Moduł 2: Kontener Spring

Hello Spring!

Sercem Spring Framework jest kontekst aplikacyjny, który umożliwia m.in. wstrzykiwanie zależności, programowanie z wykorzystaniem aspektów, wsparcie dla transakcji, pamięci podręcznej (cache), walidacji, nieskomplikowany framework webowy spring-mvc oraz szereg narzędzi do testowania aplikacji. Sam kontekst to podstawa dla szeregu bibliotek dostarczanych zarówno przez Spring Source (Spring Security, Spring Integration, Spring Web Services itd.) jak i przez niezależnych dostawców (Apache CXF).

Kontener to nietrywialna fabryka komponentów, które są zdeklarowane i wykorzystywane w tworzonej aplikacji. Komponenty mogą od siebie zależeć, mogą wykorzystywać różne zasoby systemowe (system plików, pliki konfiguracyjne, bazy danych), mogą być powoływane do życia i niszczone zależnie od potrzeb - wszystkim tym zarządza Spring.

Utworzenie kontekstu

Kontekst powołujemy do życia poprzez utworzenie pliku XML zawierającego następujące elementy:

```
<?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-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

       <context:component-scan base-package="pl.devcastzone.spring.*" />

</beans>
```

Jeżeli utworzyliśmy projekt wykorzystując jeden z wzorców oferowanych przez środowisko STS, to taki plik został już utworzony w katalogu `src/main/resources/META-INF/spring/app-context.xml`

Najistotniejsza linijka w tym pliku: `<context:component-scan base-package="pl.devcastzone.spring.*" />` informuje kontener, że wszystkie komponenty są zadeklarowane w pakiecie pl.devcastzone.spring i klasy tam znajdujące się należy przeskanować w poszukiwaniu odpowiednich adnotacji.

Jest to w zasadzie wystarczająca konfiguracja, aby rozpocząć pracę ze Springiem. Nie jest to jednak jedyna możliwość konfigurowania aplikacji. Poza skanowaniem pakietów (czyli konfiguracją pokazaną na przykładzie powyżej) można także ręcznie zadeklarować wszystkie komponenty w pliku XML lub w osobnej klasie konfiguracyjnej JavaConfig. Każda z tych metod zostanie omówiona w dalszej części modułu - tam też zostaną opisane wady i zalety każdego z rozwiązań. Na początek, aby skupić się na samym procesie tworzenia komponentów, pozostaniemy przy automatycznym ich wyszukiwaniu.

Aby uruchomić taką najprostszą aplikację Spring, należy utworzyć nową klasę w utworzony zostanie kontekst aplikacji:

```
package  pl.devcastzone.spring;

import  org.springframework.context.support.AbstractApplicationContext;
import  org.springframework.context.support.ClassPathXmlApplicationContext;

import  pl.devcastzone.spring.example001.ExampleService;

public  class App {

	public static void main(String[] args) {

		AbstractApplicationContext context =
		new ClassPathXmlApplicationContext("META-INF/spring/app-context.xml");

	}

}
```

Oczywiście nic się nie wydarzyło, ponieważ poza samym kontenerem, nie zostały zdefiniowane żadne komponenty, utworzona aplikacja nie realizuje jeszcze żadnych usług.

Komponenty

Aby poprawnie zdefiniować komponent wystarczy oznaczyć klasę komponentu adnotacją `@Component`. Spring, podczas skanowania pakietów, automatycznie rozpozna klasę i dołączy jej instancję do kontekstu. W zasadzie na tym można by poprzestać, ponieważ tą jedną, ogólną adnotacją można poprawnie zbudować całą aplikację. Jednakże Spring oferuje kolejne trzy, bardziej specyficzne adnotacje: `@Resource`, `@Service` i `@Controller`. Zależnie od kontekstu użycia, można oznaczyć klasy innymi adnotacjami, na przykład w odniesieniu do bazy danych (`@Resource`), serwisów (`@Service`) lub kontrolerów webowych (`@Controller`). Zasada działania wszystkich adnotacji jest taka sama, jednakże odpowiednie ich użycie sugeruje pewien dodatkowy kontekst oraz umożliwia elastyczniejszą i łatwiejszą konfigurację np. aspektów. Warto zauważyć że nie ma konieczności stosowania dodatkowych interfejsów (jak to ma miejsce w przypadku wielu innych frameworków). Wszystkie dodatkowe usługi kontenera (transakcje, logowanie, aspekty) mogą z powodzeniem zostać dodane do zwykłych klas. Dlatego komponent w najprostszej postaci może wyglądać następująco:

```
@Component
public  class ExampleService {

	public String getMessage() {
		return "Hello world!";
	}

}
```

