

Hilt



작성자 : Charlezz

작성일 : 2020.07.06

수정일 : 2020.07.08

버전 : v1.1

Hilt 공식 문서를 번역한 내용입니다. (<https://dagger.dev/hilt/>)

오역 및 오타자가 다수 있을 수 있으니 양해 부탁드립니다. 오역 제보 및 건의(charlezz@charlezz.com)

이 문서의 내용을 블로그 및 웹사이트에 게시하는 경우 출처(<https://charlezz.com>)를 꼭 남겨주시기 바랍니다.

License

Copyright 2012 The Dagger Authors

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

1. Hilt

Hilt는 안드로이드 애플리케이션에 의존성주입하는 Dagger의 표준적인 방법을 제공한다.

Hilt의 목적은 다음과 같다.

- 안드로이드 애플리케이션을 위한 Dagger와 관련 기반 코드들을 간소화 한다.
- 쉬운 설정과 가독성/이해도 그리고 앱간 코드 공유를 위한 표준 컴포넌트, 스코프 세트를 생성한다.
- 다양한 빌드 유형에 대한 서로 다른 바인딩을 제공하는 쉬운 방법을 제공한다

Hilt 설계 개요

Hilt는 Dagger를 기반하여 코드를 생성하고 작동한다. 이것은 Dagger를 사용하는 대부분의 보일러플레이트 코드를 없애고 실제로 객체 생성 방법과 주입 위치를 정의하는 측면만을 남겨 둔다. Hilt는 Dagger 컴포넌트

와 코드를 생성하여 Activity 및 Fragment와 같은 안드로이드 클래스를 자동으로 주입한다.

Hilt는 전이 클래스 경로를 기반으로 표준 안드로이드 Dagger 컴포넌트 세트를 생성한다. 이를 위해서는 Dagger 모듈에 Hilt 애노테이션(`@InstallIn`)을 표시하여 어떤 컴포넌트에 포함시켜야 하는지 Hilt에게 알려 주어야 한다. 안드로이드 프레임 워크 클래스에서 객체를 가져 오기 위해서는 또 다른 Hilt 어노테이션(`@AndroidEntryPoint`)을 사용한다. 이는 Dagger 주입 코드를 가지고 있는 base 클래스를 생성하고 상속할 수 있도록 한다. Gradle 사용하면 클래스 상속은 내부에서 바이트 코드 변환으로 수행된다.

테스트에서 Hilt는 프로덕션과 마찬가지로 Dagger 컴포넌트를 생성한다. 테스트에는 테스트 바인딩 추가 또는 교체에 도움이 되는 다른 특별한 유틸리티가 있다.

2. Benefit

왜 Hilt를 사용해야 할까?

- 보일러플레이트 코드 감소
- 분리된 빌드 의존성
- 환경 설정의 간소화
- 개선된 테스트 환경
- 표준화된 컴포넌트들

보일러 플레이트 코드 감소

Hilt의 목표는 개발자가 Dagger 설정에 대해 걱정할 필요 없이 Dagger 바인딩 정의 및 사용법에 집중할 수 있도록 하는 것이다. 즉, 모듈 및 인터페이스 목록을 사용하여 컴포넌트 정의, 생명주기의 올바른 지점에서 컴포넌트를 작성 및 유지하는 코드, 상위 컴포넌트를 가져 오기 위한 인터페이스 및 캐스팅 등을 숨긴다.

Hilt가 단순해지는 몇가지 이유중 하나가 바로 단일 컴포넌트를 사용한다는 것이다. Hilt는 본질적으로 전역 바인딩 네임 스페이스를 장려하여 어떤 Activity나 Fragment를 주입했는지 추적하지 않고도 사용되는 바인딩 정의를 쉽게 알 수 있다.

분리된 빌드 의존성

만약 전문적 지식이 없는 상태에서 Dagger 컴포넌트를 직접적으로 참조하는 경우에는 여러가지 빌드 오류를 접하게 된다. 이러한 문제들은 Dagger 컴포넌트가 설치된 모든 모듈을 직접적으로 참조하고 있기 때문에 발생한다. 이런 점은 비대한 의존성으로 이어지고, 빌드 속도가 느려지는 원인이 된다. 이를 해결하는 자연스러운 방법은 인터페이스와 안전하지 않은 캐스팅(`unsafe cast`)이 있다. 런타임 오류가 발생할 수 있기 때문에 이는 선택적으로 절충해야 한다. 예를 들어, 새로운 인젝터 인터페이스를 도입하면 직접 컴포넌트에 의존하는 것을 피할 수 있지만 컴포넌트를 인젝터 인터페이스로 확장하지 않으면 캐스트 예외(`cast exception`)가 발생한다.

인터페이스, 안전하지 않은 캐스팅 그리고 모듈/인터페이스 목록에 대한 코드를 내부적으로 생성하기 때문에 Hilt는 런타임에 안전하지 않은 캐스팅을 안전하게 만들 수 있는데 그 이유는 코드 생성과 모듈/진입점 탐색을 보장 하기 때문이다.

환경 설정 간소화

앱에는 종종 다른 기능을 가진 프로덕션 또는 개발 빌드와 같이 다른 빌드 구성이 있다. 이러한 다른 기능 세트는 종종 다른 Dagger 모듈 세트를 의미한다. 일반적인 Dagger 빌드에서 다른 모듈 세트에는 일반적으로 많은 부분이 반복되는 모든 스코프에 대해 분리된 컴포넌트 트리가 필요하다. Hilt는 빌드 종속성을 통해 모듈

을 설치하고 컴포넌트 코드를 생성하므로, 다른 build flavor를 생성하는 것은 추가된 또는 제거된 의존성으로 컴파일 하는 것만큼 간단하다.

개선된 테스트 환경

Dagger를 테스트하는 것은 위에서 언급한 환경설정 문제 때문에 어려울 수 있다. 마찬가지로, Hilt는 컴포넌트 코드 생성으로 인해 테스트 모듈과 바인딩을 쉽게 변경할 수 있다. Hilt는 모듈을 관리하고 테스트 바인딩을 제공하여 테스트에서 Dagger를 사용할 수 있도록 특정 테스트 유틸리티를 내장하고 있다. 테스트에서 Dagger를 사용하면 테스트에서 보일러플레이트 코드를 줄이고 프로덕션에서 인스턴스화 되는 것과 같은 방식으로 코드를 인스턴스화 함으로써 테스트를 더욱 탄탄하게 만든다.

표준화된 컴포넌트들

Hilt는 컴포넌트 계층을 표준화한다. 즉, Hilt와 통합된 라이브러리도 표준화된 컴포넌트에 쉽게 바인딩을 추가하거나 소비할 수 있게 된다. 이를 통해 Hilt를 사용한 앱에 깔끔하고 간단하게 더 복잡한 라이브러리를 구축 할 수 있다.

3. Gradle 설정하기

Hilt 의존성 추가 하기

Hilt를 사용하기 위해서는 다음 빌드 의존성을 모듈의 build.gradle 파일에 추가해야 한다.

```
dependencies {
    implementation 'com.google.dagger:hilt-android:<VERSION>'
    annotationProcessor 'com.google.dagger:hilt-android-compiler:<VERSION>'

    // For instrumentation tests
    androidTestImplementation 'com.google.dagger:hilt-android-testing:<VERSION>'
    androidTestAnnotationProcessor 'com.google.dagger:hilt-android-compiler:<VERSION>'

    // For local unit tests
    testImplementation 'com.google.dagger:hilt-android-testing:<VERSION>'
    testAnnotationProcessor 'com.google.dagger:hilt-android-compiler:<VERSION>'
}
```

코틀린에서 Hilt 사용하기

만약 코틀린을 사용한다면 kapt 플러그인을 적용하고 annotationProcessor 대신 kapt를 사용한 컴파일러 의존성을 선언해야 한다.

발생할 수 있는 에러들을 수정하기 위해서 추가적으로 kapt에 corretErrorTypes를 true로 설정하자.

```
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
}
```

Hilt Gradle 플러그인

Hilt Gradle 플러그인은 바이트 코드 변환을 실행하여 보다 쉬운 API를 사용할 수 있도록 한다. 플러그인은 생성된 클래스가 기본 클래스의 메소드에 대한 코드 완료를 방해 할 수 있으므로 IDE에서 더 나은 개발자 경험을 위해 만들어졌다. 공식 문서의 예제들은 플러그인 사용을 가정하고 있다. Hilt Gradle 플러그인을 구성하려면 먼저 프로젝트의 최상위 build.gradle 파일에서 다음과 같은 의존성을 선언하자.

```
buildscript {
    repositories {
        // other repositories...
        mavenCentral()
    }
    dependencies {
        // other plugins...
        classpath 'com.google.dagger:hilt-android-gradle-plugin:<version>'
    }
}
```

프로젝트 build.gradle의 설정을 끝냈다면 모듈의 build.gradle에 플러그인을 적용한다.

```
apply plugin: 'com.android.application'
apply plugin: 'dagger.hilt.android.plugin'

android {
    // ...
}
```

⚠ Hilt Gradle 플러그인은 어노테이션 프로세서 인자를 사용한다. 어노테이션 프로세서 인자가 필요한 다른 라이브러리를 사용하는 경우에는 인자를 재정의 하는 대신 인자를 추가하도록 해야 한다.

플러그인을 사용하는 이유

Gradle 플러그인의 주요 이점 중 하나는 @AndroidEntryPoint 및 @HiltAndroidApp를 보다 쉽게 사용할 수 있다는 점이다. Gradle 플러그인이 없으면 어노테이션에 기본 클래스를 지정하고 어노테이션이 달린 클래스는 생성된 클래스를 상속해야 한다.

```
@HiltAndroidApp(MultiDexApplication::class)
class MyApplication : Hilt_MyApplication()
```

Gradle 플러그인을 사용하면 어노테이션이 달린 클래스는 기본 클래스를 직접적으로 상속할 수 있다.

```
@HiltAndroidApp
class MyApplication : MultiDexApplication()
```

로컬 테스트 환경 설정

플러그인은 기본적으로 Instrumented 테스트 클래스들을 변형한다 (일반적으로 androidTest 소스 폴더에 위치함). 그러나 로컬 장치 테스트 (일반적으로 test 소스 폴더에 위치) 에는 추가적인 설정이 필요하다. 로컬 단위 테스트에서 @AndroidEntryPoint 클래스를 변환하려면 모듈의 build.gradle에 다음의 구성을 적용해야 한다.

```
hilt {
    enableTransformForLocalTests = true
}
```

어노테이션 프로세서 인자 적용하기

Hilt Gradle 플러그인은 어노테이션 프로세서 인자를 설정한다. 만약 어노테이션 프로세서 인자를 필요로 하는 다른 라이브러리를 사용하고 있다면, 반드시 인자를 재정의 하는 대신 추가하도록 해야한다.

예를 들면 다음에 나오는 += 의 사용은 명백하게 Hilt의 인자를 재정의하는 것을 피할 수 있다.

```
javaCompileOptions {
    annotationProcessorOptions {
        arguments += ["foo" : "bar"]
    }
}
```

만약 +를 안붙이면 인자(arguments)가 재정의되고, Hilt는 컴파일에 실패 하고 다음과 같은 에러를 출력한다.

```
Expected @HiltAndroidApp to have a value. Did you forget to apply the Gradle Plugin?
```

4. Quick Start

Introduction

Hilt를 사용하면 안드로이드 앱에 의존성 주입을 쉽게 할 수 있다. 이 튜토리얼에서는 기존 앱에 Hilt를 사용하도록 안내한다.

Gradle vs 비-Gradle 사용자

Gradle 사용자의 경우, Hilt Gradle 플러그인은 Hilt 어노테이션의 사용으로 인해 Hilt가 생성하는 클래스에 대한 직접적인 참조를 피함으로써 Hilt를 보다 쉽게 사용할 수 있도록 한다.

Gradle 플러그인이 없으면 어노테이션에 기본 클래스를 지정하고 어노테이션이 달린 클래스는 생성된 클래스를 상속해야 한다

```
@HiltAndroidApp(MultiDexApplication::class)
class MyApplication : Hilt_MyApplication()
```

Gradle 플러그인을 사용한다면 어노테이션이 달린 클래스는 기본 클래스를 직접적으로 상속할 수 있게 된다.

```
@HiltAndroidApp
class MyApplication : MultiDexApplication()
```

앞으로 나오는 예제들은 Hilt Gradle 플러그인을 사용하는 것을 가정한다.

Hilt Application

Hilt를 사용하는 모든 앱은 @HiltAndroidApp이 달린 Application 클래스를 포함해야 한다.

@HiltAndroidApp은 Hilt 컴포넌트의 코드 생성과 컴포넌트를 사용하는 Application의 기본 클래스를 생성하게 된다. 코드 생성에는 모든 모듈에 대한 액세스 권한이 필요하므로 Application 클래스를 컴파일하는 대상에는 전이 의존성에 모든 Dagger 모듈이 있어야 한다.



전이 의존성이란?

어떤 라이브러리를 의존성으로 추가하면 그 라이브러리의 의존성도 함께 의존하게 되는데, 이를 전이 의존성(transitive dependencies)라고 한다.

@AndroidEntryPoint가 달린 다른 안드로이드 프레임워크 클래스와 마찬가지로 Application에도 멤버 주입이 된다. 이는 super.onCreate()가 호출 된 후 Application의 필드에 의존성 주입이 이루어지는 것을 의미한다.

예를 들어 일반적인 Dagger 사용시 MyApplication이 MyBaseApplication을 상속하는 구조이면서 멤버 변수로 Bar를 가지고 있다고 가정해보자.

```
class MyApplication : MyBaseApplication() {
    @Inject lateinit var bar: Bar override fun onCreate() {
        super.onCreate()

        val myComponent =
            DaggerMyComponent.builder()
                ...
                .build()

        myComponent.inject(this)
    }
}
```

앞에서 살펴본 코드는 Hilt를 사용하면 다음과 같이 멤버 주입이 된다.

```
@HiltAndroidApp
class MyApplication : MyBaseApplication() {
    @Inject lateinit var bar: Bar
    override fun onCreate() {
        super.onCreate() // super.onCreate()에서 의존성 주입을 하게 된다.
        // 여기에서 bar 변수를 사용할 수 있다.
    }
}
```

@AndroidEntryPoint

Application에서 멤버 주입이 가능하게 설정하고 나면, 다른 안드로이드 클래스들에서도 @AndroidEntryPoint 어노테이션을 사용하여 멤버 주입을 하는 것이 가능해진다. @AndroidEntryPoint를 사용할 수 있는 타입은 다음과 같다.

- Activity
- Fragment
- View
- Service
- BroadcastReceiver

Hilt와 ViewModel의 사용은 직접적으로 지원하지 않는 대신에 Jetpack extension을 통해 지원한다. 다음 예제는 어떻게 Activity에 어노테이션을 추가 할 수 있는지 보여준다. 다른 타입의 경우도 Activity와 동일한 방법으로 진행된다.

Activity에서 멤버 주입을 하기 위해서 @AndroidEntryPoint를 추가 하자.

```
@AndroidEntryPoint
class MyActivity : MyBaseActivity() {
    @Inject lateinit var bar: Bar // ApplicationComponent 또는 ActivityComponent으로 부터 의존성이 주입된다.

    override fun onCreate() {
        // super.onCreate()에서 의존성 주입이 발생한다.
        super.onCreate()

        // Do something with bar ...
    }
}
```



Hilt는 현재 ComponentActivity를 상속한 Activity만 지원하고 있고 Fragment의 경우 androidx 라이브러리의 Fragment를 상속한 경우에만 지원한다. 현재 안드로이드 플랫폼에 있는 Fragment는 deprecated된 상태다.

Hilt 모듈

Hilt의 모듈은 표준 Dagger 모듈로 @InstallIn이라는 추가적인 어노테이션을 갖는다. @InstallIn은 Hilt의 표준 컴포넌트들 중 어떤 컴포넌트에 모듈을 설치할지 결정한다.

Hilt 컴포넌트가 생성될 때 모듈들은 추가된 @InstallIn과 함께 알맞은 컴포넌트 또는 서브컴포넌트에 설치 된다. Dagger와 같이 컴포넌트에 모듈을 설치하면 해당 모듈에 바인딩된 의존성들은 컴포넌트 내 다른 바인딩 또는 다른 하위 컴포넌트의 바인딩이 접근하는 것을 허용한다. 바인딩 된 의존성에 @AndroidEntryPoint 클래스가 접근 하는 것 또한 가능하다. 해당 컴포넌트의 대한 바인딩 스코프를 지정할 수도 있다.

@InstallIn 사용하기

모듈에서 @InstallIn 어노테이션을 추가하는 것으로 Hilt 컴포넌트에 모듈이 설치 된다. Hilt를 사용할 때 Dagger모듈 상에 이러한 @InstallIn 어노테이션은 필수지만 이 검사는 선택적으로 비활성화 할 수 있다.



