eBook Gratuit

APPRENEZ dagger-2

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#dagger-2

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: dagger-2

It is an unofficial and free dagger-2 ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official dagger-2.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec dagger-2

Remarques

Cette section fournit une vue d'ensemble de ce qu'est dagger-2 et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les grands sujets dans Dague-2, et établir un lien avec les sujets connexes. La documentation de dagger-2 étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Versions

Version	Date de sortie
2,5	2016-06-14
2.6	2016-07-27
2.7	2016-09-13
2.8	2016-11-22
2.9	2017-02-03
2.10	2017-03-15

Examples

Description et configuration

Qu'est ce que Dagger 2?

Le site se décrit comme:

> Dagger est un framework d'injection de dépendance à la compilation, complètement statique

La bibliothèque facilite la modélisation des graphes de dépendance et la réutilisation des objets. Étant donné que la réflexion n'est utilisée qu'au moment de la compilation dans le cadre du traitement des annotations, Dagger 2 a amélioré la vitesse d'injection des dépendances.

Installer

1- Ajouter un support pour le traitement des annotations:

Android

`build.gradle` niveau `build.gradle` :

```
 repositories {
    mavenCentral()
  }
  dependencies {
    classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
  }
```

`build.gradle` niveau du module:

```
apply plugin: 'com.neenbedankt.android-apt'
```

Java

```
plugins {
  id "net.ltgt.apt" version "0.5"
}
```

2- Ajouter les dépendances dague 2

```
 dependencies {
      compile 'com.google.dagger:dagger:2.x'
      apt 'com.google.dagger:dagger-compiler:2.x'
    }
```

Exemple de base

Définir un module (le modèle des dépendances et leur graphe):

```
@Module
public class CoffeeModule{

    @Provides
    public CoffeeMaker provideCoffeeMaker(){
         return new CoffeeMaker();
    }

    @Provides
    public Coffee provideCoffee(CoffeeMaker coffeeMaker){
        return new Coffee(coffeeMaker);
    }

}
```

Définir un composant:

```
@Component(
    modules={
        CoffeeModule.class
    }
```

```
)
interface CoffeeComponent {

        DeveloperActivity inject(DeveloperActivity developerActivity);
}
```

Injecter les dépendances:

```
class DeveloperActivity extends ...{

    @Inject
    Coffee myCoffee;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        DaggerCoffeeComponent.builder()
                .coffeeModule(new CoffeeModule())
                .build()
                .inject();

    }

}
```

Exemple Android

L'une des principales difficultés lors de l'écriture d'une application Android avec Dagger est que de nombreuses classes d'infrastructure Android sont instanciées par le système d'exploitation lui-même, comme `Activity` et `Fragment` , mais Dagger fonctionne mieux s'il peut créer tous les objets injectés. Au lieu de cela, vous devez effectuer une injection de membres dans une méthode de cycle de vie. A partir de la version 2.10, dagger permet d'utiliser `dagger.android` qui simplifie l'utilisation de dagger avec les composants Android.

Injecting Activity objects

1. Installez `AndroidInjectionModule` dans votre composant d'application pour vous assurer que toutes les liaisons nécessaires pour ces types de base sont disponibles.

   ```
   @Component(modules = {AndroidInjectionModule.class})
   public interface AppComponent {}
   ```

2. Commencez par écrire un `@Subcomponent` qui implémente [AndroidInjector] [AndroidInjector], avec un `@Subcomponent.Builder` qui étend [AndroidInjector.Builder] [AndroidInjector.Builder]:

   ```
   @Subcomponent
   public interface MainActivityComponent extends AndroidInjector<MainActivity> {
       @Subcomponent.Builder
       abstract class Builder extends AndroidInjector.Builder<MainActivity> {}
   }
   ```

3. Après avoir défini le sous-composant, ajoutez-le à la hiérarchie de vos composants en

définissant un module qui lie le générateur de sous-composants et en l'ajoutant au composant qui injecte votre `Application` :

```
@Module(subcomponents = MainActivityComponent.class)
public abstract class MainActivityModule {

    @Binds @IntoMap @ActivityKey(MainActivity.class)
    abstract AndroidInjector.Factory<? extends Activity>
    bindMainActivityInjectorFactory(MainActivityComponent.Builder builder);
}
```

4. Ensuite, faites en sorte que votre `Application` implémente `HasDispatchingActivityInjector` et `@Inject` un `DispatchingAndroidInjector<Activity>` pour retourner de la méthode activityInjector ():

```
public class MyApp extends Application implements HasDispatchingActivityInjector     {

    @Inject
    DispatchingAndroidInjector<Activity> dispatchingActivityInjector;

    @Override
    public void onCreate() {
       super.onCreate();
       DaggerAppComponent.create().inject(this);
    }

    @Override
    public DispatchingAndroidInjector<Activity> activityInjector() {
        return dispatchingActivityInjector;
    }
}
```

5. Enfin, dans votre méthode `Activity.onCreate()` , appelez `AndroidInjection.inject(this)` avant d'appeler `super.onCreate();` :

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        AndroidInjection.inject(this);
        super.onCreate(savedInstanceState);
    }
}
```

Cet exemple était basé sur la documentation officielle du poignard . Un échantillon de travail peut être trouvé sur github

Apprenez Dagger2 avec un exemple simple

J'ai lu et regardé beaucoup de tutoriels Dagger2 différents, mais la plupart d'entre eux sont trop longs ou trop difficiles à comprendre. J'ai donc décidé d'écrire un nouveau tutoriel simple et court pour Dagger2, j'espère que ça vous plaira.

Pourquoi en avons-nous besoin?

- Simplifie l'accès aux instances partagées: il fournit un moyen simple d'obtenir des références à des instances partagées. Par exemple, une fois que nous avons déclaré dans Dagger nos instances singleton telles que `SharedPrefrences` nous pouvons déclarer des champs avec une simple annotation `@Inject` .
- Tests d'unité et d'intégration simplifiés: nous pouvons facilement échanger des modules qui apportent des réponses réseau et simuler ce comportement.

Commençons par un exemple simple

La source complète de l'exemple est disponible sur mon compte GitHub.

Ajouter des dépendances Dagger2

Tout d'abord, nous devons ajouter des dépendances Dagger2. Placez le code ci-dessous dans votre fichier build.gradle au niveau du module.

