TWÓJ PRZEWODNIK PO SPRINGU!

Apress

Wprowadzenie do Spring Framework dla programistów Java

Felipe Gutierrez

Helion

Tytuł oryginału: Introducing Spring Framework: A Primer

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-0439-0

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/wprsfj.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wprsfj
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 3

Stosowanie różnych konfiguracji

System szkieletowy Spring umożliwia stosowanie różnych technik konfiguracji kontenera. W tym rozdziale znajduje się opis metody opartej na formacie XML, z której skorzystaliśmy w poprzednim rozdziale. Ponadto czytając ten rozdział, poznasz inne techniki obsługi konfiguracji, takie jak adnotacje Spring, klasy konfiguracyjne ziaren Java oraz nowa klasa `GroovyBeanDefinitionReader`.

W poprzednim rozdziale zdefiniowaliśmy aplikację Spring o nazwie *Moje dokumenty*, a także za pomocą pliku konfiguracyjnego XML wstrzyknęliśmy naszą implementację interfejsu `SearchEngine`. W tym rozdziale będziemy używać tej samej konfiguracji XML, dowiesz się również, jak używać pozostałych technik konfiguracji.

Konfiguracja XML programu Moje dokumenty

Na początek przypomnimy sobie aktualnie posiadaną konfigurację XML aplikacji *Moje dokumenty* i dokładnie ją przeanalizujemy (listing 3.1).

Listing 3.1. *Zawartość pliku mydocuments-context.xml*

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

    <bean id="engine" class="com.apress.isf.java.service.MySearchEngine" />

    <bean id="documentType" class="com.apress.isf.java.model.Type">
      <property name="name" value="WEB" />
      <property name="desc" value="Łącze sieciowe" />
      <property name="extension" value=".url" />
    </bean>

</beans>
```

Na listingu 3.1 pokazano treść pliku konfiguracyjnego w formacie XML zawierającego potrzebne kontenerowi Spring informacje na temat klas i ich zależności. W pliku tym informujemy kontener, że nasza implementacja interfejsu `SearchEngine` będzie ziarnem `MySearchEngine` o identyfikatorze `engine`. Innymi słowy, przypisaliśmy identyfikator zdefiniowanemu przez nas ziarnu. Ponadto tworzymy egzemplarz klasy `Type` w postaci ziarna o identyfikatorze `documentType`.

Ale chwileczkę! Ziarno? A co to jest ziarno? W Javie pojęcie ziarna (ang. *bean*) występuje od początku istnienia tego języka programowania i dlatego w systemie Spring przyjęto podobną nazwę. Ziarna Javy mają pewne standardowe cechy, np. znormalizowane nazwy metod (z przedrostkami `set`, `get` i `is`), odpowiednie konstruktory, zachowania itd., dzięki którym mogą współpracować z innymi ziarnami oraz klasami. Później w społeczności programistów Javy pojęcie ziarna przekształciło się w obiekt POJO (ang. *plain old java object*).

Dzięki tym konwencjom system szkieletowy Spring rozpoznaje, tworzy, wstrzykuje, obsługuje, a nawet niszczy wszystkie klasy zadeklarowane na kontenerze.

Ziarna deklaruje się przy użyciu elementu `<bean/>` w pliku XML. Mogą one mieć kilka atrybutów, opisanych w tabeli 3.1.

Tabela 3.1. *Atrybuty elementu bean*

Atrybut	Opis
`id`	Identyfikator ziarna. Każdy identyfikator może być zdefiniowany tylko raz.
`class`	Wskazuje pełną nazwę wraz z nazwą pakietu konkretnej klasy.
`scope`	Informuje kontener Spring o sposobie tworzenia ziarna. Jeśli atrybut ten jest niezdefiniowany, ziarno jest domyślnie egzemplarzem singletonowym. Poza tym można ustawić zakres `prototype` (dla każdego żądania ziarna tworzony jest egzemplarz), `request` (dla każdego żądania sieciowego HTTP tworzony jest pojedynczy egzemplarz) oraz `session` (ziarno jest tworzone i istnieje przez czas trwania sesji HTTP).
`init-method`	Nazwa metody wywoływanej po utworzeniu ziarna. Metoda ta jest przydatna, gdy trzeba ustawić stan po utworzeniu obiektu.
`factory-method`	Nazwa metody służącej do tworzenia ziarna. Innymi słowy, programista powinien dostarczyć metodę tworzącą egzemplarz obiektu. Metoda ta powinna mieć parametry.
`destroy-method`	Nazwa metody wywoływanej po usunięciu ziarna.
`lazy-init`	Parametr ten należy ustawić na wartość `true`, jeśli chcemy, aby kontener tworzył ziarno przy jego wywołaniu lub użyciu przez programistę (poprzez wywołanie metody `getBean`) bądź później w innej klasie wymagającej tego obiektu.

W systemie Spring informacje na temat klas i ich zależności oraz sposobów interakcji między nimi można dodawać na różne sposoby. O tym wszystkim opowiem w tej książce przy okazji omawiania dodawania różnych funkcji do aplikacji *Moje dokumenty*.

Na listingu 3.2 pokazano implementację interfejsu `SearchEngine` o nazwie `MySearchEngine` z poprzedniego rozdziału. Wydaje się, że to dużo kodu, ale zawiera on sporo danych wpisanych na stałe. Jak w takim razie sobie poradzić, gdy trzeba będzie dodać więcej typów lub metod? W takim przypadku konieczne jest zmodyfikowanie i ponowne skompilowanie kodu. To za dużo pracy!

Listing 3.2. *Zawartość pliku MySearchEngine.java*

