненты по умолчанию распространяют все привязки. Это включает в себя множественные привязки, которые может быть трудно распространить через зависимости компонентов. Это создает объединенное пространство ключей привязки. Как правило, это упрощает понимание графа Dagger, потому что вам не нужно беспокоиться о том, распространяется ли привязка от родительского компонента на дочерний компонент. Кроме того, если привязки не распространяются с зависимостями компонентов, можно использовать два разных определения одного и того же ключа привязки в разных компонентах. Это может затруднить прохождение кода при отладке проблем, поскольку определение привязки будет основано на контексте использования.

Одним из недостатков единого пространства ключей привязки является то, что может потребоваться дополнительная работа по наложению ограничений на использование кода (например, если одна функция не должна использовать привязки из другой функции). Для этого мы обычно рекомендуем использовать аннотации квалификаторов с ограниченной видимостью или использовать подключаемый модуль SPI для принудительного разделения кода. Использование квалификатора или подключаемого модуля SPI лучше, чем встраивание этих проблем в структуру вашего графика зависимостей компонентов Dagger, потому что часто эти правила кодируют политику. Подобные политические решения часто меняются (или необходимо иметь разрешенные исключения), и необходимость реструктуризации графа зависимостей компонентов Dagger на основе этих изменений может быть дорогостоящей.

## Распространение привязок с зависимостями компонентов побеждает обрезку Dagger

Поскольку Dagger может видеть точки входа в график, он может определить, какие привязки не используются, и не сгенерировать код для этих привязок. Эта оптимизация проходит через подкомпоненты, но проигрывает из-за зависимостей компонентов, потому что распространение привязок через зависимости компонентов добавляет методы точки входа. Таким образом, даже если методы точки входа используются только другими компонентами Dagger и между компонентами привязка не используется, Dagger будет вынужден генерировать этот мертвый код, чтобы придерживаться своего контракта.

## Конфигурация в корне и скорость сборки

Одно из основных преимуществ зависимостей компонентов - это создание кода Dagger отдельно и параллельно. Это можно сделать из-за отсутствия неявного разделения, которое делает компоненты черными ящиками по отношению друг к другу. Однако Hilt уже основан на идее централизованной настройки на основе зависимостей сборки. Поскольку Hilt должен агрегировать модули, все компоненты в любом случае будут сгенерированы одновременно, поэтому мы не сможем воспользоваться преимуществами параллельной сборки.

Вместо этого, чтобы снизить скорость сборки, Hilt рекомендует создавать небольшие тестовые приложения для разработки индивидуальных функций. Без Hilt это было бы сложно сделать из-за повторяющегося шаблона Dagger для небольшого тестового приложения. Однако, поскольку Hilt генерирует всю часть Dagger на основе зависимостей сборки, создание небольшого тестового приложения должно быть намного проще.

# Dagger & Android

Одним из основных преимуществ Dagger 2 перед большинством других фреймворков внедрения зависимостей является то, что его строго сгенерированная реализация (без отражения) означает, что его можно использовать в приложениях Android. Тем не менее, *есть* еще некоторые соображения , которые будут сделаны при использовании кинжала в приложениях Android.

**Философия**

Хотя код, написанный для Android, является источником Java, он часто сильно отличается по стилю. Обычно такие различия существуют для учета уникальных [характеристик](http://developer.android.com/training/best-performance.html) мобильной платформы.

Но многие из шаблонов, обычно применяемых к коду, предназначенному для Android, противоречат шаблонам, применяемым к другому Java-коду. Даже большая часть советов в [Effective Java](https://books.google.com/books?id=ka2VUBqHiWkC) считается неуместной для Android.

Для достижения целей как идиоматического, так и переносимого кода, Dagger полагается на [ProGuard](http://proguard.sourceforge.net/) для [постобработки](http://proguard.sourceforge.net/) скомпилированного байт-кода. Это позволяет Dagger генерировать исходный код, который выглядит и воспринимается естественно как на сервере, так и на Android, при этом используя различные инструментальные цепочки для создания байт-кода, который эффективно выполняется в обеих средах. Более того, у Dagger есть явная цель - гарантировать, что исходный код Java, который он генерирует, постоянно совместим с оптимизациями ProGuard.

Конечно, не все проблемы можно решить таким образом, но это основной механизм, с помощью которого будет обеспечена совместимость с Android.

**tl; dr**

Dagger предполагает, что пользователи Android будут использовать R8 или ProGuard.

