Spring - Beans - Configuration

Spis treści

Dodajemy Spring do projektu

Zacznijmy od tego, jakie zależności trzeba dodać do projektu, żeby móc korzystać ze Springa. Na zależności Springowe składa się wiele bibliotek. W początkowych fazach nauki będziemy dużo z tych bibliotek dodawali ręcznie. Na kolejnych etapach dowiesz się, jak konfigurację Springa można upraszczać.

Gradle

Zacznijmy jednak od poniższej konfiguracji dla Gradle:

Plik build.gradle

```
plugins {
    id 'java'
}

group = 'pl.zajavka'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

ext {
    springVersion = '5.3.23' ①
    lombokVersion = '1.18.24'
    junitVersion = '5.9.0'
}

dependencies {
    implementation "org.springframework:spring-core:$springVersion"
    implementation "org.springframework:spring-beans:$springVersion"
    implementation "org.springframework:spring-context:$springVersion"
    implementation "org.springframework:spring-context-support:$springVersion"
    implementation "org.springframework:spring-expression:$springVersion"

    compileOnly "org.projectlombok:lombok:$lombokVersion"
    annotationProcessor "org.projectlombok:lombok:$lombokVersion"
```

```
    testCompileOnly "org.projectlombok:lombok:$lombokVersion"
    testAnnotationProcessor "org.projectlombok:lombok:$lombokVersion"

    testImplementation "org.junit.jupiter:junit-jupiter-api:$junitVersion"
    testRuntimeOnly "org.junit.jupiter:junit-jupiter-engine:$junitVersion"
}
```

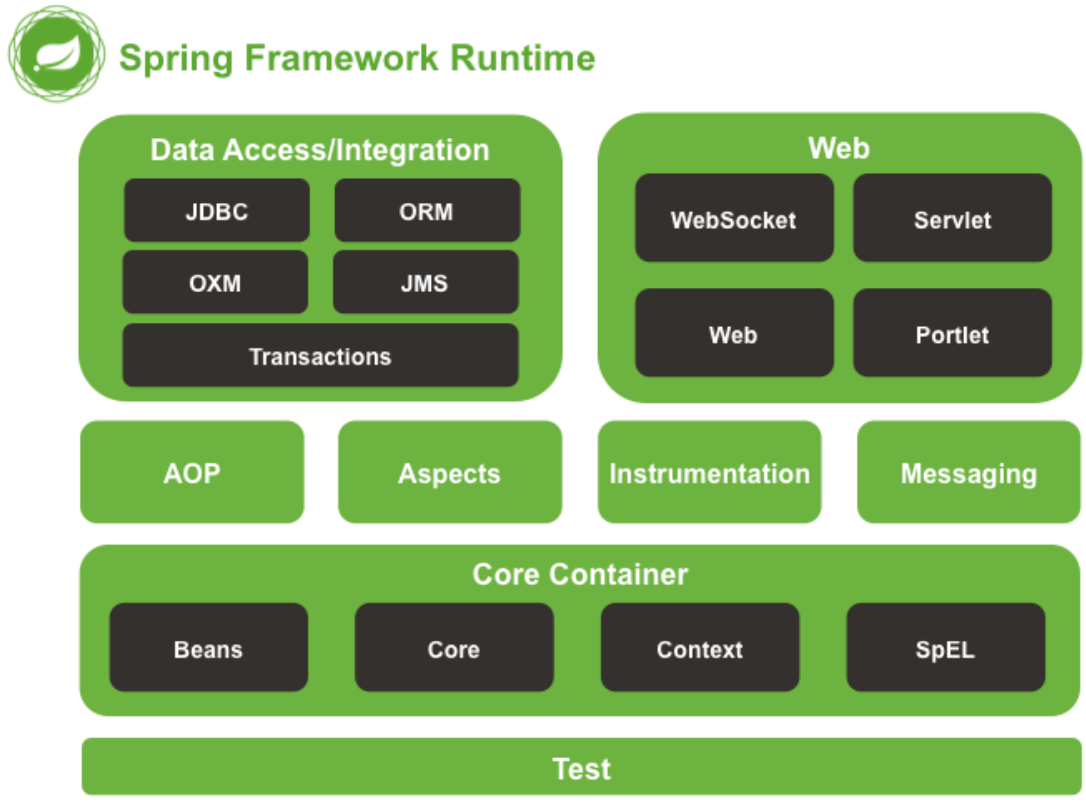
① Oczywiście w momencie, w którym czytasz tę notatkę, dostępne są nowsze wersje tych zależności, ale na tym poziomie zaawansowania nie ma to znaczenia.