```
package com.apress.isf.java.service;

import java.util.ArrayList;
import java.util.List;

import com.apress.isf.java.model.Document;
import com.apress.isf.java.model.Type;
import com.apress.isf.java.service.SearchEngine;

public class MySearchEngine implements SearchEngine {

       @Override
       public List<Document> findByType(Type documentType) {
```

```
        List<Document> result = new ArrayList<Document>();
        for(Document document : storage()){
                if(document.getType().getName().equals(documentType.getName()))
                        result.add(document);
        }
        return result;
}

@Override
public List<Document> listAll() {
        return storage();
}

private List<Document> storage(){
         List<Document> result = new ArrayList<Document>();

         Type type = new Type();
         type.setName("PDF");
         type.setDesc("Portable Document Format");
         type.setExtension(".pdf");

         Document document = new Document();
         document.setName("Szablon książki");
         document.setType(type);
         document.setLocation("/Users/felipeg/Documents/Random/Book Template.pdf");

         result.add(document);

         document = new Document();
         document.setName("Przykładowa umowa");
         document.setType(type);
         document.setLocation("/Users/felipeg/Documents/Contracts/Sample Contract.pdf");

         result.add(document);

         type = new Type();
         type.setName("NOTE");
         type.setDesc("Notatki tekstowe");
         type.setExtension(".txt");

         document = new Document();
         document.setName("Clustering with RabbitMQ");
         document.setType(type);
        document.setLocation("/Users/felipeg/Documents/Random/Clustering with RabbitMQ.txt");

         result.add(document);
         type = new Type();
         type.setName("WEB");
         type.setDesc("Łącze sieciowe");
         type.setExtension(".url");

         document = new Document();
         document.setName("Pro Spring Security Book");
         document.setType(type);
         document.setLocation("http://www.apress.com/9781430248187");

         result.add(document);
```

```
                return result;
        }
}
```

Aby wyeliminować opisane niedogodności, zaimplementujemy interfejs `SearchEngine` od nowa. Do tej implementacji wstrzykniemy typy za pomocą metody ustawiającej. Na listingu 3.3 znajduje się nowa klasa, `SearchEngineService`.

Listing 3.3. *Zawartość pliku SearchEngineService.java*

