Spring - Beans - Intro

Spis treści

Czym jest Spring Bean

Spotkaliśmy się już wcześniej ze stwierdzeniem **Java Bean**. Określenie to narzuca pewnego rodzaju standard klasy Java, w której:

- Wszystkie pola są prywatne, a dostęp do nich jest realizowany przez gettery i settery,
- Nazewnictwo getterów i setterów trzyma się konwencji `getX()` i `setX()`,
- Mamy dostępny publiczny bezargumentowy konstruktor,
- Klasa ta implementuje interfejs `Serializable`.

Nazwa **Bean** została nadana, aby uwypuklić ten standard, którego celem jest tworzenie reużywalnych komponentów (klas). Reużywalnych w tym rozumieniu, że działanie bibliotek możemy wtedy oprzeć o obiekty z naszego projektu pod warunkiem, że spełniają one standardy. Można się też pokusić o stwierdzenie, że Intellij wspiera ten standard, dając nam możliwość generowania getterów, setterów i konstruktorów.

Czym jest w takim razie Spring Bean?

Cytując dokumentację Spring:

> In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

Czyli Spring Beany są takimi obiektami, które tworzą szkielet aplikacji i które są zarządzane przez kontener Spring IoC (do tego przejdziemy). Bean to obiekt, który jest tworzony, składany i zarządzany przez kontener Spring IoC. W przeciwnym razie Bean jest po prostu jednym z wielu obiektów w Twojej aplikacji. Beany i zależności między nimi znajdują odzwierciedlenie w metadanych konfiguracyjnych używanych przez Spring IoC container.

Inversion of Control i Dependency Injection

Jeżeli nie pamiętasz o co chodziło w **IoC**, **DIP** i **DI** to zalecam przypomnienie sobie tych zagadnień.

Po wstępie już wiesz, czym jest Framework i dlaczego Spring w świecie Javy jest ważny. Zapoznając się z kolejnymi zagadnieniami zrozumiesz też jak duży jest to projekt. Mam też nadzieję, że korzyści i możliwości, jakie niesie ze sobą, zachęciły Cię do jego poznania. W szczególności, że te zalety w znacznym stopniu mają ułatwić Ci prace i pozwolić skupić się na jej najprzyjemniejszej części tj. wszystkim innym niż walka z konfiguracją i powtarzalnymi problemami technicznymi.

A zaczniemy od jednego z głównych atutów, jakie daje nam Spring, czyli implementacja **IoC**. Tematykę **IoC** (*Inversion of Control*) oraz **DI** (*Dependency Injection*) poruszaliśmy w warsztacie `Design Principles i Design Patterns`, podczas omawiania zasad **SOLID**.

Dla przypomnienia, **IoC** jest wzorcem projektowym, który zaleca odwrócenie kontroli w celu osiągnięcia luźnego powiązania pomiędzy obiektami. Natomiast **DI** jest jednym z poziomów implementacji **IoC**, który polega na wstrzyknięciu jednego obiektu w drugi, co inaczej nazywane jest *wstrzykiwaniem zależności*.

Warto dodać, że istnieją różne rodzaje wstrzykiwania. My skupimy się na dwóch głównych, które są wspierane przez Spring: **poprzez konstruktor** oraz **poprzez właściwość** (setter).

Zacznijmy jednak od przykładu kontroli nieodwróconej.

Kontrola nieodwrócona

Spójrz na poniższy przykład:

```
class ExampleBean {
    private InjectedBean injectedBean;

    public ExampleBean() {
        this.injectedBean = new InjectedBean();
    }
}

class InjectedBean {
    // ...
}
```

To, co chcemy osiągnąć to pozbycie się tworzenia klasy `InjectedBean` (a dokładnie tego: `new InjectedBean()`) z implementacji klasy `ExampleBean`. Dzięki temu `ExampleBean` straci część kontroli nad `InjectedBean`. Jest to zachowanie pożądane, bo w efekcie obiekty będą ze sobą powiązane, ale **luźno**. Za utworzenie obiektu `InjectedBean` nie będzie już odpowiadać `ExampleBean` - kontrola ta zostanie odwrócona, odpowiedzialność ta jest wypychana "na zewnątrz". W momencie gdy piszemy kod, który jest luźno powiązany, daje nam to szereg możliwości:

- ułatwia testowanie takiego kodu - możemy w testach wstrzykiwać inne zależności, a w kodzie inne,

- wymienialność komponentów - możemy wykorzystać kilka różnych implementacji klasy lub interfejsu `InjectedBean`,
- łatwiejsze zrozumienie kodu - podpunkt ten oznacza, że klasa jest odpowiedzialna tylko za wykorzystanie zależności, a nie ich konstrukcję i konfigurację, przez co jest prostsza w zrozumieniu,
- izolacja - im mniej klasy są powiązane ze sobą, tym mniej modyfikacja jednej z nich wpływa negatywnie na drugą, czyli np. wymusza modyfikację kodu.

Wstrzykiwanie poprzez konstruktor

Wstrzykiwanie poprzez konstruktor jest podejściem zalecanym dla zależności, które są wymagane tzn. takich, bez których obiekt nadrzędny nie jest w stanie poprawnie funkcjonować. Dodatkowymi cechami tego podejścia jest to, że pozwala na utworzenie obiektu niezmiennego *immutable*, pomaga w uniknięciu zależności cyklicznych *circular dependency*. Co ważne jest to podejście rekomendowane przez sam zespół Springa.

Przykład wstrzykiwania poprzez konstruktor:

```
class ExampleBean {
    private InjectedBean injectedBean;

    public ExampleBean(InjectedBean injectedBean) {
        this.injectedBean = injectedBean;
    }
}
```

Wstrzykiwanie poprzez właściwość (setter)

Wstrzykiwanie poprzez właściwość (setter) jest zalecane dla zależności opcjonalnych tzn. takich, bez których obiekt nadrzędny jest w stanie poprawnie funkcjonować. Dodatkową cechą tego podejścia jest możliwość zmiany wstrzykniętej zależności nawet po utworzeniu instancji obiektu.

Przykład wstrzykiwania poprzez właściwość (setter):

```
class ExampleBean {
    private InjectedBean injectedBean;

    public void setInjectedBean(InjectedBean injectedBean) {
        this.injectedBean = injectedBean;
    }
}
```

Istnieje jeszcze coś takiego jak wstrzykiwanie poprzez pole. Polega na użyciu refleksji, przez co nie wymaga dodatkowego kodu konstruktora i setterów, co ładnie i prosto wygląda w kodzie. W Springu wstrzykiwanie przez pole opiera się o adnotacje `@Autowired`. Wystarczy umieścić adnotacje nad polem w klasie i gotowe. Resztę pracy wykona za nas kontener Spring (o którym zaraz), a informacje o zależnościach są dla nas ukryte. Między innymi właśnie dlatego jego stosowanie nie jest zalecane.

```
public class ExampleBean {
```

```
    @Autowired
    private InjectedBean injectedBean;
}
```

Jeżeli zastanawiasz się czym jest **refleksja** - jest to jeden z mechanizmów jaki oferuje Java, o którym lekko wspomnieliśmy na etapie programowania funkcyjnego, ale nie poruszaliśmy zagadnień z tego obszaru. W skrócie w refleksji chodzi o to, że program może sam siebie modyfikować. Podczas działania programu możesz przeprowadzić inspekcję obiektu, na którym operujesz. Możesz pobrać listę pól obiektu, listę metod oraz dużo innych informacji. Następnie możesz takie pola, metody modyfikować w trakcie działania programu.

Spring pozwala na jednoczesne stosowanie wszystkich wymienionych rodzajów wstrzykiwania. Jednak dobrą praktyką jest opracowanie jednej konwencji ich stosowania i trzymanie się jej w całym projekcie.