cd.

```
repositories {
    mavenCentral()
}

test {
    useJUnitPlatform()
    testLogging {
        events "passed", "skipped", "failed"
    }
}
```

Czemu akurat takie zależności? Przypomnij sobie poniższą grafikę:



Obraz 1. Źródło: Dokumentacja Spring

Następnie odwołajmy się do dokumentacji Springa, znajdziemy tam następujący cytat:

> The Core Container consists of the spring-core, spring-beans, spring-context, spring-context-support, and spring-expression (Spring Expression Language) modules.

I właśnie dlatego zaczynamy od tych zależności 😀.

Maven

Teraz to samo, ale przy wykorzystaniu Maven:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>pl.zajavka</groupId>
  <artifactId>spring-example</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
    <java.version>17</java.version>
    <spring.version>5.3.23</spring.version>
    <lombok.version>1.18.22</lombok.version>
    <junit.version>5.9.0</junit.version>
    <maven-compiler-plugin.version>3.8.0</maven-compiler-plugin.version>
    <maven-surefire-plugin.version>3.0.0-M5</maven-surefire-plugin.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-beans</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context-support</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-expression</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
```

```
      <version>${lombok.version}</version>
      <scope>provided</scope>
    </dependency>

    <!-- test -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${maven-compiler-plugin.version}</version>
            <configuration>
                <release>${java.version}</release>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <version>${lombok.version}</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>${maven-surefire-plugin.version}</version>
        </plugin>
    </plugins>
  </build>
</project>
```

Jak wygląda konfiguracja Spring Beanów?

Rodzaje konfiguracji

Żeby kontener mógł wykonać swoje zadanie, potrzebuje informacji o Spring Beanach i ich konfiguracji. Istnieje kilka sposobów konfiguracji Beanów: w plikach XML, poprzez konfigurację w Javie oraz poprzez adnotacje Springowe. Można je dowolnie mieszać i używać jednocześnie. Pełna dowolność 😊 Dwa pierwsze sposoby to konfiguracja jawna, w której to deweloper określa sposób tworzenia beana. Konfiguracja niejawna przerzuca ten obowiązek na kontener i wiąże obiekty automatycznie. Omówimy je po kolei:

- Konfiguracja jawna:
  - **Pliki XML**
  - **Konfiguracja oparta o Jave**
- Konfiguracja automatyczna (niejawna):
  - **Adnotacje Springowe**

Pliki XML

Sposób konfiguracji stary jak sam Spring i już prawie nieużywany, dlatego ten punkt możesz traktować jak notkę historyczną. Wymaga utworzenia oddzielnego pliku **XML**, w którym definiuje się zarówno Beany, jak i jego zależności. Taki plik (lub pliki, bo może być ich więcej) jest ładowany przez implementację wcześniej wspomnianego interfejsu `ApplicationContext` np. `ClassPathXmlApplicationContext`.

Szukając pracy warto zapytać o rodzaj używanej konfiguracji Springowej w systemie u potencjalnego przyszłego pracodawcy. Jeżeli to pliki **XML**, zastanów się, czy to na pewno praca marzeń, której szukasz.

Przykładowy plik applicationContext.xml z konfiguracją wstrzykiwania poprzez `konstruktor`*:*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
            https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="exampleBean" class="pl.zajavka.ExampleBean">
        <constructor-arg>
            <ref bean="injectedBean"/>
        </constructor-arg>
        <constructor-arg ref="anotherInjectedBean"/>
    </bean>

    <bean id="injectedBean" class="pl.zajavka.InjectedBean"/>
    <bean id="anotherInjectedBean" class="pl.zajavka.AnotherInjectedBean"/>

</beans>
```