**Почему Dagger на Android - это сложно**

Одна из основных трудностей написания Android-приложения с использованием Dagger заключается в том, что многие классы фреймворка Android создаются самой ОС, например Activityи Fragment, но Dagger работает лучше всего, если он может создавать все внедренные объекты. Вместо этого вам нужно выполнить внедрение членов в методе жизненного цикла. Это означает, что многие классы в конечном итоге выглядят так:

**public** **class** **FrombulationActivity** **extends** **Activity** **{**

@Inject **Frombulator** frombulator**;**

@Override

**public** **void** **onCreate(Bundle** savedInstanceState**)** **{**

**super.**onCreate**(**savedInstanceState**);**

*// DO THIS FIRST. Otherwise frombulator might be null!*

**((SomeApplicationBaseType)** getContext**().**getApplicationContext**())**

**.**getApplicationComponent**()**

**.**newActivityComponentBuilder**()**

**.**activity**(this)**

**.**build**()**

**.**inject**(this);**

*// ... now you can write the exciting code*

**}**

**}**

У этого есть несколько проблем:

1. Копирование и вставка кода затрудняет последующий рефакторинг. Чем больше и больше разработчиков копируют этот блок, тем меньше будет знать, что он на самом деле делает.
2. Более того, он требует, чтобы тип, запрашивающий инъекцию ( FrombulationActivity), знал о своем инжекторе. Даже если это делается с помощью интерфейсов, а не конкретных типов, это нарушает основной принцип внедрения зависимостей: класс не должен ничего знать о том, как он внедряется.

**dagger.android**

Классы [dagger.android](https://dagger.dev/api/latest/dagger/android/package-summary.html)предлагают один подход к упрощению вышеуказанных проблем. Это требует изучения некоторых дополнительных API и концепций, но дает вам меньше шаблонов и инъекций в классы Android в нужном месте жизненного цикла.

Другой подход - просто использовать обычные API-интерфейсы Dagger и следовать руководствам, подобным приведенному [здесь](https://developer.android.com/training/dependency-injection/dagger-android) . Это может быть проще для понимания, но имеет обратную сторону, заключающуюся в необходимости писать дополнительный шаблон вручную.

Команды Jetpack и Dagger вместе работают над [новой инициативой](https://medium.com/androiddevelopers/dependency-injection-guidance-on-android-ads-2019-b0b56d774bc2) для Dagger на Android, которая, как надеется, существенно изменит существующее положение вещей. Хотя, к сожалению, он еще не готов, это может быть то, что нужно учитывать при выборе того, как использовать Dagger в ваших проектах Android сегодня.

**Инъекция Activityпредметов**

1. Установите [AndroidInjectionModule](https://dagger.dev/api/latest/dagger/android/AndroidInjectionModule.html)в своем приложении компонент, чтобы обеспечить доступность всех привязок, необходимых для этих базовых типов.
2. Начните с написания, @Subcomponentкоторый реализует [AndroidInjector<YourActivity>](https://dagger.dev/api/latest/dagger/android/AndroidInjector.html), с @Subcomponent.Factoryрасширением [AndroidInjector.Factory<YourActivity>](https://dagger.dev/api/latest/dagger/android/AndroidInjector.Factory.html):
3. @Subcomponent**(**modules **=** **...)**
4. **public** **interface** **YourActivitySubcomponent** **extends** **AndroidInjector<YourActivity>** **{**
5. @Subcomponent**.**Factory
6. **public** **interface** **Factory** **extends** **AndroidInjector.**Factory**<YourActivity>** **{}**
7. **}**
8. После определения подкомпонента добавьте его в иерархию компонентов, определив модуль, который связывает фабрику подкомпонентов, и добавив его к компоненту, который внедряет ваши Application:
9. @Module**(**subcomponents **=** **YourActivitySubcomponent.**class**)**
10. **abstract** **class** **YourActivityModule** **{**
11. @Binds
12. @IntoMap
13. @ClassKey**(YourActivity.**class**)**
14. **abstract** **AndroidInjector.**Factory**<?>**
15. bindYourAndroidInjectorFactory**(YourActivitySubcomponent.**Factory factory**);**
16. **}**
17. @Component**(**modules **=** **{...,** **YourActivityModule.**class**})**
18. **interface** **YourApplicationComponent** **{**
19. **void** **inject(YourApplication** application**);**
20. **}**

Совет: если ваш подкомпонент и его фабрика не имеют других методов или супертипов, кроме упомянутых в шаге № 2, вы можете использовать [@ContributesAndroidInjector](https://dagger.dev/api/latest/dagger/android/ContributesAndroidInjector.html)их для их создания. Вместо шагов 2 и 3 добавьте abstractметод модуля, который возвращает ваше действие, аннотируйте его @ContributesAndroidInjectorи укажите модули, которые вы хотите установить в подкомпонент. Если подкомпоненту требуются области действия, примените также аннотации области к методу.