만약 모듈이 `@InstallIn` 어노테이션을 가지고 있지 않다면 해당 모듈은 컴포넌트에 설치되지 않아 컴파일 에러를 발생 시킨다.

`@InstallIn` 어노테이션에 어떤 컴포넌트가 모듈이 설치될 적당한 Hilt 컴포넌트인지 명시해야한다. 예를 들면 애플리케이션 스코프에서 어떤 바인딩이든 사용할 수 있도록 모듈을 설치하려면 `ApplicationComponent`를 사용해야 한다.

```
@Module
@InstallIn(ApplicationComponent.class) // 생성되는 ApplicationComponent에 FooModule을 설치함
public final class FooModule {
    @Provides
    static Bar provideBar() {...}
}
```

5.1 Core APIs - Component



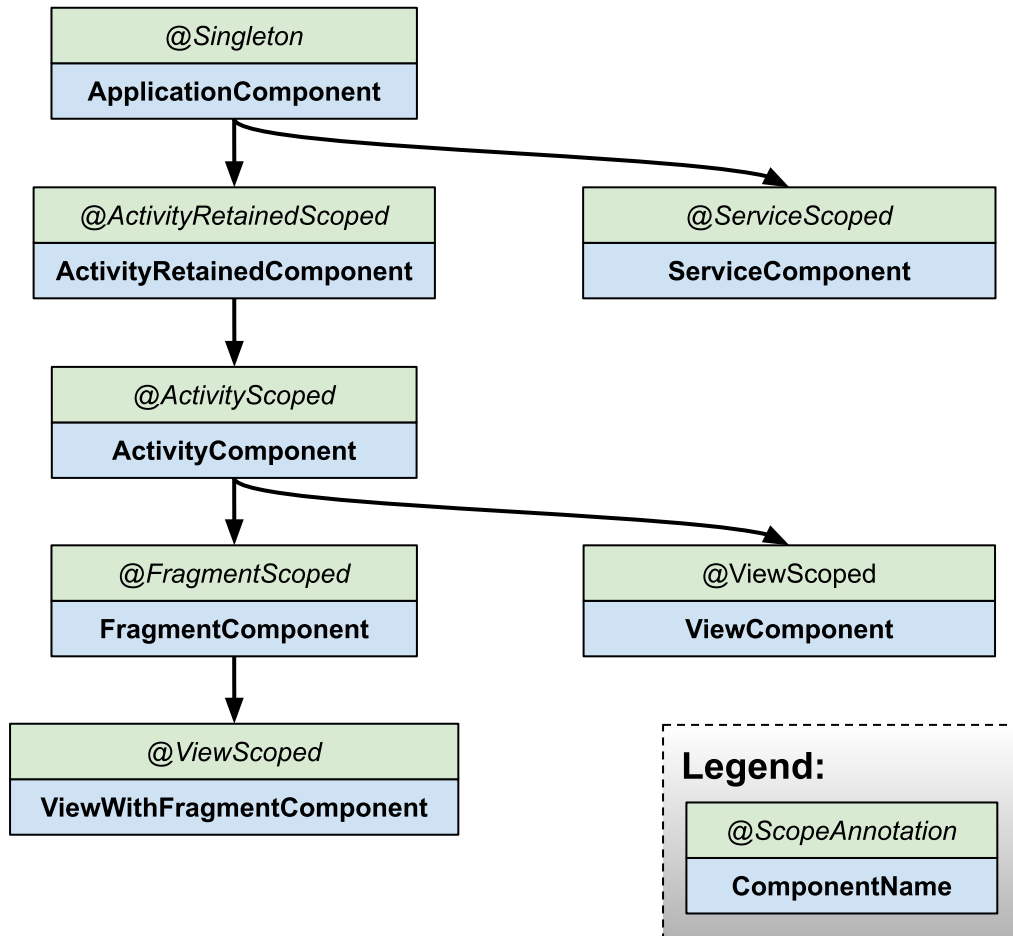
다음 나을 내용은 컴포넌트, 모듈, 스코프 그리고 바인딩을 포함하는 Dagger에 대한 기초적인 내용을 알고 있음을 가정한다.

컴포넌트 계층

기존 사용하던 Dagger와 다르게 Hilt사용자는 Dagger 컴포넌트를 직접적으로 정의하거나 인스턴스화 할 필요가 없어졌다. 대신에 Hilt는 이미 정의된 컴포넌트를 통해 생성되는 클래스들을 제공하고 있다. Hilt는 안드로이드 `Application`의 다양한 생명주기에 자동으로 통합되는 내장 컴포넌트 세트를 해당 스코프 어노테이션과 함께 제공한다. 아래에 있는 다이어그램은 표준 Hilt 컴포넌트 계층을 보여주고 있다. 각 컴포넌트에 위에 달린 어노테이션은 컴포넌트 바인딩의 생명주기를 지정하는 데 사용된다. 각 컴포넌트 아래에 있는 화살표는 하위 컴포넌트를 가르키고 있다. 보통 하위 컴포넌트의 바인딩은 상위 컴포넌트의 바인딩이 가지고 있는 의존성들을 가질 수 있다.



`@InstallIn`이 달린 모듈의 바인딩에 스코프가 지정될 때는 반드시 모듈이 설치되는 컴포넌트의 스코프와 일치해야 한다. 예를 들면, `@InstallIn(ActivityComponent.class)` 모듈은 `@ActivityScoped`만 사용할 수 있다.



컴포넌트 멤버 주입

앞에서 다룬 @AndroidEntryPoint 섹션에서는 안드로이드 클래스에 멤버 주입을 하는 방법에 대해서 다뤘다. Hilt 컴포넌트들은 각각 안드로이드 클래스에 맞는 의존성 주입을 해야 할 의무가 있다. 다음에 나올 표가 안드로이드 클래스에 적합한 Hilt 컴포넌트를 보여준다.





Component	Injector for	Property
ApplicationComponent	Application	
ActivityRetainedComponent	ViewModel	
ActivityComponent	Activity	
FragmentComponent	Fragment	
ViewComponent	View	
ViewWithFragmentComponent	View with @WithFragmentBindings	
ServiceComponent	Service	

컴포넌트의 수명

컴포넌트의 수명은 다음 두가지 관점에서 볼 때 바인딩의 수명과 관련되기 때문에 중요하다.

1. 컴포넌트가 생성되고 종료될 때, 해당 스코프 어노테이션이 지정된 바인딩 또한 수명을 함께한다.
2. 컴포넌트 수명은 멤버 주입된 값들이 사용될 수 있는 시기를 나타낸다. (@Inject 필드를 사용할 때 null이면 안된다.)

컴포넌트의 수명은 일반적으로 안드로이드 클래스에 대응하는 인스턴스 생성과 소멸을 따라간다. 다음 표는 스코프 어노테이션과 각 컴포넌트에 맞는 수명을 목록으로 보여주고 있다.

 Component	 Scope	 Created at	 Destroyed at
<u>ApplicationComponent</u>	@Singleton	Application#onCreate()	Application#onDestroy()
<u>ActivityRetainedComponent</u>	@ActivityRetainedScope	Activity#onCreate() (주석1)	Activity#onDestroy() (주석1)
<u>ActivityComponent</u>	@ActivityScoped	Activity#onCreate()	Activity#onDestroy()
<u>FragmentComponent</u>	@FragmentScoped	Fragment#onAttach()	Fragment#onDestroy()
<u>ViewComponent</u>	@ViewScoped	View#super()	View destroyed
<u>ViewWithFragmentComponent</u>	@ViewScoped	View#super()	View destroyed
<u>ServiceComponent</u>	@ServiceScoped	Service#onCreate()	Service#onDestroy()

스코프 바인딩 vs 비 스코프 바인딩

기본적으로 모든 Dagger의 바인딩은 스코프 어노테이션이 없는 비 스코프 바인딩이다. 이는 각 바인딩이 요청될 때마다 Dagger는 새로운 인스턴스를 생성하는 것을 의미 한다.

그러나 Dagger는 컴포넌트에 스코프 어노테이션을 지정할 수 있다.(바로 앞에 나온 표의 스코프 어노테이션을 살펴보자). 스코프가 지정된 컴포넌트에서 해당 스코프 바인딩은 컴포넌트 인스턴스당 한번만 생성되고, 해당 바인딩에 대한 모든 요청에 동일한 인스턴스를 제공한다.

예를 들면 다음 코드와 같다.

```
// 스코프가 지정되지 않은 이 바인딩은
// 각 바인딩의 요청에 대해 새로운 인스턴스를 제공하게 된다.
class UnscopedBinding @Inject constructor() {...}

// 스코프가 지정된 이 바인딩은
// 이 바인딩에 대한 동일한 컴포넌트 인스턴스는 각기 다른 요청에 대해 동일한 인스턴스를 제공한다.
// @FragmentScoped로 지정되었기 때문에 동일한 Fragment의 요청에 대해 동일한 인스턴스를 제공하게 된다.
@FragmentScoped
class ScopedBinding @Inject constructor() {...}
```

⚠ 일반적으로 오해하는 부분이 @FragmentScoped가 지정된 바인딩이 모든 Fragment 인스턴스에 대해 동일한 바인딩 인스턴스를 공유할 것이라고 생각하는 점이다. 하지만 실제로 그렇지 않고 각 Fragment 인스턴스는 새로운 Fragment 컴포넌트 인스턴스를 얻기 때문에 각기 다른 Fragment 인스턴스는 각자만의 스코프 된 바인딩을 얻게 된다.

모듈에서 스코프 어노테이션 사용하기

이전 섹션에서 생성자 @Inject의 바인딩에 대해 어떻게 스코프 어노테이션을 사용하는지 살펴보았다. 하지만 이 방법 말고도 모듈에서 바인딩에 대해 비슷한 방법으로 스코프 어노테이션을 사용할 수 있다.

```
@Module
@InstallIn(FragmentComponent.class)
object FooModule {
    // 이 바인딩은 스코프 어노테이션이 사용되지 않았다.
    @Provides
    fun provideUnscopedBinding() = UnscopedBinding()

    // 이 바인딩은 스코프 어노테이션이 사용되었다.
    @Provides
    @FragmentScoped
    fun provideScopedBinding() = ScopedBinding()
}
```

⚠ 일반적으로 오해하는 부분이 모듈내에 선언된 모든 바인딩이 모듈이 설치되는 컴포넌트와 수명을 함께 한다고 생각하는 것이다. 하지만 그렇지 않고, 단지 스코프 어노테이션이 지정된 바인딩 선언만 해당 컴포넌트와 수명을 함께하여 각 바인딩 요청들에 대해 동일한 인스턴스를 제공한다.

스코프 어노테이션은 언제 사용할까?

바인딩에 대해 스코프 어노테이션을 지정하는 것은 코드 생성 크기 그리고 런타임 성능에 영향을 미치므로 가능한 스코프 어노테이션을 조금만 사용하는 것이 좋다. 바인딩에 대해 스코프를 어노테이션을 사용해야 하는지 결정하는 일반적인 규칙은 동일한 인스턴스를 보장해야 할 만큼 코드의 정확성이 필요한 경우다. 성능상의 이유로만 스코프 어노테이션을 사용해야 한다면, 먼저 성능이 문제인지 확인한 뒤 표준 Hilt 컴포넌트 스코프 어노테이션 대신 @Reusable을 사용하는 것이 좋다.

컴포넌트가 제공하는 기본 바인딩

각 Hilt 컴포넌트는 기본 바인딩 세트와 함께 사용자가 정의한 바인딩들을 의존성 주입하게 된다.

Aa Component	☰ Default Bindings
<u>ApplicationComponent</u>	Application (주석2)
<u>ActivityRetainedComponent</u>	Application
<u>ActivityComponent</u>	Application, Activity
<u>FragmentComponent</u>	Application, Activity, Fragment
<u>ViewComponent</u>	Application, Activity, View
<u>ViewWithFragmentComponent</u>	Application, Activity, Fragment, View
<u>ServiceComponent</u>	Application, Service

(주석1) 구성 변경(Configuration changes)에 의해 Activity는 재생성 될 수 있으므로 ActivityRetainedComponent에서는 Activity의 인스턴스는 기본 바인딩으로 제공하지 않고 있다.

(주석2) Application 바인딩은 @ApplicationContext Context 또는 Application 으로 요청하면 제공 받을 수 있다.

5.2 Core APIs - Hilt Application



Gradle 플러그인을 사용하는 것을 가정하고 설명하고 있다. 만약 플러그인을 사용하지 않는다면 Gradle 설정하기 섹션을 참조하자.

Hilt Application

Hilt를 사용하는 모든 앱은 `@HiltAndroidApp`이 달린 `Application` 클래스를 포함해야 한다.

`@HiltAndroidApp`은 Hilt 컴포넌트의 코드 생성과 컴포넌트를 사용하는 `Application`의 기본 클래스를 생성하게 된다. 코드 생성에는 모든 모듈에 대한 액세스 권한이 필요하므로 `Application` 클래스를 컴파일하는 대상에는 전이 의존성에 모든 Dagger 모듈이 있어야 한다.

`@AndroidEntryPoint`가 달린 다른 안드로이드 프레임워크 클래스와 마찬가지로 `Application`에도 멤버 주입이 된다. 이는 `super.onCreate()`가 호출된 후 `Application`의 필드에 의존성 주입이 이루어지는 것을 의미한다.

예를 들어 일반적인 Dagger 사용시 `MyApplication`이 `MyBaseApplication`을 상속하는 구조이면서 멤버 변수로 `Bar`를 가지고 있다고 가정해보자.

```
class MyApplication : MyBaseApplication() {
    @Inject lateinit var bar: Bar override fun onCreate() {
        super.onCreate()

        val myComponent =
            DaggerMyComponent.builder()
                ...
                .build()

        myComponent.inject(this)
    }
}
```

앞에서 살펴본 코드는 Hilt를 사용하면 다음과 같이 멤버 주입이 된다.

```
@HiltAndroidApp
class MyApplication : MyBaseApplication() {
    @Inject lateinit var bar: Bar
    override fun onCreate() {
        super.onCreate() // super.onCreate()에서 의존성 주입을 하게 된다.
        // 여기에서 bar 변수를 사용할 수 있다.
    }
}
```

5.3 Core APIs - Android Entry Points



Gradle 플러그인을 사용하는 것을 가정하고 설명하고 있다. 만약 플러그인을 사용하지 않는다면 Gradle 설정하기 섹션을 참조하자.

지원하는 Android 타입

`Application`에서 멤버 주입이 가능하게 설정하고 나면, 다른 안드로이드 클래스들에서도

`@AndroidEntryPoint` 어노테이션을 사용하여 멤버 주입을 하는 것이 가능해진다. `@AndroidEntryPoint`를

사용할 수 있는 타입은 다음과 같다.

- Activity
- Fragment
- View
- Service
- BroadcastReceiver (주석1)

Hilt와 ViewModel의 사용은 직접적으로 지원하지 않는 대신에 Jetpack extension을 통해 지원한다. 다음 예제는 어떻게 Activity에 어노테이션을 추가 할 수 있는지 보여준다. 다른 타입의 경우도 Activity와 동일한 방법으로 진행된다.

Activity에서 멤버 주입을 하기 위해서 @AndroidEntryPoint를 추가 하자.

```
@AndroidEntryPoint
class MyActivity : MyBaseActivity() {
    @Inject lateinit var bar: Bar // ApplicationComponent 또는 ActivityComponent으로 부터 의존성이 주입된다.

    override fun onCreate() {
        // super.onCreate()에서 의존성 주입이 발생한다.
        super.onCreate()

        // Do something with bar ...
    }
}
```



Hilt는 현재 ComponentActivity를 상속한 Activity만 지원하고 있고 Fragment의 경우 androidx 라이브러리의 Fragment를 상속한 경우에만 지원한다. 현재 안드로이드 플랫폼에 있는 Fragment는 deprecated된 상태다.

유지되는 Fragment

Fragment의 onCreate() 메서드에서 setRetainInstance(true)를 호출하면 구성 변경이 발생해도 Fragment가 소멸-재생성 되지 않고 인스턴스가 유지된다.

Hilt와 함께 사용하는 Fragment는 (의존성 주입의 책임이 있는) 컴포넌트에 대한 참조를 가지고 있고 해당 컴포넌트는 이전 Activity의 인스턴스에 대한 참조를 가지고 있기 때문에 절대로 Fragment 인스턴스가 유지 되서는 안된다. 또한 Fragment에 주입된 스코프 된 바인딩 및 프로바이더는 Hilt와 함께 사용하는 Fragment의 인스턴스가 유지될 경우 메모리 누수가 발생할 수 있다. Hilt와 함께 사용하는 Fragment의 인스턴스가 유지되지 않도록 하기위해, Hilt와 Fragment를 같이 사용하는 경우 Fragment의 인스턴스가 유지 되면 구성 변경시 런타임 예외가 발생한다.

