Spring - Beans - Properties

Spis treści

Który Bean wstrzyknąć?

Jak Spring rozpoznaje Beany do wstrzyknięcia

Domyślnie, Spring rozpoznaje, które Beany ma wstrzyknąć przy wykorzystaniu adnotacji `@Autowired` za pomocą ich typu. Jeżeli w kontenerze zostanie zarejestrowany więcej niż jeden Bean tego samego typu, Spring nie będzie wiedział, który z nich ma zostać wstrzyknięty i zostanie wyrzucony wyjątek. Przejdźmy do przykładu:

```
package pl.zajavka;

public interface SomeCommonInterface {}
```

```
package pl.zajavka;

import org.springframework.stereotype.Component;

@Component
public class SomeBean1 implements SomeCommonInterface {}
```

```
package pl.zajavka;
```

```
import org.springframework.stereotype.Component;

@Component
public class SomeBean2 implements SomeCommonInterface {}
```

```
package pl.zajavka;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackageClasses = Marker.class)
public class ExampleConfiguration {}
```

```
package pl.zajavka;

public class Marker {}
```

```
package pl.zajavka;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class ExampleUsage {

    public static void main(String[] args) {
        ApplicationContext context
            = new AnnotationConfigApplicationContext(ExampleConfiguration.class); ①
    }
}
```

① Jeżeli uruchomimy teraz ten program, to wszystko jest w porządku, nie dostajemy błędów w trakcie działania programu.

Dodajmy natomiast do przedstawionego przykładu poniższą klasę:

```
package pl.zajavka;

import org.springframework.stereotype.Component;

@Component
public class SomeService {

    public SomeService(final SomeCommonInterface someCommonInterface) {
        this.someCommonInterface = someCommonInterface;
    }

    private SomeCommonInterface someCommonInterface;
}
```

Stworzyliśmy serwis, który jest zależny od interfejsu `SomeCommonInterface`. Interfejs ten ma dwie implementacje: `SomeBean1` oraz `SomeBean2`. Obie te klasy są oznaczone adnotacjami Springa, co oznacza, że

Spring ma je używać w swoim kontekście. Jeżeli teraz uruchomisz ten sam program, to na ekranie zostanie wydrukowany poniższy błąd:

```
org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type
'pl.zajavka.SomeCommonInterface' available: expected single matching bean but found 2:
someBean1,someBean2
```

Czyli Spring mówi nam: *Hej, mam wstrzyknąć Bean, który implementuje interface* `SomeCommonInterface`, *ale są dwa takie Beany:* `someBean1` *i* `someBean2`. *Nie wiem, który mam wziąć, musisz dać mi więcej informacji.* To pokazuje nam, że Spring domyślnie dopasowuje Beany do wstrzyknięcia na podstawie typu danych. Jeżeli w kontenerze zostanie zarejestrowany więcej niż jeden Bean tego samego typu, Spring nie będzie wiedział, który z nich ma zostać wstrzyknięty i zostanie wyrzucony wyjątek. Inaczej mówiąc, nie mamy dostępnego jednoznacznie pasującego Beana, bo pasują nam dwa.

Przypominam, że od Springa **4.3**, jeżeli klasa ma tylko jeden konstruktor, to nie ma potrzeby dodawania adnotacji `@Autowired`

Zanim przejdziemy do rozwiązania tego problemu, wyjaśnijmy w jaki sposób Spring nadaje Beanom nazwy.

W jaki sposób Spring nadaje Beanom nazwy

XML

Zwróć uwagę, że gdy definiowaliśmy konfigurację Spring przy wykorzystaniu **XML**, to sami nadawaliśmy tym Beanom identyfikatory. Przykład:

```
<bean id="exampleBean" class="pl.zajavka.ExampleBean"> ①
    <!-- treść -->
</bean>
```

① W tej linijce zostało nadane id: *exampleBean*.

Natomiast gdy definiowaliśmy konfigurację Java, bean nosił taką nazwę jak metoda, która go tworzyła. Przy konfiguracji przez adnotacje, nigdzie nie nadawaliśmy tych identyfikatorów manualnie. W pokazanych przykładach Spring robił to za nas.