@ActivityScope

@ContributesAndroidInjector**(**modules **=** **{** */\* modules to install into the subcomponent \*/* **})**

**abstract** **YourActivity** **contributeYourAndroidInjector();**

1. Затем, сделайте ваш Applicationреализовать [HasAndroidInjector](https://dagger.dev/api/latest/dagger/android/HasAndroidInjector.html) и для возврата из метода:@Inject[DispatchingAndroidInjector<Object>](https://dagger.dev/api/latest/dagger/android/DispatchingAndroidInjector.html)androidInjector()
2. **public** **class** **YourApplication** **extends** **Application** **implements** **HasAndroidInjector** **{**
3. @Inject **DispatchingAndroidInjector<Object>** dispatchingAndroidInjector**;**
4. @Override
5. **public** **void** **onCreate()** **{**
6. **super.**onCreate**();**
7. **DaggerYourApplicationComponent.**create**()**
8. **.**inject**(this);**
9. **}**
10. @Override
11. **public** **AndroidInjector<Object>** **androidInjector()** **{**
12. **return** dispatchingAndroidInjector**;**
13. **}**
14. **}**
15. Наконец, в вашем Activity.onCreate()методе вызовите *перед* вызовом :[AndroidInjection.inject(this)](https://dagger.dev/api/latest/dagger/android/AndroidInjection.html#inject-android.app.Activity-) super.onCreate();
16. **public** **class** **YourActivity** **extends** **Activity** **{**
17. **public** **void** **onCreate(Bundle** savedInstanceState**)** **{**
18. **AndroidInjection.**inject**(this);**
19. **super.**onCreate**(**savedInstanceState**);**
20. **}**
21. **}**
22. Поздравляю!

**Как это работало?**

AndroidInjection.inject()получает DispatchingAndroidInjector<Object>от Applicationи передает информацию о вашей деятельности inject(Activity). Он DispatchingAndroidInjectorищет AndroidInjector.Factoryкласс вашей активности (который есть YourActivitySubcomponent.Factory), создает AndroidInjector(который есть YourActivitySubcomponent) и передает вашу активность в inject(YourActivity).

**Инъекция Fragmentпредметов**

Ввести a Fragmentтак же просто, как ввести Activity. Таким же образом определите свой подкомпонент.

Вместо того, чтобы вводить, onCreate()как это делается для Activity типов, [вводите Fragments в inonAttach()](https://dagger.dev/dev-guide/android.html#when-to-inject) .

В отличие от модулей, определенных для Activitys, у вас есть выбор, где установить модули для Fragments. Вы можете сделать свой Fragmentкомпонент подкомпонентом другого Fragmentкомпонента, Activityкомпонента или Applicationкомпонента - все зависит от того, какие другие привязки вам Fragment требуются. После определения местоположения компонента создайте соответствующий тип реализации HasAndroidInjector(если это еще не сделано). Например, если вам Fragment нужны привязки из YourActivitySubcomponent, ваш код будет выглядеть примерно так:

**public** **class** **YourActivity** **extends** **Activity**

**implements** **HasAndroidInjector** **{**

@Inject **DispatchingAndroidInjector<Object>** androidInjector**;**

@Override

**public** **void** **onCreate(Bundle** savedInstanceState**)** **{**

**AndroidInjection.**inject**(this);**

**super.**onCreate**(**savedInstanceState**);**

*// ...*

**}**

@Override

**public** **AndroidInjector<Object>** **androidInjector()** **{**

**return** androidInjector**;**

**}**

**}**

**public** **class** **YourFragment** **extends** **Fragment** **{**

@Inject **SomeDependency** someDep**;**

@Override

**public** **void** **onAttach(Activity** activity**)** **{**

**AndroidInjection.**inject**(this);**

**super.**onAttach**(**activity**);**

*// ...*

**}**

**}**

@Subcomponent**(**modules **=** **...)**

**public** **interface** **YourFragmentSubcomponent** **extends** **AndroidInjector<YourFragment>** **{**

@Subcomponent**.**Factory

**public** **interface** **Factory** **extends** **AndroidInjector.**Factory**<YourFragment>** **{}**

**}**

@Module**(**subcomponents **=** **YourFragmentSubcomponent.**class**)**

**abstract** **class** **YourFragmentModule** **{**

@Binds

@IntoMap

@ClassKey**(YourFragment.**class**)**

**abstract** **AndroidInjector.**Factory**<?>**

bindYourFragmentInjectorFactory**(YourFragmentSubcomponent.**Factory factory**);**