Czym jest Service

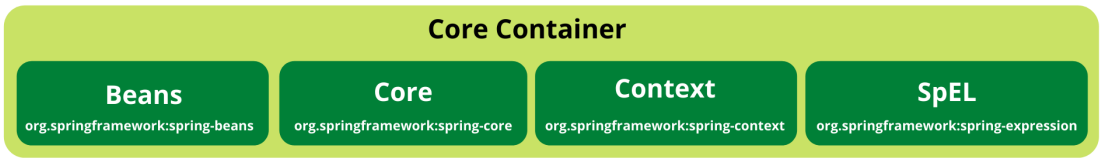
Spotkaliśmy się już wcześniej z klasami, których nazwa kończyła się słówkiem `···Service`. Klasy tego typu są odpowiedzialne za realizację jakiejś funkcji biznesowej, np. za obliczenie części składowej jakiegoś wyniku końcowego itp. Klasy takie służą do wykonywania operacji, a nie do przetrzymywania danych - dlatego mówi się na nie *usługi*. Każdy serwis powinien odpowiadać za specyficzną dla niego funkcjonalność.

Podział funkcjonalności pomiędzy oddzielne serwisy ułatwia utrzymanie porządku w projekcie. Pozwala na logiczne pogrupowanie serwisów, przez co upraszcza prace z nimi i ich późniejsze rozszerzanie. W dużych projektach jest to szczególnie przydatne, bo np. minimalizuje ilość zduplikowanego kodu. Poprzez mechanizm wstrzykiwania różne serwisy udostępniają innym serwisom swoje usługi. W praktyce przypomina to klocki lego, gdzie każdy klocek odpowiada za mniejszą funkcjonalność i może być użyty w budowaniu większej logiki biznesowej. Z tą ważną różnicą, że taki klocek może być użyty wielokrotnie w różnych budowlach.

Spring IoC Container

O kontenerze **IoC** też już w pewnym sensie słyszałeś/słyszałaś na poprzednich warsztatach. Kontener **IoC** jest rodzajem frameworka, który zajmuje się automatyzacją **DI**. Tym razem zajmiemy się nim bardziej szczegółowo. W przypadku Springa jest to jego kluczowy element. To on odpowiada za wstrzykiwanie obiektów w aplikacji i robi to genialnie, bo co najważniejsze odciąża nas z obowiązku zarządzania tym ręcznie. My tylko definiujemy co i gdzie, a resztą zajmuje się **Spring IoC Container**.

Patrząc na ogólny zarys modułów Spring Framework, można zauważyć wydzieloną grupę Core Container składającą się z czterech modułów.

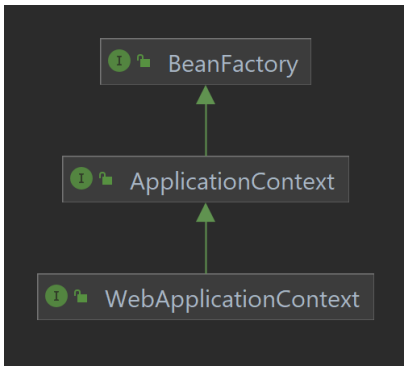


Obraz 1. Moduły Core Container

> Warto wiedzieć, że na kontener IoC składają się jedynie moduły: *Beans*, *Core* i *Context*.