Odzwierciedlenie w postaci Javy:

```
public class ExampleBean {

    private InjectedBean injectedBean;
    private AnotherInjectedBean anotherInjectedBean;

    public ExampleBean(InjectedBean injectedBean, AnotherInjectedBean anotherInjectedBean) {
        this.injectedBean = injectedBean;
        this.anotherInjectedBean = anotherInjectedBean;
    }

    public void exampleMethod() {
        // ...
    }
```

```
}
```

Przykładowy plik applicationContext.xml z konfiguracją wstrzykiwania poprzez właściwość (setter):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
            https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="exampleBean" class="pl.zajavka.ExampleBean">
        <property name="injectedBean">
            <ref bean="injectedBean"/>
        </property>
        <property name="anotherInjectedBean" ref="anotherInjectedBean"/>
    </bean>

    <bean id="injectedBean" class="pl.zajavka.InjectedBean"/>
    <bean id="anotherInjectedBean" class="pl.zajavka.AnotherInjectedBean"/>

</beans>
```

Odzwierciedlenie w postaci Javy:

```
public class ExampleBean {

    private InjectedBean injectedBean;
    private AnotherInjectedBean anotherInjectedBean;

    public void setInjectedBean(InjectedBean injectedBean) {
        this.injectedBean = injectedBean;
    }
    public void setAnotherInjectedBean(AnotherInjectedBean anotherInjectedBean) {
        this.anotherInjectedBean = anotherInjectedBean;
    }

    public void exampleMethod() {
        // ...
    }
}
```

Użyte elementy konfiguracji XML:

- <**beans**></**beans**> — sekcja dla definicji Beanów,
- <**bean id**="..." **class**="..."/> — definicja pojedynczego Beana,
- **id** — identyfikator definicji Beana,
- **class** — pakiet i nazwa klasy Beana,
- <**constructor-arg ref**="..."/> — element definiujący konstruktor,
- **ref** — referencja do innego Beana,
- <**property name**="..." **ref**="..."/> — element definiujący właściwość (setter),
- **name** — nazwa własności,

Przykładowy kod użycia kontenera:

```
package pl.zajavka;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import pl.zajavka.ExampleBean;

public class ExampleSpringUsage {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml"); ①
        ExampleBean exampleBean = context.getBean("exampleBean", ExampleBean.class); ②
        exampleBean.exampleMethod(); ③
    }
}
```

① Tworzenie i konfigurowanie Beanów.

② Wybranie Beana.

③ Użycie Beana.

W powyższym przykładzie mamy do czynienia z `ClassPathXmlApplicationContext`, czyli implementacją `ApplicationContext` pobierającą plik definicji kontekstu z pliku **XML** podanego w konstruktorze.

Powyższe wywołanie metody `main()` tworzy kontekst na podstawie dostarczonego pliku *applicationContext.xml*. Następnie wyciąga z kontekstu serwis `ExampleBean` i wywołuje zdefiniowaną w nim metodę `exampleMethod()`.

Konfiguracja oparta o Jave

Ten sposób konfiguracji Beanów wiąże się ze stworzeniem klasy konfiguracyjnej. Klasa taka będzie oznaczona adnotacją `@Configuration`. Dodatkowo trzeba użyć kolejnej adnotacji `@Bean` nad metodą tworzącą obiekt. Adnotacja `@Bean` jest tym samym co element `<bean>` w konfiguracji **XML** z poprzedniej sekcji. W aplikacji możemy mieć wiele takich klas konfiguracyjnych i mogą być one uzupełniane konfiguracją w plikach XML.

Chcę tutaj jednak wyraźnie zaznaczyć, że w praktyce raczej nie stosuje się konfiguracji przez XML. Raczej używa się konfiguracji opartej o Javę w zestawieniu z adnotacjami.

Rodzaje adnotacji:

- `@Configuration` — Znacznik klasy konfiguracyjnej, która zawiera deklaracje Beanów,
- `@Bean` — Znacznik deklaracji beana.

Przykład konfiguracji opartej o Jave:

```
package pl.zajavka;

public class ExampleBean {
    private InjectedBean injectedBean;

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

```
    }

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

```
package pl.zajavka;

public class InjectedBean {
    public void anotherExampleMethod() {
        // ...
    }
}
```

```
package pl.zajavka;

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

@Configuration
public class ExampleConfigurationClass {

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

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

}
```

Przykładowy kod użycia kontenera:

```
package pl.zajavka;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import pl.zajavka.ExampleBean;

public class ExampleSpringUsage {

    public static void main(String[] args) {
        ApplicationContext context
            = new AnnotationConfigApplicationContext(ExampleConfigurationClass.class); ①
        ExampleBean exampleBean = context.getBean(ExampleBean.class); ②
        exampleBean.exampleMethod(); ③
    }
}
```

① Tworzenie i konfigurowanie Beanów.

② Wybranie Beana.

③ Użycie Beana.

W powyższym przykładzie mamy do czynienia z `AnnotationConfigApplicationContext`, czyli implementacją `ApplicationContext` przyjmującą klasy oznaczone adnotacją `@Configuration`, albo `@Component` (o tym zaraz).

Wywołanie metody `main()` tworzy kontekst na podstawie dostarczonej klasy `ExampleConfigurationClass`. Następnie wyciąga z kontekstu serwis `ExampleBean` i wywołuje zdefiniowaną w nim metodę `exampleMethod()`, która to wywołuje metodę `anotherExampleMethod()` ze wstrzykniętego serwisu `InjectedBean`.

Modularyzacja konfiguracji

Powyższy rodzaj konfiguracji pozwala na rozłożenie definicji tworzenia Beanów w różnych klasach. W niektórych przypadkach taka możliwość może być przydatna np. w celu zmniejszeniu złożoności konfiguracji. Każda klasa konfiguracyjna musi posiadać adnotację `@Configuration`, a metody definiujące tworzenie serwisów adnotację `@Bean`. Tak przygotowane klasy można importować do innej klasy konfiguracyjnej za pomocą adnotacji `@Import`. To samo jest możliwe do osiągnięcia w konfiguracji **XML**owej, ale nie zagłębiamy się w ten temat.

Przykład modularyzacji konfiguracji.

Serwis `ExampleBean`*:*

```
package pl.zajavka;

public class ExampleBean {

    public ExampleBean() {
        System.out.println("Initialize class ExampleBean");
    }

    public void exampleMethod() {
        System.out.println("ExampleBean#exampleMethod");
    }
}
```

Serwis `AnotherExampleBean`*:*

```
package pl.zajavka;

public class AnotherExampleBean {

    public AnotherExampleBean() {
        System.out.println("Initialize class AnotherExampleBean");
    }

    public void anotherExampleMethod() {
        System.out.println("AnotherExampleBean#anotherExampleMethod");
    }
}
```

Definicja klas konfiguracyjnych `ConfigurationClassFirst` *i* `ConfigurationClassSecond`*:*

```
package pl.zajavka;

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

@Configuration
public class ConfigurationClassFirst {

    @Bean
    public ExampleBean exampleBean() {
        return new ExampleBean();
    }
}
```

```
package pl.zajavka;