```
package com.apress.isf.spring.service;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import com.apress.isf.java.model.Document;
import com.apress.isf.java.model.Type;
import com.apress.isf.java.service.SearchEngine;
import com.apress.isf.spring.data.DocumentDAO;

public class SearchEngineService implements SearchEngine {

        private DocumentDAO documentDAO;

        public DocumentDAO getDocumentDAO() {
                return documentDAO;
        }

        public void setDocumentDAO(DocumentDAO documentDAO) {
                this.documentDAO = documentDAO;
        }

        public List<Document> findByType(Type documentType) {
                List<Document> result = new ArrayList<Document>();
                for(Document doc : listAll()){
                        if(doc.getType().getName().equals(documentType.getName()))
                                result.add(doc);
                }
                return result;
        }
        public List<Document> listAll() {
                return Arrays.asList(documentDAO.getAll());
        }
}
```

Porównamy kod znajdujący się na listingu 3.2 z kodem z listingu 3.3. Na tym ostatnim brak sztywnych fragmentów kodu i metody `storage`, natomiast dodano nowy atrybut o nazwie `documentDAO`, który będzie wstrzykiwany przez metodę ustawiającą, tylko trzeba poinformować kontener Spring o tym nowym atrybucie i klasie zawierającej dane.

Teraz zdefiniujemy nowe klasy: `DocumentDAO` i `DocumentRepository` (listingi 3.4 i 3.5). Pokazana na listingu 3.4 klasa `DocumentDAO` przechowuje wszystkie informacje na temat dokumentów, które na razie będą zapisywane w pamięci. Natomiast widoczna na listingu 3.5 klasa `DocumentRepository` to jej implementacja.

Listing 3.4. *Zawartość pliku DocumentDAO.java*

```
package com.apress.isf.spring.data;

import com.apress.isf.java.model.Document;

public interface DocumentDAO {
        public Document[] getAll();
}
```

Na listingu 3.5 przedstawiono klasę `DocumentRepository` zawierającą cztery własności typu `Document` z własnymi metodami ustawiającymi i pobierającymi. Tak, będziemy wstrzykiwać cztery dokumenty za pomocą ich metod ustawiających.

Listing 3.5. *Zawartość pliku DocumentRepository.java*

```
package com.apress.isf.spring.data;

import com.apress.isf.java.model.Document;

public class DocumentRepository implements DocumentDAO {

        private Document doc1;
        private Document doc2;
        private Document doc3;
        private Document doc4;

        public Document getDoc1() {
               return doc1;
        }

        public void setDoc1(Document doc1) {
               this.doc1 = doc1;
        }

        public Document getDoc2() {
               return doc2;
        }

        public void setDoc2(Document doc2) {
               this.doc2 = doc2;
        }

        public Document getDoc3() {
               return doc3;
        }

        public void setDoc3(Document doc3) {
               this.doc3 = doc3;
        }

        public Document getDoc4() {
               return doc4;
        }

        public void setDoc4(Document doc4) {
               this.doc4 = doc4;
```

```
        }

        public Document[] getAll() {
               return new Document[] { doc1, doc2, doc3, doc4 };
        }
}
```

To wygląda już trochę lepiej. Oddzieliliśmy metodę wydobywania danych, ale jak to robimy? Przypomnę, że wcześniej do pobierania informacji używaliśmy metody `storage` (listing 3.2). Później postanowiliśmy od nowa zaimplementować interfejs `SearchEngine`. W tej nowej implementacji zmieniliśmy sposób pobierania danych i utworzyliśmy interfejs, który będzie wstrzykiwany niezależnie od swojej implementacji, dzięki czemu klasa stanie się solidniejsza i łatwiejsza w obsłudze. Ale zobaczmy teraz, co trzeba zmienić w pliku XML, aby poinformować kontener Spring o tych wszystkich nowych modyfikacjach. Nowa wersja pliku *mydocuments-context.xml* jest przedstawiona na listingu 3.6. Znajdują się w nim wszystkie informacje dotyczące implementacji klasy `DocumenDAO` (`DocumentRepository`) oraz sposobu jej wstrzykiwania do implementacji `SearchEngine`.

Listing 3.6. *Zawartość pliku mydocuments-context.xml*