Konfiguracja Java

```
@Configuration
public class ExampleConfigurationClass {

    @Bean
    public ExampleBean exampleBean(InjectedBean injectedBean) { ①
        return new ExampleBean(injectedBean);
    }

    @Bean
    public InjectedBean injectedBean() { ②
```

```
        return new InjectedBean();
    }

}
```

① Ten Bean będzie nosił identyfikator: *exampleBean*.

② Ten Bean będzie nosił identyfikator: *injectedBean*.

W powyższym przypadku, Spring nazwie Bean adekwatnie do nazwy metody, która służy do jego stworzenia. Czyli Bean będzie nazywał się tak samo jak metoda, która go tworzy.

Konfiguracja przez adnotacje

Przejdźmy do sytuacji, gdzie konfigurujemy Beany przy wykorzystaniu adnotacji:

```
@Service
public class CalculationService {}
```

W powyższym przykładzie Bean będzie nosił nazwę *calculationService*. Czyli Spring bierze nazwę klasy i nazywa Bean tak samo, tylko pierwsza litera jest zamieniana na małą literę.

W sytuacji która była przedstawiona wcześniej, gdy mamy dostępnych kilka Beanów tego samego typu (mówimy tutaj w odniesieniu do relacji IS-A), musimy być dokładniejsi. Możemy do tego wykorzystać adnotację `@Qualifier`.

Qualifier

Spróbujmy przepisać teraz klasę `SomeService` wykorzystując jednocześnie adnotację `@Qualifier`:

```
@Component
public class SomeService {

    public SomeService(final @Qualifier("someBean2") SomeCommonInterface someCommonInterface) {
        this.someCommonInterface = someCommonInterface;
    }

    private SomeCommonInterface someCommonInterface;
}
```

Mówimy w tym momencie Springowi, że w miejsce zmiennej `someCommonInterface` ma zostać wstrzyknięty Bean, który nosi identyfikator `someBean2`. Bean taki został nazwany domyślnie przez Springa na podstawie klasy `SomeBean2`.

Jak możemy sami nadać Beanom nazwy

Konfiguracja Java

Skoro w trakcie tworzenia konfiguracji przy wykorzystaniu adnotacji `@Bean`, Beany nazywają się tak jak metody, które je tworzą, to najprostszym sposobem jest odpowiednie nazwanie metody.

Konfiguracja przez adnotacje

W tym przypadku możemy również narzucić swoją własną nazwę Beanów w poniższy sposób:

```
package pl.zajavka;

import org.springframework.stereotype.Component;

@Component(value = "myOwnBeanName")
public class SomeBean2 implements SomeCommonInterface {}
```

W tym przypadku, Bean powstały na podstawie definicji klasy `SomeBean2` będzie nosił nazwę `myOwnBeanName`. Będziemy mogli ponownie odwołać się do niego w poniższy sposób:

```
@Component
public class SomeService {

    public SomeService(final @Qualifier("myOwnBeanName") SomeCommonInterface someCommonInterface) {
        this.someCommonInterface = someCommonInterface;
    }

    private SomeCommonInterface someCommonInterface;
}
```

Autowire by Name

Istnieje prostszy sposób na rozwiązanie tego problemu. Klasę `SomeService` możemy zapisać w poniższy sposób:

```
@Component
public class SomeService {

    public SomeService(final SomeCommonInterface myOwnBeanName) {
        this.someCommonInterface = myOwnBeanName;
    }

    private SomeCommonInterface someCommonInterface;
}
```

W takiej sytuacji nie potrzebuję adnotacji `@Qualifier`. Wystarczy, że nazwę parametr konstruktora dokładnie tak samo jak nazywa się Bean, który chcę wstrzyknąć. Należy jednak pamiętać, że domyślnym sposobem jest wstrzykiwanie na podstawie typu. Nazwa Beana będzie użyta dopiero gdy nie uda się znaleźć Beana, który będzie mógł być dopasowany na podstawie typu.