**}**

@Subcomponent**(**modules **=** **{** **YourFragmentModule.**class**,** **...** **}**

**public** **interface** **YourActivityOrYourApplicationComponent** **{** **...** **}**

**Базовые типы каркасов**

Поскольку DispatchingAndroidInjectorво AndroidInjector.Factoryвремя выполнения выполняется поиск соответствующего класса, базовый класс может реализовывать, HasAndroidInjectorа также вызывать AndroidInjection.inject(). Все, что нужно сделать каждому подклассу, - это привязать соответствующий @Subcomponent. Dagger предоставляет несколько базовых типов, которые делают это, например [DaggerActivity](https://dagger.dev/api/latest/dagger/android/DaggerActivity.html)и [DaggerFragment](https://dagger.dev/api/latest/dagger/android/support/DaggerFragment.html), если у вас нет сложной иерархии классов. Dagger также предоставляет [DaggerApplication](https://dagger.dev/api/latest/dagger/android/DaggerApplication.html)для той же цели - все, что вам нужно сделать, это расширить его и переопределить applicationInjector()метод, чтобы вернуть компонент, который должен вводить Application.

Также включены следующие типы:

* [DaggerService](https://dagger.dev/api/latest/dagger/android/DaggerService.html) и [DaggerIntentService](https://dagger.dev/api/latest/dagger/android/DaggerIntentService.html)
* [DaggerBroadcastReceiver](https://dagger.dev/api/latest/dagger/android/DaggerBroadcastReceiver.html)
* [DaggerContentProvider](https://dagger.dev/api/latest/dagger/android/DaggerContentProvider.html)

**Примечание:** [DaggerBroadcastReceiver](https://dagger.dev/api/latest/dagger/android/DaggerBroadcastReceiver.html) следует использовать только тогда, когда BroadcastReceiverдомен зарегистрирован в AndroidManifest.xml. Когда BroadcastReceiverсоздается в вашем собственном коде, предпочтите вместо этого внедрение конструктора.

**Библиотеки поддержки**

Для пользователей библиотеки поддержки Android в dagger.android.supportпакете существуют параллельные типы .

TODO (ronshapiro): мы должны начать разбивать это на пакеты androidx

**Как мне это получить?**

Добавьте в свой build.gradle следующее:

dependencies **{**

compile 'com.google.dagger:dagger-android:2.x'

compile 'com.google.dagger:dagger-android-support:2.x' *// if you use the support libraries*

annotationProcessor 'com.google.dagger:dagger-android-processor:2.x'

annotationProcessor 'com.google.dagger:dagger-compiler:2.x'

**}**

**Когда вводить**

Внедрение конструктора предпочтительнее, когда это возможно, потому javacчто гарантирует, что ни одно поле не будет указано до его установки, что помогает избежать NullPointerExceptions. Когда требуется инъекция членов (как обсуждалось выше), предпочитайте делать инъекцию как можно раньше. По этой причине DaggerActivity вызывается AndroidInjection.inject()немедленно onCreate(), перед вызовом super.onCreate(), и DaggerFragmentделает то же самое onAttach(), что также предотвращает несоответствия при Fragmentповторном подключении.

Крайне важно выполнить вызов AndroidInjection.inject()до super.onCreate()в an Activity, поскольку вызов superприсоединяет Fragments из предыдущего экземпляра действия во время изменения конфигурации, который, в свою очередь, вводит Fragments. Чтобы Fragmentинъекция была успешной, она Activity должна быть уже введена. Для пользователей [ErrorProne](https://github.com/google/error-prone) , это ошибка компилятора звонить AndroidInjection.inject()после super.onCreate().

**часто задаваемые вопросы**

**Определение объема AndroidInjector.Factory**

AndroidInjector.Factoryпредназначен для интерфейса без сохранения состояния, поэтому разработчикам не нужно беспокоиться об управлении состоянием, связанным с объектом, который будет внедрен. Когда DispatchingAndroidInjectorзапрашивает a AndroidInjector.Factory, он делает это через a, Providerпоэтому явно не сохраняет никаких экземпляров фабрики. Поскольку некоторые реализации могут сохранять экземпляр вводимого Activity/ Fragment/ etc, применение области к методам, которые их предоставляют, является ошибкой времени компиляции. Если вы уверены, что AndroidInjector.Factoryне сохраняете экземпляр внедренного объекта, вы можете подавить эту ошибку, применив @SuppressWarnings("dagger.android.ScopedInjectorFactory")метод вашего модуля.