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

@Configuration
public class ConfigurationClassSecond {

    @Bean
    public AnotherExampleBean innyExampleBean() {
        return new AnotherExampleBean();
    }
}
```

W tym przykładzie mamy dwa proste, niezależne od siebie, obiekty `ExampleBean` i `AnotherExampleBean`. Ich definicje znajdują się w dwóch różnych klasach konfiguracyjnych `ConfigurationClassFirst` i `ConfigurationClassSecond`. Używając adnotacji `@Import` w kolejnej klasie konfiguracyjnej `ConfigurationClassMain` mamy dostęp do obu Beanów, pomimo tego, że sama `ConfigurationClassMain` nie posiada żadnej definicji Beanów.

Osadzenie klas konfiguracyjnych `ConfigurationClassFirst` *i* `ConfigurationClassSecond` *za pomocą adnotacji* `@Import`*:*

```
package pl.zajavka;

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

@Configuration
@Import({ ConfigurationClassFirst.class, ConfigurationClassSecond.class })
public class ConfigurationClassMain {
}
```

Przykład użycia serwisów skonfigurowanych za pomocą adnotacji `@Import`*:*

```
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(ConfigurationClassMain.class);

        ExampleBean exampleBean = context.getBean(ExampleBean.class);
        AnotherExampleBean innyExampleBean = context.getBean(AnotherExampleBean.class);

        exampleBean.exampleMethod();
        innyExampleBean.anotherExampleMethod();
    }
}
```

Wywołanie metody `main()` tworzy kontekst na podstawie dostarczonej klasy `ConfigurationClassMain`. Następnie wyciąga z kontekstu oba zdefiniowane serwisy i wywołuje metody w nich zaimplementowane.

Widok konsoli po wykonaniu metody main():

```
Initialize class ExampleBean
Initialize class AnotherExampleBean
ExampleBean#exampleMethod
AnotherExampleBean#anotherExampleMethod
```

Jeżeli pracujemy z wieloma klasami `@Configuration`, a metody `@Bean` się powtarzają, to każda kolejna metoda nadpisuje poprzednią.

Adnotacje Springowe

Poprzez użycie adnotacji Springowych można tak skonfigurować Springa, żeby wyszukał definicje Beanów. Wtedy Spring sam zadba o to, żeby Beany znalazły się w kontenerze. Utworzy wtedy również zależności między tymi Beanami. Jest to konfiguracja niejawna, ponieważ informacje o wiązaniach znajdują się bezpośrednio w klasach, a nie w jakimś jednym centralnym miejscu z konfiguracją. Spring dostarcza własne adnotacje do takiej konfiguracji nazywane **stereotypami** (*Spring stereotypes*). Jest ich kilka i niosą ze sobą dodatkowe informacje na temat oznaczonych nimi Beanów. Podstawowym jest stereotyp `@Component`, a kolejne wywodzą się od niego.

Rodzaje stereotypów tworzących Beany:

- **@Component** — podstawowe oznaczenie Beana zarządzanego przez Springa, nie ma tutaj żadnego dodatkowego znaczenia,
- **@Service** — dla Beanów zawierających logikę biznesową. Przypomnij sobie teraz o klasach, które nazywaliśmy serwisami. Właśnie takie klasy oznacza się przy wykorzystaniu tej adnotacji,
- **@Repository** — dla Beanów z dostępem do danych. Przykładowo będą to Beany, które będą nam umożliwiały realizację zapytań do baz danych - jeszcze do tego przejdziemy,
- **@Controller** — dla Beanów obsługujących zapytania do API. Ta adnotacja będzie nam bardzo przydatna, gdy będziemy umożliwiali komunikację sieciową z naszą aplikacją. Przejdziemy jeszcze do tego zagadnienia.

Wszystkie powyższe adnotacje są umieszczone w pakiecie `org.springframework.stereotype`.

Ciekawostka dotycząca @Configuration

`@Configuration` - ta adnotacja nie pochodzi z pakietu stereotypów, ale wykorzystuje pod spodem stereotyp `@Component`

Kod źródłowy adnotacji @Configuration

```
package org.springframework.context.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.annotation.AliasFor;
import org.springframework.stereotype.Component;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {

    @AliasFor(annotation = Component.class)
    String value() default "";

    boolean proxyBeanMethods() default true;

