https://dagger.dev/hilt/

**Внедрение зависимости** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Dependency injection*, DI) — процесс предоставления внешней зависимости [программному компоненту](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). Является специфичной формой «[инверсии управления](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)» ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Inversion of control*, IoC), когда она применяется к управлению зависимостями. В полном соответствии с [принципом единственной обязанности](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B5%D0%B4%D0%B8%D0%BD%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BE%D0%B1%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8) объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму[[1]](https://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8#cite_note-_2f16e3a9227f810c-1).

При использовании паттерна "внедрение зависимости" объект пассивен и не предпринимает вообще никаких шагов для выяснения зависимостей, а предоставляет для этого [сеттеры](https://ru.wikipedia.org/wiki/Setter) и/или принимает своим конструктором аргументы, посредством которых внедряются зависимости[[1]](https://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8#cite_note-_2f16e3a9227f810c-1).

Работа фреймворка, обеспечивающая внедрение зависимости, описывается следующим образом. Приложение, независимо от оформления, исполняется внутри контейнера [IoC](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F), предоставляемого фреймворком. Часть объектов в программе по-прежнему создается обычным способом языка программирования, часть создается контейнером на основе предоставленной ему конфигурации.

Условно, если [объекту](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) нужно получить доступ к определенному [сервису](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%B5%D1%80%D0%B2%D0%B8%D1%81_(%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC)&action=edit&redlink=1), объект берет на себя обязанность по доступу к этому сервису: он или получает прямую [ссылку](https://ru.wikipedia.org/wiki/%D0%A1%D1%81%D1%8B%D0%BB%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) на местонахождение сервиса, или обращается к известному [«сервис-локатору»](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%BA%D0%B0%D1%82%D0%BE%D1%80_%D1%81%D0%BB%D1%83%D0%B6%D0%B1) и запрашивает ссылку на реализацию определенного типа сервиса. Используя же внедрение зависимости, объект просто предоставляет свойство, которое в состоянии хранить ссылку на нужный тип сервиса; и когда объект создается, ссылка на реализацию нужного типа сервиса автоматически вставляется в это свойство (поле), используя средства среды.

Внедрение зависимости более гибко, потому что становится легче создавать альтернативные реализации данного типа сервиса, а потом указывать, какая именно реализация должна быть использована в, например, [конфигурационном файле](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB_%D0%BA%D0%BE%D0%BD%D1%84%D0%B8%D0%B3%D1%83%D1%80%D0%B0%D1%86%D0%B8%D0%B8), без изменений в объектах, которые этот сервис используют. Это особенно полезно в [юнит-тестировании](https://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D1%82-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), потому что вставить реализацию «[заглушки](https://ru.wikipedia.org/wiki/Mock-%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82)» сервиса в тестируемый объект очень просто.

С другой стороны, излишнее использование внедрения зависимостей может сделать приложения более сложными и трудными в сопровождении: так как для понимания поведения программы программисту необходимо смотреть не только в исходный код, а еще и в конфигурацию, а конфигурация, как правило, невидима для [IDE](https://ru.wikipedia.org/wiki/%D0%A1%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), которые поддерживают анализ ссылок и [рефакторинг](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3), если явно не указана поддержка [фреймворков](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA) с внедрениями зависимостей.

При использовании внедрения зависимостей, как правило, существует конфигурационный механизм или архитектура, которая определяет целесообразность выбора той или иной реализации в зависимости от поставленных целей.

**<service-point** id="CarBuilderService"**>**

**<invoke-factory>**

**<construct** class="Car"**>**

**<service>**DefaultCarImpl**</service>**

**<service>**DefaultEngineImpl**</service>**

**</construct>**

**</invoke-factory>**

**</service-point>**

*/\*\* Неявная реализация \*\*/*

**public** **class** **MyApplication** {

**public** **static** void main(String[] args) {

Service service = (Service)DependencyManager.get("CarBuilderService");

ICar car = (ICar)service.getService(Car.class);

car.setPedalPressure(5);

float speed = car.getSpeed();

System.out.println("Speed of the car is " + speed);}}

**Внедрение зависимостей в Android**

### Что такое внедрение зависимостей?

Классы часто требуют ссылок на другие классы. Например, Car классу может потребоваться ссылка на Engine класс. Эти обязательные классы называются зависимостями, и в этом примере Car класс зависит от наличия экземпляра Engine класса для запуска.

Жесткая зависимость от Engine затрудняет тестирование. Car использует реальный экземпляр Engine, что предотвращает использование [тестового двойника](https://en.wikipedia.org/wiki/Test_double) для изменения Engine для разных тестовых случаев.

Существуют библиотеки, которые решают эту проблему, автоматизируя процесс создания и предоставления зависимостей. Их можно разделить на две категории:

* Решения на основе отражения, которые связывают зависимости во время выполнения.
* Статические решения, которые генерируют код для подключения зависимостей во время компиляции.

[Dagger](https://dagger.dev/) — это популярная библиотека внедрения зависимостей для Java, Kotlin и Android, поддерживаемая Google. Dagger упрощает использование DI в вашем приложении, создавая и управляя графом зависимостей для вас. Он обеспечивает полностью статические зависимости и зависимости во время компиляции, решая многие проблемы разработки и производительности решений на основе отражения, таких как [Guice](https://en.wikipedia.org/wiki/Google_Guice) .( [программный фреймворк](https://en.wikipedia.org/wiki/Software_framework) с [открытым исходным](https://en.wikipedia.org/wiki/Open-source_software)[кодом](https://en.wikipedia.org/wiki/Software_framework) для [платформы Java,](https://en.wikipedia.org/wiki/Java_platform) выпущенный [Google](https://en.wikipedia.org/wiki/Google) по [лицензии Apache](https://en.wikipedia.org/wiki/Apache_License) . Он обеспечивает поддержку [внедрения зависимостей](https://en.wikipedia.org/wiki/Dependency_injection) с помощью [аннотаций](https://en.wikipedia.org/wiki/Java_annotation) для настройки объектов Java)

## Альтернативы внедрению зависимостей

Альтернативой внедрению зависимостей является использование [локатора служб](https://en.wikipedia.org/wiki/Service_locator_pattern).  Шаблон проектирования локатора сервисов также улучшает отделение классов от конкретных зависимостей. Вы создаете класс, известный как локатор сервисов, который создает и хранит зависимости, а затем предоставляет эти зависимости по запросу.

Набор зависимостей, требуемых локатором сервисов, затрудняет тестирование кода, поскольку все тесты должны взаимодействовать с одним и тем же глобальным локатором сервисов.

**Используйте Hilt в своем приложении для Android**

[Hilt](https://developer.android.com/training/dependency-injection/hilt-android) — это рекомендуемая библиотека Jetpack для внедрения зависимостей в Android. Hilt определяет стандартный способ выполнения DI в вашем приложении, предоставляя контейнеры для каждого класса Android в вашем проекте и автоматически управляя их жизненными циклами.

Hilt построен на основе популярной библиотеки DI [Dagger,](https://developer.android.com/training/dependency-injection/dagger-basics) чтобы извлечь выгоду из корректности времени компиляции, производительности во время выполнения, масштабируемости и поддержки Android Studio, которые предоставляет Dagger.

## **Вывод**

Внедрение зависимостей дает вашему приложению следующие преимущества:

* Возможность повторного использования классов и разделение зависимостей: проще поменять местами реализации зависимости. Повторное использование кода улучшено за счет инверсии управления, и классы больше не контролируют создание своих зависимостей, а вместо этого работают с любой конфигурацией.
* Легкость рефакторинга: зависимости становятся проверяемой частью поверхности API, поэтому их можно проверять во время создания объекта или во время компиляции, а не скрывать как детали реализации.
* Легкость тестирования: класс не управляет своими зависимостями, поэтому, когда вы его тестируете, вы можете передавать разные реализации для тестирования всех ваших различных случаев.