Możesz pobawić się w ten sam sposób stosując adnotację `@Autowired` na polu, natomiast nie jest to zalecane podejście, dlatego skupiliśmy się na wstrzykiwaniu przez konstruktor.

Cykl życia Beanów

W języku angielskim: *The Spring Bean Lifecycle*. Wcześniej powiedzieliśmy sobie, że Bean ma swój cykl życia, za który odpowiada kontener. Cykl życia takiego obiektu można w uproszczeniu przedstawić w postaci trzech faz: *inicjalizacji*, *użycia* i *zniszczenia*.

Spring zarówno podczas tworzenia Beanów jak i podczas ich niszczenia, wykonuje kolejne kroki - przechodzi przez kolejne fazy. Chcę tutaj jednak zaznaczyć, że tematyka ta jest poruszona nie po to, żeby kazać Ci nauczyć się tych kroków na pamięć, tylko żeby zaznaczyć, że takie zjawisko istnieje i możemy kiedyś potrzebować je wykorzystać na własne potrzeby. Nie chcę nawet wypisywać tych wszystkich kroków i poruszać wszystkich możliwych sposobów na dodanie swojego kodu na etapie tworzenia i niszczenia Beana. Pokażę jeden z możliwych sposobów, żeby zobrazować na czym to polega, co możemy dzięki temu osiągnąć i w jaki sposób możemy to zrobić. Spójrz na przykład poniżej:

```
public class SomeService {

    public void initMethod() {
        System.out.println("SomeService init-method");
    }

    public void destroyMethod() {
        System.out.println("SomeService destroy-method");
    }

    public SomeService(final SomeCommonInterface someBean1) {
        this.someCommonInterface = someBean1;
        System.out.println("SomeService constructor");
    }

    private SomeCommonInterface someCommonInterface;
}
```

```
@Configuration
@ComponentScan(basePackageClasses = Marker.class)
public class ExampleConfiguration {

    @Bean(initMethod = "initMethod", destroyMethod = "destroyMethod")
    public SomeService someService(final SomeCommonInterface someBean1) {
        return new SomeService(someBean1);
    }
}
```

```
public class ExampleUsage {

    public static void main(String[] args) {
        System.out.println("Before context creation");
        AbstractApplicationContext context
            = new AnnotationConfigApplicationContext(ExampleConfiguration.class);
        System.out.println("After context creation");

        System.out.println("Before Bean retrieval");
        context.getBean(SomeService.class);
        System.out.println("After Bean retrieval");
```

```
        System.out.println("Before closing context");
        context.close();
        System.out.println("After closing context");
    }
}
```

Jeżeli postawimy teraz Spring Context to na ekranie zostanie wydrukowane:

```
Before context creation
SomeService constructor
SomeService init-method
After context creation
Before Bean retrieval
After Bean retrieval
Before closing context
SomeService destroy-method
After closing context
```

Powyższy przykład obrazuje kilka kwestii:

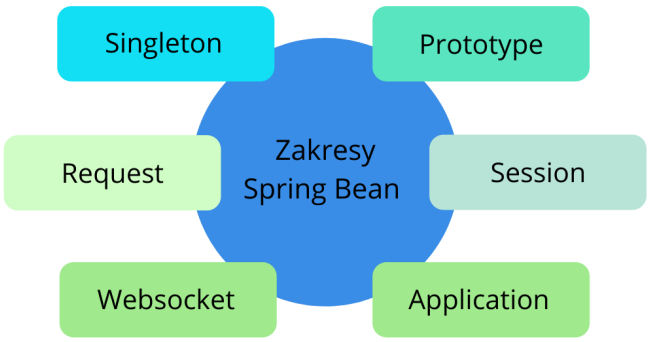
- Metoda `destroyMethod()` zostanie wywołana tylko gdy ręcznie zamkniemy Spring context. Jeżeli program zakończy się samoistnie, a my nie wywołamy `context.close();`, metoda ta nie zostanie wywołana,
- Spring daje nam możliwość "opakowania" się wokół tworzenia Beanów, przedstawiony sposób jest jednym z wielu możliwych,
- Na podstawie przykładu widać, że `initMethod()` została wywołana zanim Spring context został utworzony,