```
<?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
http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="engine" class="com.apress.isf.spring.service.ServiceSearchEngine">
    <property name="documentDAO" ref="documentDAO"/>
  </bean>
  <bean id="documentDAO" class="com.apress.isf.spring.data.DocumentRepository">
    <property name="doc1" ref="doc1"/>
    <property name="doc2" ref="doc2"/>
    <property name="doc3" ref="doc3"/>
    <property name="doc4" ref="doc4"/>
  </bean>
  <bean id="doc1" class="com.apress.isf.java.model.Document">
    <property name="name" value="Szablon książki"/>
    <property name="type" ref="pdfType"/>
    <property name="location" value="/Users/felipeg/Documents/Random/Book Template.pdf"/>
  </bean>
  <bean id="doc2" class="com.apress.isf.java.model.Document">
    <property name="name" value="Przykładowa umowa"/>
    <property name="type">
      <bean id="pdfType" class="com.apress.isf.java.model.Type">
        <property name="name" value="PDF" />
        <property name="desc" value="Portable Document Format" />
        <property name="extension" value=".pdf" />
      </bean>
    </property>
    <property name="location" value="/Users/felipeg/Documents/Contracts/Sample Contract.pdf"/>
  </bean>
  <bean id="doc3" class="com.apress.isf.java.model.Document">
    <property name="name" value="Clustering with RabbitMQ"/>
    <property name="type" ref="noteType"/>
    <property name="location" value="/Users/felipeg/Documents/Random/Clustering with
RabbitMQ.txt"/>
  </bean>
  <bean id="doc4" class="com.apress.isf.java.model.Document">
    <property name="name" value="Pro Spring Security Book"/>
    <property name="type" ref="webType"/>
```

```
    <property name="location" value="http://www.apress.com/9781430248187"/>
  </bean>
  <bean id="webType" class="com.apress.isf.java.model.Type">
    <property name="name" value="WEB" />
    <property name="desc" value="Łącze sieciowe" />
    <property name="extension" value=".url" />
  </bean>
  <bean id="pdfType" class="com.apress.isf.java.model.Type">
    <property name="name" value="PDF" />
    <property name="desc" value="Portable Document Format" />
    <property name="extension" value=".pdf" />
  </bean>
  <bean id="noteType" class="com.apress.isf.java.model.Type">
    <property name="name" value="NOTE" />
    <property name="desc" value="Notatki tekstowe" />
    <property name="extension" value=".txt" />
  </bean>
</beans>
```

Analizując kod przedstawiony na listingu 3.6, można zauważyć, że do przypisywania wartości użyto referencji, takich jak atrybut `ref`. Ponadto w kodzie znajduje się nowa deklaracja klasy `ServiceSearchEngine`, ustawiono także własność `documentDao` i odniesiono jego wartość do innego ziarna o identyfikatorze `documentDAO`.

Spójrz też na ziarno o identyfikatorze `doc2`. Osadzamy nowe ziarno jako wartość, co jest zgodne z zasadami konfiguracji Spring.

Jak widać, wszystkie dane dotyczące typów i dokumentów zostały umieszczone w pliku XML. Może istnieje jeszcze lepszy sposób, ale na razie zajmiemy się utworzeniem testu jednostkowego. Na listingu 3.7 pokazano zmodyfikowaną wersję naszego testu.

Listing 3.7. *Zawartość pliku MyDocumentsTest.java*

```
package com.apress.isf.spring.test;

import static org.junit.Assert.*;

import java.util.List;

import org.junit.Before;
import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.apress.isf.java.model.Document;
import com.apress.isf.java.model.Type;
import com.apress.isf.java.service.SearchEngine;

public class MyDocumentsTest {

        private ClassPathXmlApplicationContext context;
        private SearchEngine engine;
        private Type webType;

        @Before
        public void setup(){
               context = new ClassPathXmlApplicationContext("META-INF/spring/mydocuments-context.xml");
                engine = context.getBean(SearchEngine.class);
                webType = context.getBean("webType",Type.class);
        }
```

```
        @Test
        public void testWithSpringFindByType() {
                List<Document> documents = engine.findByType(webType);
                assertNotNull(documents);
                assertTrue(documents.size() == 1);
                assertEquals(webType.getName(),documents.get(0).getType().getName());
                assertEquals(webType.getDesc(),documents.get(0).getType().getDesc());

assertEquals(webType.getExtension(),documents.get(0).getType().getExtension());
        }