Hilt를 사용하는 Activity에 Hilt없이 Fragment만 사용하는 경우에는 Fragment의 인스턴스가 유지될 수 있다. 하지만 해당 Fragment가 Hilt를 사용하는 하위 Fragment를 포함한다면 마찬가지로 구성 변경시 런타임 예외가 발생하게 된다.



권장하지는 않지만, Hilt를 사용하는 Fragment들은 동일한 Activity의 인스턴스로부터 떨어졌다가 다시 붙을 수 있다. Hilt를 사용하는 Fragment는 `onAttach()` 메서드를 처음 호출 할 때만 의존성 주입이 수행된다. 유지된 Fragment가 다른 Activity 인스턴스에 다시 붙을 수 있기 때문에 이는 Fragment를 유지하는 것과 동일하지 않다. 다시 한번 강조하지만 이 방법은 권장하지 않으며, 각 사용법에 대해 새로운 Fragment 인스턴스를 만드는 것이 간단하다.

Fragment 바인딩과 View

기본적으로 `ApplicationComponent` 와 `ActivityComponent` 바인딩은 `View`에 주입될 수 있다. `Fragment`에서 이를 가능하게 하기 위해서는 `@WithFragmentBindings` 어노테이션을 클래스에 추가해야 한다.

```
@AndroidEntryPoint
@WithFragmentBindings
class MyView : MyBaseView {
    // ApplicationComponent, ActivityComponent, FragmentComponent,
    // ViewComponent의 바인딩들로부터 의존성 주입을 받을 수 있음
    @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();
        // 전개된 뷰 계층으로 부터 하위 뷰를 찾고 적용하자
    }
}
```

(주석1) 지원하는 다른 안드로이드 클래스와는 다르게 `BroadcastReceiver`만의 Dagger 컴포넌트를 가지고 있지 않고 대신에 `ApplicationComponent`로부터 간단히 의존성을 주입 받는다.

5.4 Core APIs - Modules

Hilt의 모듈은 표준 Dagger 모듈로 `@InstallIn`이라는 추가적인 어노테이션을 갖는다. `@InstallIn`은 Hilt의 표준 컴포넌트들 중 어떤 컴포넌트에 모듈을 설치할지 결정한다.

Hilt 컴포넌트가 생성될 때 모듈들은 추가된 `@InstallIn`과 함께 알맞은 컴포넌트 또는 서브 컴포넌트에 설치된다. Dagger와 같이 컴포넌트에 모듈을 설치하면 해당 모듈에 바인딩된 의존성들은 컴포넌트 내 다른 바인딩 또는 다른 하위 컴포넌트의 바인딩이 접근하는 것을 허용한다. 바인딩 된 의존성에 `@AndroidEntryPoint` 클래스가 접근 하는 것 또한 가능하다. 해당 컴포넌트의 대한 바인딩 스코프를 지정할 수도 있다.

@InstallIn 사용하기

모듈에서 `@InstallIn` 어노테이션을 추가하는 것으로 Hilt 컴포넌트에 모듈이 설치 된다. Hilt를 사용할 때 Dagger모듈 상에 이러한 `@InstallIn` 어노테이션은 필수지만 이 검사는 선택적으로 비활성화 할 수 있다.



만약 모듈이 `@InstallIn` 어노테이션을 가지고 있지 않다면 해당 모듈은 컴포넌트에 설치되지 않아 컴파일 에러를 발생 시킨다.

`@InstallIn` 어노테이션에 어떤 컴포넌트가 모듈이 설치될 적당한 Hilt 컴포넌트인지 명시해야 한다. 예를 들면 애플리케이션에서 범위에서 어떤 바인딩이든 사용할 수 있도록 모듈을 설치하려면 `ApplicationComponent`를 사용해야 한다.

```
@Module
@InstallIn(ApplicationComponent.class) // 생성되는 ApplicationComponent에 FooModule을 설치함
public final class FooModule {
    @Provides
    static Bar provideBar() {...}
}
```

각 컴포넌트에는 스코프 어노테이션이 존재하고, 이것은 컴포넌트의 수명동안 바인딩된 인스턴스가 유지될 수 있도록 한다. 예를 들면, `ApplicationComponent`에 설치되는 모듈의 바인딩에서는 `@Singleton` 어노테이션을 사용할 수 있다.

```
@Module
@InstallIn(ApplicationComponent::class)
object class FooModule {
    // @Singleton 프로바이더 메서드는 ApplicationComponent 인스턴스당 한번만 호출된다.
    @Provides
    @Singleton
    fun provideBar(): Bar {...}
}
```

추가적으로 각 컴포넌트는 컴포넌트 별로 기본적으로 가지고 있는 바인딩들이 있는데 5.1 Components 섹션에서 '컴포넌트가 제공하는 기본 바인딩'을 참조하자. 예를 들어, `ApplicationComponent`에서는 `Application` 바인딩이 기본적으로 제공된다.

```
@Module
@InstallIn(ApplicationComponent::class)
object class FooModule {
    // @InstallIn(ApplicationComponent.class) 모듈의 프로바이더 메서드들은
    // 바인딩 된 Application 객체를 참조하는 것이 가능하다.
    fun provideBar(app: Application): Bar {...}
}
```

여러 컴포넌트에 하나의 모듈 설치하기

여러 컴포넌트에 하나의 모듈을 설치하는 것이 가능하다. 예를 들면, `ViewComponent`와 `ViewWithFragmentComponent`에 있는 바인딩을 가지고 있다고 가정할 때 모듈을 복사해서 사용하는 것을 원치 않을 것이다. `@InstallIn({ViewComponent.class, ViewWithFragmentComponent.class})` 으로 작성하면 두 컴포넌트 모듈을 설치할 수 있게 된다.

여러 컴포넌트에 하나의 모듈을 설치할 때 지켜야 할 세가지 규칙이 있다.

- 모든 컴포넌트가 동일한 스코프 어노테이션을 지원하는 경우에만 프로바이더 메서드의 스코프 어노테이션을 지정할 수 있다. 예를 들면, `ViewComponent`와 `ViewWithFragmentComponent` 제공된 바인딩

은 @ViewScoped를 사용할 수 있다. Fragment와 Service에서 제공된 바인딩은 어떤 스코프 어노테이션을 지정하더라도 같이 사용할 수 없다.

- 모든 컴포넌트가 이러한 바인딩에 대해 접근이 가능하려면 프로바이더는 스코프 어노테이션 없이 바인딩 주입만 가능해야한다. 예를 들면, ViewComponent와 ViewWithFragmentComponent에 있는 바인딩은 View에 주입할 수 있는 반면에 FragmentComponent와 ServiceComponent의 바인딩은 Service 또는 Fragment에 주입 될 수 없다.
- 상위 컴포넌트와 하위 컴포넌트는 같은 모듈을 설치 할 수 없다. (대신, 상위 컴포넌트에 모듈을 설치하고 하위 컴포넌트에 해당 모듈의 바인딩에 접근할 수 있다.)

앱 빌드 변형

대부분의 안드로이드 앱은 앱의 빌드 변형에 따라 다른 모듈과 바인딩을 가져오게 된다. (예: 프로덕션, 디버그, 테스트, 기타)

Hilt에서 바이너리의 빌드 대상이 전이적으로 모듈에 의존하는 경우 해당 모듈은 앱의 적절한 컴포넌트에 설치된다. 이를 통해 다른 빌드 대상을 정의하고 바이너리 정의에 다른 의존성을 가져 오는 것처럼 쉽게 구성 할 수 있다.

Bazel: 빌드 파일들 정리하기

Bazel은 더 세밀하게 빌드 대상별로 분리하는 경향이 있기 때문에 @UninstallModules을 사용하여 테스트에서 교체하려는 모듈을 명시하는 대신 모듈을 설치하지 않는 것이 낫다. 왜냐하면 테스트에서 빌드 의존성을 줄이기 때문에 전체적으로 빌드 시간이 빨라질 수 있다.

모듈의 빌드 대상을 구성 할 때 테스트 또는 앱의 다른 구성에서 이 모듈을 교체할 수 있는지 고려해야 한다. 만약 해당 모듈이 교체 될 일이 없다면 자유롭게 다른 소스 코드와 함께 모듈을 포함해도 좋다.

모듈이 교체 가능해야 하는 경우 모듈에 분리된 대상을 생성해야 한다. 그런 다음 각 테스트 루트 (또는 다른 구성 루트)가 모듈 사용 여부를 결정할 수 있도록 앱의 루트에서 이 대상을 가져올 수 있다.

상황에 따른 모듈들들을 고려하여 빌드 대상을 구성하는 두가지 방법이 존재한다.

- 일반적인 빌드 대상과 함께 간단히 모듈들을 포함한다. 이는 사용자들이 사용하는 라이브러리에 당신이 정의한 내용을 항상 얻게 되는 것을 의미한다.
- 테스트에서 교체 가능한 바인딩들이 되려면, 안드로이드 바이너리 단계에서 모듈을 나누어야 한다.

기본적으로 첫번째 방식을 선택하는 것을 추천한다. 두번째 방법은 테스트에서 바인딩의 교체가 필요한 경우에만 사용하자. 그렇지만 많은 라이브러리들이 두가지 방식을 모두 사용한다.

5.5 Core APIs - Entry Points

Entry point란 무엇인가?

Entry point(진입점)는 Dagger를 사용하여 의존성 주입을 할 수 없는 코드에서 제공된 Dagger 객체를 얻을 수 있는 방법이다. Dagger가 관리 하는 오브젝트 그래프에 코드가 처음 들어가는 지점이다.

만약 Dagger 컴포넌트에 익숙하다면, Entry point는 Hilt가 상속하여 생성할 컴포넌트의 인터페이스다.

Entry point는 언제 필요할까?

Dagger를 적용하지 않은 라이브러리를 인터페이스링 하거나 Hilt에서 지원하지 않는 안드로이드 구성요소가 Dagger 객체에 접근이 필요할 때 Entry point가 필요하다.

일반적으로, 대부분의 entry point는 Activity, Fragment 와 같은 안드로이드 플랫폼이 인스턴스화 하는 곳에서 필요하다. @AndroidEntryPoint는 Entry point들을 다루기 위해 특화된 도구이며 안드로이드 클래스의 Entry point에 접근할 수 있도록 한다. 이 작업은 이미 해당 안드로이드 클래스들에서는 특별히 다루고 있기 때문에, 다음 나올 문서에서는 다른 타입의 클래스에 Entry point가 필요한 경우를 다룬다.

Entry Point, 어떻게 사용하나?

EntryPoint 생성하기

Entry point를 생성하기 위해서는 각 바인딩 타입에 대한 접근 가능한 메서드를 사용하여 인터페이스를 정의하고 @EntryPoint 어노테이션을 추가해야 한다. 그런 다음 @InstallIn을 추가하여 Entry point가 설치될 컴포넌트를 지정한다.

```
@EntryPoint
@InstallIn(ApplicationComponent::class)
interface FooBarInterface {
    @Foo fun getBar(): Bar
}
```

EntryPoint에 접근하기

Entry point에 접근하기 위해서는 EntryPoints클래스를 사용하여 컴포넌트 인스턴스를 매개변수로 전달하거나 컴포넌트 홀더 역할을 하는 @AndroidEntryPoint 객체를 전달한다. 매개변수로 전달하는 컴포넌트가 @EntryPoint 인터페이스에 추가되어 있는 @InstallIn 어노테이션이 전달하는 컴포넌트와 일치하는지 확인하자.

앞에서 정의한 Entry point 인터페이스를 사용하는 방법은 다음과 같다.

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

추가적으로 EntryPointAccessors의 메서드는 표준 안드로이드 컴포넌트로부터 Entry point들을 가져오는 것에 대해 더 적합하며 타입에 안전하다.

모범 사례 : Entry point 인터페이스는 어디에 정의해야 할까?

만약 Hilt를 사용하지 않는 라이브러리 그리고 Dagger로부터 Foo 클래스를 필요로 하는 클래스 인스턴스화를 구현해야 한다면, Entry point 인터페이스는 사용하고 있는 클래스 또는 Foo와 함께 정의 되어야 할까?

일반적으로 해당 클래스가 Foo 아닌 Entry point 인터페이스를 필요로 한다면, Entry point를 사용하고 있는 클래스와 함께 정의되는 것이 맞다. 해당 클래스에 더 많은 의존성이 필요하면, 추가 메서드를 Entry point 인터페이스에 쉽게 추가하여 가져 올 수 있다. 본질적으로 Entry point 인터페이스는 해당 클래스의 @Inject 생성자의 위치에서 기능을 하게 된다. 대신 Entry point가 Foo에 정의 된 경우 다른 사람들이 Foo를 주입하거나 Entry point 인터페이스를 사용해야 하는지 혼동 될 수 있다. 또한 나중에 다른 의존성이 필요한 경우 더 많은 Entry point 인터페이스가 추가 될 수 있다.

모범 사례

```
// Dagger에서 인스턴스화되지 않기 때문에 @Inject가 없다.
class MyClass : NonHiltLibraryClass() {

    @EntryPoint
    @InstallIn(ApplicationComponent::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()
    }
}
```

안 좋은 사례

```
@Module
@InstallIn(ApplicationComponent::class)
object FooModule {
    @Provides
    fun provideFoo(): Foo {
        return Foo()
    }

    @EntryPoint
    @InstallIn(ApplicationComponent::class)
    interface FooInterface {
        fun getFoo(): Foo
    }
}
```

5.6 Core APIs - Custom Components

사용자화 컴포넌트가 필요한가?

Hilt는 개발자를 위해 관리되는 안드로이드용으로 미리 정의된 컴포넌트들을 가지고 있다. 하지만 표준 Hilt 컴포넌트가 객체의 수명과 맞지 않거나 특정 기능을 필요로 하는 상황들이 생기기 마련이다. 이런 경우에는 사용자화 컴포넌트가 필요하다. 그러나 사용자화 컴포넌트를 생성하기 전에 논리적으로 사용자화 컴포넌트가 반드시 필요한 것인지 고려해야한다.

예를 들면 백그라운드 작업을 고려해볼 때, 작업이 스코프에 대해 타당하고 명확한 수명을 가지고 있다고 가정하자. 또한 해당 작업을 위해 요청된 객체들이 있는 경우 Dagger에 있는 바인딩을 작업 매개변수 일부로 전달할 수 있다. 하지만 대부분의 백그라운드 작업은 컴포넌트가 실제로 필요하지 않거나 단순히 몇개의 객체를 전달하기 위해 복잡도만 늘어나게 된다. 사용자화 컴포넌트를 추가하기 전에 다음과 같은 문제점들을 집고 넘어가자.

- 각 컴포넌트/스코프에 복잡성이 더해진다(cognitive overhead)
- 사용자화 컴포넌트와 스코프는 조합론적으로 봤을 때 그래프를 복잡하게 만든다. (예: 컴포넌트가 개념적으로 ViewComponent의 하위 컴포넌트라면, ViewComponent 및 ViewWithFragmentComponent에 대해 두 컴포넌트를 추가해야한다)

- 컴포넌트들은 단지 하나의 상위 컴포넌트만 가질 수 있다. 컴포넌트 계층은 다이아몬드형으로 구성될 수 없다. 더 많은 컴포넌트를 생성하는 것은 다이아몬드 의존 관계를 필요로 하는 상황의 가능성을 증대시킨다. 불행히도 이런 다이아몬드 의존 관계에 대한 문제의 솔루션은 없고, 이를 예측하거나 회피하는 것도 어렵다.
- 사용자화 컴포넌트는 표준화된 규격에 반하기 때문에 더 많은 사용자화 컴포넌트가 사용될 수록 공용 라이브러리 사용은 더 어려워지게 된다.

이를 염두에 두고 사용자화 컴포넌트가 필요한지 결정하는데 사용해야 하는 몇가지 기준이 있다.

- 컴포넌트는 명확한 수명을 가져야 한다.
- 컴포넌트의 개념은 이해하기 쉽고 넓게 적용 가능해야 한다. Hilt 컴포넌트는 앱에 전반적으로 적용되므로 개념적으로 모든 곳에 적용 가능해야 한다. 전반적으로 이해하기 쉽다는 것은 복잡성 문제를 해결한다. (Issues with cognitive overhead)
- Hilt가 아닌(일반적인 Dagger) 컴포넌트로 충분한지 고려해야 한다. 제한된 목적을 갖는 컴포넌트의 경우 때로는 Hilt 이외의 컴포넌트를 사용하는 것이 좋다. 예를 들어, 단일 백그라운드 작업을 표현하는 프로덕션 컴포넌트가 있다고 생각해보자. Hilt 컴포넌트들은 코드가 분리형 / 모듈형 코드로 작업해야 하는 상황에 탁월하다. 컴포넌트가 실제로 확장성을 갖지 못하는 경우 사용자화 컴포넌트를 사용하는 것이 맞지 않을 수 있다.