Może pojawić Ci się pytanie, czemu wołać `initMethod()` zamiast konstruktora? Powodów jest więcej niż te poniżej wymienione, ale na naszym poziomie zaawansowania wystarczą dwa poniższe:

- Jeżeli stosujemy wstrzykiwanie przez setter, to metoda `initMethod()` zostanie wywołana po zakończeniu wstrzykiwania. Jeżeli stosujemy wstrzykiwanie przez setter i chcemy, żeby po wstrzyknięciu wszystkich zależności została wykonana jakaś operacja, możemy wtedy łatwo skorzystać z metody `initMethod()`. Inaczej musielibyśmy określać, który setter wykona się jako ostatni i do niego dodawać jakiś kod. Łatwiej jest się oprzeć o `initMethod()`,
- Jeżeli dodajemy w konstruktorze jakąś logikę, to może być ona problematyczna w testowaniu. W kolejnych warsztatach przejdziemy do testowania aplikacji Spring.

Zakresy Spring Beanów

W kontekście pracy z Beanami musimy jeszcze omówić coś takiego jak zakres Spring Beana (*Spring Beans Scope*). Spring Framework obsługuje kilka takich zakresów. Domyślny zakres nazywa się `Singleton` i nie wymaga on jawnego definiowania. Zwróć uwagę, że w poprzednich przykładach nigdzie nie określaliśmy czegoś takiego jak **scope** - wykorzystywany był ten domyślny.



Obraz 1. Zakresy Spring Beanów

Możemy wyróżnić kilka zakresów Spring Beanów:

- **singleton** - jedna instancja serwisu na `ApplicationContext`. Pamiętasz wzorzec **Singleton**? Właśnie tutaj jest on używany. W tym przypadku w całej aplikacji jest używana jedna i ta sama instancja danego Beana.
- **prototype** - dowolna liczba instancji serwisu na `ApplicationContext`. W tym przypadku Spring będzie tworzył nową instancję danego Beana, za każdym razem gdy będziemy potrzebowali pobrać jego instancję.
- Zakresy dostępne wyłącznie dla kontekstu webowego (jeszcze o tym nie rozmawialiśmy, będzie później, ale wymienimy je teraz):
  - **request** - jedna instancja serwisu na jedno żądanie HTTP
  - **session** - jedna instancja serwisu na jedną sesje HTTP
  - **application** - jedna instancja serwisu na ServerContext
  - **websocket** - jedna instancja serwisu na WebSocket

Istnieje też możliwość utworzenia własnego zakresu.

Przykłady definicji dla zakresu **Singleton**:

w klasach możemy wykorzystać jedną z kilku możliwości:

```
@Scope("singleton")
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
```

```
@Component
@Scope("singleton")
public class ExampleBean {
}
```

w plikach XML:

```
<bean id="exampleBean" class="pl.zajavka.ExampleBean" scope="singleton"/>
```

Przykłady działania zakresów Spring Beanów

Tworzenie instancji Spring Beanów

Do zobrazowania różnicy pomiędzy **Singletonem**, a **Prototypem** napiszemy prosty kod, który będzie tworzył dwa odpowiadające tym zakresom serwisy: `ExampleSingletonBean` i `ExamplePrototypeBean` oraz trzeci serwis: `InjectedBean`, od którego będą zależeć dwa poprzednie. Ich kod poniżej:

Definicja serwisu `InjectedBean` *jako Singleton (domyślna konfiguracja, nie wymaga jawnego ustawiania).*

```
package pl.zajavka;

import org.springframework.stereotype.Service;

@Service
public class InjectedBean {}
```

Definicja `ExampleBeansScanningConfiguration`.

```
package pl.zajavka.imported.annotationbased.autowiring.annotation;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("pl.zajavka")
public class ExampleBeansScanningConfiguration {
}
```