Чтобы полностью понять преимущества внедрения зависимостей, вы должны попробовать его вручную в своем приложении, как показано в [разделе Внедрение зависимостей вручную](https://developer.android.com/training/dependency-injection/manual) .

## **Вывод**

Внедрение зависимостей - хороший метод создания масштабируемых и тестируемых приложений для Android. Используйте контейнеры как способ совместного использования экземпляров классов в разных частях вашего приложения и как централизованное место для создания экземпляров классов с помощью фабрик.

Когда ваше приложение станет больше, вы начнете замечать, что пишете много шаблонного кода (например, фабрики), что может быть подвержено ошибкам. Вы также должны самостоятельно управлять областью действия и жизненным циклом контейнеров, оптимизируя и удаляя контейнеры, которые больше не нужны, чтобы освободить память. Неправильное выполнение этого может привести к незначительным ошибкам и утечкам памяти в вашем приложении.

В разделе [Dagger](https://developer.android.com/training/dependency-injection/dagger-basics) вы узнаете, как с помощью Dagger автоматизировать этот процесс и сгенерировать тот же код, который вы в противном случае написали бы вручную.

# Внедрение зависимостей с помощью Hilt

Hilt предоставляет стандартный способ использования DI в вашем приложении, предоставляя контейнеры для каждого класса Android в вашем проекте и автоматически управляя их жизненными циклами. Hilt построен на основе популярной библиотеки DI [Dagger,](https://developer.android.com/training/dependency-injection/dagger-basics) чтобы извлечь выгоду из корректности времени компиляции, производительности во время выполнения, масштабируемости и [поддержки Android Studio,](https://medium.com/androiddevelopers/dagger-navigation-support-in-android-studio-49aa5d149ec9) которые предоставляет Dagger. Для получения дополнительной информации см. [Рукоять и кинжал](https://developer.android.com/training/dependency-injection/hilt-android#hilt-and-dagger) .

В этом руководстве объясняются основные концепции Hilt и его созданных контейнеров. Он также включает демонстрацию того, как настроить существующее приложение для использования Hilt.

classpath 'com.google.dagger:hilt-android-gradle-plugin:2.28-alpha'

apply plugin: 'kotlin-kapt'  
apply plugin: 'dagger.hilt.android.plugin'  
implementation "com.google.dagger:hilt-android:2.28-alpha"  
kapt "com.google.dagger:hilt-android-compiler:2.28-alpha"

Все приложения, использующие Hilt, должны содержать Application класс с аннотацией @HiltAndroidApp.

## **Внедрить зависимости в классы Android**

После того, как Hilt настроен в вашем Application классе и доступен компонент уровня приложения, Hilt может предоставить зависимости другим классам Android, имеющим @AndroidEntryPoint аннотацию:

В настоящее время Hilt поддерживает следующие классы Android:

* Application(с помощью @HiltAndroidApp)
* Activity
* Fragment
* View
* Service
* BroadcastReceiver

Если вы аннотируете класс Android с помощью @AndroidEntryPoint, вы также должны аннотировать классы Android, которые зависят от него. Например, если вы аннотируете фрагмент, вы также должны аннотировать любые действия, в которых вы используете этот фрагмент.

@AndroidEntryPoint генерирует отдельный компонент Hilt для каждого класса Android в вашем проекте. Эти компоненты могут получать зависимости от своих соответствующих родительских классов, как описано в [разделе «Иерархия компонентов»](https://developer.android.com/training/dependency-injection/hilt-android#component-hierarchy) .

Чтобы получить зависимости от компонента, используйте @Inject аннотацию для внедрения поля:

@AndroidEntryPoint  
class ExampleActivity : AppCompatActivity() {  
  @Inject lateinit var analytics: AnalyticsAdapter  
  ...  
}

class AnalyticsAdapter @Inject constructor(  
  private val service: AnalyticsService  
) { ... }

### Внедрить экземпляры интерфейса с помощью @Binds

### Внедрить экземпляры с помощью @Provides

# Интеграция рукояти и Jetpack

Hilt включает расширения для предоставления классов из других библиотек Jetpack. В настоящее время Hilt поддерживает следующие компоненты Jetpack:

* ViewModel
* WorkManager

## Внедрение объектов ViewModel с помощью Hilt

implementation 'androidx.hilt:hilt-lifecycle-viewmodel:1.0.0-alpha02'  
  // When using Kotlin.  
  kapt 'androidx.hilt:hilt-compiler:1.0.0-alpha02'

class ExampleViewModel @ViewModelInject constructor(  
  private val repository: ExampleRepository,  
  @Assisted private val savedStateHandle: SavedStateHandle  
) : ViewModel() {  
  ...  
}

@AndroidEntryPoint  
class ExampleActivity : AppCompatActivity() {  
  private val exampleViewModel: ExampleViewModel by viewModels()  
  ...  
}

### Интеграция с навигационной библиотекой Jetpack

val viewModel: ExampleViewModel by navGraphViewModels(R.id.my\_graph) {  
  defaultViewModelProviderFactory  
}

## Ввести WorkManager с рукояткой

implementation 'androidx.hilt:hilt-work:1.0.0-alpha01'  
  // When using Kotlin.  
  kapt 'androidx.hilt:hilt-compiler:1.0.0-alpha01'

class ExampleWorker @WorkerInject constructor(  
  @Assisted appContext: Context,  
  @Assisted workerParams: WorkerParameters,  
  workerDependency: WorkerDependency  
) : Worker(appContext, workerParams) { ... }

@HiltAndroidApp  
class ExampleApplication : Application(), Configuration.Provider {  
  @Inject lateinit var workerFactory: HiltWorkerFactory  
  override fun getWorkManagerConfiguration() =  
      Configuration.Builder()  
            .setWorkerFactory(workerFactory)  
            .build()  
}

# Руководство по тестированию рукояти

## Модульные тесты

Hilt не требуется для модульных тестов, поскольку при тестировании класса, использующего внедрение конструктора, вам не нужно использовать Hilt для создания экземпляра этого класса. Вместо этого вы можете напрямую вызвать конструктор класса, передав поддельные или фиктивные зависимости, как если бы конструктор не был аннотирован:

@ActivityScoped  
class AnalyticsAdapter @Inject constructor(  
  private val service: AnalyticsService  
) { ... }  
  
class AnalyticsAdapterTest {  
  @Test  
  fun `Happy path`() {  
    // You don't need Hilt to create an instance of AnalyticsAdapter.  
    // You can pass a fake or mock AnalyticsService.  
    val adapter = AnalyticsAdapter(fakeAnalyticsService)  
    assertEquals(...)  
  }  
}

## Сквозные тесты

Для интеграционных тестов Hilt внедряет зависимости так же, как в ваш производственный код. Тестирование с Hilt не требует обслуживания, поскольку Hilt автоматически генерирует новый набор компонентов для каждого теста.

dependencies {  
    // For Robolectric tests.  
    testImplementation 'com.google.dagger:hilt-android-testing:2.28-alpha'  
    // ...with Kotlin.  
    kaptTest 'com.google.dagger:hilt-android-compiler:2.28-alpha'  
       // For instrumented tests.  
    androidTestImplementation 'com.google.dagger:hilt-android-testing:2.28-alpha'  
    // ...with Kotlin.  
    kaptAndroidTest 'com.google.dagger:hilt-android-compiler:2.28-alpha'  
   }

### Настройка теста пользовательского интерфейса

@HiltAndroidTest  
class SettingsActivityTest {  
  @get:Rule  
  var hiltRule = HiltAndroidRule(this)  
  // UI tests here.  
}

………………………………………………..

**Использование Hilt в вашем приложении для Android**

Внедрение зависимостей — это метод, широко используемый в программировании и хорошо подходящий для разработки под Android. Следуя принципам DI, вы закладываете основу для хорошей архитектуры приложения.

