Dependency Injection med Dagger 2

SB 2015-08-31/tra

Hvad ER problemet?

```
X x = new ...()
```

- "new" kræver vi ved hvilken implementation af X vi vil bruge, når koden skrives.
- En Factory flytter bare problemet til et andet sted.
- Ender med "new" eller *Stringly typed.* (hvor javac ikke hjælper dig).

Løsning: Parametre og knofedt

- Løsningen er at de ting som koden skal bruge for at kunne virke, sendes ind “udefra”.
- Hvis hele ens applikation bygges sådan, ender man med en bunke moduler, og en main-rutine der skal sætte det hele sammen inden programmet skal køres. → Javakodebøvl.

Knofedt?!? “Føj da, lad os lave et framework!”

- “Lad os lave et framework der hjælper med at sætte ting sammen” → XML bøvl, Reflection bøvl, Bøvl bøvl.
- Fantastisk når det virker, umuligt at fejlfinde på.
- Min konklusion: INGEN magi på runtime.

“Ikke runtime? Dvs det er ok på compile-time?”

- Jep.
- Genistreg#1: Lad java compileren gøre arbejdet vha annotations!
- Genistreg#2: Generér gode, letlæste Java-sourcer der udfører samme trin som en dygtig programmør ville have lavet i hånden.

  Dvs ingen magi - kun almindelig Java-kode på runtime.

Hello World i Dagger2

```java
package com.example.dagger;

import dagger.Component;
import dagger.Module;
import dagger.Provides;

public class Example {

  interface Printer {
    void printMsg(String msg);
  }

  static class ConsolePrinter implements Printer {
    @Override
    public void printMsg(String msg) {
      System.out.println(msg);
    }
  }

  @Component(modules = ConsoleModule.class)
  interface HelloWorldApp {
    Printer getPrinter();
  }

  @Module
  static class ConsoleModule {
    @Provides
    Printer providePrinter() {
      return new ConsolePrinter();
    }
  }

  public static void main(String[] args) {
    HelloWorldApp app = DaggerExample_HelloWorldApp.create();
    app.getPrinter().printMsg("Hello World!");
  }
}
```

Hello World i Dagger2

`@Component` angiver de metoder der skal kunne kaldes fra main og de moduler Dagger skal bruge hér.

`@Module` opremser de @Providers som Dagger ikke selv kan gætte sig til.

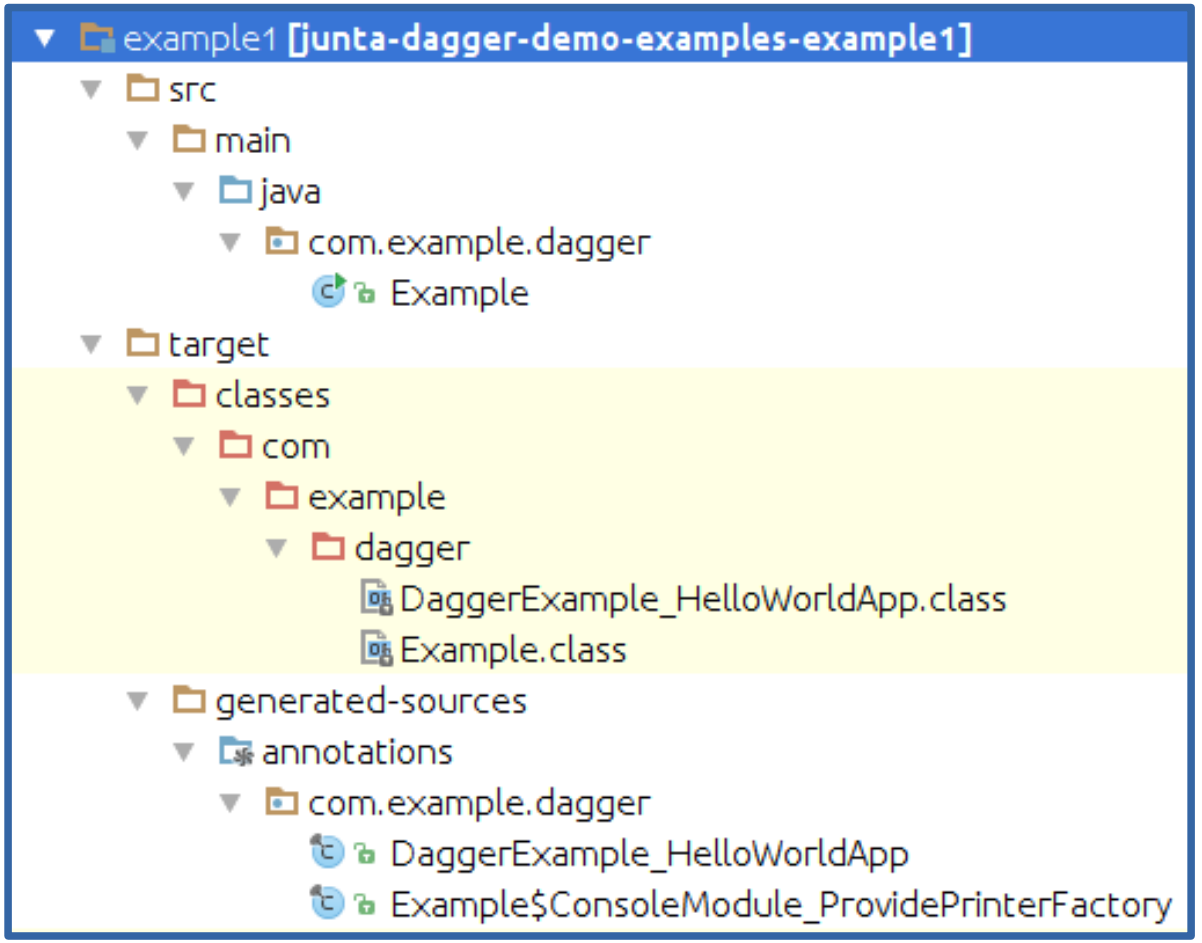
`Dagger`… klasserne er automatisk genereret af javac.

```java
package com.example.dagger;

import dagger.Component;
import dagger.Module;
import dagger.Provides;

public class Example {

  interface Printer {
    void printMsg(String msg);
  }

  static class ConsolePrinter implements Printer {
    @Override
    public void printMsg(String msg) {
      System.out.println(msg);
    }
  }

  @Component(modules = ConsoleModule.class)
  interface HelloWorldApp {
    Printer getPrinter();
  }

  @Module
  static class ConsoleModule {
    @Provides
    Printer providePrinter() {
      return new ConsolePrinter();
    }
  }

  public static void main(String[] args) {
    HelloWorldApp app = DaggerExample_HelloWorldApp.create();
    app.getPrinter().printMsg("Hello World!");
  }
}
```