}
```

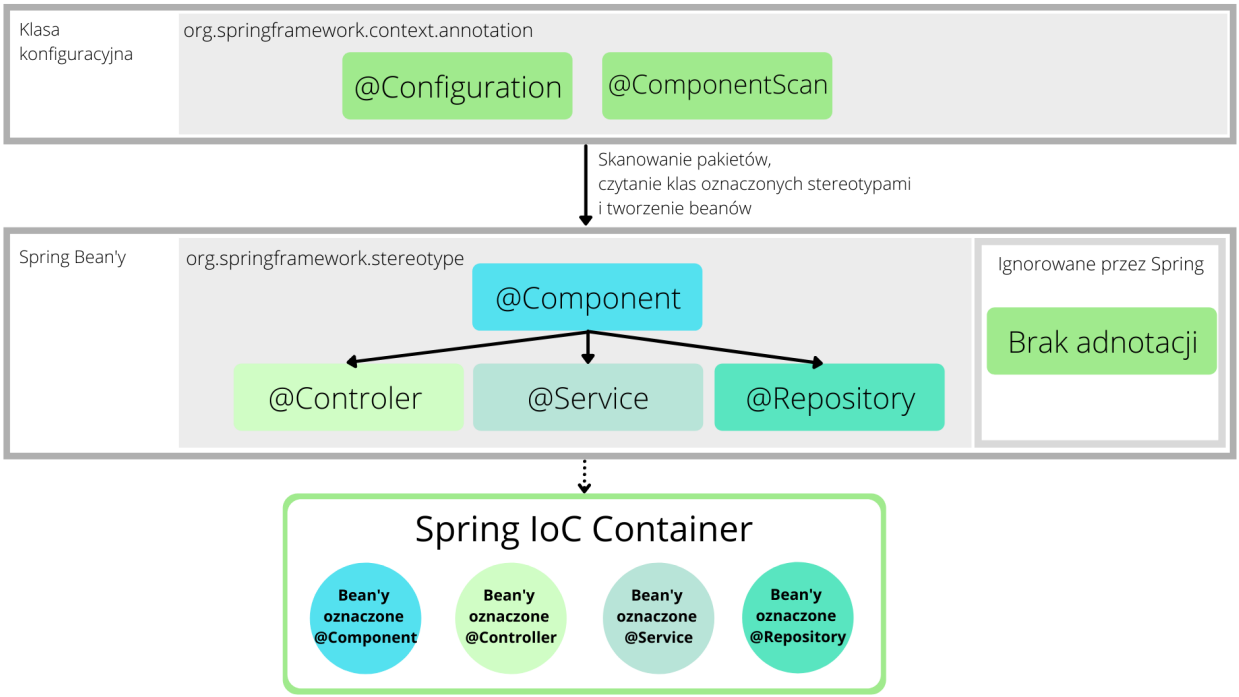
Skoro wiemy już jak skonfigurować aplikację Spring przy wykorzystaniu adnotacji, należy również dodać, skąd Spring wie gdzie ma szukać takich klas, które są oznaczone jako `@Component`, `@Service` lub `@Repository`. Gdyby się nad tym zastanowić, to przecież Spring nie wie gdzie takie klasy mogą być. Do poprawnej konfiguracji przy wykorzystaniu adnotacji Spring potrzebne zatem są:

- Skanowanie komponentów - dzięki temu Spring wie gdzie ma szukać konfiguracji Spring Beanów określonych przez adnotacje,
- Automatyczne wiązanie Beanów - dzięki temu Spring wie jak ma ze sobą wiązać Beany określone przez adnotacje.

Omawiamy je poniżej.

Skanowanie komponentów

Spring wyszukuje wszystkie Beany oznaczone stereotypem `@Component` (bądź stereotypami pokrewnymi), ale żeby to się zadziało, trzeba to Springowi zlecić - trzeba mu powiedzieć gdzie w ogóle ma takich klas szukać. Można to zrobić poprzez kombinacje adnotacji `@Configuration` i `@ComponentScan` nad klasą konfiguracyjną.



Obraz 2. Automatyczne wiązanie obiektów poprzez skanowanie

Wyszukanie można dodatkowo skierować na konkretne paczki, w naszym przypadku jest to `pl.zajavka`.

Przykładowa klasa, skanująca Beany w pakiecie `pl.zajavka`