Definicja serwisu `ExampleSingletonBean` *jako Singleton.*

```
package pl.zajavka:

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
public class ExampleSingletonBean {

    private InjectedBean injectedBean;

    @Autowired
    public ExampleSingletonBean(InjectedBean injectedBean) {
        this.injectedBean = injectedBean;
    }

    public void exampleMethod() {
        injectedBean.anotherExampleMethod();
    }

    public InjectedBean getInjectedBean() {
        return injectedBean;
    }
```

```
}
```

Definicja serwisu `ExamplePrototypeBean` *jako Prototype.*

```
package pl.zajavka;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class ExamplePrototypeBean {

    private InjectedBean injectedBean;

    @Autowired
    public ExamplePrototypeBean(InjectedBean injectedBean) {
        this.injectedBean = injectedBean;
    }

    public void exampleMethod() {
        injectedBean.anotherExampleMethod();
    }

    public InjectedBean getInjectedBean() {
        return injectedBean;
    }
}
```

Żeby pokazać Ci różnice pomiędzy zakresami, stworzyliśmy klasę z metodą `main()`, która z kontekstu Springowego dwa razy bierze serwis `ExampleSingletonBean` oraz dwa razy serwis `ExamplePrototypeBean`. Oczekiwanym wynikiem tego sprawdzenia jest potwierdzenie, że dla serwisów zdefiniowanych jako **Singleton** kontekst zawsze zwraca tę samą instancję serwisu. A dla serwisów zdefiniowanych jako **Prototype** kontekst zawsze zwraca nową instancję serwisu.

Przykładowe użycie zdefiniowanych serwisów.

```
package pl.zajavka;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class ExampleSpringUsage {

    public static void main(String[] args) {

        ApplicationContext context
            = new AnnotationConfigApplicationContext(ExampleBeansScanningConfiguration.class);

        ExampleSingletonBean singleton1 = context.getBean(ExampleSingletonBean.class);
        ExampleSingletonBean singleton2 = context.getBean(ExampleSingletonBean.class);

        ExamplePrototypeBean prototype1 = context.getBean(ExamplePrototypeBean.class);
        ExamplePrototypeBean prototype2 = context.getBean(ExamplePrototypeBean.class);
```

```
        System.out.println("singleton1 == singleton2? " + (singleton1 == singleton2));
        System.out.println("prototype1 == prototype2? "  + (prototype1 == prototype2));

        InjectedBean injectedBean = context.getBean(InjectedBean.class);
        boolean isInjectedBeanASingleton =
                injectedBean == singleton1.getInjectedBean()
                    && injectedBean == singleton2.getInjectedBean()
                    && injectedBean == prototype1.getInjectedBean()
                    && injectedBean == prototype2.getInjectedBean();
        System.out.println("Is injectedBean a Singleton? " + isInjectedBeanASingleton);
    }

}
```

Tak jak oczekiwaliśmy, dwa obiekty `ExampleSingletonBean` są tą samą instancją, natomiast obiekty `ExamplePrototypeBean` są różnymi instancjami. Dodatkowo ta sama instancja `InjectedBean`, z domyślną konfiguracją zakresu jako **Singleton**, jest wstrzyknięta zarówno w `ExampleSingletonBean`, jak i w `ExamplePrototypeBean`.

Widok konsoli po wykonaniu metody `main()`

```
singleton1 == singleton2? true
prototype1 == prototype2? false
Is injectedBean a Singleton? true
```

Zmiana stanu w Spring Beanie

Ale dlaczego te zakresy Spring Beanów są tak ważne? Już wyjaśniamy! Tak jak poprzednio mamy `ExampleSingletonBean` i `ExamplePrototypeBean`, ale już bez `InjectedBean`, za to ze zmienną `exampleValue`. W poniższym przykładzie będziemy manipulować wartością zmiennej `exampleValue` w identyczny sposób dla obu serwisów i zobaczymy, co z tego wyniknie.