```
compile "com.google.dagger:dagger:$dagger_version"
compile "com.google.dagger:dagger-android:$dagger_version"
compile "com.google.dagger:dagger-android-support:$dagger_version"
annotationProcessor "com.google.dagger:dagger-compiler:$dagger_version"
```

Si vous obtenez une erreur comme Error: Conflict with dependency 'com.google.code.findbugs: jsr305' dans le projet ': app', vous devez ajouter ce qui suit dans votre fichier app / build.gradle principal.

```
configurations.all {
   resolutionStrategy.force 'com.google.code.findbugs:jsr305:3.0.1'
}
```

Deux classe simple

Nous avons deux classes (véhicule et moteur), la classe de véhicule a besoin de la classe de moteur pour fonctionner et la classe de véhicule de besoins de MainActivity. Nous utiliserons Dagger2 pour fournir ces instances.

```
class Vehicle {
   private Motor motor;

  @Inject
  Vehicle(Motor motor) {
     this.motor = motor;
  }

  void increaseSpeed(int value) {
     motor.accelerate(value);
  }

  void decreaseSpeed(int value) {
     motor.decelerate(value);
```

```
  }

  void stop() {
     motor.brake();
  }

  int getSpeed() {
     return motor.getRpm();
  }
}
```

Classe de moteur:

```
class Motor {
  private int rpm;

  Motor() {
    this.rpm = 0;
  }

  int getRpm() {
    return rpm;
  }

  void accelerate(int value) {
    rpm += value;
  }

  void decelerate(int value) {
    rpm -= value;
  }

  void brake() {
    rpm = 0;
  }
}
```

Classe de module

La classe de module est chargée de fournir les objets pouvant être injectés. Dans cet exemple, nous voulons injecter la classe Motor dans la classe Vehicle et injecter la classe Vehicle dans MainActivity. Nous devons donc créer MyModule pour fournir ces instances.

```
@Module
class MyModule {

  @Provides
  @Singleton
  Motor provideMotor() {
    return new Motor();
  }

  @Provides
  @Singleton
  Vehicle provideVehicle() {
    return new Vehicle(new Motor());
```

```
  }
}
```

Annotation @Provide: l' objet renvoyé par cette méthode est disponible pour l'injection de dépendance.

Interface @Component

Dagger2 a besoin d'une interface de composant pour savoir comment créer des instances de nos classes.

```
@Singleton
@Component(modules = {MyModule.class})
interface MyComponent {
  Vehicle provideVehicle();

  void inject(MainActivity main);
}
```

@Component interface: connexion entre le fournisseur d'objet et les objets qui expriment une dépendance.

Injecter la dépendance dans le constructeur

En ajoutant une annotation @Inject, dagger2 peut créer automatiquement une instance à partir de cet objet, comme notre exemple d'objet moteur dans la classe Vehicle.

Injecter la dépendance dans MainClass

Dagger2 peut automatiquement injecter des dépendances dans des constructeurs, mais les composants Android (activités, fragments, etc.) sont instanciés par un framework Android, ce qui rend difficile l'utilisation de l'injection de dépendance sur ces composants.

```
public class MainActivity extends AppCompatActivity {
  @Inject
  Vehicle vehicle;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    MyComponent component = DaggerMyComponent.builder().build();
    component.inject(this);
  }
}
```

C'est ça, j'espère que vous apprécierez.

Lire Commencer avec dagger-2 en ligne: https://riptutorial.com/fr/dagger-2/topic/7680/commencer-

avec-dagger-2

Chapitre 2: @Annotation nommée dans Kotlin

Introduction

Comment utiliser correctement l'annotation nommée dans Kotlin v1.1

Examples

Déclarer une dépendance qualifiée

```
@Module
class AppModule(val app: Application) {

    @Provides @Named("the_answer")
    fun providesTheAnswer(): Int {
        return 42
    }
}
```

Injection de dépendance basée sur Setter

```
class MyClass{
    @field:[Inject Named("the_answer")] lateinit var answer: Int
}
```

Dans Android Development, vous injectez des dépendances dans `Activity` , `Fragment` ou tout autre objet instancié directement par le système d'exploitation.

Pour en savoir plus sur le `@field:` annotation dans Kotlin visitez la documentation

Injection de dépendance basée sur le constructeur

```
class MyClass @Inject constructor(@Named val answer: Int){
    /* The nuts and bolts of your class */
}
```

Lire @Annotation nommée dans Kotlin en ligne: https://riptutorial.com/fr/dagger-2/topic/10812/-annotation-nommee-dans-kotlin

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec dagger-2	Community, Hani, Jacob, Volodymyr Khodonovych
2	@Annotation nommée dans Kotlin	Nicolás Carrasco