Внедрение внедрения зависимостей дает вам следующие преимущества:

* Возможность повторного использования кода
* Легкость рефакторинга
* Легкость тестирования

Hilt — это самоуверенная библиотека внедрения зависимостей для Android, которая сокращает шаблон использования ручного DI в вашем проекте. Выполнение [инъекции вручную зависимостей](https://developer.android.com/training/dependency-injection/manual) требует построения каждого класса и его зависимости от стороны и использования контейнеров для повторного использования и управления зависимостями.

Hilt предоставляет стандартный способ внедрения DI в ваше приложение, предоставляя контейнеры для каждого компонента Android в вашем проекте и автоматически управляя жизненным циклом контейнера для вас. Это делается с помощью популярной библиотеки DI: [Dagger](https://developer.android.com/training/dependency-injection/dagger-basics) .

## **Предпосылки**

* У вас есть опыт работы с синтаксисом Kotlin.
* Вы понимаете, почему внедрение зависимостей важно в вашем приложении.

Дополнительную информацию о внедрении зависимостей можно найти здесь:

1. [Основы внедрения зависимостей](https://developer.android.com/training/dependency-injection)
2. [Внедрение зависимостей вручную в Android](https://developer.android.com/training/dependency-injection/manual)

## **Что ты узнаешь**

* Как использовать Hilt в приложении для Android.
* Соответствующие концепции Hilt для создания устойчивого приложения.
* Как добавить несколько привязок к одному типу с квалификаторами.
* Как использовать @EntryPoint для доступа к контейнерам из классов, которые Hilt не поддерживает.
* Как использовать модульные и инструментальные тесты для тестирования приложения, использующего Hilt.
* git clone https://github.com/googlecodelabs/android-hilt

В этой кодовой лаборатории вы собираетесь добавить Hilt в приложение, которое **регистрирует взаимодействия пользователя** и использует [Room](https://developer.android.com/topic/libraries/architecture/room) для хранения данных в локальной базе данных.

Как видите, журнал создается и сохраняется каждый раз, когда вы нажимаете одну из пронумерованных кнопок. На экране «Просмотреть **все журналы»** вы увидите список всех предыдущих взаимодействий. Чтобы удалить журналы, нажмите кнопку «Удалить **журналы»**.

## **3. Добавление Hilt в проект**

Если вы посмотрите на начальный код, вы увидите экземпляр ServiceLocator класса, хранящийся в LogApplication классе. ServiceLocator создаёт и хранят зависимости, которые получены по требованию, классы, которые в этом нуждаются. Вы можете думать об этом как о **контейнере** зависимостей, который прикреплен к жизненному циклу приложения, поскольку он будет уничтожен, когда приложение это сделает.

Как объясняется в руководстве [Android DI](https://developer.android.com/training/dependency-injection#choosing-right-di-tool) , локаторы сервисов начинаются с относительно небольшого шаблонного кода, но также плохо масштабируются. Чтобы разработать масштабное приложение для Android, следует использовать Hilt.

Hilt удаляет ненужный шаблон, необходимый для использования ручного DI или шаблона Service Locator в приложении Android, путем генерации кода, который вы бы создали вручную (например, кода в ServiceLocator классе).

Узнайте больше о [шаблоне Service Locator](https://developer.android.com/training/dependency-injection) в нашем руководстве по внедрению зависимостей на Android.

На следующих шагах вы будете использовать Hilt для замены ServiceLocator класса. После этого мы добавим в проект новые функции, чтобы изучить дополнительные возможности Hilt.

buildscript {  
    ...  
    ext.hilt\_version = '2.30.1-alpha' // (24 ноября 2020 года)  
    dependencies {  
        ...  
        classpath "com.google.dagger:hilt-android-gradle-plugin:$hilt\_version"  
    }  
}

Затем, чтобы использовать плагин gradle в app модуле, мы указываем его в app/build.gradle файле, добавляя плагин в начало файла, ниже kotlin-kapt плагина:

...  
apply plugin: 'kotlin-kapt'  
apply plugin: 'dagger.hilt.android.plugin'  
android {  
    ...  
}

Наконец, зависимости Hilt включены в наш проект в том же app/build.gradle файле:

...  
dependencies {  
    ...  
    implementation "com.google.dagger:hilt-android:$hilt\_version"  
    kapt "com.google.dagger:hilt-android-compiler:$hilt\_version"  
}

## **4. Рукоятка в вашем приложении**

Аналогично тому, как экземпляр ServiceLocator в LogApplication классе используется и инициализируется, чтобы добавить ***контейнер*, прикрепленный к жизненному циклу приложения**, нам нужно аннотировать Application класс с помощью @HiltAndroidApp. Откройте LogApplication.kt и добавьте аннотацию к классу:

@HiltAndroidApp  
class LogApplication : Application() {  
    ...  
}

@HiltAndroidApp запускает генерацию кода Hilt, включая базовый класс для вашего приложения, которое может использовать внедрение зависимостей. Контейнер приложения является родительским контейнером приложения, что означает, что другие контейнеры могут получить доступ к зависимостям, которые он предоставляет.

Теперь наше приложение готово к использованию Hilt!

## **5. Полевая инъекция с рукоятью**

Вместо того, ServiceLocator чтобы получать зависимости по запросу из наших классов, мы будем использовать Hilt, чтобы предоставить нам эти зависимости. Приступим к замене вызовов ServiceLocator из наших классов.

Откройте ui/LogsFragment.kt файл. LogsFragment заполняет свои поля в onAttach. Вместо того, чтобы заполнять экземпляры LoggerLocalDataSourceи DateFormatter использовать вручную ServiceLocator, мы можем использовать Hilt для создания экземпляров этих типов и управления ими.

Чтобы LogsFragment использовать Hilt, мы должны аннотировать его @AndroidEntryPoint:

@AndroidEntryPoint  
class LogsFragment: Fragment() {  
    ...  
}

Аннотирование классов Android с помощью @AndroidEntryPoint создает контейнер зависимостей, который следует жизненному циклу класса Android.

Рукоять в настоящее время поддерживает следующие типы Android: **Application** (с помощью **@HiltAndroidApp**), **Activity**, **Fragment**, **View**, **Service**и **BroadcastReceiver**.

Hilt поддерживает только действия, которые расширяют [**FragmentActivity**](https://developer.android.com/reference/androidx/fragment/app/FragmentActivity) (например [**AppCompatActivity**](https://developer.android.com/reference/kotlin/androidx/appcompat/app/AppCompatActivity)), и фрагменты, которые расширяют библиотеку Jetpack **Fragment**, а не (теперь не рекомендуется) **Fragment** из платформы Android.

Предупреждение: рукоять не поддерживает сохраненные фрагменты.

С помощью @AndroidEntryPoint Hilt будет создан контейнер зависимостей, который присоединен к LogsFragment жизненному циклу и сможет внедрять экземпляры в LogsFragment. Как мы можем получить поля, введенные Hilt?

Мы можем **заставить Hilt вводить экземпляры разных типов с**@Inject **аннотацией** к полям, которые мы хотим **внедрить** (т.е. loggerи dateFormatter):

@AndroidEntryPoint  
class LogsFragment : Fragment() {  
  
    @Inject lateinit var logger: LoggerLocalDataSource  
    @Inject lateinit var dateFormatter: DateFormatter  
  
    ...  
}

Это то, что называется **полевой инъекцией**.

Чтобы выполнить внедрение поля, используйте **@Inject** аннотацию к полям классов Android, которые должны быть предоставлены (или введены) Hilt.

Предупреждение: поля, введенные Hilt, не могут быть частными.

Поскольку Hilt будет отвечать за заполнение этих полей за нас, этот populateFields метод нам больше не нужен. Удалим метод из класса:

@AndroidEntryPoint  
class LogsFragment : Fragment() {  
    // Remove following code from LogsFragment  
    override fun onAttach(context: Context) {  
        super.onAttach(context)  
        populateFields(context)  
    }  
    private fun populateFields(context: Context) {  
        logger = (context.applicationContext as LogApplication).serviceLocator.loggerLocalDataSource  
        dateFormatter =  
            (context.applicationContext as LogApplication).serviceLocator.provideDateFormatter()  
    }  
    ...  
}

Под капотом Hilt заполнит эти поля в onAttach()методе жизненного цикла экземплярами, встроенными в автоматически созданный LogsFragment контейнер зависимостей.