        @Test
        public void testWithSpringListAll() {
                List<Document> documents = engine.listAll();
                assertNotNull(documents);
                assertTrue(documents.size() == 4);
        }
}
```

W metodzie `setup()` (metoda ta jest uruchamiana przed wykonaniem każdej metody w klasie) używamy klasy `ClassPathXmlApplicationContext`, która uruchamia kontener Spring przez utworzenie i powiązanie wszystkich egzemplarzy oraz przygotowanie ich do użytku w momencie, gdy będą potrzebne.

Teraz uruchom ten test za pomocą poniższego polecenia:

```
gradle test
```

W katalogu głównym projektu można użyć polecenia:

```
gradle :r03:test
```

■ **Uwaga** Każdy rozdział zawiera kilka plików z testami jednostkowymi, więc poniżej przedstawiam polecenie wykonujące jeden konkretny test:

```
gradle -Dtest.single=MyDocumentsTest test
```

Na razie pokazałem Ci, jak skonfigurować kontener Spring poprzez dodanie ziaren i utworzenie do nich odwołań, aby kontener wiedział, jak je tworzyć i jakie łączą je relacje, oraz by mógł je przygotować, gdy będą potrzebne. Ale przypomnę, że w systemie Spring konfigurację można tworzyć także innymi sposobami, i dlatego w następnym podrozdziale pokazuję, jak utworzyć taką samą konfigurację jak wcześniej przy użyciu adnotacji.

Używanie adnotacji Spring

Adnotacje do języka Java wprowadzono w jego wersji 5. Było to znakomite posunięcie ze strony twórców Javy, ponieważ za pomocą adnotacji do klas można dodawać metadane stosowane zarówno podczas kompilacji, jak i działania programu, co stwarza nowe możliwości dla programistów. Programiści systemu Spring postanowili wykorzystać tę okazję do budowy mechanizmu konfiguracyjnego opartego na adnotacjach. Mechanizm ten pojawił się w wersji 2.5 systemu.

Ale wystarczy tego gadania. Czas się wziąć za kod źródłowy i sprawdzić, co trzeba zmienić, aby zastosować konfigurację opartą na adnotacjach. Spójrz na listing 3.8.

Listing 3.8. *Zawartość pliku AnnotatedSearchEngine.java*

```
package com.apress.isf.spring.annotated.service;

import java.util.ArrayList;
```

```
import java.util.Arrays;
import java.util.List;

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

import com.apress.isf.java.model.Document;
import com.apress.isf.java.model.Type;
import com.apress.isf.java.service.SearchEngine;
import com.apress.isf.spring.data.DocumentDAO;

@Service("engine")
public class AnnotatedSearchEngine implements SearchEngine {
        @Autowired
        private DocumentDAO documentDAO;

        public List<Document> findByType(Type documentType) {
                List<Document> result = new ArrayList<Document>();
                for(Document doc : listAll()){
                        if(doc.getType().getName().equals(documentType.getName()))
                                result.add(doc);
                }
                return result;
        }

        public List<Document> listAll() {
                return Arrays.asList(documentDAO.getAll());
        }
}
```

Na powyższym listingu przedstawiono nową implementację interfejsu `SearchEngine` w postaci klasy `AnnotatedSearchEngine`. Jak widać, zostały w niej użyte adnotacje `@Service("engine")` i `@Autowired`. System Spring obsługuje kilka adnotacji, które zostały wymienione w tabeli 3.2. Są one jedynie markerami albo stereotypami, podobnie jak adnotacja `@Service`. Adnotacja ta może mieć wartość. W tym przypadku jest to wartość `engine`, która oznacza, że kontener utworzy ziarno o identyfikatorze `engine`. W tym przykładzie wskazywana jest klasa `AnnotatedSearchEngine`. To tak samo, jakbyśmy w pliku XML wpisali poniższy element:
`<bean id="engine" class="com.apress.isf.spring.annotated.service AnnotatedSearchEngine" />`

Tabela 3.2. *Stereotypy*

Stereotyp/marker	Opis
`@Component`	Jest to marker, ogólny stereotyp, który przez system Spring jest rozpoznawany jako zarządzany przez niego składnik.
`@Repository`	Jest to specjalizacja adnotacji `@Component` dotycząca obiektu obsługującego dostęp do danych. Klasy oznaczone tą adnotacją mogą być przetwarzane przez inne narzędzia, a nawet aspekty w kontenerze Spring.
`@Service`	Jest to specjalizacja adnotacji `@Component` dotycząca warstwy usługowej.
`@Controller`	To również jest specjalizacja adnotacji `@Component`, której zazwyczaj używa się w kontekście sieciowym.

Zastosowaliśmy też adnotację `@Autowired`. Nakazuje ona systemowi Spring utworzenie egzemplarza i przypisanie go do zadeklarowanej zmiennej. Ma takie samo znaczenie jak poniższy element XML:

```
<property name="documentDAO" ref="documentDAO" />
```

Podsumowując, konfiguracja klasy `AnnotatedSearchEngine` będzie wyglądała następująco:

```
<bean id="engine" class="com.apress.isf.spring.annotated.service AnnotatedSearchEngine">
        <property name="documentDAO" ref="documentDAO" />
</bean>
```

Na listingu 3.9 znajduje się kod źródłowy klasy `AnnotatedDocumentRepository`. Zawiera ona marker `@Repository` i będzie wstrzykiwana do implementacji `SearchEngine` dzięki adnotacji `@Autowired` (listing 3.8).

Listing 3.9. *AnnotatedDocumentRespository.java*

```
package com.apress.isf.spring.annotated.data;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Repository;

import com.apress.isf.java.model.Document;
import com.apress.isf.java.model.Type;
import com.apress.isf.spring.data.DocumentDAO;

@Repository("documentDAO")
public class AnnotatedDocumentRepository implements DocumentDAO {

        public Document[] getAll() {
                return storage();
        }

        private Document[] storage(){
                List<Document> result = new ArrayList<Document>();

                Type type = new Type();
                type.setName("PDF");
                type.setDesc("Portable Document Format");
                type.setExtension(".pdf");

                Document document = new Document();
                document.setName("Szablon książki");
                document.setType(type);
                document.setLocation("/Users/felipeg/Documents/Random/Book Template.pdf");

                result.add(document);

                document = new Document();
                document.setName("Przykładowa umowa");
                document.setType(type);
                document.setLocation("/Users/felipeg/Documents/Contracts/Sample Contract.pdf");

                result.add(document);

                type = new Type();
                type.setName("NOTE");
                type.setDesc("Notatki tekstowe");
                type.setExtension(".txt");

                document = new Document();
                document.setName("Clustering with RabbitMQ");
```

```
                document.setType(type);
                document.setLocation("/Users/felipeg/Documents/Random/Clustering with RabbitMQ.txt");

                result.add(document);

                type = new Type();
                type.setName("WEB");
                type.setDesc("Łącze sieciowe");
                type.setExtension(".url");

                document = new Document();
                document.setName("Pro Spring Security Book");
                document.setType(type);
                document.setLocation("http://www.apress.com/9781430248187");

                result.add(document);

                return result.toArray(new Document[result.size()]);
        }
}
```

Teraz skierujemy uwagę na listing 3.10. Znajduje się na nim treść pliku konfiguracyjnego XML zawierającego elementy nakazujące kontenerowi Spring poszukać klas z adnotacjami i ich adnotacji. W pliku tym użyty został też specjalny element `<context:component-scan-base-package/>`. Należy on do tej samej przestrzeni nazw XML co nasza konfiguracja. W przestrzeni tej będzie dodanych jeszcze więcej znaczników, o których przeczytasz w dalszych rozdziałach tej książki. Na razie wystarczy wiedzieć, że opisywany element nakazuje kontenerowi Spring wyszukiwanie klas z adnotacjami z podanego pakietu, tu `com.apress.isf.spring.annotated`, i wszystkich podpakietów.

Listing 3.10. *Zawartość pliku Mydocuments-annotations-context.xml*