사용자화 컴포넌트의 제약

사용자화 컴포넌트는 현재 다음과 같은 제약조건을 가지고 있다.

- 사용자화 컴포넌트는 반드시 `ApplicationComponent`의 직접적 또는 간접적으로 하위 컴포넌트여야 한다.
- 사용자화 컴포넌트는 표준 컴포넌트 사이에 추가되면 안된다. 예를 들어 사용자화 컴포넌트는 `ActivityComponent`와 `FragmentComponent` 사이에 추가 될 수 없다.

사용자화 Hilt 컴포넌트 추가하기

사용자화 Hilt 컴포넌트를 생성하기 위해서는 `@DefineComponent`와 함께 클래스를 생성해야 한다. 이는 `@InstallIn` 어노테이션에서 사용되는 클래스가 된다.

상위 컴포넌트는 `@DefineComponent` 어노테이션의 멤버 값으로 정의 되어야 한다. `@DefineComponent` 클래스는 스코프 어노테이션을 추가하여 객체가 해당 컴포넌트의 수명과 함께 하도록 할 수 있다.

예를 들면 다음과 같다.

```
@DefineComponent(parent = ApplicationComponent::class)
interface MyCustomComponent
```

빌더 인터페이스가 반드시 정의되어야 한다. 빌더 인터페이스는 상위 컴포넌트에서 주입 가능해야하고, 빌더 인터페이스를 통해 컴포넌트의 인스턴스를 생성할 수 있어야 한다. 이런 사용자 컴포넌트들은 일단 인스턴스가 생성되고 나면 적절한 시점에 컴포넌트 인스턴스를 유지하거나 해제하는 작업이 된다.

빌더 인터페이스는 `@DefineComponent.Builder` 어노테이션을 사용하여 정의된다. 빌더는 반드시 `@DefineComponent` 어노테이션을 갖는 타입을 반환하는 메서드 하나를 가져야 한다. 빌더는 `@BindsInstance` 메서드와 같이 일반적인 Dagger 컴포넌트 빌더의 메서드를 추가적으로 가질 수 있다.

```
@DefineComponent.Builder
interface MyCustomComponentBuilder {
    fun fooSeedData(@BindsInstance Foo foo): MyCustomComponentBuilder
    fun build(): MyCustomComponent
}
```

@DefineComponent.Builder 클래스가 @DefineComponent내에 내재 될 수 있지만, 일반적으로는 별도의 클래스로 사용하는 것이 좋다. @HiltAndroidApp 클래스 또는 @HiltAndroidTest 클래스가 전이적 의존성인 경우 다른 클래스로 분리 될 수 있다. 많은 곳에서 @InstallIn을 통해 @DefineComponent 클래스가 참조되기 때문에 빌더의 의존성이 컴포넌트에 설치된 모든 모듈에 전이 의존성이 되지 않도록 빌더를 분리하는 것이 좋다.

과도한 의존성을 피하기 위해 @DefineComponent 인터페이스에서는 메서드를 사용할 수 없고, 대신에 Entry point를 통해 Dagger 객체에 접근해야 한다.

```
@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()
        // 필요하다면 컴포넌트 인스턴스를 이곳에서 멤버 변수로 관리 할 수 있다.
    }
}
```

6.1 Testing - Testing 개요

소개



현재 Hilt는 안드로이드 Instrumentation 과 Robolectric 테스트만 지원한다. Hilt는 vanilla JVM 테스트에서는 사용할 수 없지만 평소와 같이 이러한 테스트를 작성하지 못하게 막지는 않는다.

Hilt는 안드로이드 테스트에 의존성 주입의 기능을 제공하여 테스트를 더 쉽게 만든다. Hilt는 테스트를 통해 Dagger 바인딩에 쉽게 접근하거나 새로운 바인딩을 제공하여 바인딩을 대체 할 수 있다. 각 테스트마다 Hilt 컴포넌트 세트를 가져오므로 테스트 레벨별로 바인딩을 쉽게 사용자화 할 수 있다.

이 문서에 설명된 많은 테스트 API들과 기능은 훌륭한 테스트를 수행하는 것에 대한 무언의 철학을 기반으로 하고 있다. Hilt의 테스트 철학에 대한 자세한 내용은 '10.2 테스트 철학'을 참고하자.

테스트 설정하기



Gradle 사용자는 '3. Gradle 설정하기' 편을 참고하여 Hilt 테스트 빌드 의존성을 먼저 추가해야 한다.

테스트에서 Hilt를 사용하기 위해서는:

1. 테스트 클래스에 `@HiltAndroidTest`를 추가한다.
2. `HiltAndroidRule` 테스트 룰을 추가한다.
3. 안드로이드 Application 클래스를 위해 `HiltTestApplication`을 사용한다.

예를 들면,

```
@HiltAndroidTest
class FooTest {
    @get:Rule val rule = HiltAndroidRule(this)
    ...
}
```

앞에서 언급한 3번째에서, 테스트를 위한 Application 클래스를 설정하는 것은 테스트가 Robolectric 테스트인지 Instrumentation 테스트인지에 따라 다르다. 자세한 내용은 '6.2 Robolectric 테스트하기' 또는 '6.3 Instrumentation 테스트하기'를 참고하자. 이 장에서 다루는 내용은 Robolectric 그리고 Instrumentation 테스트 둘다 적용 가능한 부분을 다룬다.

바인딩에 접근하기

테스트는 종종 Hilt 컴포넌트로부터 바인딩을 요청하게 된다. 이 섹션에서는 각기 다른 컴포넌트의 바인딩을 어떻게 요청하는지 살펴본다.

ApplicationComponent 바인딩에 접근하기

ApplicationComponent 바인딩은 `@Inject`를 사용하여 필드에 직접적으로 주입이 가능하다. `HiltAndroidRule#inject()`를 호출하기 전까지 주입은 일어나지 않는다.

```
@HiltAndroidTest
class FooTest {

    @get:Rule HiltAndroidRule hiltRule = HiltAndroidRule(this)

    @Inject foo: Foo@Test

    fun testFoo() {
        assertThat(foo).isNull()
        hiltRule.inject()
        assertThat(foo).isNotNull()
    }
}
```

ActivityComponent 바인딩에 접근하기

ActivityComponent 바인딩을 요청하는 것은 Hilt를 사용하는 Activity의 인스턴스를 필요로 한다. 한가지 방법은 작성한 테스트에 필요한 바인딩에 대해 `@Inject` 필드를 포함하는 내재된 Activity를 정의하는 것이다. 그런 다음에 테스트 Activity 인스턴스를 생성하여 바인딩을 얻을 수 있다.

```

@HiltAndroidTest
class FooTest {

    @AndroidEntryPoint
    class TestActivity : AppCompatActivity() {
        @Inject foo: Foo
    }

    ...
    val foo = testActivity.foo
}

```

다른 방법으로, 테스트에서 Hilt를 사용하는 Activity 인스턴스를 가지고 있다면, EntryPoint를 사용하여 어떤 ActivityComponent 바인딩도 얻을 수 있게 된다.

```

@HiltAndroidTest
class FooTest {

    @EntryPoint
    @InstallIn(ActivityComponent::class)
    interface FooEntryPoint {
        fun getFoo() : Foo
    }

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

```

FragmentComponent 바인딩 접근하기

ActivityComponent 바인딩에 접근하는 방법과 유사하게 FragmentComponent 바인딩에 접근할 수 있다. 주요 차이점은 FragmentComponent 바인딩에 접근하는 것은 Hilt를 사용하는 Activity와 Fragment 인스턴스를 모두 필요로 한다는 점이다.

```

@HiltAndroidTest
class FooTest {

    @AndroidEntryPoint
    class TestFragment : Fragment() {
        @Inject foo: Foo
    }

    ...
    val foo = testFragment.foo
}

```

다른 방법으로, 테스트에서 Hilt를 사용하는 Fragment 인스턴스를 가지고 있다면, EntryPoint를 사용하여 어떤 FragmentComponent 바인딩도 얻을 수 있게 된다.

```

@HiltAndroidTest
class FooTest {

    @EntryPoint
    @InstallIn(FragmentComponent::class)
    interface FooEntryPoint {
        fun getFoo() : Foo
    }

```

```

    }

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

```

⚠ Hilt는 Activity 클래스를 지정할 방법이 없고 Hilt를 사용하는 Fragment는 Hilt를 사용하는 Activity에 포함되어야 하기 때문에 현재는 FragmentScenario를 지원하고 있지 않다. 한 가지 해결책은 Hilt를 사용하는 Activity를 실행하고 Fragment를 붙이는 것이다.

바인딩 추가하기

테스트는 애플리케이션의 프로덕션 빌드에 포함되지 않은 추가 Dagger 바인딩을 추가적으로 제공해야 할 수도 있다. 또한 테스트들은 테스트마다 다른 값으로 동일한 바인딩을 제공해야 할 수도 있다. 이 섹션에서는 Hilt를 사용하여 테스트에 바인딩을 제공하는 방법에 대해 설명한다.

내재된 모듈

보통 `@InstallIn` 모듈은 모든 테스트의 Hilt 컴포넌트에서 설치된다. 하지만 바인딩이 특정 테스트에서만 설치되어야 한다면 테스트 클래스내에 `@InstallIn` 모듈을 내재하여 수행 할 수 있다.

```

@HiltAndroidTest
class FooTest {

    // 내재된 모듈은 테스트 밖의 Hilt 컴포넌트에서만 설치된다
    @Module
    @InstallIn(ApplicationComponent::class)
    object FakeBarModule {
        @Provides
        fun provideBar() = Bar()
    }
    ...
}

```

따라서 다른 구현으로 동일한 바인딩을 제공해야 하는 다른 테스트가 있는 경우 중복 바인딩 충돌없이 수행할 수 있다

Hilt는 정적으로 내재된 `@InstallIn` 모듈 외에도 테스트 내에서 내부 (비-정적) `@InstallIn` 모듈을 지원한다. 내부 모듈을 사용하면 `@Provides` 메소드가 테스트 인스턴스의 멤버를 참조 할 수 있다.

⚠ Hilt는 생성자 매개 변수가 있는 `@InstallIn` 모듈을 지원하지 않는다.

@BindValue

간단한 바인딩, 특히 테스트 메서드에서 접근해야 하는 바인딩의 경우, 모듈과 바인딩을 프로비전 메서드를 생성하는 보일러플레이트 코드를 회피하기 위해 Hilt는 편리한 어노테이션을 제공한다.

`@BindValue`는 쉽게 테스트에서 필드를 Dagger 그래프에 바인딩 할 수 있는 어노테이션이다. 이를 사용하기 위해, `@BindValue` 어노테이션을 필드에 추가하면 선언된 한정자 그리고 필드 타입과 함께 바인딩 된다.

```
@HiltAndroidTest
class FooTest {
    ...
    @BindValue fakeBar: Bar = new FakeBar()
}
```

바인딩의 스코프가 필드에 연결되고 테스트에 의해 제어되므로 @BindValue는 스코프 어노테이션의 사용을 지원하지 않는다. 필드 값은 요청 될 때마다 쿼리되므로 테스트에서 필요한 대로 변경 될 수 있다. 바인딩을 실질적으로 싱글톤으로 만드려면 필드가 테스트당 한번만 설정되도록 해야한다. 예를들어 필드의 이니셜라이저 또는 테스트의 @Before 메소드에서 필드 값을 설정한다.

마찬가지로 Hilt에는 @IntoSet, @ElementIntoSet 및 @IntoMap을 각각 지원하기 위해 @BindValueIntoSet, @BindElementsIntoSet 및 @BindValueIntoMap을 이용한 멀티 바인딩에 대한 편리한 어노테이션도 있다. (@BindValueIntoMap을 사용하려면 필드에 @MapKey 어노테이션을 추가해야 한다.)

주의사항

@BindValue를 사용하거나 비-정적인 내부 모듈과 함께 ActivityScenarioRule을 사용할 때는 주의해야 한다. ActivityScenarioRule은 @Before 메서드를 호출하기 전에 Activity를 생성하므로, @BindValue 필드가 @Before(또는 그 이후)에 초기화 된다면 필드가 초기화 되지 않은 상태에서 바인딩을 Activity에 주입할 가능성이 있다. 이런 경우를 피하려면 @BindValue를 필드에서 바로 초기화 하면 된다.

바인딩 교체하기

@UninstallModules

일반적인 테스트 유즈 케이스에서는 프로덕션용 바인딩을 테스트용 바인딩으로 교체할 필요성이 있다. 예를 들면 Fake 또는 Mock이 있는데, Hilt 테스트에서 프로덕션용 바인딩은 @UninstallModules를 사용하여 포함된 프로덕션 모듈을 먼저 제거하여 프로덕션 바인딩을 교체 할 수 있다. 그런 뒤 테스트에서 새로운 바인딩을 제공한다.

```
@UninstallModules(ProdFooModule::class)
@HiltAndroidTest
class FooTest {
    ...
    @BindValue fakeFoo: Foo = new FakeFoo()
}
```

@UninstallModules는 주어진 테스트와 관련하여 해당 모듈에서 @InstallIn 어노테이션을 제거하는 것과 같다. Hilt는 개별 바인딩 제거를 직접 지원하지 않지만 특정 모듈에 단일 바인딩만 포함하면 실질적으로 가능하다.

사용자와 테스트 애플리케이션

모든 Hilt 테스트는 반드시 안드로이드 Application 클래스 대용으로 HiltTestApplication을 사용해야 한다. Hilt는 기본 테스트용 Application인 HiltTestApplication을 제공한다. HiltTestApplication은 MultiDexApplication을 확장한 클래스다. 그러나 다른 기본 클래스를 사용해야 해서 테스트하기 힘든 몇 가지 경우가 있다.

@CustomTestApplication

만약 테스트에서 사용자화 기본 클래스가 필요한 경우에는 @CustomTestApplication 어노테이션을 통해 주어진 Application 클래스를 확장하는 Hilt용 테스트 애플리케이션을 생성할 수 있다.

@CustomTestApplication을 사용하려면 클래스 또는 인터페이스에 @CustomTestApplication 어노테이션을 추가하고 어노테이션 값으로 기본 클래스를 명시하면 된다.

```
// MyCustom_Application.class를 생성하게 된다.  
@CustomTestApplication(MyBaseApplication::class)  
interface MyCustom
```

앞의 예제는 Hilt가 MyCustom_Application이라는 MyBaseApplication을 확장한 클래스를 생성한다. 일반적으로 생성된 Application 이름은 어노테이션이 달린 클래스의 이름 뒤에 _Application을 붙인다. 어노테이션이 달린 클래스가 내재된 클래스라면 클래스의 이름은 바깥 클래스의 이름도 밀줄표시로 분리하여 포함하게 된다. 어노테이션이 달린 클래스는 생성된 Application의 이름 외에는 관련이 없다.

모범 사례

모범 사례로 @CustomTestApplication 사용을 피하고 대신에 테스트에서 HiltTestApplication을 사용하자. 일반적으로 Activity, Fragment 등이 포함된 상위 클래스와 독립적으로 만들면 나중에 쉽게 구성하고 재사용 할 수 있다.

하지만 사용자화 기본 Application을 반드시 사용해야 한다면 프로덕션 생명주기에 대해 알아야 할 몇가지 미묘한 차이점이 있다.

첫번째 차이점은 Instrumentation 테스트는 같은 Application 인스턴스를 매 테스트마다 사용하게 된다. 그러므로 사용자화 테스트 Application을 사용할 때는 테스트 전반에 걸쳐 뜻하지 않은 구멍이 생기기 마련이다. Application 상태에 의존적이거나 상태를 저장하는 테스트를 만들지 않는 것이 중요하다.

또 다른 차이점은 테스트 Application에서 Hilt 컴포넌트는 super#onCreate에서 생성되지 않는다. 이런 제약은 주로 Hilt의 일부 기능이(예: @BindValue) 테스트 인스턴스에 의존하기 때문에 Application#onCreate가 호출 될 때까지 테스트에서 사용할 수 없기 때문에 발생한다. 따라서 프로덕션용 Application과 달리 사용자화 기본 Application은 Application#onCreate 중에 컴포넌트를 호출하지 않아야 한다. 여기에는 Application을 멤버로 주입하는 것이 포함된다. 이런 문제를 방지하기 위해서 Hilt는 기본 Application에서 의존성 주입하는 것을 허용하고 있지 않다.

Hilt 규칙 순서

여러가지 테스트 규칙을 사용한다면 HiltAndroidRule이 Hilt 컴포넌트에 접근을 필요로 하는 다른 테스트 규칙 전에 실행되도록 해야 한다. 예를 들면 ActivityScenarioRule은 Activity#onCreate를 호출한다(Hilt를 사용하는 Activity의 경우 Hilt 컴포넌트가 의존성 주입을 수행해야 한다). 따라서 ActivityScenarioRule은 컴포넌트가 적절히 초기화 된 것을 보장하기 위해 HiltAndroidRule 후에 실행되어야 한다.