Definicja serwisu `ExampleSingletonBean` *jako Singleton.*

```
package pl.zajavka;

import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
public class ExampleSingletonBean {
    private int exampleValue = 1;

    public int getExampleValue() {
        return exampleValue;
    }

    public void setExampleValue(int exampleValue) {
        this.exampleValue = exampleValue;
    }
}
```

Definicja serwisu `ExamplePrototypeBean` *jako Prototype.*

```
package pl.zajavka;

import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class ExamplePrototypeBean {

    private int exampleValue = 1;

    public int getExampleValue() {
        return exampleValue;
    }

    public void setExampleValue(int exampleValue) {
        this.exampleValue = exampleValue;
    }
}
```

Definicja serwisu konfiguracyjnego.

```
package pl.zajavka.imported.annotationbased.autowiring.annotation;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("pl.zajavka")
public class ExampleBeansScanningConfiguration {
}
```

Tym razem w metodzie `main()` wyciągamy z kontekstu po dwa obiekty `ExampleSingletonBean` i `ExamplePrototypeBean`. Później zmieniamy stan zmiennej `exampleValue` dla jednego z nich i sprawdzamy co i gdzie tak naprawdę się zmieniło.

Przykładowe użycie zdefiniowanych serwisów.

```
package pl.zajavka;


import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class ExampleSpringUsage {

    public static void main(String[] args) {
        ApplicationContext context =
                new AnnotationConfigApplicationContext(ExampleBeansScanningConfiguration.class);

        ExampleSingletonBean singleton1 = context.getBean(ExampleSingletonBean.class);
        ExampleSingletonBean singleton2 = context.getBean(ExampleSingletonBean.class);

        System.out.println("--- Singleton ---");
```

```
        System.out.println("singleton1.exampleValue = " + singleton1.getExampleValue());
        System.out.println("singleton2.exampleValue = " + singleton2.getExampleValue());

        System.out.println("singleton1.setExampleValue(2)");
        singleton1.setExampleValue(2);

        System.out.println();
        System.out.println("singleton1.exampleValue = " + singleton1.getExampleValue());
        System.out.println("singleton2.exampleValue = " + singleton2.getExampleValue());

        ExamplePrototypeBean prototype1 = context.getBean(ExamplePrototypeBean.class);
        ExamplePrototypeBean prototype2 = context.getBean(ExamplePrototypeBean.class);

        System.out.println();
        System.out.println("--- Prototype ---");
        System.out.println("prototype1.exampleValue = " + prototype1.getExampleValue());
        System.out.println("prototype2.exampleValue = " + prototype2.getExampleValue());

        System.out.println("prototype1.setExampleValue(2)");
        prototype1.setExampleValue(2);
        System.out.println();

        System.out.println("prototype1.exampleValue = " + prototype1.getExampleValue());
        System.out.println("prototype2.exampleValue = " + prototype2.getExampleValue());
    }

}
```

Wyniki są podzielone na części dotyczące **Singletona** i **Prototypu**. W przypadku **Singletona** zmiana stanu zmiennej `exampleValue` z `1` na `2` skutkuje zmianą stanu we wszystkich instancjach **Singletona**. W przypadku **Prototypu** ta sama operacja skutkuje zmianą stanu zmiennej `exampleValue` jedynie w konkretnej instancji serwisu.

Widok konsoli po wykonaniu metody main()

```
--- Singleton ---
singleton1.exampleValue = 1
singleton2.exampleValue = 1

singleton1.setExampleValue(2)

singleton1.exampleValue = 2
singleton2.exampleValue = 2

--- Prototype ---
prototype1.exampleValue = 1
prototype2.exampleValue = 1

prototype1.setExampleValue(2)

prototype1.exampleValue = 2
prototype2.exampleValue = 1
```

Problem wstrzykiwania Prototype w Singleton

Ale dlaczego od razu problem? Przecież wcześniej były opisane Spring scopes, jest **Singleton**, jest