Rodzaje dostępnych kontenerów

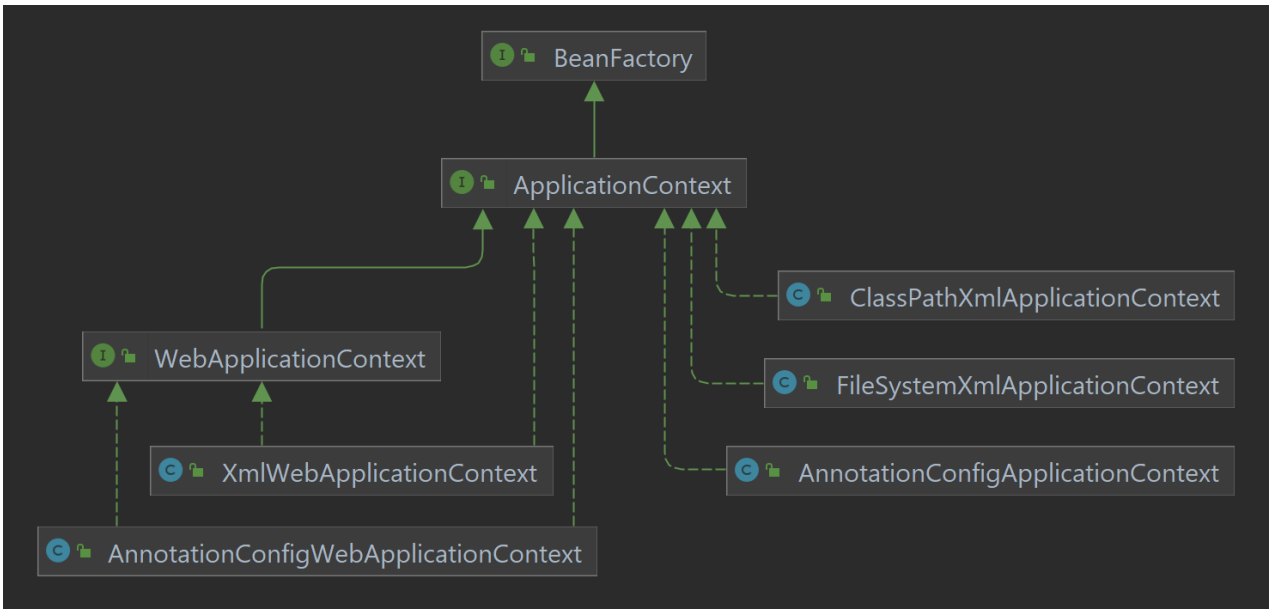
Jak to ze Springiem bywa, mamy opcje do wyboru. Framework dostarcza kilka kontenerów, które można wybrać w zależności od potrzeb projektu. Interfejsy reprezentujące te kontenery to:



Obraz 2. Hierarchia zależności interfejsów Spring IoC Container

- **BeanFactory** jest to podstawowy interfejs kontenera **IoC**, zalecany do użytku jedynie w przypadku potrzeby wyjątkowej oszczędności pamięci,
- **ApplicationContext** jest to rozszerzenie `BeanFactory`, ulepszone o funkcje przydane aplikacjom enterprise – `rekomendowany do używania`,
- **WebApplicationContext** jest to rozszerzenie `ApplicationContext` dedykowane dla aplikacji webowych.

Na swojej drodze możesz natknąć się na nazwę **Spring Context**. Jest to po prostu inne określenie na Spring IoC Container. Żeby zrozumieć skąd ono pochodzi spojrzymy na dostępne implementacje kontenera.

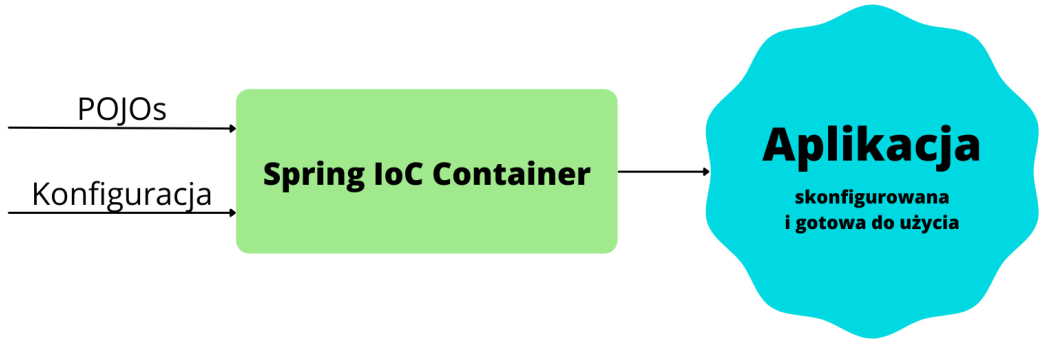


Obraz 3. Implementacje ApplicationContext

Jako że `ApplicationContext` "umie" wszystko to, co "umie" `BeanFactory`, a dodatkowo daje wiele więcej, to `BeanFactory` w większości przypadków nie jest stosowany. Za to są stosowane implementacje

`ApplicationContext`, które dziedziczą też nazwę *...ApplicationContext*. To którą implementację użyć, zależy już od nas, ale o tym później.