만약 4.13 버전 미만의 JUnit을 사용한다면 RuleChain을 사용하여 순서를 명시하자.

```

@HiltAndroidTest
class FooTest {
    // ActivityScenarioRule을 실행하기 전에 Hilt 컴포넌트가 초기화되었는지 확인한다.
    @get:Rule(order = 0)
    val hiltRule = HiltAndroidRule(this);

    @get:Rule(order = 1)
    val scenarioRule = ActivityScenarioRule(MyActivity::class.java)
}

```

6.2 Testing - Robolectric Testing

테스트 Application 설정하기

Hilt의 테스트 API는 특정 테스트 환경에 무관하게 설계되었다. 그러나 테스트에서 Application 클래스를 설정하기 위한 방법은 Robolectric 또는 안드로이드 Instrumentation 테스트를 사용 중인지 여부에 따라 다르다.

Robolectric 테스트에서는 국소적으로 `@Config`를 사용하거나 전역적으로 `robolectric.properties`를 사용하여 Application을 설정할 수 있다. Hilt를 위한 테스트에서 Application은 반드시 HiltTestApplication 또는 사용자화 테스트 Application 중 하나가 되어야 한다.



이 설정은 Hilt에 특정되지 않는다. [Robolectric 공식 문서](#)를 통해 더 많은 정보를 참조하자.

@Config 사용하기

Hilt용 Application 클래스는 국소적으로 `@Config` 어노테이션을 사용하여 설정될 수 있다. Application 클래스를 설정하기 위해서는 테스트 (또는 테스트 메서드)에 `@Config` 어노테이션을 추가하고 어노테이션의 값으로 원하는 Application 클래스를 지정하면 된다.

```

@HiltAndroidTest
@Config(application = HiltTestApplication::class)
class FooTest {...}

```

robolectric.properties 사용하기

Hilt 애플리케이션은 robolectric.properties 파일을 사용하여 전역적으로 설정될 수 있다. Application 클래스를 설정하기 위해서는 적절한 resource 패키지에 `robolectric.properties` 파일을 생성하고 Hilt 테스트 Application을 설정하면 된다.

```

application=dagger.hilt.android.testing.HiltTestApplication

```

안드로이드 Instrumentation 테스트와 함께 `@Config` 어노테이션을 사용할 수 없을 때 이러한 접근 방식을 사용하면, 테스트를 Robolectric 그리고 안드로이드 Instrumentation 환경에서 실행할 때 유용하다.

6.3 Testing - Instrumentation Testing

테스트 Application 설정하기

Hilt의 테스트 API는 특정 테스트 환경에 무관하게 설계되었다. 그러나 테스트에서 Application 클래스를 설정하기 위한 방법은 Robolectric 또는 안드로이드 Instrumentation 테스트를 사용 중인지 여부에 따라 다르다.

안드로이드 Instrumentation 테스트에서 AndroidJUnitRunner를 확장하는 사용자화 테스트 러너(runner)를 사용하여 Application이 설정될 수 있다. 러너를 사용하는 Application을 설정하기 위해서 newApplication 메서드를 재정의 하고 Application 클래스 이름을 넘기면 된다. Hilt 테스트에서는 Application은 반드시 HiltTestApplication 또는 사용자화 테스트 Application 중 하나가 되어야 한다.

```
package my.pkg

class MyTestRunner extends AndroidJUnitRunner {
    override fun newApplication(
        cl: ClassLoader,
        appName: String,
        context: Context) : Application {
        return super.newApplication(
            cl, HiltTestApplication::class.java.getName(), context)
    }
}
```

추가적으로 주어진 Gradle 모듈에 대한 testInstrumentationRunner는 반드시 build.gradle 파일에서 설정되어야 한다.

```
android {
    defaultConfig {
        testInstrumentationRunner "my.pkg.MyTestRunner"
    }
}
```

7.1 Migration - Guide

Hilt로 마이그레이션 하는 것은 코드베이스 상태와 코드베이스가 따르는 관행 또는 패턴에 따라 매우 다양할 수 있다. 이 페이지는 앱 마이그레이션시 발생할 수 있는 몇가지 일반적인 문제에 대한 권고사항을 제공한다. 이 페이지는 여러분이 이미 기본 Hilt API를 이해하고 있다고 가정한다. 그렇지 않은 경우 Hilt를 위한 'Quick Start'를 먼저 참조하자. 이 페이지는 또한 Dagger에 대한 일반적인 이해를 가정하고 있다. 왜냐하면 이 페이지는 Dagger를 코드베이스로 이미 사용하고 있어 이를 마이그레이션 하려는 사람들에게만 필요하기 때문이다. 코드베이스가 Dagger를 사용하지 않는 경우, Dagger 설정으로부터 마이그레이션이 필요 없는 'Quick Start' 가이드 문서를 통해 앱에 Hilt를 추가하자.



리팩토링 Tip : 클래스 코드를 수정할 때, 사용하지 않거나 더이상 존재하지 않는 import들이 파일에서 삭제 되었는지 확인하자.

0. 마이그레이션 계획하기

Hilt로 마이그레이션 할 때, 작업을 단계별로 구성하고 싶을 것이다. 이 가이드는 대부분의 경우에 알맞은 일반적인 접근 방식을 제시하지만 경우에 따라서 다를 수는 있다. 권장하는 접근 방식은 Application 또는

@Singleton 컴포넌트에서 시작여 점진적으로 확장해 나가는 방법이다. Application 그리고 @Singleton 이후에 Activity를 마이그레이션 한 후 Fragment들을 작업하자. 전반적으로 점진적인 마이그레이션을 해야 한다. 상대적으로 적은 코드베이스가 있는 경우에도 마이그레이션을 점진적으로 수행하면 단계별로 진행 상황을 확인할 수 있다.

컴포넌트 계층 비교하기

가장 먼저해야 할 일은 현재 컴포넌트 계층 구조를 Hilt의 계층 구조와 비교하는 것이다. 어떤 컴포넌트를 어떤 Hilt 컴포넌트에 매핑할지 결정해야 한다. 이 방법은 비교적 간단해야 하지만, 명확한 매핑이 없는 경우 Dagger 컴포넌트를 구성하듯 사용자화 컴포넌트를 구성할 수 있다. 이러한 컴포넌트는 Hilt 컴포넌트의 하위 항목이 될 수 있다. 그러나 Hilt는 컴포넌트 계층 사이에 삽입하는 것은 허용하지 않는다 (예: Hilt 컴포넌트의 상위컴포넌트를 변경 하는 것). 이 가이드 다음에 나올 내용인 '사용자화 컴포넌트' 섹션을 확인하자. **이 가이드의 나머지 부분에서는 컴포넌트가 모든 Hilt 컴포넌트에 직접 매핑되는 마이그레이션을 한다고 가정한다.**

또한 코드에서 컴포넌트 의존성을 사용하는 경우 다음에 나올 '컴포넌트 의존성' 섹션을 먼저 읽도록 하자. **이 가이드 문서의 나머지 부분에서는 서브 컴포넌트를 사용한다고 가정한다.**

dagger.android의 @ContributesAndroidInjector를 사용 중이고 컴포넌트 계층 구조가 확실하지 않은 경우, 계층 구조를 Hilt 컴포넌트와 대략적으로 맞추도록 하자.

Hilt가 언제 클래스들을 주입하는지 알자

Hilt가 각 Android 클래스에 대한 클래스를 언제 주입하는지 '컴포넌트의 수명'에서 확인할 수 있다. 이것들은 코드가 현재 주입하는 위치와 유사해야 하지만 그렇지 않은 경우 코드에 차이가 있는 경우를 염두에 두자.

마이그레이션 개요

마이그레이션이 끝나면 코드를 다음과 같이 변경해야 한다:

- 모든 @Component / @Subcomponent (또는 dagger.android @ContributesAndroidInjector를 사용하는 경우) 해당 사용방법은 제거해야 한다.
- 모든 @Module 클래스는 @InstallIn 어노테이션을 추가해야 한다.
- 모든 Application / Activity / Fragment / View / Service / BroadcastReceiver 클래스에는 @AndroidEntryPoint 어노테이션을 추가해야 한다.
- 컴포넌트를 인스턴스화 하거나 전파하는 코드 (예: 컴포넌트를 노출시키기 위한 Activity의 인터페이스)를 제거해야 한다.
- 모든 dagger.android에 대한 참조를 제거해야 한다.

1. Application 마이그레이션 하기

가장 먼저 변경해야 할 것은 Application 및 @Singleton 컴포넌트를 생성되는 Hilt의 ApplicationComponent로 마이그레이션하는 것이다. 이를 위해 먼저 현재 컴포넌트에 설치된 모든 것이 Hilt의 ApplicationComponent에 설치되어 있는지 확인해야 한다.

컴포넌트 마이그레이션하기

Application을 마이그레이션 하기 위해, 기존에 존재하던 @Singleton 컴포넌트의 모든 것을 ApplicationComponent로 마이그레이션 한다.

a. 모듈 다루기

먼저 모든 모듈을 `ApplicationComponent`에 설치해야 한다. 컴포넌트에 현재 설치된 각 모듈에 `@InstallIn` (`ApplicationComponent.class`) 어노테이션을 추가 한다. 많은 모듈이 있는 경우 지금 모든 모듈을 변경하는 대신 모든 현재 모듈을 포함하는 단일로 통합하는 `@Module` 클래스를 만들고 설치할 수 있다. 그러나 이는 `@UninstallModules`와 같은 Hilt 기능을 최대한 활용하려면 나중에 통합된 모듈을 분리해야 하기 때문에 임시 솔루션 일 뿐이다.

```
// 이 컴포넌트는 이렇게 시작해서
@Component(modules = [
    FooModule::class,
    BarModule::class,
    ...
])
interface MySingletonComponent {
}

// 다음과 같이 바뀐다.
@InstallIn(ApplicationComponent::class)
@Module(includes = [
    FooModule::class,
    BarModule::class,
    ...
])
interface AggregatorModule {}
```

⚠ `@InstallIn` 어노테이션을 달지 않은 모듈은 Hilt에서 사용되지 않는다. 어노테이션이 없는 모듈이 발견되면 기본적으로 Hilt에서 오류가 발생하지만 이 오류는 비활성화 할 수 있다.

b. 확장한 인터페이스 또는 모듈 다루기

`@EntryPoint`를 사용하여 현재 컴포넌트를 확장하는 모든 인터페이스에 대해 비슷한 처리를 할 수 있다.

컴포넌트의 인터페이스는 일반적으로 주입 방법을 추가하거나 바인딩 또는 서브컴포넌트와 같은 타입에 접근할 때 사용된다. Hilt로 마이그레이션이 완료되면, Hilt가 이를 생성하거나 Hilt 도구로 대체하기 때문에 이 중 많은 것들이 필요하지 않게 된다. 그러나 마이그레이션을 위해, 이 섹션에서는 코드가 계속 작동하도록 현재 동작을 유지하는 방법에 대해 설명한다. 하지만 이러한 모든 방법을 살펴보고, 마이그레이션을 진행하는 것이 여전히 필요한지 확인하자.

`@EntryPoint`로 모든 것을 옮기기

`@EntryPoint` 및 `@InstallIn(ApplicationComponent.class)`을 컴포넌트를 확장하는 인터페이스에 추가한다. 인터페이스가 많은 경우 하나로 통합한 인터페이스를 작성하여 모듈처럼 모든 인터페이스를 수집하자. 컴포넌트 인터페이스에 직접 정의된 모든 메소드는 통합된 인터페이스 또는 통합된 인터페이스를 확장한 인터페이스로 옮길 수 있다.

예제 코드:

```
// 이 컴포넌트는 이렇게 시작해서
@Component
@Singleton
interface MySingletonComponent : FooInjector, BarInjector {
    fun inject(myApplication: MyApplication)

    fun getFoo() : Foo
}
```

```
// 다음과 같이 바뀐다.
@InstallIn(ApplicationComponent::class)
@EntryPoint
interface AggregatorEntryPoint : FooInjector, BarInjector {
    // 이는 예시로 이동시킨 것이다. 그러나 다음에 나올 내용에서는
    // inject 메서드를 제거할 수 있음을 보여준다.
    fun inject(myApplication: MyApplication)

    fun getFoo() : Foo
}
```

Inject 메서드

Hilt는 내부에서 Application 클래스 주입을 처리하므로 Application에 대한 주입 메소드가 있으면 제거 할 수 있다. @AndroidEntryPoint를 사용하도록 나중에 마이그레이션되므로 다른 Android 타입의 주입 메소드도 결국 제거해야 한다.

```
@Component
@Singleton
interface MySingletonComponent {
    // Hilt는 Application을 관리하므로 이 부분은 삭제될 수 있다.
    fun inject(myApplication: MyApplication)

    // @AndroidEntryPoint를 사용한다면 이 부분은 삭제될 수 있다.
    fun inject(fooActivity: FooActivity)
}
```

인터페이스에 접근하기

코드는 컴포넌트를 직접적으로 반환하거나 인터페이스 타입 중 하나를 반환하는 메서드를 가지므로 다른 코드는 주입 메서드 또는 접근자 메서드를 얻을 수 있게 된다. 마이그레이션 할 때 이 코드가 계속 동작하려면 EntryPoints 클래스를 사용하여 참조 할 수 있다. 마이그레이션이 계속됨에 따라 이러한 메소드를 제거하고 호출 코드가 Hilt EntryPoints API를 직접 사용하도록 해야 한다.

```
// 이와 같은 코드로 시작 한다면
class MyApplication : Application() {
    fun component(): MySingletonComponent {
        return component
    }
}

// 통합적인 Entry point 를 추가한 후, 코드는 다음과 같은 형태를 갖는다.:
@InstallIn(ApplicationComponent::class)
@EntryPoint
interface AggregatorEntryPoint : LegacyInterface, ... {
}

@HiltAndroidApp
class MyApplication : Application() {
    // 반환형이 AggregatorEntryPoint로 변경되었지만,
    // 이전 컴포넌트가 사용하던 인터페이스를 모두 구현하므로 괜찮다.
    fun component(): AggregatorEntryPoint {
        // EntryPoints를 사용하여 AggregatorEntryPoint의 인스턴스를 가져온다.
        return EntryPoints.get(this, AggregatorEntryPoint::class.java)
    }
}
```

c. 스코프

Hilt로 컴포넌트를 마이그레이션 할 때, Hilt 스코프 어노테이션을 사용하는 바인딩으로 마이그레이션 하는 작업 또한 필요하다. ApplicationComponent의 경우 @Singleton 어노테이션을 사용한다. '컴포넌트 수명'에서 어떤 어노테이션이 어떤 컴포넌트에 해당하는지 확인할 수 있다. @Singleton을 사용하지 않고 자체 스코프 어노테이션이 있는 경우, 스코프 별칭(scope alias)를 사용하여 해당 어노테이션이 Hilt 스코프 어노테이션과 동등하다고 Hilt에게 알려줄 수 있다. 이렇게 하면 프로세스 후반에 스코프 어노테이션을 마이그레이션 하고 제거 할 수 있다.

d. 컴포넌트 인자 다루기

컴포넌트 초기화 코드가 숨겨져 있으므로, Hilt 컴포넌트는 컴포넌트 인자를 취할 수 없다. 일반적으로 Application 인스턴스 (또는 다른 컴포넌트의 경우 Activity / Fragment 인스턴스)를 Dagger 그래프로 가져 오는데 사용된다. 이러한 경우 '컴포넌트가 제공하는 기본 바인딩'에 나열된 Hilt의 사전 정의된 바인딩을 사용하도록 전환해야 한다.

컴포넌트가 빌더를 통한 모듈 인스턴스를 넘겨 받거나 @BindsInstance를 통해 어떤 인자를 갖는다면, 이를 다루기 위해 '컴포넌트 인자' 섹션을 읽도록 하자. 이러한 것들을 다루면 @Component.Builder 인터페이스를 사용하지 않게 되므로 삭제가 가능하다.

e. 통합적인 모듈 및 인터페이스 정리하기

통합적인 모듈 또는 Entry point를 사용한 경우, 결국에는 해당 모듈 및 Entry point 클래스를 제거해야 한다. 포함 된 모든 모듈과 구현된 인터페이스에 개별적으로 통합 모듈에 사용된 동일한 @InstallIn 어노테이션을 추가하여 진행할 수 있다.

```
@InstallIn(ApplicationComponent::class)
@Module(includes = [FooModule::class, ...])
interface AggregatorModule {
}

// 위에 있는 목록에서 FooModule을 제거하고, 직접적으로 @InstallIn 어노테이션을 추가 하자.
@InstallIn(ApplicationComponent::class)
@Module
interface FooModule {
}
```

애플리케이션에 Hilt 추가하기

이제 'Quick Start'에 설명 한대로 @HiltAndroidApp을 Application 클래스에 추가 할 수 있다. 그 외에도, 컴포넌트의 인스턴스를 작성하거나 저장하는 데 관련된 코드가 비어 있어야 한다. @Component 클래스와 @Component.Builder 클래스를 아직 삭제하지 않은 경우 삭제할 수 있다.

dagger.android의 Application

Application이 DaggerApplication으로 확장되거나 HasAndroidInjector를 구현하는 경우, 모든 dagger.android의 Activity / Fragment가 마이그레이션 될 때까지 이 코드를 유지해야 한다. 이는 마이그레이션의 마지막 단계 중 하나다. dagger.android의 이 부분은 의존성을 얻는 부분이 제대로 작동하는지 확인하기 위함이다 (예 : Activity가 자신에게 의존성을 주입하려고 할 때). 차이점은 이제 앞의 단계에서 제거된 컴포넌트 대신 Hilt ApplicationComponent로 만족한다는 것이다.