```
package pl.zajavka;

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

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

Adnotacja `@ComponentScan` daje nam możliwość określenia, gdzie Spring ma szukać klas, z których ma następnie utworzyć Spring Beany. Teoretycznie oznacza to, że nie cała nasza aplikacja musi być oparta o Spring context - możemy skanować tylko część aplikacji w poszukiwaniu tego rodzaju klas. Jest to tylko przypadek teoretyczny, bo w praktyce najczęściej skanuje się całą aplikację w celu poszukiwania Spring Beanów.

Adnotacja `@ComponentScan` przyjmuje również wiele argumentów:

- **basePackages** — lista nazw pakietów do przeskanowania,
- **basePackagesClasses** — lista nazw klas, których pakiety są do przeskanowania - w ten sposób oznaczamy pakiet poprzez klasę, która się w takim pakiecie znajduje,
- **includeFilters** — dodatkowe filtry do skanowania obiektów,
- **excludeFilter** — dodatkowe filtry do pominięcia obiektów podczas skanowania,
- **useDefaultFilters** — boolean determinujący czy skanowanie ma wykrywać adnotacje `@Component`, `@Repository`, `@Service` albo `@Controller` (domyślnie: true).

Ten sam efekt można uzyskać poprzez konfigurację w pliku XML.

Przykładowy plik applicationContext.xml z konfiguracją skanującą Beany w pakiecie `pl.zajavka`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="pl.zajavka"/>

</beans>
```

Jak przygotować Beany, żeby Spring je znalazł podczas skanowania?

Żeby mechanizm skanowania odnalazł Beany, którymi kontener ma zarządzać, wystarczy użyć jednego ze stereotypów. W poniższym przykładzie mamy dwa `@Component` dla klasy `ExampleBean` i `@Service` dla klasy `InjectedBean`. Spring będzie skanował wszystkie klasy w pakiecie zdefiniowanym w klasie `ExampleBeansScanningConfiguration` w poszukiwaniu stereotypów, a resztę klas ominie.

Przykład konfiguracji opartej o stereotypy:

```
package pl.zajavka;

import org.springframework.stereotype.Component;

@Component
public class ExampleBean {
    public void exampleMethod() {
        // ...
    }
}
```

```
package pl.zajavka;

import org.springframework.stereotype.Service;

@Service
public class InjectedBean {
    public void anotherExampleMethod() {
        // ...
    }
}
```

Przykładowy kod użycia kontenera:

```
package pl.zajavka;

import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import pl.zajavka.ExampleBean;

public class ExampleSpringUsage {

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

        ExampleBean exampleBean = context.getBean(ExampleBean.class);
        InjectedBean injectedBean = context.getBean(InjectedBean.class);

        exampleBean.exampleMethod();
        injectedBean.anotherExampleMethod();
    }
}
```

Wywołanie metody `main()` tworzy kontekst na podstawie dostarczonej klasy `ExampleBeansScanningConfiguration`. Następnie wyciąga z kontekstu oba zdefiniowane serwisy i wywołuje metody w nich zaimplementowane.

Automatycznie wiązanie Beanów

Powyżej mamy już przygotowaną konfigurację odpowiedzialną za wyszukiwanie Beanów. Mamy też zdefiniowane Beany dzięki użyciu adnotacji `@Component` i `@Service`. Podczas skanowania Spring je wykryje i umieści w swoim kontekście.

Musimy teraz zatroszczyć się o to, żeby Spring wiedział jak ma powiązać ze sobą Beany zdefiniowane przez adnotacje. Do takiego automatycznego wiązania stosuje się adnotację `@Autowired`. Może być ona umieszczona nad konstruktorem, nad setterem, bądź nad polem. W taki sposób "mówimy" Springowi, żeby sam odnalazł potrzebny Bean w Spring kontekście. Takie podejście zmniejsza ilość kodu, za to rozprasza konfigurację po całej aplikacji.

Adnotacja nosi nazwę `@Autowired`, bo **Autowire** to inaczej automatyczne wstrzykiwanie zależności.

Wcześniej wspomnieliśmy, o adnotacji `@Autowired` w kontekście stosowania jej do wstrzykiwania przez pole. Adnotacja ta może jest również stosowana przy wstrzykiwaniu przez konstruktor lub przez setter, ale mogą być od tego wyjątki. Zostanie to wyjaśnione poniżej.

Konfiguracja `@Autowired` poprzez konstruktor

