FREE eBook

# LEARNING dagger-2

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#dagger-2

# Table of Contents

# About

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

# Chapter 1: Getting started with dagger-2

## Remarks

This section provides an overview of what dagger-2 is, and why a developer might want to use it.

It should also mention any large subjects within dagger-2, and link out to the related topics. Since the Documentation for dagger-2 is new, you may need to create initial versions of those related topics.

## Versions

| Version | Release Date |
|---|---|
| 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 and Setup

**What is Dagger 2 ?**

The website describes itself as:

> Dagger is a fully static, compile-time dependency injection framework

The library makes it easy to model dependency graphs as well as to reuse objects. Since reflection is only used at compile time as part of the annotation processing Dagger 2 has improved speed for dependency injection.

**Setup**

1- Add support for annotation processing:

*Android*

Top level `build.gradle` script:

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

Module level `build.gradle` script:

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

*Java*

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

2- Add the dagger 2 dependencies

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

## Basic Example

### Define a module (the model of dependencies and their graph):

```
@Module
public class CoffeeModule{

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

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

}
```

Define a component:

```
@Component(
    modules={
        CoffeeModule.class
    }
)
interface CoffeeComponent {
```

```
        DeveloperActivity inject(DeveloperActivity developerActivity);
}
```

Inject the dependencies:

```
class DeveloperActivity extends ...{

    @Inject
    Coffee myCoffee;

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

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

    }

}
```

## Android example

One of the central difficulties of writing an Android application using Dagger is that many Android framework classes are instantiated by the OS itself, like `Activity` and `Fragment`, but Dagger works best if it can create all the injected objects. Instead, you have to perform members injection in a lifecycle method. Starting from version 2.10 dagger allows using `dagger.android` which simplify using dagger with android components.

### Injecting Activity objects

1. Install `AndroidInjectionModule` in your application component to ensure that all bindings necessary for these base types are available.

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

2. Start off by writing a `@Subcomponent` that implements [AndroidInjector][AndroidInjector], with a `@Subcomponent.Builder` that extends [AndroidInjector.Builder][AndroidInjector.Builder]:

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

3. After defining the subcomponent, add it to your component hierarchy by defining a module that binds the subcomponent builder and adding it to the component that injects your `Application`:

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

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

4. Next, make your `Application` implement `HasDispatchingActivityInjector` and `@Inject` a `DispatchingAndroidInjector<Activity>` to return from the activityInjector() method:

```
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. Finally, in your `Activity.onCreate()` method, call `AndroidInjection.inject(this)` before calling `super.onCreate();`:

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

This example was based on official dagger documentation. Working sample can be found on github

### Learn Dagger2 with simple example

I have read and watched a lot of different Dagger2 tutorials but most of them are too long or hard to understand so I decided to write a new simple and short tutorial for Dagger2, I hope you like it.

# Why we need it?

- Simplifies access to shared instances: It provides a simple way to obtain references to shared instances, for example once we declare in Dagger our singleton instances such as `SharedPrefrences` then we can declare fields with a simple `@Inject` annotation.

- Easier unit and integration testing: We can easily swap out modules that make network responses and mock out this behavior.

# Lest's start with a simple example

Full source of example is available on my GitHub account.

## Add Dagger2 dependencies

First of all we need to add Dagger2 dependencies, Put below code to your module-level build.gradle file.

```
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"
```

If you are getting an error like Error:Conflict with dependency 'com.google.code.findbugs:jsr305' in project ':app' you should add the following to your main app/build.gradle file.

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

## Two simple class

We have two classes (Vehicle and Motor), Vehicle class needs Motor class to run and MainActivity needs Vehicle class. We will use Dagger2 to provide these 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();
```

```
  }
}
```

Motor class:

```
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;
  }
}
```

## Module class

Module class is responsible for providing objects which can be injected, In this example we want to inject Motor class to Vehicle class and inject Vehicle class to MainActivity so  we should create MyModule to providing these instances.

```
@Module
class MyModule {

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

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

**@Provide annotation:** returned object from this method is available for dependency injection.

## @Component interface

Dagger2 needs component interface to know how should it create instances from our classes.

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

  void inject(MainActivity main);
}
```

**@Component interface:** connection between the provider of object and the objects which express a dependency.

## Inject Dependency in Constructor

By adding @Inject annotation, dagger2 can automatically create an instance from that object like our example Motor object in Vehicle class.

## Inject dependency in MainClass

Dagger2 can automatically inject dependencies in constructors, but Android components (activities, fragments, etc.) are instantiated by Android framework which makes it difficult to use dependency injection on them, so we should inject them manually like below code:

```
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);
  }
}
```

That's it, I hope you enjoy.

Read Getting started with dagger-2 online: https://riptutorial.com/dagger-2/topic/7680/getting-started-with-dagger-2

# Chapter 2: @Named annotation in Kotlin

## Introduction

How to correctly use the named annotation in Kotlin v1.1

## Examples

### Declaring a qualified dependency

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

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

### Setter based dependency injection

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

In Android Development, this is the way in which you inject dependencies into `Activity`, `Fragment` or any other object that is instantiated directly by the OS.

To learn more about the `@field:` annotation in Kotlin visit the documentation

### Constructor based dependency injection

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

Read @Named annotation in Kotlin online: https://riptutorial.com/dagger-2/topic/10812/-named-annotation-in-kotlin

# Credits

| S. No | Chapters | Contributors |
|---|---|---|
| 1 | Getting started with dagger-2 | Community, Hani, Jacob, Volodymyr Khodonovych |
| 2 | @Named annotation in Kotlin | Nicolás Carrasco |