Дополнительную информацию о том, в какой обратный вызов жизненного цикла внедряется класс Android, можно найти в документации: [раздел «Время жизни компонентов»](https://developer.android.com/training/dependency-injection/hilt-android#component-lifetimes) .

Чтобы выполнить внедрение поля, Hilt необходимо знать, как предоставить экземпляры этих зависимостей! В этом случае Hilt необходимо знать, как предоставить экземпляры LoggerLocalDataSource и DateFormatter. Однако Hilt еще не знает, как предоставить эти экземпляры.

## **Сообщите Hilt, как предоставлять зависимости с помощью @Inject**

Откройте ServiceLocator.kt файл, чтобы увидеть, как ServiceLocator это реализовано. Вы можете видеть, как при вызове provideDateFormatter() всегда возвращается другой экземпляр DateFormatter.

Это точно такое же поведение, которого мы хотим добиться с Hilt. К счастью, DateFormatter не зависит от других классов, поэтому нам пока не нужно беспокоиться о транзитивных зависимостях.

**Чтобы сообщить Hilt, как предоставлять экземпляры типа, добавьте аннотацию @Inject к конструктору класса, который вы хотите внедрить**.

Откройте util/DateFormatter.kt файл и аннотируйте DateFormatter конструктор с помощью @Inject. Помните, что для аннотирования конструктора в Kotlin вам также понадобится constructor ключевое слово:

class DateFormatter @Inject constructor() { ... }

Благодаря этому Hilt знает, как предоставить экземпляры DateFormatter. То же самое нужно сделать с LoggerLocalDataSource. Откройте data/LoggerLocalDataSource.kt файл и аннотируйте его конструктор @Inject:

class LoggerLocalDataSource @Inject constructor(private val logDao: LogDao) {  
    ...  
}

Информация Hilt о том, как предоставлять экземпляры разных типов, также называется **привязками** .

На данный момент у Hilt есть две привязки: как предоставить экземпляры 1) **DateFormatter** и 2) **LoggerLocalDataSource**.

Если мы ServiceLocator снова откроем класс, вы увидите, что у нас есть публичное LoggerLocalDataSource поле. Это означает, что ServiceLocator всегда будет возвращать один и тот же экземпляр LoggerLocalDataSource всякий раз, когда он вызывается. Это то, что называется «**привязкой экземпляра к контейнеру**». Как мы можем сделать это в Hilt?

## **6. Копирование экземпляров в контейнеры**

Мы можем использовать аннотации для привязки экземпляров к контейнерам. Поскольку Hilt может создавать разные контейнеры с разным жизненным циклом, для этих контейнеров существуют разные аннотации.

Аннотация, которая связывает экземпляр с контейнером приложения, имеет вид @Singleton. Эта аннотация заставит контейнер приложения всегда предоставлять один и тот же экземпляр, независимо от того, используется ли тип как зависимость другого типа или его нужно вводить в поле.

Эту же логику можно применить ко всем контейнерам, прикрепленным к классам Android. Вы можете найти список всех аннотаций в [документации](https://developer.android.com/training/dependency-injection/hilt-android#component-bindings) . Например, если вы хотите, чтобы контейнер действий всегда предоставлял один и тот же экземпляр типа, вы можете аннотировать этот тип с помощью @ ActivityScoped.

Как упоминалось выше, поскольку мы хотим, чтобы контейнер приложения всегда предоставлял один и тот же экземпляр LoggerLocalDataSource, мы аннотируем его класс с помощью @Singleton:

@Singleton  
class LoggerLocalDataSource @Inject constructor(private val logDao: LogDao) {  
    ...  
}

**Привязки, доступные в контейнерах на более высоком уровне иерархии, также доступны на более низких уровнях иерархии**. Следовательно, если экземпляр **LoggerLocalDataSource** доступен в контейнере приложения, он также будет доступен в контейнерах активности и фрагментов.

Теперь Hilt знает, как предоставлять экземпляры LoggerLocalDataSource. Однако на этот раз тип имеет транзитивные зависимости! Чтобы предоставить экземпляр LoggerLocalDataSource, Hilt также необходимо знать, как предоставить экземпляр LogDao.

Однако, поскольку LogDao это интерфейс, мы не можем аннотировать его конструктор, так @Inject как интерфейсы его не имеют. Как мы можем сказать Hilt, как предоставлять экземпляры этого типа?

## **7. Модули рукояти.**

**Модули используются для добавления привязок к Hilt**, или, другими словами, чтобы сообщить Hilt, как предоставлять экземпляры различных типов. В модули Hilt вы включаете привязки для **типов, которые не могут быть введены конструктором,** таких как интерфейсы или классы, не содержащиеся в вашем проекте. Пример этого OkHttpClient вам нужно использовать его построитель для создания экземпляра.

**Модуль Hilt — это класс, помеченный значками**@Module **и** @InstallIn. @Module сообщает Hilt, что это модуль, и @InstallIn сообщает Hilt, в каких контейнерах доступны привязки, путем указания компонента Hilt. Вы можете думать о Hilt Component как о контейнере, и полный список компонентов можно найти [здесь](https://developer.android.com/training/dependency-injection/hilt-android#generated-components) .

|  |  |
| --- | --- |
| **Hilt component** | **Injector for** |
| ApplicationComponent | Application |
| ActivityRetainedComponent | ViewModel |
| ActivityComponent | Activity |
| FragmentComponent | Fragment |
| ViewComponent | View |
| ViewWithFragmentComponent | View annotated with @WithFragmentBindings |
| ServiceComponent | Service |

**Для каждого класса Android, который может быть внедрен Hilt, существует связанный компонент Hilt**. Например, Application контейнер связан с ApplicationComponent, а Fragment контейнер связан с FragmentComponent.

## **Создание модуля**

Давайте создадим модуль Hilt, в который мы сможем добавлять привязки. Создайте новый пакет с именем di в hilt пакете и создайте новый файл с именем DatabaseModule.kt внутри пакета.

Поскольку LoggerLocalDataSource область действия ограничена контейнером приложения, LogDao привязка должна быть доступна в контейнере приложения. Мы указываем это требование с помощью @InstallIn аннотации, передавая класс компонента Hilt, связанный с ним (т.е. ApplicationComponent:class):

package com.example.android.hilt.di  
  
@InstallIn(ApplicationComponent::class)  
@Module  
object DatabaseModule {  
  
}

В ServiceLocator реализации класса экземпляр LogDao получается путем вызова logsDatabase.logDao(). Следовательно, чтобы предоставить экземпляр LogDao, у нас есть транзитивная зависимость от AppDatabase класса.

В Kotlin модули, содержащие только **@Provides** функции, могут быть **object** классами. Таким образом, провайдеры оптимизируются и почти встраиваются в сгенерированный код.

## **Предоставление экземпляров с помощью @Provides**

Мы можем аннотировать функцию с помощью @Provides модулей в Hilt, чтобы сообщить Hilt, как предоставлять типы, которые не могут быть введены конструктором.

Тело @Provides аннотированной функции будет выполняться каждый раз, когда Hilt необходимо предоставить экземпляр этого типа. Тип возврата @Provides аннотированной функции сообщает Hilt тип привязки или способ предоставления экземпляров этого типа. Параметры функции — это зависимости типа.

В нашем случае мы включим эту функцию в DatabaseModule класс:

@Module  
object DatabaseModule {  
  
    @Provides  
    fun provideLogDao(database: AppDatabase): LogDao {  
        return database.logDao()  
    }  
}

Приведенный выше код сообщает Hilt, что database.logDao()необходимо выполнить при предоставлении экземпляра LogDao. Поскольку у нас AppDatabase есть транзитивная зависимость, нам также нужно сообщить Hilt, как предоставлять экземпляры этого типа.