Fra kommandolinien:

```
tra@tra-laptop:~/x/src$ ls -l ..
total 2736
-rw-r----- 1 tra tra   17559 aug 27 13:32 dagger-2.0.1.jar
-rw-r----- 1 tra tra  472766 aug 27 13:32 dagger-compiler-2.0.1.jar
-rw-r----- 1 tra tra   14216 aug 27 13:37 dagger-producers-2.0-beta.jar
-rw-rw-r-- 1 tra tra     722 aug 27 13:45 Example.java
-rw-r----- 1 tra tra 2256213 aug 27 13:34 guava-18.0.jar
-rw-r----- 1 tra tra    2497 aug 27 13:38 javax.inject-1.jar
drwxrwxr-x 2 tra tra    4096 aug 27 13:52 src
tra@tra-laptop:~/x/src$ mkdir -p com/example/dagger
tra@tra-laptop:~/x/src$ cp ../Example.java com/example/dagger
tra@tra-laptop:~/x/src$ javac -cp ../dagger-2.0.1.jar:../dagger-compiler-2.0.1.jar:../guava-18.0.jar:../d
agger-producers-2.0-beta.jar:../javax.inject-1.jar com/example/dagger/Example.java
tra@tra-laptop:~/x/src$ ls -l com/example/dagger/
total 48
-rw-rw-r-- 1 tra tra  267 aug 27 13:53 DaggerExample_HelloWorldApp$1.class
-rw-rw-r-- 1 tra tra 1465 aug 27 13:53 DaggerExample_HelloWorldApp$Builder.class
-rw-rw-r-- 1 tra tra 2156 aug 27 13:53 DaggerExample_HelloWorldApp.class
-rw-rw-r-- 1 tra tra 1491 aug 27 13:53 DaggerExample_HelloWorldApp.java
-rw-rw-r-- 1 tra tra  845 aug 27 13:53 Example.class
-rw-rw-r-- 1 tra tra  629 aug 27 13:53 Example$ConsoleModule.class
-rw-rw-r-- 1 tra tra 1642 aug 27 13:53 Example$ConsoleModule_ProvidePrinterFactory.class
-rw-rw-r-- 1 tra tra  901 aug 27 13:53 Example$ConsoleModule_ProvidePrinterFactory.java
-rw-rw-r-- 1 tra tra  554 aug 27 13:53 Example$ConsolePrinter.class
-rw-rw-r-- 1 tra tra  521 aug 27 13:53 Example$HelloWorldApp.class
-rw-rw-r-- 1 tra tra  722 aug 27 13:53 Example.java
-rw-rw-r-- 1 tra tra  238 aug 27 13:53 Example$Printer.class
tra@tra-laptop:~/x/src$ java -cp ../dagger-2.0.1.jar:../javax.inject-1.jar:. com.example.dagger.Example
Hello World!
tra@tra-laptop:~/x/src$
```

Maven modul i IntelliJ



Note: target/generated-sources/annotations skal eksistere ved import i IntelliJ!
(Hey, men virkede for Kåre da han gjorde det)

Konfigurationsstrenge:

```
String password = <mumle>.get(“password”);
```

- Slæb konfigurationsobjekt med rundt fra main helt ned til laveste niveau der faktisk skal bruge det.
- Hav “globalt” konfigurationobjekt.
- Misbrug de globale system properties.
- JNDI (ja, også udenfor Java EE)
- Ingen central styring uanset valg.