```
package pl.zajavka;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class ExampleBean {
    private InjectedBean injectedBean;

    public ExampleBean(){
    }

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

Wstrzyknięcie obiektu `InjectedBean` nastąpi podczas tworzenia obiektu `ExampleBean`. Od wersji Spring **4.3**, jeżeli klasa posiada tylko jeden konstruktor, adnotacja `@Autowired` przy konstruktorze jest opcjonalna - Spring sam się domyśli, o który konstruktor chodzi, bo przecież będzie tylko jeden.

Konfiguracja `@Autowired` poprzez właściwość (setter)

```
package pl.zajavka;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class ExampleBean {
    private InjectedBean injectedBean;

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

Wstrzyknięcie obiektu `InjectedBean` nastąpi podczas wykonywania aplikacji (runtime) na podstawie typu i nazwy obiektu.

Adnotacja `@Autowired` może być użyta na dowolnej metodzie. Nie ma konieczności użycia jej wyłącznie na setterze.

Konfiguracja `@Autowired` poprzez pole

```
package pl.zajavka;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class ExampleBean {

    @Autowired
    private InjectedBean injectedBean;

}
```

Wstrzyknięcie obiektu `InjectedBean` nastąpi podczas wykonywania aplikacji (runtime).

Użycie adnotacji `@ComponentScan` bądź elementu `<context:component-scan/>` włącza automatyczne wykrywanie Beanów oraz wiązanie ich zależności. Element `<context:annotation-config />` włącza jedynie automatyczne wiązanie zależności, ale nadal wymaga definicji Beanów `<bean/>` w pliku XML.

Przykładowy plik applicationContext.xml z konfiguracją włączającą wiązania `@Autowired`

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config />

    <bean id="exampleBean" class="pl.zajavka.ExampleBean"/>
    <bean id="injectedBean" class="pl.zajavka.InjectedBean"/>

</beans>
```

Pełny przykład działania adnotacji `@Autowired` poprzez konstruktor.

Definicja klasy konfiguracyjnej skanującej pakiet `pl.zajavka` w poszukiwaniu Beanów:

```
package pl.zajavka;

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

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

Definicje Beanów `ExampleBean` *i* `InjectedBean` *ze wstrzykiwaniem poprzez konstruktor:*

```
package pl.zajavka;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class ExampleBean {
    private InjectedBean injectedBean;

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

    public void exampleMethod() {
        injectedBean.anotherExampleMethod(); ①
    }
}
```

① Użycie wstrzykniętego Beana

```
package pl.zajavka;

import org.springframework.stereotype.Service;

@Service
public class InjectedBean {
    public void anotherExampleMethod() {
        // ...
    }
}
```

Przykładowe użycie wcześniej zdefiniowanych Beanów:

```
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);
        ExampleBean exampleBean = context.getBean(ExampleBean.class);
        exampleBean.exampleMethod();
    }
}
```

Wywołanie metody `main()` tworzy kontekst na podstawie dostarczonej klasy `ExampleBeansScanningConfiguration`. Następnie wyciąga z kontekstu serwis `ExampleBean` i wywołuje zdefiniowaną w nim metodę `exampleMethod()`, która to wywołuje metodę `anotherExampleMethod()` ze wstrzykniętego serwisu `InjectedBean`.