EBook Gratis

APRENDIZAJE

dagger-2

Free unaffiliated eBook created from
Stack Overflow contributors.

#dagger-2

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con la daga-2

Observaciones

Esta sección proporciona una descripción general de qué es dagger-2 y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema grande dentro de dagger-2, y vincular a los temas relacionados. Dado que la Documentación para dagger-2 es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Versiones

Versión	Fecha de lanzamiento
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

Descripción y configuración

¿Qué es la daga 2?

El sitio web se describe a sí mismo como:

> Dagger es un framework de inyección de dependencia totalmente estático y en tiempo de compilación

La biblioteca facilita el modelado de gráficos de dependencia, así como la reutilización de objetos. Dado que la reflexión solo se utiliza en tiempo de compilación como parte del proceso de anotación, Dagger 2 ha mejorado la velocidad para la inyección de dependencia.

Preparar

1- Añadir soporte para el procesamiento de anotaciones:

Androide

`build.gradle` nivel superior de `build.gradle` :

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

Nivel de `build.gradle` script `build.gradle` :

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

Java

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

2- Añadir las dagas 2 dependencias.

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

Ejemplo básico

Defina un módulo (el modelo de dependencias y su gráfica):

```
@Module
public class CoffeeModule{

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

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

}
```

Definir un componente:

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

```
)
interface CoffeeComponent {

        DeveloperActivity inject(DeveloperActivity developerActivity);
}
```

Inyectar las dependencias:

```
class DeveloperActivity extends ...{

    @Inject
    Coffee myCoffee;

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

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

    }

}
```

Ejemplo de Android

Una de las dificultades centrales de escribir una aplicación de Android con Dagger es que muchas clases de framework de Android son creadas por el propio sistema operativo, como `Activity` y `Fragment` , pero Dagger funciona mejor si puede crear todos los objetos inyectados. En su lugar, debe realizar la inyección de miembros en un método de ciclo de vida. A partir de la versión 2.10, dagger permite usar `dagger.android` que simplifica el uso de dagger con componentes de Android.

Inyectando objetos de actividad

1. Instale `AndroidInjectionModule` en su componente de aplicación para asegurarse de que todos los enlaces necesarios para estos tipos básicos estén disponibles.

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

2. Comience escribiendo un `@Subcomponent` que implementa [AndroidInjector] [AndroidInjector], con un `@Subcomponent.Builder` que se extiende [AndroidInjector.Builder] [AndroidInjector.Builder]:

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

3. Después de definir el subcomponente, agréguelo a la jerarquía de sus componentes

definiendo un módulo que vincule el generador de subcomponentes y agregándolo al componente que inyecta su `Application` :

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

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

4. A continuación, haga que su `Application` implemente `HasDispatchingActivityInjector` y `@Inject` a `DispatchingAndroidInjector<Activity>` para regresar del método 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. Finalmente, en su método `Activity.onCreate()` , llame a `AndroidInjection.inject(this)` antes de llamar a `super.onCreate();` :

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

Este ejemplo se basó en la documentación oficial de la daga . Muestra de trabajo se puede encontrar en github

Aprende Dagger2 con un simple ejemplo.

He leído y visto muchos tutoriales diferentes de Dagger2 pero la mayoría de ellos son demasiado largos o difíciles de entender, así que decidí escribir un nuevo tutorial simple y corto para Dagger2, espero que les guste.

¿Por qué lo necesitamos?

- Simplifica el acceso a instancias compartidas: proporciona una forma sencilla de obtener referencias a instancias compartidas, por ejemplo, una vez que declaramos en Dagger nuestras instancias singleton como `SharedPrefrences` , podemos declarar los campos con una simple anotación `@Inject` .
- Pruebas de unidad e integración más sencillas: podemos cambiar fácilmente los módulos que hacen que las respuestas de la red se burlen de este comportamiento.

Empezamos con un simple ejemplo.

La fuente completa del ejemplo está disponible en mi cuenta de GitHub.

Añadir Dagger2 dependencias

En primer lugar, debemos agregar las dependencias de Dagger2. Ponga el código debajo de su archivo build.gradle a nivel de módulo.

```
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 recibe un error como Error: Conflicto con la dependencia 'com.google.code.findbugs: jsr305' en el proyecto ': aplicación', debe agregar lo siguiente a su aplicación principal / build.gradle archivo.

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

Dos clases simples

Tenemos dos clases (vehículo y motor), la clase de vehículo necesita clase de motor para funcionar y MainActivity necesita la clase de vehículo. Usaremos Dagger2 para proporcionar estas instancias.

```
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();
  }
}
```

Clase motora

```
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;
  }
}
```

Clase de modulo

La clase de módulo es responsable de proporcionar objetos que se pueden inyectar. En este ejemplo, queremos inyectar clase de motor en clase de vehículo e clase de vehículo en MainActivity, por lo que deberíamos crear MyModule para proporcionar estas instancias.

```
@Module
class MyModule {

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

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

```
}
```

@Provide anotación: el objeto devuelto de este método está disponible para la inyección de dependencia.

Interfaz @Component

Dagger2 necesita una interfaz de componentes para saber cómo debe crear instancias de nuestras clases.

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

  void inject(MainActivity main);
}
```

Interfaz @Component: conexión entre el proveedor del objeto y los objetos que expresan una dependencia.

Inyectar dependencia en el constructor

Al agregar la anotación @Inject, dagger2 puede crear automáticamente una instancia de ese objeto como nuestro objeto Motor de ejemplo en la clase Vehículo.

Inyectar dependencia en MainClass

Dagger2 puede inyectar automáticamente dependencias en los constructores, pero los componentes de Android (actividades, fragmentos, etc.) están instanciados por el marco de trabajo de Android, lo que dificulta el uso de la inyección de dependencias en ellos, por lo que deberíamos inyectarlos manualmente, como en el siguiente código:

```
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);
  }
}
```

Eso es todo, espero que disfrutes.

Lea Empezando con la daga-2 en línea: https://riptutorial.com/es/dagger-

2/topic/7680/empezando-con-la-daga-2

Capítulo 2: @Named anotación en Kotlin

Introducción

Cómo usar correctamente la anotación nombrada en Kotlin v1.1

Examples

Declarar una dependencia cualificada

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

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

Inyección de dependencia basada en Setter

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

En el desarrollo de Android, esta es la forma en la que inyectas dependencias en `Activity` , `Fragment` o cualquier otro objeto que se ejecute directamente por el sistema operativo.

Para obtener más información sobre `@field:` anotación en Kotlin, visite la documentación

Inyección de dependencia basada en constructor

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

Lea @Named anotación en Kotlin en línea: https://riptutorial.com/es/dagger-2/topic/10812/-named-anotacion-en-kotlin

Creditos

S. No	Capítulos	Contributors
1	Empezando con la daga-2	Community, Hani, Jacob, Volodymyr Khodonovych
2	@Named anotación en Kotlin	Nicolás Carrasco