Поскольку AppDatabase это еще один класс, которым наш проект не владеет, потому что он сгенерирован Room, мы также можем предоставить его, используя @Provides функцию, аналогичную тому, как мы создаем экземпляр базы данных в ServiceLocator классе:

@Module  
object DatabaseModule {  
  
    @Provides  
    @Singleton  
    fun provideDatabase(@ApplicationContext appContext: Context): AppDatabase {  
        return Room.databaseBuilder(  
            appContext,  
            AppDatabase::class.java,  
            "logging.db"  
        ).build()  
    }  
  
    @Provides  
    fun provideLogDao(database: AppDatabase): LogDao {  
        return database.logDao()  
    }  
}

Поскольку мы всегда хотим, чтобы Hilt предоставлял один и тот же экземпляр базы данных, мы аннотируем @Provides provideDatabase метод с помощью @Singleton.

Каждый контейнер Hilt поставляется с набором привязок по умолчанию, которые можно внедрить в качестве зависимостей в ваши пользовательские привязки. Это случай applicationContext: чтобы получить к нему доступ, вам необходимо аннотировать поле с помощью @ApplicationContext.

Вы можете увидеть список предопределенных привязок на этой [странице документации](https://developer.android.com/training/dependency-injection/hilt-android#component-default) .

## **Запуск приложения**

Теперь у Hilt есть вся необходимая информация для внедрения экземпляров LogsFragment. Тем не менее, перед запуском приложения, Эфес должен быть в курсе того, Activity что Саваофа Fragment для работы. Нам нужно аннотировать MainActivity с помощью @AndroidEntryPoint.

Открыть ui/MainActivity.kt файл и аннотирования MainActivity с @AndroidEntryPoint:

@AndroidEntryPoint  
class MainActivity : AppCompatActivity() { ... }

Теперь вы можете запустить приложение и убедиться, что все работает нормально, как и раньше.

Давайте продолжим рефакторинг приложения, чтобы удалить ServiceLocator вызовы из MainActivity.

## **8. Обеспечение интерфейсов с @Binds**

MainActivity получает экземпляр AppNavigator от ServiceLocatorвызова provideNavigator(activity: FragmentActivity)функции.

Поскольку AppNavigator это интерфейс, мы не можем использовать внедрение конструктора. **Чтобы сообщить Hilt, какую реализацию использовать для интерфейса, вы можете использовать**@Binds **аннотацию к функции внутри модуля Hilt**.

@Binds должен аннотировать абстрактную функцию (поскольку она абстрактная, она не содержит кода, и класс тоже должен быть абстрактным). Тип возврата абстрактной функции — это интерфейс, для которого мы хотим реализовать (т.е. AppNavigator). Реализация указывается путем добавления уникального параметра с типом реализации интерфейса (т.е. AppNavigatorImpl).

Можем ли мы добавить информацию в DatabaseModule созданный ранее класс или нам нужен новый модуль? Есть несколько причин, по которым мы должны создать новый модуль:

* Для лучшей организации имя модуля должно отражать тип предоставляемой информации. Например, не имеет смысла включать привязки навигации в модуль с именем DatabaseModule.
* DatabaseModule Модуль установлен в ApplicationComponent, так что переплеты доступны в прикладном контейнере. Наша новая навигационная информация (т.е. AppNavigator) нуждается в информации, специфичной от деятельности (как AppNavigatorImpl имеет Activity как зависимость). Следовательно, **он должен быть установлен в**Activity **контейнере,** а не в Application контейнере, поскольку именно там Activity доступна информация о.
* Модули Hilt не могут содержать как нестатические, так и абстрактные методы привязки, поэтому вы не можете размещать аннотации @Bindsи @Provides аннотации в одном классе.

Создайте новый файл с именем NavigationModule.kt в di папке. Здесь давайте создадим новый абстрактный класс NavigationModule с аннотациями @Module и, @InstallIn(ActivityComponent::class)как объяснено выше:

@InstallIn(ActivityComponent::class)  
@Module  
abstract class NavigationModule {  
  
    @Binds  
    abstract fun bindNavigator(impl: AppNavigatorImpl): AppNavigator  
}

Внутри модуля мы можем добавить привязку для AppNavigator. Это абстрактная функция, которая возвращает интерфейс, о котором мы информируем Hilt (т.е. AppNavigator), а параметр - реализация этого интерфейса (т.е. AppNavigatorImpl).

Теперь мы должны сообщить Hilt, как предоставлять экземпляры AppNavigatorImpl. Поскольку этот класс может быть внедрен в конструктор, мы просто аннотируем его конструктор @Inject.

Откройте navigator/AppNavigatorImpl.kt файл и сделайте это:

class AppNavigatorImpl @Inject constructor(  
    private val activity: FragmentActivity  
) : AppNavigator {  
    ...  
}

AppNavigatorImpl зависит от FragmentActivity. Поскольку AppNavigator экземпляр предоставляется в Activityконтейнере (он также доступен в Fragmentконтейнере и в Viewконтейнере, поскольку NavigationModule он установлен в ActivityComponent), FragmentActivity он уже доступен, поскольку он поставляется как [предопределенная привязка](https://developer.android.com/training/dependency-injection/hilt-android#component-default) .

## **Использование рукояти в упражнении**

Теперь у Hilt есть вся информация, чтобы внедрить AppNavigator экземпляр. Откройте MainActivity.kt файл и сделайте следующее:

1. Аннотировать navigator поле с @Inject помощью Hilt,
2. Удалите private модификатор видимости и
3. Удалите navigator код инициализации в onCreate функции.

Новый код должен выглядеть так:

@AndroidEntryPoint  
class MainActivity : AppCompatActivity() {  
  
    @Inject lateinit var navigator: AppNavigator  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity\_main)  
          
        if (savedInstanceState == null) {  
            navigator.navigateTo(Screens.BUTTONS)  
        }  
    }  
  
    ...  
}

## **Запуск приложения**

Вы можете запустить приложение и посмотреть, как оно работает.

## **Завершение рефакторинга**

Единственный класс, который все еще использует ServiceLocator для получения зависимостей, — это ButtonsFragment. Поскольку Hilt уже знает, как предоставить все необходимые типы ButtonsFragment, мы можем просто выполнить внедрение поля в класс.

Как мы узнали ранее для того, чтобы класс был внедрен Hilt в поле, мы должны:

1. Аннотируйте с ButtonsFragment помощью @AndroidEntryPoint,
2. Удалить модификатор из частного logger и navigator полей и аннотировать их @Inject,
3. Удалить код инициализации полей (т.е. onAttachи populateFields методы).

Код для ButtonsFragment:

@AndroidEntryPoint  
class ButtonsFragment : Fragment() {  
  
    @Inject lateinit var logger: LoggerLocalDataSource  
    @Inject lateinit var navigator: AppNavigator  
  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? {  
        return inflater.inflate(R.layout.fragment\_buttons, container, false)  
    }  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
        ...  
    }  
}

Обратите внимание, что экземпляр LoggerLocalDataSource будет таким же, как и тот, который мы использовали, LogsFragment поскольку тип ограничен контейнером приложения. Однако экземпляр AppNavigator будет отличаться от экземпляра, MainActivity поскольку мы не ограничили его соответствующим Activity контейнером.

На этом этапе ServiceLocator класс больше не предоставляет зависимости, поэтому мы можем полностью удалить его из проекта. Единственное использование остается в LogApplication классе, в котором мы сохранили его экземпляр. Давайте очистим этот класс, поскольку он нам больше не нужен.

Откройте LogApplication класс и удалите ServiceLocator использование. Новый код Application класса:

@HiltAndroidApp  
class LogApplication : Application()

Теперь вы можете полностью удалить ServiceLocator класс из проекта. Поскольку ServiceLocator он все еще используется в тестах, удалите его использования из AppTest класса.