예를 들어, Hilt의 Activity와 dagger.android의 Activity를 모두 지원하는 마이그레이션 된 dagger.android의 Application은 다음과 같다.

```

@HiltAndroidApp
class MyApplication : HasAndroidInjector {
    @Inject
    lateinit var dispatchingAndroidInjector: DispatchingAndroidInjector<Object>

    override fun androidInjector() = dispatchingAndroidInjector
}

```

또는 DaggerApplication을 사용중인 경우 다음을 수행 할 수 있다. @EntryPoint 클래스는 Dagger 컴포넌트가 AndroidInjector<MyApplication>을 구현하도록 한다. 이것은 이전 Dagger 컴포넌트가 전에 수행했던 작업 일 수 있다.

```

@HiltAndroidApp
class MyApplication : DaggerApplication() {
    @EntryPoint
    @InstallIn(ApplicationComponent::class)
    interface ApplicationInjector : AndroidInjector<MyApplication>

    override fun applicationInjector(): AndroidInjector<MyApplication> {
        return EntryPoints.get(this, ApplicationInjector::class.java)
    }
}

```

다른 모든 dagger.android 사용법을 마이그레이션하고 이 코드를 제거 할 준비가 되었으면 단순히 Application을 확장하고 대체된 메소드 및 DispatchingAndroidInjector 클래스를 제거하도록 하자.

빌드 확인하기

이 시점에서 앱을 중지하고 빌드 / 실행 할 수 있는지 확인하자. 아마도 앱이 Hilt의 ApplicationComponent를 성공적으로 사용하고 있을 것이다.

2. Activity와 Fragment (그리고 다른 클래스들) 마이그레이션 하기

이제 Application이 Hilt를 지원하므로 Activity 마이그레이션을 시작한 다음 Fragment 마이그레이션을 진행 할 수 있다. 앱을 마이그레이션하는 동안 @AndroidEntryPoint Activity와 @AndroidEntryPoint를 사용하지 않는 Activity를 함께 사용하는 것은 괜찮다. Activity 내의 Fragment에 대해서도 마찬가지다. Hilt를 비-Hilt 코드와 혼합하는 유일한 제약조건은 상위 개념에 달려 있다. Hilt의 Activity는 Hilt의 Application에 포함되고, Hilt의 Fragment는 Hilt의 Activity에 포함되어야 한다. Fragment 마이그레이션을 진행하기 전에 모든 Activity에 대해 마이그레이션을 먼저 진행하는 것을 권장하지만, 문제가 있는 경우 선택적 주입(optional injection)으로 해당 제약조건을 완화 할 수 있다.

Activity와 Fragment를 마이그레이션 하는 것은 기술적 측면에서 Application 컴포넌트와 매우 유사하다. 현재 컴포넌트에서 모든 모듈을 가져와 @InstallIn 모듈과 함께 적절한 컴포넌트에 설치해야 한다. 마찬가지로 현재 컴포넌트의 확장 인터페이스를 모두 가져와 @InstallIn Entry point가 있는 적절한 컴포넌트에 설치하도록 한다. 자세한 내용은 위의 '컴포넌트 마이그레이션 하기' 섹션을 다시 살펴보자. 또한 Activity 및 Fragment에 대해 고려해야 할 몇 가지 추가사항도 다음 나올 섹션에서 살펴보자.



dagger.android의 @ContributesAndroidInjector를 사용하는 경우, '컴포넌트 마이그레이션 하기' 섹션을 따르면 @ContributesAndroidInjector의 모듈이 마이그레이션을 해야 하는 모듈이다. @EntryPoint로 마이그레이션 할 인터페이스는 없다.

단일로 된 컴포넌트의 차이점을 알자

Hilt의 설계 결정 중 하나는 모든 Activity에 단일 컴포넌트와 모든 Fragment에 단일 컴포넌트를 사용하는 것이다. 이 부분에 관심이 있다면 그 이유를 '단일 컴포넌트'편에서 자세히 확인할 수 있다. 이것이 중요한 이유는 dagger.android의 기본적인 설정과 같이 각 Activity에 대해 별도의 컴포넌트가 있는 경우 Hilt로 마이그레이션 할 때 컴포넌트를 단일 컴포넌트로 병합하기 때문이다. 코드베이스에 따라 문제가 발생할 수 있다.

흔하게 겪는 두가지 문제는 이렇다.

바인딩 충돌

두 Activity에서 동일한 바인딩 키를 다르게 정의한 경우에 발생한다. 병합되면 중복된 바인딩을 얻게 된다. 이는 Hilt의 전역적 바인딩 키 공간에 대한 제한 사항이며 단일 정의를 갖도록 해당 바인딩을 재정의해야 한다. 일반적으로 이는 나쁘지 않으며 주입된 Activity를 기반으로 논리를 수행하여 동작한다. 예제는 '컴포넌트 인자' 섹션을 참조하자.

특정 Activity 타입에 의존 하는 것

병합된 컴포넌트로 인해 컴포넌트가 BarActivity (또는 다른 Activity)에 사용될 때 FooActivity 바인딩을 충족 할 수 없으므로 FooActivity 또는 BarActivity에 대한 바인딩이 더 이상 의미가 없는 경우가 많다. 일반적으로 코드는 꼭 실제 하위 타입의 Activity에 의존하지 않으며 Activity 또는 FragmentActivity와 같은 공통 하위 타입만 필요하다. 보다 일반적인 유형을 사용하려면 하위 타입을 사용하는 코드를 리팩토링해야 한다. Hilt에서 자동으로 제공하지 않는 공통 하위 타입이 필요한 경우 캐스팅하여 바인딩을 제공 할 수 있지만 (예 : 'Component 인자') 조심해야 한다.

일반적인 하위 타입 사용법을 교체하는 예제:

```
// 이 클래스는 FragmentManager를 얻기 위해서 Activity만을 사용하고 있는데,
// FooActivity 대신 FragmentActivity 클래스를 사용할 수도 있다.
class Foo @Inject constructor(private val activity: FooActivity) {
    fun doSomething() {
        activity.getSupportFragmentManager()...
    }
}

// Hilt용으로 Foo 클래스를 마이그레이션할 때는 FragmentActivity로 변경한다.
class Foo @Inject constructor(private val activity: FragmentActivity) {
    fun doSomething() {
        activity.getSupportFragmentManager()...
    }
}
```

Activity/Fragment에 Hilt 추가하기

이제 'Quick Start' 가이드에 설명된 대로 Activity 또는 Fragment에 @AndroidEntryPoint 어노테이션을 추가 할 수 있다. 기본 클래스는 필드 주입을 수행하더라도 어노테이션을 추가 할 필요가 없다 (가장 하위 클래스로 직접 인스턴스화 되는 상황이 아닌 한).

```
@AndroidEntryPoint
class FooActivity : AppCompatActivity() {
    @Inject lateinit var foo: Foo
}
```



Activity에 필드 주입이 필요하지 않더라도 @AndroidEntryPoint를 사용하는 Fragment가 첨부되어 있으면 @AndroidEntryPoint를 사용하도록 Activity를 마이그레이션해야 한다.

Dagger

이제 컴포넌트 초기화 코드 또는 주입 인터페이스가 있으면 제거 할 수 있다.

dagger.android

이 클래스에 @ContributesAndroidInjector를 사용중인 경우 지금 제거 할 수 있다. AndroidInjection / AndroidSupportInjection에 대한 호출이 있으면 그것도 제거 할 수 있다. 클래스가 HasAndroidInjector를 구현하고 비-Hilt Fragment의 상위 클래스가 아닌 경우 해당 코드를 지금 제거 할 수 있다.

Activity 또는 Fragment가 DaggerAppCompatActivity, DaggerFragment 또는 이와 유사한 클래스에서 확장 된 경우, 이를 제거하고 비 Dagger 등가물 (예 : AppCompatActivity 또는 일반 Fragment)로 바꿔야 한다. 여전히 dagger.android를 사용하는 하위 Fragment 또는 View가 있는 경우

DispatchingAndroidInjector를 주입하여 HasAndroidInjector를 구현해야 한다. (다음에 나올 예제 참조)

dagger.android에서 모든 하위 항목을 마이그레이션 한 후 나중에 다시 HasAndroidInjector 코드를 제거 하자.

간단한 dagger.android 예제

다음 예제는 Activity를 마이그레이션 하면서 Hilt 및 dagger.android Fragment를 모두 지원할 수 있도록 한다.

초기 상태:

```
class MyActivity : DaggerAppCompatActivity() {
    @Inject lateinit var foo: Foo
}

@Module
interface MyActivityModule {
    // 스코프 어노테이션을 가지고 있다면, 스코프 별칭 섹션을 확인하자
    @ContributesAndroidInjector(modules = [ FooModule::class, ... ])
    fun bindMyActivity(): MyActivity
}
```

Hilt와 dagger.android Fragment를 모두 허용하는 중간 상태 :

```
@AndroidEntryPoint
class MyActivity : AppCompatActivity(), HasAndroidInjector {
    @Inject lateinit var foo: Foo

    // 모든 하위 항목이 마이그레이션이 되면 아래의 코드는 삭제한다.
    @Inject lateinit var androidInjector: DispatchAndroidInjector<Object>

    override fun androidInjector() = androidInjector
}

// 모듈의 목록이 매우 짧다면 이런 통합 모듈은 필요가 없다.
// 모듈의 includes 목록에서 FooModule과 같이 모든 모듈에 대해
// @InstallIn(ActivityComponent.class) 어노테이션을 추가하자.
@Module(includes = [ FooModule::class, ...])
@InstallIn(ActivityComponent::class)
interface MyActivityAggregatorModule
```

최종 상태:

```
@AndroidEntryPoint
class MyActivity : AppCompatActivity() {
    @Inject lateinit var foo: Foo
}

// 각 Activity 모듈은 @InstallIn(ActivityComponent::class)가 달려있다.
```

빌드 상태 확인하기

Activity 또는 Fragment를 마이그레이션 한 후 앱을 중지하고 빌드 / 실행할 수 있어야 한다. 각 클래스를 마이그레이션 한 후 올바른 길을 가고 있는지 확인하도록 하자.

3. 다른 안드로이드 컴포넌트

View, Service 및 BroadcastReceiver 타입은 위와 동일한 공식을 따라야하며, 지금 마이그레이션 할 준비가 되었다. 모든 것을 옮기면 끝이다.

이것만 기억하자:

- 사용하고 있는 HasAndroidInjector 을 정리하자.
- 남은 통합 모듈 또는 Entry point 인터페이스를 정리하자. 일반적으로 Hilt와 함께 @Module (includes =)을 사용할 필요가 없으므로 이 모듈을 제거하고 포함된 모듈에 @InstallIn 주석을 추가하면 된다.
- 필요한 경우 이전 스코프 어노테이션을 스코프 별칭으로 마이그레이션 하자
- 컴포넌트 인자 바인딩을 일치시키기 위해 배치해야 하는 @Binds 어노테이션을 마이그레이션 하자.

무엇을 할까?

한정자

프로젝트에서 사용하고 있는 한정자는 여전히 유효하고, 그것들은 Dagger에서 사용하던 방식과 동일하게 Hilt에서 사용된다.

앱에서 서로 다른 Context를 구별하기 위해 자체 @ApplicationContext 및 @ActivityContext 한정자가 있는 경우 @Binds를 추가하여 함께 매핑한 다음 남은 시간에 Hilt 한정자로 바꿀 수 있다.

```
@InstallIn(ApplicationComponent::class)
@Module
interface ApplicationContextModule {
    @Binds
    @my.app.ApplicationContextfun bindAppContext(
        @dagger.hilt.android.qualifiers.ApplicationContext context: Context
    ): Context
}
```

컴포넌트 인자

Hilt를 사용할 때는 컴포넌트를 인스턴스화하는 코드가 숨겨져 있으므로 모듈 인스턴스 또는 @BindsInstance 호출을 사용하여 자체 컴포넌트 인자를 추가 할 수 없다. 컴포넌트에 이러한 코드가 있으면 코드를 리팩토링하여 사용하지 않도록 한다. Hilt는 각 컴포넌트에 기본 바인딩 세트가 있으며 '컴포넌트가 제공하는 기본 바인딩'에서 볼 수 있다. 컴포넌트 인자가 무엇인지에 따라 일부 기본 바인딩에 의존하도록 할 수

있다. 때로는 약간의 재설계가 필요하지만 대부분의 경우 다음 전략을 사용하여 이 방법으로 해결할 수 있다. 그렇지 않은 경우 사용자화 컴포넌트 사용을 고려해야 한다.

예를 들어, 가장 간단한 경우는 바인딩을 전혀 전달할 필요가 없고 일반적인 정적 @Provides 메소드인 경우다. 또 다른 간단한 경우는 인자가 사용자화 BaseFragment 타입과 같은 기본 바인딩의 변형 일 수 있다. Hilt는 모든 Fragments가 BaseFragment의 인스턴스가 될 것임을 알 수 없으므로 BaseFragment로 바인딩된 실제 타입이 필요한 경우 캐스팅하여 진행한다.

```
@Component.Builderinterface Builder {
    @BindsInstance
    fun fragment(fragment: BaseFragment): Builder
}

@InstallIn(FragmentComponent::class)
@Module
object BaseFragmentModule {
    @Provides
    fun provideBaseFragment(fragment: Fragment) : BaseFragment {
        return fragment as BaseFragment
    }
}
```

다른 경우에는 인자가 Activity의 Intent와 같은 기본 바인딩 중 하나에 대한 것 일 수 있다.

```
@Component.Builderinterface Builder {
    @BindsInstance
    fun intent(intent: Intent): Builder
}

@InstallIn(ActivityComponent::class)
@Module
object IntentModule {
    @Provides
    fun provideIntent(activity: Activity) : Intent {
        return activity.getIntent()
    }
}
```

마지막으로, 다른 Activity 또는 Fragment 컴포넌트에 대해 다르게 구성된 경우 일부를 다시 설계해야 할 수도 있다. 예를 들어, Activity에 새 인터페이스를 사용하여 오브젝트를 제공 할 수 있다.

```
@Component.Builderinterface Builder {
    @BindsInstance
    fun foo(foo: Foo): Builder // Activity별로 Foo는 다르다
}

// Activity가 구현하여 사용자화 Foo를 제공하는 인터페이스를 정의한다.
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
    }
}

```

사용자와 컴포넌트

Hilt 컴포넌트에 매핑되지 않은 다른 컴포넌트가 있는 경우, Hilt 컴포넌트로 단순화 될 수 있는지 먼저 고려해야 한다. 그렇지 않은 경우 컴포넌트를 수동 Dagger 컴포넌트로 유지할 수 있다. 컴포넌트 의존성 또는 서브 컴포넌트를 사용하려면 아래 섹션을 살펴보자.

컴포넌트 의존성

컴포넌트 의존성 @EntryPoint로 연결될 수 있다.

예를 들어 ApplicationComponent에서 컴포넌트 의존성이 없는 경우 필요한 메소드를 @EntryPoint로 어노테이션이 달린 인터페이스로 분리하여 작동을 유지할 수 있다.

```

// 컴포넌트 의존성과 함께 시작하면
@Component
interface MyApplicationComponent {
    // MyCustomComponent에서 이 바인딩들이 노출 된다
    fun getFoo(): Foo
    fun getBar(): Bar
    fun getBaz(): Baz
    ...
}

@Component(dependencies = [MyApplicationComponent::class])
interface MyCustomComponent {
    @Component.Builder
    interface Builder {
        fun appComponent(appComponent: MyApplicationComponent): Builder

        fun build(): MyCustomComponent
    }
}

// 다음 클래스들과 함께 Hilt로 마이그레이션 될 수 있다.
@InstallIn(ApplicationComponent::class)
@EntryPoint
interface CustomComponentDependencies {
    fun getFoo(): Foo
    fun getBar(): Bar
    fun getBaz(): Baz
    ...
}

@Component(dependencies = [CustomComponentDependencies::class])
interface MyCustomComponent {
    @Component.Builderinterface Builder {
        fun appComponentDeps(deps: CustomComponentDependencies): Builder
        fun build(): MyCustomComponent
    }
}

```

사용자와 컴포넌트를 빌드 할 때 EntryPoint를 사용하여 CustomComponentDependencies의 인스턴스를 얻을 수 있다.

```

DaggerMyCustomComponent.builder()
    .appComponentDeps(
        EntryPoints.get(
            applicationContext,

```

```
CustomComponentDependencies::class.java))
.build()
```

서브 컴포넌트

서브 컴포넌트는 Dagger에서 주입 가능한 서브 컴포넌트 빌더를 사용하여 일반 서브 컴포넌트를 설치하는 것과 같은 방식으로 Hilt 컴포넌트의 하위에 추가 할 수 있다. 상위의 적절한 @InstallIn을 사용하여 서브 컴포넌트를 모듈에 설치하자.