Tak utworzone komponenty możemy ze sobą łączyć (wstrzykiwać) z wykorzystaniem adnotacji `@Autowired` (org.springframework.beans.factory.annotation.Autowired). Niezależnie czy adnotacją oznaczymy pole klasy, modyfikator (getter / setter), konstruktor - Spring będzie w stanie poprawnie powiązać ze sobą komponenty. Dla porządku i dla zachowania pewnej przejrzystości i konwencji - w niniejszym szkoleniu komponenty będziemy łączyć poprzez adnotację na atrybutach klasy. Wybór jednej z metod nie ma wpływu na sposób działania aplikacji, jednakże ze względu na czystość i jakość kodu dobrze jest trzymać się przyjętej i jednolitej konwencji.

Począwszy od wersji 3.0 Spring, pojawiły się dodatkowe możliwość konfiguracyjne - pojawiła się także możliwość łączenia komponentów (wstrzykiwania) poprzez adnotację `@Inject` oraz deklarowania komponentów poprzez adnotację `@Named`. Te nowości wynikają m.in. z faktu, że po wielu latach kwestie wstrzykiwania zależności (*Inversion of control*) doczekały się własnej specyfikacji (JSR 330 - Dependency Injection for Java) której głównym autorem był Rod Johnson - autor Springa. Nie jest zatem zaskakujące, że Spring Framework w całości implementuje tę specyfikację, dodając do istniejących już adnotacji, kilka nowych wynikających ze standardu.

Komponent utworzony zgodnie z JSR 330 będzie wyglądał następująco:

```
package  pl.devcastzone.spring.example001;

import  javax.inject.Named;


@Named
public  class ExampleService {

	public String getMessage() {
		return "Hello world!";
	}

}
```

Jego wstrzyknięcie odbędzie się natomiast poprzez adnotację javax.inject.Inject:

```
	@Inject
	ExampleService exampleService;
```

Standardowe adnotację (`@Named` i `@Inject`) są zdecydowanie uboższe od swoich odpowiedników dostępnych w Springu. Z drugiej jednak strony, gwarantują przenaszalność kodu pomiędzy różnymi kontenerami wstrzykiwania zależności, takimi jak Spring, Google Guice lub CDI (Context and Dependency Injection - będącego elementem platformy JEE6). Wsparcie dla podstawowych adnotacji umożliwia łatwe tworzenie modułów, które mogą być używane poza aplikacją napisaną w Springu – na przykład narzędzia, które będę współdzielone pomiędzy kilka projektów, bez konieczności korzystania z frameworku Spring.

Komponenty bezstanowe

Koncepcja komponentów nie jest nowa, nie pojawiała się wraz ze Springiem – jest dobrze znana chociażby ze specyfikacji EJB. Jednakże w Springu temat podjęto w nieco innym ujęciu. W przywołanym EJB wyróżniamy 2 zasadnicze typy komponentów: stanowe i bezstanowe, które tworzone są i niszczone zależnie od potrzeb (np. od obciążenia systemu). W Springu natomiast wszystkie komponenty nie podlegają zarządzaniu w oparciu o pulę – zostało to rozwiązanie w zupełnie inny z wykorzystaniem zakresów (ang. *bean scope*). Każda klasa jest pewnym 'przepisem na komponent', jest jego wzorcem, jednak dopiero konkretna konfiguracja definiuje sposób tworzenia instancji komponentu. W przypadku ogólnym, w aplikacji istnieje tylko jednak instancja każdego komponentu (taka jest domyślna konfiguracja zalecana przez Spring). Niemniej jednak, każdy utworzony komponent może występować w jednym z kilku dostępnych zakresów (lub też w dowolnym innym utworzonym przez programistę). Na etapie budowy komponentów biznesowych aplikacji to najbardziej interesujące są dwa zakresy: singleton oraz prototype.

Jak już pisałem, domyślnym zakresem jest singleton, który powoduje, że tylko jedna instancja klasy istnieje w kontenerze. Instancja ta jest przetrzymywana w podręcznej pamięci i wszystkie odwołania do komponentu są realizowane przez tę jedną instancję. Implikuje jedną, bardzo istotną rzecz z punktu widzenia architektury komponentów: jeżeli istnieje tylko jedna instancja danej klasy, to pod żadnym pozorem komponent taki nie może przechowywać wewnętrznego stanu, komponent musi być bezstanowy. Jeżeli wywołanie którejś z metod zmieni wewnętrzny stan obiektu, ta zmiana będzie widoczna także dla innych wywołań – nie tylko kolejnych wywołań w obrębie relacji pomiędzy dwiema instancjami, ale dla każdego kolejnego requestu w obrębie aplikacji (np. pochodzącego od innego użytkownika). To z kolei może to prowadzić do niedeterministycznego zachowania się komponentu.