## **Охвачено базовое содержание**

**То, что вы только что узнали, должно быть достаточно, чтобы использовать Hilt в качестве инструмента внедрения зависимостей в вашем приложении Android.**

С этого момента мы добавим новые функции в наше приложение, чтобы узнать, как использовать более продвинутые функции Hilt в различных ситуациях.

## **9. Отборочные.**

Теперь, когда мы удалили ServiceLocator класс из нашего проекта и вы узнали основы Hilt, давайте добавим в приложение новые функции для изучения других функций Hilt.

**В этом разделе вы узнаете о:**

* Как перейти к контейнеру Activity.
* Что такое **квалификаторы**, какие задачи они решают и как их использовать.

Чтобы показать это, нам нужно другое поведение в нашем приложении. Мы заменим хранилище журналов из базы данных на список в памяти с намерением записывать журналы только во время сеанса приложения.

## **Интерфейс LoggerDataSource**

Начнем абстрагировать источник данных в интерфейс. Создайте новый файл с именем LoggerDataSource.kt в data папке со следующим содержимым:

package com.example.android.hilt.data  
  
// Common interface for Logger data sources.  
interface LoggerDataSource {  
    fun addLog(msg: String)  
    fun getAllLogs(callback: (List<Log>) -> Unit)  
    fun removeLogs()  
}

LoggerLocalDataSource используется в обоих фрагментах: ButtonsFragment и LogsFragment. Нам нужно реорганизовать их, чтобы использовать LoggerDataSource вместо них экземпляр.

Откройте LogsFragment и сделайте переменную регистратора типа LoggerDataSource:

@AndroidEntryPoint  
class LogsFragment : Fragment() {  
  
    @Inject lateinit var logger: LoggerDataSource  
    ...  
}

Сделайте то же самое в ButtonsFragment:

@AndroidEntryPoint  
class ButtonsFragment : Fragment() {  
  
    @Inject lateinit var logger: LoggerDataSource  
    ...  
}

Теперь давайте LoggerLocalDataSource реализуем этот интерфейс. Откройте data/LoggerLocalDataSource.kt файл и:

1. Сделайте так, чтобы он реализовал LoggerDataSource интерфейс, и
2. Отметьте его методы с помощью override

@Singleton  
class LoggerLocalDataSource @Inject constructor(  
    private val logDao: LogDao  
) : LoggerDataSource {  
    ...  
    override fun addLog(msg: String) { ... }  
    override fun getAllLogs(callback: (List<Log>) -> Unit) { ... }  
    override fun removeLogs() { ... }  
}

Теперь давайте создадим еще одну реализацию LoggerDataSource вызова, LoggerInMemoryDataSource которая хранит журналы в памяти. Создайте новый файл с именем LoggerInMemoryDataSource.kt в data папке со следующим содержимым:

package com.example.android.hilt.data  
  
import java.util.LinkedList  
  
class LoggerInMemoryDataSource : LoggerDataSource {  
  
    private val logs = LinkedList<Log>()  
  
    override fun addLog(msg: String) {  
        logs.addFirst(Log(msg, System.currentTimeMillis()))  
    }  
  
    override fun getAllLogs(callback: (List<Log>) -> Unit) {  
        callback(logs)  
    }  
  
    override fun removeLogs() {  
        logs.clear()  
    }  
}

## **Переход к контейнеру Activity**

Чтобы использовать его LoggerInMemoryDataSource в качестве детали реализации, нам нужно сообщить Hilt, как предоставлять экземпляры этого типа. Как и раньше, мы аннотируем конструктор класса @Inject:

class LoggerInMemoryDataSource @Inject constructor(  
) : LoggerDataSource { ... }

Поскольку наше приложение состоит только из одного Activity (также называемого приложением single-Activity ), мы должны иметь экземпляр класса LoggerInMemoryDataSourceв Activity контейнере и повторно использовать этот экземпляр в Fragments.

Мы можем добиться ведения журнала в памяти, привязав его LoggerInMemoryDataSourceк Activity контейнеру: каждый Activity созданный будет иметь свой собственный контейнер, другой экземпляр. В каждом контейнере LoggerInMemoryDataSource будет предоставлен один и тот же экземпляр, когда регистратор потребуется в качестве зависимости или для внедрения поля. Кроме того, тот же экземпляр будет предоставлен в контейнерах ниже [иерархии компонентов](https://developer.android.com/training/dependency-injection/hilt-android#component-hierarchy) .

После [определения области видимости в документации по компонентам](https://developer.android.com/training/dependency-injection/hilt-android#component-scopes) , чтобы охватить тип Activity контейнером, нам необходимо аннотировать тип с помощью @ActivityScoped:

@ActivityScoped  
class LoggerInMemoryDataSource @Inject constructor(  
) : LoggerDataSource { ... }

На данный момент, Эфес знает, как обеспечить экземпляры LoggerInMemoryDataSource и , LoggerLocalDataSource но как насчет LoggerDataSource? Hilt не знает, какую реализацию использовать, когда требуется LoggerDataSource.

Как мы знаем из предыдущих разделов, мы можем использовать @Binds аннотацию в модуле, чтобы сообщить Hilt, какую реализацию использовать. Однако, **что, если нам нужно предоставить обе реализации в одном проекте?**Например, при использовании LoggerInMemoryDataSource во время работы приложения и LoggerLocalDataSource в Service.

## **Две реализации для одного интерфейса**

Создадим новый файл в di папке с именем LoggingModule.kt. Поскольку различные реализации LoggerDataSource находятся в области видимости в различные контейнеры, мы не можем использовать один и тот же модуль: LoggerInMemoryDataSource в области видимости в Activity контейнер и LoggerLocalDataSource в Application контейнер.

К счастью, мы можем определить привязки для обоих модулей в том же файле, который мы только что создали:

package com.example.android.hilt.di  
  
@InstallIn(ApplicationComponent::class)  
@Module  
abstract class LoggingDatabaseModule {  
  
    @Singleton  
    @Binds  
    abstract fun bindDatabaseLogger(impl: LoggerLocalDataSource): LoggerDataSource  
}  
  
@InstallIn(ActivityComponent::class)  
@Module  
abstract class LoggingInMemoryModule {  
  
    @ActivityScoped  
    @Binds  
    abstract fun bindInMemoryLogger(impl: LoggerInMemoryDataSource): LoggerDataSource  
}

@Binds **методы должны иметь аннотации области видимости, если тип имеет область видимости**, поэтому приведенные выше функции помечены значками @Singletonи @ActivityScoped. Если @Bindsили @Providesиспользуются в качестве привязки для типа, аннотации области действия в типе больше не используются, поэтому вы можете продолжить и удалить их из различных классов реализации.

Если вы попытаетесь собрать проект сейчас, вы увидите DuplicateBindings ошибку!

error: [Dagger/DuplicateBindings] com.example.android.hilt.data.LoggerDataSource is bound multiple times

Это связано с тем, что LoggerDataSource тип вводится в наши Fragments, но **Hilt не знает, какую реализацию использовать, потому что есть две привязки одного типа!**Как Hilt узнать, какой из них использовать?

## **Использование квалификаторов**

**Чтобы указать Hilt, как предоставлять разные реализации (несколько привязок) одного и того же типа, вы можете использовать квалификаторы** \*\*. \*\*

Квалификатор - это аннотация, используемая для идентификации привязки.

Нам необходимо определить квалификатор для каждой реализации, поскольку каждый квалификатор будет использоваться для идентификации привязки. При внедрении типа в класс Android или наличии этого типа как зависимости от других классов необходимо использовать аннотацию квалификатора, чтобы избежать двусмысленности.