예를들어 ApplicationComponent의 하위 항목인 FooSubcomponent를 가지고 있다면, 다음과 같이 설정할 수 있다.

```
@InstallIn(ApplicationComponent::class)
@Module(subcomponents = FooSubcomponent::class)
interface FooModule {}
```

Hilt 컴포넌트에 매핑되는 컴포넌트를 위한 컴포넌트 의존성

현재 컴포넌트 의존성을 사용하고 컴포넌트가 Hilt 컴포넌트에 상대적으로 잘 매핑되어 있으면 마이그레이션 할 때 컴포넌트 의존성과 서브 컴포넌트간의 차이점을 염두에 두어야 한다. Hilt가 서브 컴포넌트를 사용하기로 선택한 몇 가지 이유를 설명하는 '서브 컴포넌트 vs 컴포넌트 의존성' 섹션을 확인하자.

알아야 할 주요 차이점은 바인딩이 상위 항목으로부터 자동으로 상속된다는 점이다. 이는 바인딩을 노출시키기 위한 추가 메서드를 제거하고, 상위 및 하위 컴포넌트에서 정의된 바인딩에 대해 발생할 수 있는 중복 바인딩을 처리 할 가능성이 있음을 의미한다. 바인딩을 노출시키는 추가 메서드를 제거하는 것은 기술적으로 빌드를 깨지 않기 때문에 선택 사항이지만, 죽은 코드들을 정리 할 수 있으므로 제거하는 것을 권장한다. 'b. 확장한 인터페이스 또는 모듈 다루기' 섹션에서 설명한대로 안전하게 마이그레이션 할 수 있다.

다음은 노출된 바인딩의 예제다.

```
@Component
interface MySingletonComponent {
    // 이런 바인딩들은 컴포넌트 의존성을 위해 노출 될 수 있다.
    // 이 바인딩들을 제거하는 것을 고려해보자.
    fun getFoo(): Foo
    fun getBar(): Bar
    fun getBaz(): Baz
    ...
}
```

앞의 단계를 따라 컴포넌트를 마이그레이션 할 때, 컴포넌트에 상위 Hilt와 동등한 컴포넌트에 대한 의존성이 있는 경우 나머지 컴포넌트들을 제거하므로 해당 의존성을 제거하자.

```
// 나머지 컴포넌트들을 마이그레이션 가이드에 따라 마이그레이션 하므로
// 이 의존성들을 제거하자
@Component(dependencies = [MySingletonComponent::class])
interface MyActivityComponent {
    ...
}
```

7.2 Migration - Optional Injection

왜 선택적 주입이 필요한가?

Hilt를 사용하는 Fragment는 Hilt를 사용하는 Activity에 속해야 하고, Hilt를 사용하는 Activity는 Hilt를 사용하는 Application에 속해 있어야 한다. 이는 순수한 Hilt 코드베이스를 위해서는 자연스러운 제약조건이며, Hilt를 사용하지 않는 Fragment 또는 Activity를 가지고 있다면 마이그레이션 할때 문제가 될 수 있다. 예를 들면, Hilt로 Fragment를 마이그레이션 하길 원하지만 한번에 마이그레이션 하기에는 너무 많은 곳에서 사용되고 있을 수 있다. 선택적 주입(Optional Injection)없이 Hilt를 사용하는 Fragment 사용하기 위해서는 해당 Fragment를 사용하는 모든 Activity를 먼저 마이그레이션 해야 한다. 그렇지 않으면, Fragment는 Fragment 자신에게 의존성을 주입하려고 할 때 Hilt 컴포넌트를 찾으려다가 크래시가 발생한다. 코드베이스 사이즈에 따라 이는 대공사가 될 수 있다.

@OptionalInject를 사용하는 방법

@AndroidEntryPoint를 사용하는 클래스와 함께 @OptionalInject 어노테이션을 추가하면 Hilt를 사용하는 상위 항목이 있을때만 주입을 시도 하게 된다. 이 어노테이션을 사용하는 것은 생성된 기본 클래스에서 주입 성공시 true를 반환하는 wasInjectedByHilt() 메서드를 생성하는데 영향을 끼친다.



기본 클래스에서 생성된 API는 gradle 플러그인 사용자가 액세스 할 수 없으므로, OptionalInjectCheck의 정적 헬퍼 메서드를 사용하여 이 기능에 액세스 할 수 있는 대체 API가 있다.

이를 통해 다른 방법으로 의존성을 제공 할 수 있다 (일반적으로 Hilt를 사용하기 전에 의존성을 얻는 방법과는 다르다).

예를 들면 다음과 같다.

```
@OptionalInject
@AndroidEntryPoint
class MyFragment : Fragment() {

    @Inject lateinit var foo: Foo
    override fun onAttach(activity: Activity) {
        super.onAttach(activity) // 주입은 여기서 발생한다. 하지만 Activity가 Hilt를 사용할 때만 발생한다.
        if (!wasInjectedByHilt()) {
            // 이전 방법으로 Dagger를 얻고 주입한다
        }
    }
}
```

7.3 Scope aliases

스코프 별칭은 왜 필요한가?

현재 많은 코드에서 사용중인 스코프 어노테이션 중 Hilt가 제공하는 스코프 어노테이션으로 변경하고 싶다면, 스코프 별칭(Scope alias)은 마이그레이션 할 때 유용하다. 코드베이스에 따라 스코프 어노테이션을 변경하는 것은 많은 작업을 필요로 한다. 스코프 별칭을 추가하는 것으로 이러한 전환 작업을 점진적으로 할 수 있다.

스코프 별칭을 사용하는 것은 Dagger와 Hilt에게 이러한 스코프 어노테이션을 같은 스코프 어노테이션으로 취급해달라고 미리 말하는 것이다.

@AliasOf 사용하는 방법

@AliasOf 어노테이션을 스코프 어노테이션과 함께 사용하면 Hilt에게 해당 스코프 어노테이션은 @AliasOf 어노테이션의 값과 동일하게 취급해 달라고 요청하게 된다. 어노테이션 값은 반드시 @DefineComponent에서 사용되는 어노테이션이 되어야 Hilt가 무엇을 할지 알게 된다.

다음 나오는 예제에서는 이전에 사용하던 @MyActivityScoped를 Hilt의 @ActivityScoped와 동등하게 만드는 방법을 보여준다. 지금부터는 @MyActivityScoped를 Hilt용 어노테이션으로 점진적으로 교체해 나가기가 수월해진다.

```
@Scope
@AliasOf(dagger.hilt.android.scopes.ActivityScoped::class)
annotation class MyActivityScoped {}
```

8. Compiler Options

@InstallIn 검사 비활성화 하기

기본적으로 Hilt는 @InstallIn 어노테이션에 대한 @Module 클래스를 검사하고 @InstallIn이 없다면 에러를 나타낸다. 누군가 실수로 모듈에 @InstallIn을 누락시켰을까봐 이러한 기능이 사용되고 있으며, 이는 Hilt가 해당 모듈을 찾지 못해 디버깅을 어렵게 만들 수 있다.

이런 검사는 때로는 지나치게 광범위 할 수 있다. 특히 마이그레이션 진행중이라면 말이다. 이러한 기능을 비활성화 하기 위해서는 다음과 같은 플래그를 사용할 수 있다.

```
-Adagger.hilt.disableModulesHaveInstallInCheck=true.
```

또는 @DisableInstallInCheck를 모듈에 추가하여 개별 모듈 레벨에서 검사를 비활성화 할 수 있다.

9. Creating Extensions

모듈과 Entry point 생성하기

Hilt는 표준 컴포넌트와 클래스 경로에서 모듈 및 Entry point가 선택되는 방식으로 인해 Hilt와 통합하려는 확장(extension) 또는 라이브러리에 특히 적합하다.

그러나, @InstallIn 모듈을 생성하는 확장 또는 Entry point는 Hilt가 올바르게 선택할 수 있도록 생성된 클래스에 추가 정보를 넣어야 한다.

@GeneratesRootInput

Hilt는 클래스 경로에서 모듈과 Entry point를 암시적으로 선택하기 때문에, Dagger 컴포넌트를 생성하기 전에 확장이 코드를 생성 할 때까지 기다려야 하는지 알기 위한 추가 정보가 필요하다. 이것은 코드 생성을 트리거하는 어노테이션 클래스에 @GeneratesRootInput을 달아서 수행된다.

예를 들어, 누군가 @GenerateMyModule 어노테이션을 사용할 때마다 확장에서 모듈을 생성 한 경우, @GenerateMyModule 어노테이션을 다음과 같이 사용해야 한다.

```
@GeneratesRootInput
annotation class GenerateMyModule {}
```


어노테이션을 달지 않는다고 해서, Hilt가 반드시 모듈을 놓치는 것은 아니다. 다른 항목이 생성되기를 기다리는 경우에도 여전히 모듈을 생성할 수 있기 때문이다. 이것도 물론 신뢰할 수 없다.

@OriginatingElement

테스트 페이지에 설명 된대로 테스트의 내재된 모듈은 테스트를 둘러싸는 곳에 격리된다. 그러나 테스트를 위해 생성된 모듈은 내재된 클래스로 생성될 수 없다. 이를 올바르게 지원하려면 생성된 코드에 최상위 클래스가 값인 @OriginatingElement 어노테이션을 달아야 한다. 내재된 계층이 많을 수 있으므로 이 클래스는 항상 클래스와 동일하진 않다.

예를 들어, 확장은 다음 코드에 의해 트리거 되고 FooTest_FooModule이라는 모듈을 생성한다고 가정하자.

```
@HiltAndroidTest
class FooTest {
    @GenerateMyModule
    val foo: Foo = Foo()
    ...
}
```

그런 다음 생성된 FooTest_FooModule에 다음과 같이 어노테이션을 달아야 한다.

```
@OriginatingElement(FooTest::class)
@Module
@InstallIn(ApplicationComponent::class)
interface FooTest_FooModule {
    ...
}
```

10.1 Design Decisions - Design Overview

컴포넌트 생성과 모듈/Entry point 설치

Hilt는 전이 클래스 경로에서 모든 모듈과 Entry point를 찾아 컴포넌트를 생성한다. 모든 모듈의 @InstallIn 어노테이션과 Entry point는 정의된 패키지에서 작은 메타데이터 클래스를 생성하게 된다.

@HiltAndroidApp를 처리 할 때 컴포넌트에 설치해야 하는 모든 집계된 항목을 찾기 위해 특수 패키지를 검사한다. @DefineComponent 및 @AliasOf와 같은 다른 헬퍼 클래스에도 동일한 전략이 사용된다.

동시에 안드로이드 Application이 생성되기 때문에, 생성된 Application은 최상위 컴포넌트인 ApplicationComponent를 직접 참조 할 수 있게 된다.

HiltTestApplication은 반드시 여러가지 테스트들을 지원해야하기 때문에, 프로덕션 애플리케이션에서와는 달리, 생성된 컴포넌트를 찾기 위해 리플렉션이 사용된다. 이는 테스트 Application을 테스트와 빌드에서 분리 할 수 있기 때문에 각 프로젝트에서 테스트 Application을 코드로 작성하는 대신 Hilt가 편리한 기본값을 제공 할 수 있기 때문에 유용하다. 리플렉션은 가치가 낮고 더 많은 비용이 들어가기 때문에 프로덕션에서는 사용되지 않는다.

클래스 경로에서 모든 모든 모듈을 통합하는 것은 테스트에서 잘 동작한다. 왜냐하면 테스트는 테스트 클래스에서 내재된 클래스에 의한 바인딩을 쉽게 추가 할 수 있기 때문이다 (또는 모듈을 생성하는 @BindValue를 사용하는 것 보다 낫다). 마찬가지로 모듈 감지를 통해 클래스가 @Module 클래스를 내부 클래스로 포함 할 수도 있다. 이를 통해 Dagger 바인딩 없이 클래스를 사용할 수 없고 오류 발생 가능성이 줄어들게 된다(예. 클래스를 @BindsOptionalOf와 연결하거나 소비하는 @Binds와 인터페이스).

@AndroidEntryPoint injection

@AndroidEntryPoint는 Gradle 플러그인의 변환을 통해 사용자 코드가 직접 또는 간접적으로 확장되는 기본 클래스를 생성하여 작동한다. 이 기본 클래스는 (상위 Hilt 인터페이스를 통해) 부모 컴포넌트를 검색하고, 컴포넌트를 만들고, 클래스를 주입하고, Hilt 인터페이스를 통해 하위에 컴포넌트를 노출시키는 역할을 한다.

예를 들면, Activity에 주입하기 위해 생성된 코드는 다음과 같은 작업을 수행한다.(가독성을 위해 간소화함)

```
@Override public void onCreate(Bundle savedInstanceState) {
    // 이는 Application에서 상위 컴포넌트를 가져온다.
    // (실제로 부모로써 컴포넌트를 유지하는 Activity가 있다)
    Object parentComponent = ((GeneratedComponentManager) getApplication()).generatedComponent();
    // 이것은 Activity 컴포넌트를 생성한다. 이는 상위 컴포넌트가 Activity 컴포넌트를 빌드하는 메소드를
    // 가지고 있다는 것을 알기 위해 안전하지 않은 캐스팅을 포함한다.
    Object activityComponent = ((ActivityComponentBuilderEntryPoint) parentComponent)
        .activityComponentBuilder()
        .activity(this)
        .build();
    // 이는 Activity에 주입하고, 또한 Activity주입 메서드에 접근하기 위해
    // 안전하지 않은 캐스팅을 포함하는 것을 의미 한다. 다른 안전하지 않은 캐스팅과 같이
    // 이러한 캐스팅은 빌드 의존성들을 깨고, 코드생성과
    // 모듈/인터페이스의 클래스 경로 발견을 통해 보장되기 때문에 안전하다
    (MyActivity_GeneratedInjector) activityComponent).inject(this);
}
```

이러한 모든 글루 코드(glue code)의 생성으로 안전하지 않은 캐스팅과 함께 의존성을 깨뜨리고 안전하고 쉽게 만들 수 있다. 또한 생성된 인터페이스를 사용하는 Activity와 결합된 자동 검색은 @AndroidEntryPoint를 포함하거나 제거하면 @AndroidEntryPoint에 대한 종속성이 모두 추가/제거된다.

대부분의 경우 상위 컴포넌트를 쉽게 얻을 수 있지만, View와 Fragment의 경우 View가 Activity의 Context를 가져 오기 때문에 쉽지 않다. Fragment 바인딩이 있는 View를 지원하기 위해 Fragment에 대해 생성된 기본 클래스는 getLayoutInflater를 재정의하여 View에 대한 Dagger 컴포넌트를 보유하는 ContextWrapper에서 Context를 감싼다.

Hilt에서 이런 모든 설계 결정은 표준화됨으로써, Activity/Fragment/View와 함께 라이브러리를 구현하는 것은 더욱 쉬워 진다.

10.2 Design Decisions - Testing Philosophy

개요

이 페이지는 Hilt를 사용하는 테스트 사례를 설명하는 것을 목표로 한다. Hilt의 많은 API와 기능 (그리고 특정 기능의 부족함)은 좋은 테스트를 만드는 것에 대한 무언의 철학을 바탕으로 만들어졌다. 좋은 테스트의 개념은 보편적으로 합의 된 것은 아니기 때문에 이 문서는 Hilt 팀의 테스트 철학을 명확히 하는 것을 목표로 한다.

테스트해야 하는 것

Hilt는 외부 사용자의 관점에서 가능한 많은 테스트하는 것을 장려한다. 외부 사용자 관점에서라는 것은 많은 것을 의미 한다. 그것은 앱 또는 서비스의 실제 사용자가 될 수도 있고, API 또는 클래스를 사용하는 더 많은 범위의 사용자가 될 수도 있다.

핵심적인 부분은 테스트가 세부적인 구현에 대한 내용까지 작성하지 않는다는 점이다. 내부 메서드가 호출되었는지 확인하는 것 처럼 세부적인 구현에 대한 것은 불안정한 테스트를 유발한다. 리팩토링시 내부 메서드의 이름이 변경된다면 좋은 테스트의 경우에는 갱신이 필요 없을 것이다. 기존 테스트가 깨지는 유일한 변경사항은 사용자가 볼 수있는 동작을 변경하는 것이다.