Warto nadmienić, że nie jest to singleton w rozumienia wzorców projektowych Gang of Four. Framework nie narzuca specjalnej konstrukcji obiektu, a decyzja o tym czy instancja jest singletonem, czy niepodejmowana jest w momencie tworzenia kontekstu.

Alternatywą wobec pojedynczej instancji klasy w kontekście jest zakres prototype. Oznaczenie komponentu w ten sposób powoduje utworzenie nowego obiektu wraz z każdym wywołaniem; niezależnie czy komponent pobierany jest jawnie z kontekstu, czy wstrzykiwany do innego komponentu, zawsze tworzona jest nowa instancja. W związku z tym, tak zadeklarowane komponenty nadają się do przechowywania stanu (mogę funkcjonować jako komponenty stanowe). W przypadku komponentów niebędących singletonami rola kontenera ogranicza się jedynie do utworzenia obiektu. Nie są natomiast wywoływane którekolwiek metody związane z niszczeniem instancji (a co za tym idzie np. proces zwolnienia zasobów leży po stronie klienta komponentu).

Framework oferuje także cztery dodatkowe zakresy: thread (jak sama nazwa wskazuje ograniczający dostęp do komponentu do jednego wątku) oraz trzy zakresy wykorzystywane w aplikacjach webowych, bezpośrednio związane z sesją HTTP: request, session oraz global session. Dobrą analogią dla zakresu thread jest znany z Javy SE obiekt ThreadLocal, będący swego rodzaju pamięcią podręczną dostępną tylko dla jednego wątku.

Konfiguracja kontekstu

Wróćmy do pliku `app-context.xml`, który został utworzony na samym początku. Aby aplikacja zadziałała, wystarczył prosty wpis nakazujący skanowanie pakietów oraz określenie które ścieżki mają być skanowane. Całość konfiguracji odbyła się w sposób automatyczny. Przeskanowane zostały pakiety aplikacji w poszukiwaniu adnotacji, które są rozpoznawane przez framework: `@Component`, `@Repository`, `@Service` oraz `@Controller`. O ile w podstawowym przykładzie sprawdza się to doskonale, to przy bardziej rozbudowanych projektach lepiej mieć kontrolę nad tym, jakie elementy projektu są skanowane. W praktyce warto ograniczyć zakres skanowania aplikacji do konkretnych pakietów, uzupełniając atrybuty base-package o kolejne pakiety, oddzielone przecinkami.

Dodatkowo istnieje możliwość rozbudowanego filtrowania co zostanie przeskanowane; poprzez użycie tagów include-filter oraz exclude-filter można regulować czy konkretne klasy zostaną dołączone do projektu. Automatyczna konfiguracja może na przykład dotyczyć tylko klas oznaczonych odpowiednimi adnotacjami (np. stereotypami dostarczanymi przez framework jak `@Component`), klas rozszerzających wybrany przez nas typ bazowy lub implementujących interfejs aż po bardzo elastyczne włączenie klas, których nazwa odpowiada zapisanemu w konfiguracji wyrażeniu regularnemu.

Posłużywszy się naszym przykładem, jeżeli konfigurację zapiszemy w następujący sposób, żaden komponent nie zostanie włączony do kontekstu, ponieważ wyłączone zostaną wszystkie klasy których nazwa kończy się na *Service*

```
<context:component-scan base-package="pl.devcastzone.spring.*">
	<context:exclude-filter type="regex" expression=".*Service"/>
</context:component-scan>
```

Przedstawiona konfiguracja kontekstu jest możliwie najprostsza (minimalna) i budowa aplikacji, czyli powiązań pomiędzy komponentami, odbywa się automatycznie. Mają wtedy zastosowanie konwencje programistyczne zalecane przez Spring, a niewyrażona explicite konfiguracja (ang. *convention over configuration*). Takie podejście sprawdza się doskonale na wczesnych etapach budowy aplikacji, kiedy częstotliwość zmian jest bardzo duża i sam projekt mocno ewoluuje. Programowanie z zastosowaniem konwencji powoduje, że wiele rzeczy dzieje się automatycznie (automagicznie) co daje bardzo dużą elastyczność i pozwala skupić uwagę na realizacji wymagań biznesowych, a nie na pisaniu kodu konfiguracyjnego (tzw. *boilerplate code*). Alternatywą wobec konwencji jest konfiguracja deklaratywna (zapisana explicite w plikach XML).