Поскольку квалификатор — это просто аннотация, мы можем определить их в LoggingModule.kt файле, в который мы добавили модули:

package com.example.android.hilt.di  
  
@Qualifier  
annotation class InMemoryLogger  
  
@Qualifier  
annotation class DatabaseLogger

Теперь эти квалификаторы должны аннотировать @Binds(или, @Provides если нам это нужно) функции, которые обеспечивают каждую реализацию. Посмотрите полный код и обратите внимание на использование квалификаторов в @Binds методах:

package com.example.android.hilt.di  
  
@Qualifier  
annotation class InMemoryLogger  
  
@Qualifier  
annotation class DatabaseLogger  
  
@InstallIn(ApplicationComponent::class)  
@Module  
abstract class LoggingDatabaseModule {  
  
    @DatabaseLogger  
    @Singleton  
    @Binds  
    abstract fun bindDatabaseLogger(impl: LoggerLocalDataSource): LoggerDataSource  
}  
  
@InstallIn(ActivityComponent::class)  
@Module  
abstract class LoggingInMemoryModule {  
  
    @InMemoryLogger  
    @ActivityScoped  
    @Binds  
    abstract fun bindInMemoryLogger(impl: LoggerInMemoryDataSource): LoggerDataSource  
}

Кроме того, эти квалификаторы должны использоваться в точке внедрения с реализацией, которую мы хотим внедрить. В этом случае мы собираемся использовать LoggerInMemoryDataSource реализацию в нашем Fragments.

**Важно:** поскольку **@DatabaseLogger** квалификатор установлен **ApplicationComponent**, он может быть введен в **LogApplication** класс. Однако, поскольку **@InMemoryLogger** он установлен **ActivityComponent**, его нельзя внедрить в **LogApplication** класс, потому что контейнер приложения не знает об этой привязке.

Откройте LogsFragment и используйте @InMemoryLogger квалификатор в поле регистратора, чтобы сообщить Hilt, что нужно ввести экземпляр LoggerInMemoryDataSource:

@AndroidEntryPoint  
class LogsFragment : Fragment() {  
  
    @InMemoryLogger  
    @Inject lateinit var logger: LoggerDataSource  
    ...  
}

Сделайте то же самое для ButtonsFragment:

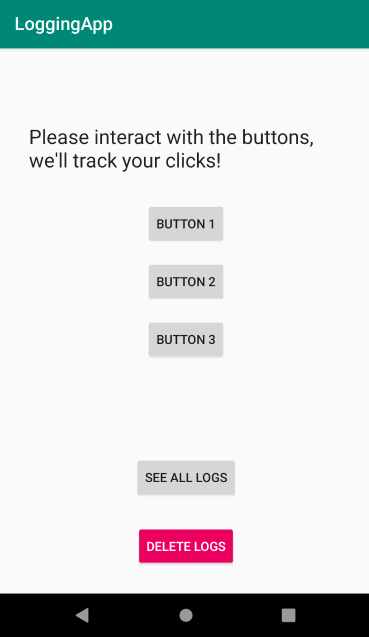
@AndroidEntryPoint  
class ButtonsFragment : Fragment() {  
  
    @InMemoryLogger  
    @Inject lateinit var logger: LoggerDataSource  
    ...  
}

Если вы хотите изменить реализацию базы данных, которую хотите использовать, вам просто нужно аннотировать введенные поля @DatabaseLogger вместо @InMemoryLogger.

## **Запуск приложения**

Мы можем запустить приложение и подтвердить, что мы сделали, взаимодействуя с кнопками и наблюдая, как соответствующие журналы появляются на экране «Просмотреть все журналы».

Обратите внимание, что журналы больше не сохраняются в базе данных. Они не сохраняются между сеансами, когда вы закрываете и снова открываете приложение, экран журнала пуст.



10. Тестирование пользовательского интерфейса

Теперь, когда приложение полностью перенесено на Hilt, мы также можем перенести инструментальный тест, который у нас есть в проекте. Тест, проверяющий функциональность приложения, находится в AppTest.kt файле в app/androidTest папке. Открой это!

Вы увидите, что он не компилируется, потому что мы удалили ServiceLocator класс из нашего проекта. Удалите ссылки на то, ServiceLocator что мы больше не используем, удалив @After tearDown метод из класса.

androitTest тесты запускаются на эмуляторе. В happyPath тесте подтверждает, что кран на «Button 1» был внесен в базу данных. Поскольку приложение использует базу данных в памяти, после завершения теста все журналы исчезнут.

Тестирование пользовательского интерфейса с Hilt

Hilt будет внедрять зависимости в ваш UI-тест, как это происходит в вашем производственном коде.

**Тестирование с помощью Hilt не требует обслуживания, поскольку Hilt автоматически создает новый набор компонентов для каждого теста** .

Добавление тестовых зависимостей

Hilt использует дополнительную библиотеку со специфическими для тестирования аннотациями, которые упрощают тестирование вашего кода, называемые, hilt-android-testing которые необходимо добавить в проект. Кроме того, поскольку Hilt необходимо сгенерировать код для классов в androidTest папке, его обработчик аннотаций также должен иметь возможность работать там. Чтобы включить это, вам нужно включить в app/build.gradle файл две зависимости .

Чтобы добавить эти зависимости, откройте app/build.gradle и добавьте эту конфигурацию в конец dependencies раздела:

...  
dependencies {  
  
    // Hilt testing dependency  
    androidTestImplementation "com.google.dagger:hilt-android-testing:$hilt\_version"  
    // Make Hilt generate code in the androidTest folder  
    kaptAndroidTest "com.google.dagger:hilt-android-compiler:$hilt\_version"  
}

Пользовательский TestRunner

Инструментальные тесты с использованием Hilt необходимо выполнять в среде Application, поддерживающей Hilt. Уже поставляется библиотека, HiltTestApplication которую мы можем использовать для запуска наших тестов пользовательского интерфейса. Указание Application для использования в тестах выполняется путем создания нового средства выполнения тестов в проекте.

На том же уровне AppTest.kt файл находится в androidTest папке, создайте новый файл с именем CustomTestRunner. Наш CustomTestRunner расширяется от [AndroidJUnitRunner](https://developer.android.com/training/testing/junit-runner) и реализован следующим образом:

class CustomTestRunner : AndroidJUnitRunner() {  
  
    override fun newApplication(cl: ClassLoader?, name: String?, context: Context?): Application {  
        return super.newApplication(cl, HiltTestApplication::class.java.name, context)  
    }  
}

Затем нам нужно указать проекту, чтобы он использовал этот бегун тестов для инструментальных тестов. Это указано в testInstrumentationRunner атрибуте app/build.gradle файла. Откройте файл и замените testInstrumentationRunner содержимое по умолчанию следующим:

...  
android {  
    ...  
    defaultConfig {  
        ...  
        testInstrumentationRunner "com.example.android.hilt.CustomTestRunner"  
    }  
    ...  
}  
...

Теперь мы готовы использовать Hilt в наших UI-тестах!

Запуск теста с использованием Hilt

Затем, чтобы тестовый класс эмулятора мог использовать Hilt, он должен:

1. Аннотировать того, @HiltAndroidTest кто отвечает за создание компонентов Hilt для каждого теста.
2. Используйте, HiltAndroidRule который управляет состоянием компонентов и используется для выполнения инъекции в ваш тест.

Включим их в AppTest:

@RunWith(AndroidJUnit4::class)  
@HiltAndroidTest  
class AppTest {  
  
    @get:Rule  
    var hiltRule = HiltAndroidRule(this)  
  
    ...  
}

Теперь, если вы запустите тест, используя кнопку воспроизведения рядом с определением класса или определением метода тестирования, эмулятор запустится, если он у вас настроен, и тест пройдет.

Чтобы узнать больше о тестировании и таких функциях, как внедрение поля или замена привязок в тестах, ознакомьтесь [с документацией](https://developer.android.com/training/dependency-injection/hilt-testing) .

11. Аннотация @EntryPoint

В этом разделе кодовой лаборатории мы узнаем, как использовать @EntryPoint аннотацию, которая используется для **внедрения зависимостей в классы, не поддерживаемые Hilt** .