실제 의존성 사용하기

Hilt의 테스트 철학은 모든 클래스가 자체 테스트를 가져야 하는 엄격한 규칙을 규정하지는 않는다. 실제로 이러한 규칙은 일반적으로 사용자의 관점에서 위의 테스트 원칙을 위반한다. 테스트는 작성 및 실행이 편리하도록 필요한 만큼만 작아야 한다(예. 빠르거나 또는 리소스 집약이지 않게 작아야 한다.). 다른 모든것이 동일하다면 테스트는 이 순서대로 진행되는 것이 좋다.

- 의존성에 대한 실제코드를 사용하라
- 라이브러리에 의해 제공된 표준 Fake를 사용하라
- Mock은 최후의 수단으로 사용하라

그렇지만 트레이드 오프도 존재한다. 테스트에서 실제 의존성/실제 의존성 주입을 사용하는 것은 하나 또는 다음과 같은 이유들로 인해 제한적으로 어려울 수 있다:

- 실제 의존성 주입을 설정하고 인스턴스화하는 것은 너무 많은 보일러플레이트 코드 또는 반복되는 코드를 만든다.
- 실제 의존성을 사용하는 것은 퍼포먼스 문제에 직면하게 된다.(백엔드 서버를 시작하는 것을 필요로 하는 것 처럼)

Hilt는 첫번째와 같은 문제를 해결하기 위해 설계되었다. 성능은 문제일 수 있지만, 성능은 대부분의 의존성에 대해 문제가 되지 않는다. 이는 중대한 I/O를 가진 의존성을 사용할 때만 문제가 될 수 있다. 그래서 성능을 크게 떨어뜨리지 않으면서 더 많은 실제 의존성을 사용하여 테스트를 보다 편리하고 강력하게 작성할 수 있는 경우, 실제 의존성을 사용하여 작성해야 한다. 테스트에서 큰 부정적인 영향을 미치는 클래스의 경우 Hilt는 바인딩을 교체하는 수단을 제공한다.

더 많은 실제 의존성을 사용하면 다음과 같은 중요한 이점이 있다:

- 실제 의존성은 실제 문제를 잡아낼 가능성이 더 높다. 실제 의존성은 Mocking 하는 것과 달리 항상 최신 상태를 유지 한다.
- 사용자 관점에서 위의 테스트 원칙과 부합하여 동일한 적용 범위에 대해 더 적은 테스트를 작성하게 된다.
- 테스트가 깨진다는 것은 잘못 구성된 Fake 또는 Mock 대신 실제 문제를 나타낸다.
- 더 많은 실제 의존성을 사용하면 사용자의 관점에서 테스트 하는 위의 원칙과 종종 관련이 있다.

그래도 실제 의존성을 사용할 수 없으면 일반적으로 라이브러리에서 제공하는 표준 Fake가 다음으로 좋은 선택지다. 라이브러리 제작자가 유지보수하는 프로덕션 코드와 동기화 될 가능성이 높기 때문에 표준 Fake는 Mock보다 낫다. 이러한 이유로 Mock은 일반적으로 최후의 수단으로 선택된다.

Hilt, 의존성 주입 그리고 테스트

테스트에 대한 기초가 설명되었으므로 이제 Hilt, 의존성 주입 및 테스트의 특성에 대해 알아보자. 실제 객체를 사용한다는 철학에 따라 Hilt는 테스트에 의존성 주입 / Dagger를 사용한다. 이는 프로덕션 코드에서 객체가 생성되기 때문에 더욱 현실적이다. 즉, 테스트는 프로덕션 코드보다 깨지기 쉽지 않으며 실제 객체를 보다 쉽게 사용할 수 있다. 실제로 @Inject 생성자를 가진 타입의 경우, 실제로 이 조언을 따르고 실제 코드를 사용하는 것이 Mock를 구성하고 바인딩하는 것보다 쉽고 적은 코드를 갖는다.

불행히도, Hilt가 없는 이런 종류의 테스트는 전통적으로 보일러플레이트 코드와 테스트에서 Dagger를 설정하기 위한 추가 작업으로 인해 실제로 어려웠다. 그러나 Hilt는 보일러플레이트 코드를 생성하고 Fake 또는 Mock이 필요할 때 테스트를 위한 다양한 바인딩 구성을 설정하는 명확한 스토리를 가지고 있다. Hilt와 함께 이 문제는 더 이상 Dagger로 테스트를 작성하는 데 방해가 되지 않으므로 실제 의존성을 쉽게 사용할 수 있다.

Dagger를 사용하지 않아 생기는 단점

유닛 테스트에서 Dagger를 사용하지 않는 것은 실제로 매우 일반적이다. 이런 점은 상당히 단점으로 작용하지만, 테스트에서 Hilt 없이 Dagger를 사용하는 것이 더 어렵다는 점을 감안하면 이해할 만한 방법이다.

예를 들어 테스트하려는 Foo 클래스가 있다고 가정하자:

```
class Foo @Inject constructor(bar: Bar) {  
}
```

이 경우 Dagger를 사용하지 않으면 테스트는 생성자를 호출하여 Foo를 직접 인스턴스화한다. 언뜻 보기에 이것은 매우 간단해 보이지만, Foo의 생성자에 Bar 인스턴스를 제공해야만 한다.

직접 인스턴스화 하는 대신 Mock으로 테스트를 하자.

앞에서 설명한 테스트 철학에 따르면, 실제 Bar 클래스를 사용하는 것이 좋다. 하지만, 어떻게 해야할까? 이는 사실 테스트 하기 위해 실제 Foo 클래스를 얻는 것을 반복하는 것이다: Foo를 직접 인스턴스화 해야 하며 Bar가 자체의 의존성을 가지고 있다면, 그것 또한 비슷하게 인스턴스화 해야하는 작업이 필요하다. 너무 복잡하지 않게 하기 위해 Fake 또는 Mock을 사용해야 한다. 꼭 테스트 속도나 성능에 때문은 아니지만, 유지보수 문제를 야기하는 불안정한 보일러 플레이트 코드의 사용을 피하기 위해서다. 이것이 Fake 또는 Mock을 사용하는 타당한 이유는 아니지만 어쨌든 그렇게 하는 것이 좋다.

위에서 언급한 대안은 일반적인 Fake를 사용하는 것이다. 이는 의존관계를 끊고 직접적인 인스턴스화의 유지 관리의 부담을 감소시킨다. 그러나 항상 모든게 그렇게 간단하지는 않다. 대부분의 경우에 Fake는 기존 클래스와 비슷한 의존성을 필요로 하게 된다. 예를 들면 실제 Bar가 Clock을 필요로 한다면, FakeBar도 결국은 FakeClock을 필요로 한다. FakeClock이 서로 다른 클래스간에 흔히 조정을 해주기 때문이다 (Foo에 Clock을 사용하는 다른 의존성 Baz가 있다고 가정하면, FakeBaz는 동일하게 FakeClock 인스턴스를 사용하여 시간이 지날 때 기능이 잘 동작하도록 할 것이다). 이런식으로 의존성을 관리하다보면 급속도로 감당할 수 없게 된다.

이런 부분 때문에 일반적으로 Mock으로 테스트를 작성해야 한다. Mock으로 테스트를 하는 것은 이러한 의존성 체이닝 문제를 해결해 준다. 하지만 자동으로 쉽게 최신 정보를 얻지 못하고, 실제 버그를 찾는 전체 목표에서 테스트를 쓸모 없게 만들 수 있다는 중대한 단점을 갖게 될 수도 있다. 테스트 작성자 외에 Mock 테스트의 동작을 확인하는 사람이 없기 때문에 일반적으로 충분한 시간이 지나면 테스트에서 더 이상 유용한 시나리오를 테스트하지 않을 가능성이 높다.

직접 인스턴스화 하는 것은 테스트에서 세세한 구현까지 해야 한다.

직접적으로 인스턴스화를 하면 생성자 호출이 의존성의 세부 정보를 표현하기 때문에 테스트에서 구현 세부 정보까지 작성하지 않는다는 철학을 깨뜨리게 된다. Bar를 @Inject 생성자의 매개변수로 갖는 경우 Foo를 사용하는 입장에서는 Bar 클래스의 존재에 대해 알아야 할 이유가 없다. Foo의 리팩토링 로직이 다른 라이브러리의 private 클래스에 세부 구현사항이 될 수 있기 때문이다.

이를 실제로 보기 위해서는, Foo가 Foo(Bar,Baz) 생성자와 같이 Bar와 Baz의존성을 필요로 하는 경우를 생각해보자. Dagger에서는 @Inject 생성자 매개변수의 순서를 바꿔도 무관하다. 그러나 직접 인스턴스화하여

Foo를 테스트 해야 한다면, 테스트 또한 변경이 필요하다. 마찬가지로, 새로운 @Inject 클래스의 사용을 추가하거나 또는 옵셔널 바인딩의 사용은 최종 사용자에게는 보이지 않지만, 테스트를 지속적으로 갱신해야만 한다.

요약

Hilt는 실제 의존성과 함께 쉬운 테스트를 진행하기 위해 Dagger의 단점을 보완하여 설계되었다. Hilt를 사용하여 작성한 테스트는 이러한 원리 원칙을 잘 따른다면 전반적으로 더 나은 테스트 경험을 보여줄 것이다.

10.3 Design Decisions - Monolithic Components

개요

Hilt는 단일 컴포넌트 체제를 사용한다. 이것이 의미하는 점은 모든 Activity 클래스들에 의존성 주입을 하는 것에 대해 단일 Activity 컴포넌트의 정의가 사용된다는 것이다. Fragment와 다른 안드로이드 타입들도 같은 맥락이다. 각 Activity는 분리된 컴포넌트 인스턴스를 갖지만, 정의된 컴포넌트 클래스는 공유된다. 이는 각 Activity가 분리된 컴포넌트 정의를 갖는 다형적 컴포넌트 체제와는 반대된다. dagger.android의 @ContributesAndroidInjector를 사용할 때는 다형적 체제를 기본적으로 사용한다. 이 페이지는 Hilt가 두 가지 모델 사이에서 몇가지 트레이드 오프와 함께 단일 컴포넌트를 채택하여 설계된 이유에 대해서 알아본다.

단일 바인딩 키 공간

Hilt에서 단일 체제를 사용하는 주된 이유 중 하나는 바인딩 키 공간이 통합된다는 점이다. Fragment에 Foo 클래스를 주입한다고 가정하면, Fragment가 어느 Activity에 붙었는지에 따라 다르지 않기 때문에 Foo바인딩이 어디로 부터 왔는지 찾기가 쉬워진다. 다형 체제는 Activity마다 다른 바인딩을 정의하여 더 나은 유연성을 제공하지만, 이는 보통 코드량이 더 많아지고 추적하기 힘들어 지기 때문에 혼란을 야기한다.

바인딩들을 사용해야 하는 코드에만 바인딩을 private으로 유지하려면, 제한된 가시성을 통해 보호되는 한정자를 사용하거나 SPI 플러그인을 사용하여 코드를 분리하는 것이 좋다.

간단한 설정

단일 바인딩 키 공간은 설정을 매우 쉽게 만든다. 이는 모듈이 설치될 수 있는 곳을 줄이기 때문에 테스트를 위해 바인딩을 쉽게 교체할 수 있게 한다. 이런 부분은 해당 기능을 사용하는 모든 곳에 대해 모듈을 전파하는 점을 걱정할 필요가 없다는 것을 의미한다. 이것은 다른 스코프를 사용하는 기능들에 대해 정말 유용할 수 있다. 다형성의 세계에서 보면 Fragment 스코프 내 객체와 Activity 스코프 내 객체를 사용하는 기능은 개발자가 Fragment에 모듈을 포함시킨 다음 해당 Fragment를 사용하는 모든 Activity에 포함시켜야 한다. 종종 이러한 코드 설정은 보일러플레이트 코드를 더하고 캡슐화를 깨뜨린다.

생성되는 코드 줄이기

단일 체제를 사용하면 생성되는 코드를 줄일 수 있다. 일반적으로 모듈이 여러 서브 컴포넌트에서 사용되면 (일반적인 Activity 헬퍼 클래스의 경우와 마찬가지로), 모든 서브 컴포넌트에 대해 Dagger 코드가 반복적으로 생성되어야 한다. 이게 처음엔 별거 아닌 것 같지만, 많은 Activity들을 통해 빠르게 더해지고, 많은 Fragment 또는 View로 인해 급격하게 늘어 날 수 있다.

fastinit 및 시작 대기시간

일부 개발자들은 이것이 어떻게 시작 대기시간(startup latency)에 영향을 미치는지 걱정할 것이다. 만약 fastinit 컴파일 옵션을 사용하고 있다면, 단일 컴포넌트는 시작 대기시간에 눈에 띄만한 영향을 주지 못한다.

이는 그레이들을 사용하여 Hilt를 사용하는 경우 기본값이며, 일반적으로 안드로이드에서 사용되는 Dagger 컴파일모드여야 한다.

10.4 Design Decisions - Subcomponents vs Component dependencies

개요

Hilt는 컴포넌트 의존성과 대조적으로 Dagger의 서브컴포넌트를 기본적으로 사용한다. 이 페이지에서는 Hilt가 왜 이러한 방식으로 설계되었는지 설명한다.

단일 바인딩 키 공간

서브 컴포넌트는 기본적으로 모든 바인딩을 전파한다. 여기에는 컴포넌트 의존성을 통해 전파하기 어려운 멀티 바인딩도 포함된다. 멀티 바인딩은 병합된 바인딩 키 공간을 만들고, 이는 보통 바인딩이 전파되거나 또는 상위 컴포넌트로부터 하위 컴포넌트로 전파되는지에 대한 걱정을 할 필요가 없기 때문에. Dagger의 오브젝트 그래프를 이해하기 쉽도록 만든다. 또한 컴포넌트 의존성으로 바인딩이 전파되지 않는다면, 서로 다른 컴포넌트에서 동일한 바인딩 키에 대해 다른 두가지 정의를 사용할 수 있다. 바인딩 정의가 사용되는 문맥을 기반으로 하기 때문에 디버깅시 코드를 살펴보는 것이 어려울 수 있다.

단일 바인딩 키 공간의 한가지 단점은 코드 사용에 대한 제약조건으로 인해 추가적인 작업이 어려워 질 수 있다는 점이다 (예. 다른 기능의 바인딩을 어떤 기능에서 사용하지 않을 때). 이럴때 보통은 한정자 어노테이션을 사용하여 가시성을 제한하거나 SPI 플러그인의 사용으로 코드를 분리하는 것을 권장한다. 한정자를 사용하거나 SPI 플러그인을 사용하면 이러한 규칙이 있는 정책을 표현하기 때문에 Dagger 컴포넌트 의존성 그래프의 구조에서 이러한 걱정들을 기르는 것보다는 낫다. 이와 같은 정책 결정은 종종 유동적이고 (예외가 허용되어야하며) 이러한 변경을 기반으로 Dagger 컴포넌트 의존성 그래프를 재구성해야하는 경우 비용이 많이 들 수 있다.

컴포넌트 의존성으로 바인딩을 전파하는 것은 Dagger에서 문제가 될 수 있다.

Dagger는 그래프의 진입점을 알 수 있기 때문에, 사용되지 않는 바인딩을 파악하고 해당 바인딩에 대한 코드를 생성하지 않을 수 있다. 이런 최적화는 서브컴포넌트들에서도 이루어지지만, 컴포넌트 의존성을 통해 바인딩을 전파하고자 하면 진입점 메서드가 추가되기 때문에 문제가 생긴다. 따라서 진입점 메서드가 다른 Dagger 컴포넌트와 컴포넌트 전체에서 바인딩이 사용되지 않더라도, Dagger는 강제로 관련된 코드를 생성한다.

상위 및 빌드 속도 구성하기

컴포넌트 의존성의 주된 장점 중 하나는 Dagger 코드를 개별적 그리고 병렬적으로 만든다는 것이다. 이는 컴포넌트간 관계에 있어 컴포넌트를 블랙박스로 만드는 암시적인 공유의 부재로 수행된다. 그러나 Hilt는 이미 빌드 의존성에 기반한 중앙 구성 개념을 기반으로 하고 있다. Hilt는 모듈을 통합해야하기 때문에 모든 컴포넌트가 동시에 생성되므로 병렬로 만들어질 수는 없다.

대신 빌드 속도를 해결하기 위해, Hilt는 독립적인 기능 개발에 대해 작은 테스트 앱을 만드는 것을 권장하고 있다. Hilt가 없다면 Dagger의 반복되는 보일러플레이트 코드로 인해 테스트를 수행하기가 어려울 것이다. 하지만 Hilt는 빌드 의존성에 기반한 모든 Dagger 관련 코드를 생성하므로 작은 테스트 앱을 구성하는 것이 훨씬 쉬워진다.