Konfiguracja deklaratywna

Wykorzystywana dotychczas konfiguracja poprzez adnotacje pojawiła się w Springu 2.5 (od momentu, gdy Spring zaczął wykorzystywać Java5). Jednak, historycznie rzecz ujmując, podstawową konfiguracją aplikacji wykorzystujących Spring Framework były pliki XML. To, co dotychczas działo się automatycznie, można z powodzeniem utworzyć w sposób deklaratywny z wykorzystaniem następującego pliku konfiguracyjnego:

```
<?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-3.0.xsd
	http://www.springframework.org/schema/context
	http://www.springframework.org/schema/context/spring-context-3.0.xsd">

	<bean class="pl.devcastzone.spring.example001.ExampleService" />
</beans>
```

Analogicznie do konfiguracji XML, konfiguracja aplikacji może się odbywać poprzez użycie odpowiednio przygotowanej klasy Java. Z jednej strony jest to wciąż konfiguracja deklaratywna, nic nie dzieje się automatycznie - analogicznie do przykładu opartego o XML. Z drugiej strony (co może być zarówno zaleta jak też wada) kompilator już na etapie stworzenia klasy sprawdza poprawność typów. W przypadku plików XML odbywa się to dopiero w momencie uruchamiania konfiguracji - dopiero wtedy sprawdzane są np. literówki. Konfiguracja JavaConfig dla naszego prostego przykładu wyglądałaby następująco:

```
package  pl.devcastzone.spring;

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

import  pl.devcastzone.spring.example001.ExampleService;

@Configuration
public  class AppConfig {

	@Bean
	public ExampleService service() {
		return new ExampleService();
	}

}
```

Jeżeli decydujemy się na korzystanie z konfiguracji JavaConfig, zmienia się wtedy nieznacznie sposób budowania kontekstu aplikacji: zamiast klasy `ClassPathXmlApplicationContext` (jak w poprzednich przykładach) korzystamy z klasy `AnnotationConfigApplicationContext`, gdzie jako parametr konstruktora podajemy klasę konfiguracyjną.

Różnice i podobieństwa poszczególnych konfiguracji zostaną dokładnie omówione w kolejnych modułach; w jakich sytuacjach użycie poszczególnych konfiguracji jest najodpowiedniejsze oraz jakie konsekwencje niesie wybranie każdego z podejść.

Cykl życia komponentów

Niezależnie od sposobu, w jaki będzie konfigurowana aplikacja (automatyczne skanowanie, pliki XML, klasy konfiguracyjne) wszystkie tworzone komponenty podlegają takim samym prawom i zarządzane są przez kontener na analogicznych zasadach. Jak już podkreśliłem podczas omawiania zakresu komponentów, pojedyncze instancje klas (singletony) podlegają pełnemu cyklowi życia, natomiast w przypadku komponentów oznaczonych jako prototype kontener odpowiada tylko i wyłącznie za utworzenie instancji klasy. Jako, że jest to funkcjonalność dostępna we frameworku od samego początku, więc podobnie jak omówiona powyżej konfiguracja ewoluowała z biegiem czasu.

Obecnie, najpopularniejszą metodą połączenia komponentu ze zdarzeniami kontenera jest użycie adnotacji `@PostConstruct` i `@PreDestroy`. Adnotacje pojawiły się wraz z Java 5 i są opisane w JSR 250: Common Annotations for the Java Platform.

```
@Named
public  class ExampleService {

	@PostConstruct
	public void startup() {
		System.out.println("Preparing the message...");
	}

	public String getMessage() {
		return "Hello world!";
	}

	@PreDestroy
	public void shutdown() {
		System.out.println("...finishing.");
	}

}
```

Po wywołaniu takiego komponentu, następujące wpisy pojawią się w konsoli:

```
Preparing the message...
Hello world!
...finishing.
```

Nie jest to oczywiście jedyny sposób, jednakże że względu na brak jawnego połączenia komponentu z frameworkiem, wydaje się być najodpowiedniejszy. Ten sam komponent, z tak samo oznaczonymi metodami, mógłby zostać także użyty w aplikacji opartej o inny framework *Dependency Injection* i (lub) też jako element aplikacji JEE. Idea wywołania metod związanych z cyklem życia pozostaje taka sama.