Как мы видели ранее, Hilt поддерживает наиболее распространенные компоненты Android. Однако вам может потребоваться выполнить внедрение поля в классах, которые либо не поддерживаются непосредственно Hilt, либо не могут использовать Hilt.

В этих случаях вы можете использовать @EntryPoint. Точка входа — это граничное место, где вы можете получить объекты, предоставляемые Hilt, из кода, который не может использовать Hilt для внедрения своих зависимостей. Это момент, когда код сначала попадает в контейнеры, управляемые Hilt.

Вариант использования

Мы хотим иметь возможность экспортировать наши журналы вне нашего процесса приложения. Для этого нам нужно использовать файл [ContentProvider](https://developer.android.com/reference/android/content/ContentProvider). Мы разрешаем потребителям запрашивать только один конкретный журнал (при наличии id) или все журналы из приложения с использованием файла ContentProvider. Мы будем использовать базу данных Room для получения данных. Следовательно, LogDao класс должен предоставлять методы, которые возвращают требуемую информацию с помощью базы данных [Cursor](https://developer.android.com/reference/android/database/Cursor). Откройте LogDao.kt файл и добавьте в интерфейс следующие методы.

@Dao  
interface LogDao {  
    ...  
  
    @Query("SELECT \* FROM logs ORDER BY id DESC")  
    fun selectAllLogsCursor(): Cursor  
  
    @Query("SELECT \* FROM logs WHERE id = :id")  
    fun selectLogById(id: Long): Cursor?  
}

Затем мы должны создать новый ContentProvider класс и переопределить query метод для возврата Cursor с журналами. Создайте новый файл с именем LogsContentProvider.kt в новом contentprovider каталоге со следующим содержимым:

package com.example.android.hilt.contentprovider  
  
import android.content.ContentProvider  
import android.content.ContentUris  
import android.content.ContentValues  
import android.content.Context  
import android.content.UriMatcher  
import android.database.Cursor  
import android.net.Uri  
import com.example.android.hilt.data.LogDao  
import dagger.hilt.EntryPoint  
import dagger.hilt.EntryPoints  
import dagger.hilt.InstallIn  
import dagger.hilt.android.components.ApplicationComponent  
import java.lang.UnsupportedOperationException  
  
/\*\* The authority of this content provider.  \*/  
private const val LOGS\_TABLE = "logs"  
  
/\*\* The authority of this content provider.  \*/  
private const val AUTHORITY = "com.example.android.hilt.provider"  
  
/\*\* The match code for some items in the Logs table.  \*/  
private const val CODE\_LOGS\_DIR = 1  
  
/\*\* The match code for an item in the Logs table.  \*/  
private const val CODE\_LOGS\_ITEM = 2  
  
/\*\*  
 \* A ContentProvider that exposes the logs outside the application process.  
 \*/  
class LogsContentProvider: ContentProvider() {  
  
    private val matcher: UriMatcher = UriMatcher(UriMatcher.NO\_MATCH).apply {  
        addURI(AUTHORITY, LOGS\_TABLE, CODE\_LOGS\_DIR)  
        addURI(AUTHORITY, "$LOGS\_TABLE/\*", CODE\_LOGS\_ITEM)  
    }  
  
    override fun onCreate(): Boolean {  
        return true  
    }  
  
    /\*\*  
     \* Queries all the logs or an individual log from the logs database.  
     \*  
     \* For the sake of this codelab, the logic has been simplified.  
     \*/  
    override fun query(  
        uri: Uri,  
        projection: Array<out String>?,  
        selection: String?,  
        selectionArgs: Array<out String>?,  
        sortOrder: String?  
    ): Cursor? {  
        val code: Int = matcher.match(uri)  
        return if (code == CODE\_LOGS\_DIR || code == CODE\_LOGS\_ITEM) {  
            val appContext = context?.applicationContext ?: throw IllegalStateException()  
            val logDao: LogDao = getLogDao(appContext)  
  
            val cursor: Cursor? = if (code == CODE\_LOGS\_DIR) {  
                logDao.selectAllLogsCursor()  
            } else {  
                logDao.selectLogById(ContentUris.parseId(uri))  
            }  
            cursor?.setNotificationUri(appContext.contentResolver, uri)  
            cursor  
        } else {  
            throw IllegalArgumentException("Unknown URI: $uri")  
        }  
    }  
  
    override fun insert(uri: Uri, values: ContentValues?): Uri? {  
        throw UnsupportedOperationException("Only reading operations are allowed")  
    }  
  
    override fun update(  
        uri: Uri,  
        values: ContentValues?,  
        selection: String?,  
        selectionArgs: Array<out String>?  
    ): Int {  
        throw UnsupportedOperationException("Only reading operations are allowed")  
    }  
  
    override fun delete(uri: Uri, selection: String?, selectionArgs: Array<out String>?): Int {  
        throw UnsupportedOperationException("Only reading operations are allowed")  
    }  
  
    override fun getType(uri: Uri): String? {  
        throw UnsupportedOperationException("Only reading operations are allowed")  
    }  
}

Вы увидите, что getLogDao(appContext)вызов не компилируется! Нам нужно реализовать это, взяв LogDao зависимость из контейнера приложения Hilt. Тем не менее, Hilt не поддерживает инъекцию в исходный код, ContentProvider как, например, в Activity с @AndroidEntryPoint.

Нам нужно создать новый интерфейс с аннотациями @EntryPointдля доступа к нему.

@EntryPoint в действии

**Точка входа — это интерфейс с методом доступа для каждого типа привязки, который нам нужен** (включая его квалификатор). Кроме того, интерфейс должен быть аннотирован, @InstallIn чтобы указать компонент, в котором нужно установить точку входа.

Лучшая практика - добавить новый интерфейс точки входа в класс, который его использует. Поэтому включите интерфейс в LogsContentProvider.kt файл:

class LogsContentProvider: ContentProvider() {  
  
    @InstallIn(ApplicationComponent::class)  
    @EntryPoint  
    interface LogsContentProviderEntryPoint {  
        fun logDao(): LogDao  
    }  
  
    ...  
}

Обратите внимание, что интерфейс снабжен аннотацией, @EntryPoint и он установлен в, ApplicationComponent поскольку нам нужна зависимость от экземпляра Application контейнера. Внутри интерфейса мы предоставляем методы для привязок, к которым мы хотим получить доступ, в нашем случае LogDao.

Для доступа к точке входа используйте соответствующий статический метод из EntryPointAccessors. Параметр должен быть либо экземпляром компонента, либо @AndroidEntryPoint объектом, который действует как держатель компонента. Убедитесь, что компонент, который вы передаете в качестве параметра, и EntryPointAccessors статический метод, соответствуют классу Android в @InstallIn аннотации @EntryPoint интерфейса:

Теперь мы можем реализовать getLogDao метод, которого нет в приведенном выше коде. Давайте использовать интерфейс точки входа, который мы определили выше в нашем LogsContentProviderEntryPoint классе:

class LogsContentProvider: ContentProvider() {  
    ...  
  
    private fun getLogDao(appContext: Context): LogDao {  
        val hiltEntryPoint = EntryPointAccessors.fromApplication(  
            appContext,  
            LogsContentProviderEntryPoint::class.java  
        )  
        return hiltEntryPoint.logDao()  
    }  
}

Обратите внимание, как мы передаем applicationContext статическому EntryPoints.get методу и классу интерфейса, который помечен @EntryPoint.

12. Поздравляю!

Теперь вы знакомы с Hilt и сможете добавить его в свое приложение для Android. В этой кодовой лаборатории вы узнали о:

* Как настроить Hilt в своем классе Application с помощью @HiltAndroidApp.
* Как добавить контейнеры зависимостей к различным компонентам жизненного цикла Android с помощью @AndroidEntryPoint.
* Как использовать модули, чтобы сообщить Hilt, как предоставлять определенные типы.
* Как использовать квалификаторы для предоставления нескольких привязок для определенных типов.
* Как протестировать свое приложение с помощью Hilt.
* Когда @EntryPoint это полезно и как им пользоваться.