Prototype i wszystko powinno działać. No właśnie nie! Poniższy problem nazwany jest **scoped bean injection problem** i już przedstawiamy przykład.

```
package pl.zajavka;

import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.*;

@Configuration
@ComponentScan(basePackageClasses = Marker.class)
public class SomeApplicationConfiguration {

    @Bean
    public SingletonBean singletonBean(final PrototypeBean prototypeBean) {
        return new SingletonBean(prototypeBean);
    }

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public PrototypeBean prototypeBean() {
        return new PrototypeBean();
    }
}
```

```
package pl.zajavka;

public class Marker {}
```

```
package pl.zajavka;

import java.time.LocalTime;

public class SingletonBean {

    private PrototypeBean prototypeBean;

    public SingletonBean(final PrototypeBean prototypeBean) {
        this.prototypeBean = prototypeBean;
        System.out.println("Singleton created");
    }

    public PrototypeBean callPrototype() {
        System.out.println("Calling callPrototype: " + LocalTime.now());
        return prototypeBean;
    }
}
```

```
package pl.zajavka;

public class PrototypeBean {

    public PrototypeBean() {
        System.out.println("Prototype created");
    }
```

```
}
```

```
package pl.zajavka;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class ExampleUsage {

    public static void main(String[] args) {
        ApplicationContext context
            = new AnnotationConfigApplicationContext(SomeApplicationConfiguration.class);

        SingletonBean firstSingleton = context.getBean(SingletonBean.class);
        PrototypeBean firstPrototype = firstSingleton.callPrototype();

        SingletonBean secondSingleton = context.getBean(SingletonBean.class);
        PrototypeBean secondPrototype = secondSingleton.callPrototype();

        System.out.println("firstPrototype == secondPrototype? " + (firstPrototype == secondPrototype));
    }
}
```

Jeżeli uruchomimy przygotowany powyżej przykład, na ekranie zobaczymy przykładowy wydruk:

```
Prototype created
Singleton created
Calling callPrototype: 20:42:16.651032200
Calling callPrototype: 20:42:16.653944700
firstPrototype == secondPrototype? true
```

Na tyle co już zdążyliśmy poznać scope **Prototype**, powinniśmy za każdym razem otrzymać nową instancję - a jednak wiadomość "Prototype created" drukowana jest tylko raz. Co więcej, `firstPrototype == secondPrototype` zwróciło `true`, czyli mamy nadal do czynienia z jednym i tym samym Beanem.

Przedstawiony sposób obejścia tego problemu nie jest jedynym możliwym, jest jednym z wielu. Jeżeli teraz zmodyfikujemy klasę `SingletonBean` do takiej postaci:

```
package pl.zajavka;

import org.springframework.beans.factory.ObjectFactory;

import java.time.LocalTime;

public class SingletonBean {

    private ObjectFactory<PrototypeBean> factory;

    public SingletonBean(final ObjectFactory<PrototypeBean> factory) {
        this.factory = factory;
        System.out.println("Singleton created");
    }

    public PrototypeBean getFactory() {
```

```
        System.out.println("Calling getPrototypeBean: " + LocalTime.now());
        return factory.getObject();
    }
}
```

Będzie wymagało to również modyfikacji klasy `SomeApplicationConfiguration`:

```
package pl.zajavka;

import org.springframework.beans.factory.ObjectFactory;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.*;

@Configuration
@ComponentScan(basePackageClasses = Marker.class)
public class SomeApplicationConfiguration {

    @Bean
    public SingletonBean singletonBean(final ObjectFactory<PrototypeBean> factory) {
        return new SingletonBean(factory);
    }

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public PrototypeBean prototypeBean() {
        return new PrototypeBean();
    }
}
```

I ponownie teraz uruchomimy ten program, to na ekranie zostanie wydrukowane przykładowo:

```
Singleton created
Calling getPrototypeBean: 20:50:28.861749800
Prototype created
Calling getPrototypeBean: 20:50:28.870645800
Prototype created
firstPrototype == secondPrototype? false
```

Co pokazuje, że przedstawione podejście zadziałało i udało nam się w ten sposób rozwiązać problem. Także jeżeli od tego momentu ktoś zada Ci pytanie: *Czy jeżeli Bean, który jest Singletonem jest zależny od Beana, który jest Prototype to wszystko zadziała normalnie, czy trzeba na coś uważać?*, to wiesz już jaka jest odpowiedź i znasz jeden ze sposobów na rozwiązanie tego problemu.