W starszych wersjach frameworka, analogiczne zachowanie można było osiągnąć poprzez implementację specyficznych dla Spring interfejsów (był to jedyny sposób w czasach sprzed adnotacji, czyli np. Spring 1.0). Implementując interfejsy `InitializingBean` oraz `DisposableBean`, wymuszana jest implementacja metod `afterPropertiesSet()` oraz `destroy()`. Zaprezentowany powyżej komponent przyjmie wówczas następującą postać:

```
@Named
public  class ExampleService implements InitializingBean, DisposableBean {

	public void afterPropertiesSet() throws Exception {
		System.out.println("Preparing the message...");
	}

	public String getMessage() {
		return "Hello world!";
	}

	public void destroy() throws Exception {
		System.out.println("...finishing.");
	}

}
```

Aby uniknąć bezpośredniego powiązania komponentu z frameworkiem (a tutaj ma to miejsce poprzez implementację specyficznych dla Springa interfejsów), metody związane z cyklem życia można zadeklarować w pliku konfiguracyjnym XML. Jeżeli zatem utworzony komponent będzie wyglądał następująco:

```
public  class ExampleService {

	public void afterPropertiesSet() throws Exception {
		System.out.println("Preparing the message...");
	}

	public String getMessage() {
		return "Hello world!";
	}

	public void destroy() throws Exception {
		System.out.println("...finishing.");
	}

}
```

Aby metody związane z cyklem życia zostały wywołane, konfiguracja w pliku XML musi wyglądać następująco:

```
	<bean class="pl.devcastzone.spring.example001.ExampleService"
		init-method="afterPropertiesSet"
		destroy-method="destroy"
	/>
```

Jak widać na powyższym przykładzie (a także na wcześniejszych przykładach konfiguracyjnym), Spring jest frameworkiem niezwykle elastycznym, gdzie każda czynność może być wykonana na co najmniej kilka sposobów. Dla niektórych osób jest to zaleta, podczas gdy inny postrzegają to jako wadę. Niewątpliwie, wymaga to znacznie większej wiedzy i doświadczenia, połączonego często z umiejętnością holistycznego spojrzenia na problem, gdyż nie ma tutaj jedynie słusznej drogi dojścia do rozwiązania problemów projektowych.

Zasoby i wykorzystanie @Resource

Reagowanie na zdarzenia związane z utworzeniem komponentu i (lub) usuwaniem komponentu z kontekstu to najczęstszy przypadek użycia metod związanych z cyklem życia. Niemniej jednak, nie są to jedyne możliwość współpracy pomiędzy komponentem a kontenerem. W przeciwności np. do platformy JEE, Spring oferuje także szereg interfejsów niebędących bezpośrednio powiązanych z cyklem życia komponentu (z jego tworzeniem oraz niszczeniem) ale umożliwiających komponentowy 'odnalezienie się' w kontekście aplikacji. Mam tutaj na myśli *Aware interfaces*, czyli interfejsy łączące komponent z zasobami dostarczanymi przez framework (`ApplicationContext`, `BeanFactory`, `MessageSource`). Począwszy od wersji 2.0, Spring umożliwia także bezpośrednie wstrzykiwanie elementów frameworka, wykorzystując adnotację `@Resource`. Tak więc poniższe dwie klasy są równoważne co do funkcjonalności:

```
@Named
public  class ExampleService {

	@Resource
	ApplicationContext ctx;

	@PostConstruct
	public void setup() {
		System.out.println("Context started: "
				+ new Date(ctx.getStartupDate()));
	}

	public String getMessage() {
		return "Hello world!";
	}

}
```

```
@Named
public  class ExampleService implements ApplicationContextAware {

	ApplicationContext ctx;

	public void setApplicationContext(ApplicationContext applicationContext)
			throws BeansException {
		this.ctx = applicationContext;
	}

	@PostConstruct
	public void setup() {
		System.out.println("Context started: "
				+ new Date(ctx.getStartupDate()));
	}

	public String getMessage() {
		return "Hello world!";
	}

}
```

W kolejnym module...

Przedstawione tutaj przykłady były stosunkowo podstawowe, ale dość dobrze opisują podstawowe elementy, z których składa się aplikacja oparta o Spring Framework oraz w jaki sposób łączyć te elementy ze sobą. W kolejnej części rozwinę temat wstrzykiwania zależności, opisując bardziej skomplikowane zagadnienie przy okazji budowania bardziej skomplikowanej aplikacji niźli proste „Hello World”.