String kan injectes!

```
@Inject
public X(@Named(“password”) String password) {
  this.password = password;
}
```

- String er en klasse som alle andre.
- Hvis en DI-engine skal kunne skelne mellem flere af samme klasse, brug JSR-330 `@Named`.
- Dagger kræver en provider-metode pr `@Named` string i `@Module`-modulet.

```
@Provides
@Named(“password”) String providePassword() {
  return “swordfish”;
}
```

“Before - Hello World - After”

```java
package com.example.dagger;
import dagger.Component;
import dagger.Module;
import dagger.Provides;
import javax.inject.Named;
import java.sql.SQLException;
public class NamedStringExample {
    public static void main(String[] args) throws SQLException {
        Example2App app = DaggerNamedStringExample_Example2App.create();
        app.getPrinter().printMsg("Hello World");
    }
    interface Printer {
        void printMsg(String msg);
    }
    @Component(modules = ConsoleModule.class)
    interface Example2App {
        Printer getPrinter();
    }
    @Module
    static class ConsoleModule {
        @Provides
        Printer providePrinter(@Named("before") String before, @Named("after") String after) {
            return msg -> System.out.println(before + " - " + msg + " - " + after);
        }
        @Provides @Named("before") String provideBefore() {
            return "Before";
        }
        @Provides @Named("after") String provideAfter() {
            return "After";
        }
    }
}
```

For at kunne kalde en given metode – først providePrinter(…) - skal Dagger kunne fremskaffe dens parametre. Her ved at kalde provideBefore() og provideAfter().

Denne process er rekursiv hvilket muliggør injektion af konfigurationsstrenge vilkårligt dybt.

```java
package com.example.dagger;
import dagger.Component;
import dagger.Module;
import dagger.Provides;

import javax.inject.Named;
import java.sql.SQLException;
public class NamedStringExampleConstants {
    public static final String BEFORE = "before";
    public static final String AFTER = "after";
    public static void main(String[] args) throws SQLException {
        Example2App app = DaggerNamedStringExampleConstants_Example2App.create();
        app.getPrinter().printMsg("Hello World");
    }
    interface Printer {
        void printMsg(String msg);
    }
    @Component(modules = ConsoleModule.class)
    interface Example2App {
        Printer getPrinter();
    }
    @Module
    static class ConsoleModule {
        @Provides
        Printer providePrinter(@Named(BEFORE) String before, @Named(AFTER) String after) {
            return msg -> System.out.println(before + " - " + msg + " - " + after);
        }
        @Provides @Named(BEFORE) String provideBefore() { return "Before"; }
        @Provides @Named(AFTER) String provideAfter() { return "After"; }
    }
}
```

Konstanter hjælper!

Konfiguration via @Named Strings

- Konfigurationsværdier styres centralt.
- Moduler kan genbruges.
- Vi rammer konkret kode når en værdi skal slås op og ikke når den beregnes. (Fordel ved debugning).
- Hver konfigurationsværdi har sin egen metode som kan have fuld javadoc og individuel fejlhåndtering (defaultværdi? RuntimeException?)

Constructor versus Setter DI

- Eksisterende kode er lettest at tilrette via setter injection (`@Inject void setX(…)`)
- Ny kode bliver efter TRA's mening bedst (mest modulær up front) med constructor injection (hvor alle dependencies angives som parametre til constructoren).

Diverse

- `@Singleton` – lav kun et eksemplar uanset hvor mange gange det skal injectes.
- `Provider<X>` er DI-udgaven af Factory pattern.
- `Lazy<X> x`. Instansen af X konstrueres først når "x.get()" kaldes, og kun en gang.
- IntelliJ skal have `Settings → Build,Execution,Deployment → Compiler → Make project automatically` vinget af for at virke godt.

SB relevans.

- Forøget brug af interfaces → Nemmere at mocke “opad” og helst bedre snitflader generelt i modulerne.
- Bedre konvention for at erkende og udskille projektspecifikke ting fra generel kode. Det er et paradigmeskift at få ting ind “udefra” som det kan tage tid at vænne sig til, men som resulterer i bedre kode.
- Konfigurationsværdihåndtering kan centraliseres i en given applikation, og kun de værdier der faktisk BRUGES behøver @Providers.
- Ikke helt trivielt at lægge om, så nok kun for ny kode. dpaviser-qa-tool kunne være en god test.

Resourcer

- Spartansk hjemmeside:
  http://google.github.io/dagger/
- Intro-video – viser alle principperne:
  https://www.youtube.com/watch?v=oK_XtfXPkqw
- Slides plus kode:
  https://github.com/ravn/junta-dagger-demo-examples
-