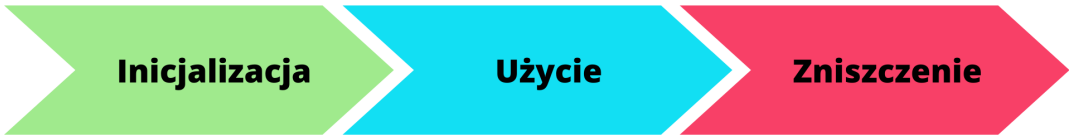
No dobra, ale jak to działa? Powołując się na oficjalną dokumentację, w dużym uproszczeniu działanie Spring IoC Container można przedstawić tak jak poniżej:



Obraz 4. Działanie Spring IoC Container

POJO czyli **P**lain **O**ld **J**ava **O**bject. Oznacza klasę, która w żaden sposób nie jest zależna od zewnętrznych bibliotek. Nie oznacza to stosowania żadnej konwencji, tzn, posiadania getterów, setterów itp. Ważne jest to, że klasa taka może być używana bez zależności do jakichkolwiek bibliotek czy frameworków.

Spring IoC Container na podstawie zdefiniowanej konfiguracji składa obiekty biznesowe w działającą aplikację. Konfiguracja zawiera informacje o zależnościach pomiędzy obiektami biznesowymi, dzięki którym kontener jest w stanie stworzyć instancje i odpowiednio skonfigurować obiekty. Przypomnijmy, że obiekt zarządzany przez kontener nazywa się **Spring Bean**. Ma on swój cykl życia, za który odpowiada kontener. Cykl życia takiego obiektu można w uproszczeniu przedstawić w postaci trzech faz: *inicjalizacji*, *użycia* i *zniszczenia*.



Obraz 5. Cykl życia Spring Bean

Najciekawsza dla nas jest faza pierwsza, inicjalizacja. To w tym momencie zachodzi proces odczytywania konfiguracji (o której zaraz), inicjalizowania obiektów i wstrzykiwania zależności (o tym już było, ale jeszcze do tego wrócimy). Większość swojego życia Beany spędzają z fazie drugiej (użycie), podczas której są wykorzystywane w aplikacji. Natomiast podczas fazy zniszczenia Spring uwalnia zajęte zasoby, aż w końcu w ruch wchodzi Garbage Collector.

Nie wszystkie klasy muszą być używane w kontekście Springowym. Nie każda klasa wymaga zarządzania przez Spring Container, bo np. klasa nie posiada zależności lub klasa służy do przechowywania danych - nie jest serwisem realizującym logikę biznesową. Przykładowo proste obiekty **POJO** możemy tworzyć w kodzie sami, bez zaangażowania do tego kontekstu.

Gdy zaczniesz już pracować ze Springiem to zwrócisz uwagę, że aplikacje konfiguruje się w taki sposób, żeby Spring container zarządzał serwisami (czyli tymi klasami realizującymi logikę biznesową), natomiast klasami przechowującymi dane najczęściej

> zarządza deweloper.

Podsumowując: Stwierdzenie *Spring IoC Container* używane jest wymiennie z *Spring Context*. Kontekst na podstawie zdefiniowanej konfiguracji, tworzy w projekcie Spring Beany, wstrzykuje zależności i zarządza nimi przez całe ich życie. Rozporządza nimi w aplikacji, kiedy zgłaszają się do niego potrzebujące ich serwisy. Jest z nimi do samego końca, a kiedy sam jest zamykany, sprząta też Beany, którymi zarządzał.