```
<?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-4.0.xsd">

  <context:component-scan base-package="com.apress.isf.spring.annotated"/>

  <bean id="webType" class="com.apress.isf.java.model.Type">
    <property name="name" value="WEB" />
    <property name="desc" value="Łącze sieciowe" />
    <property name="extension" value=".url" />
  </bean>
</beans>
```

Stereotypy Spring

Stereotypy są markerami pomagającymi kontenerowi Spring w zidentyfikowaniu składników zarządzanych przez system Spring. Markerów tych można używać do oznaczania zewnętrznych narzędzi przetwarzających albo jako referencji dla punktów przecięcia (ang. *pointcat*) w programowaniu aspektowym. Obecnie stereotypy

znajdujące się w kodzie źródłowym pomagają programistom zrozumieć ten kod, ponieważ sprawiają, że jest on bardziej czytelny oraz że są w nim zaznaczone warstwy strukturalne i architektoniczne. Do najczęściej wykorzystywanych stereotypów (w społeczności skupionej wokół systemu Spring i oczywiście w naszej aplikacji) zaliczają się te, których opis znajduje się w tabeli 3.2.

Konfiguracja w postaci klasy ziarna Java

W wersji 3.0 systemu Spring wprowadzono nową technikę konfigurowania kontenera Spring polegającą na użyciu nowej klasy konfiguracyjnej w postaci ziarna Java. Początkowo był to osobny projekt, ale postanowiono wcielić go do rdzenia systemu w wersji 3.0. Aktualnie jest to jedna z zalecanych metod konfigurowania kontenera, ponieważ daje wyraźny obraz relacji występujących między klasami i pokazuje interakcje pomiędzy nimi. A w niektórych przypadkach pomaga nawet uniknąć całego tego bałaganu związanego z plikami XML.

Na listingu 3.11 znajduje się kod źródłowy klasy konfiguracyjnej w Javie. Klasa ta jest równoważna z pokazanym na listingu 3.6 kodem konfiguracyjnym w formacie XML. Każda definicja ziarna z tego pliku ma odpowiednik w postaci adnotacji `@Bean` w metodzie.

Listing 3.11. *Zawartość pliku MyDocumentsContext.java*

```
package com.apress.isf.spring.config;

import java.util.HashMap;
import java.util.Map;

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

import com.apress.isf.java.model.Document;
import com.apress.isf.java.model.Type;
import com.apress.isf.java.service.SearchEngine;
import com.apress.isf.spring.data.DocumentDAO;
import com.apress.isf.spring.data.DocumentRepository;
import com.apress.isf.spring.service.ServiceSearchEngine;

@Configuration
public class MyDocumentsContext {

        private Map<String,Document> documents = new HashMap<String,Document>();
        private Map<String,Type> types = new HashMap<String,Type>();

        @Bean
        public Type webType(){
                return getTypeFromMap("web");
        }

        @Bean
        public SearchEngine engine(){
                ServiceSearchEngine engine = new ServiceSearchEngine();
                engine.setDocumentDAO(documentDAO());
                return engine;
        }

        public MyDocumentsContext(){
                Type type = new Type();
                type.setName("PDF");
                type.setDesc("Portable Document Format");
                type.setExtension(".pdf");
```

```
                Document document = new Document();
                document.setName("Szablon książki");
                document.setType(type);
                document.setLocation("/Users/felipeg/Documents/Random/Book Template.pdf");

                documents.put("doc1", document);
                types.put("pdf",type);

                document = new Document();
                document.setName("Przykładowa umowa");
                document.setType(type);
                document.setLocation("/Users/felipeg/Documents/Contracts/Sample Contract.pdf");

                documents.put("doc2",document);

                type = new Type();
                type.setName("NOTE");
                type.setDesc("Notatki tekstowe");
                type.setExtension(".txt");

                document = new Document();
                document.setName("Clustering with RabbitMQ");
                document.setType(type);
                document.setLocation("/Users/felipeg/Documents/Random/Clustering with RabbitMQ.txt");

                documents.put("doc3",document);
                types.put("note",type);

                type = new Type();
                type.setName("WEB");
                type.setDesc("Łącze sieciowe");
                type.setExtension(".url");

                document = new Document();
                document.setName("Pro Spring Security Book");
                document.setType(type);
                document.setLocation("http://www.apress.com/9781430248187");

                documents.put("doc4",document);
                types.put("web",type);
                }

        private DocumentDAO documentDAO(){
                DocumentRepository documentDAO = new DocumentRepository();
                documentDAO.setDoc1(getDocumentFromMap("doc1"));
                documentDAO.setDoc2(getDocumentFromMap("doc2"));
                documentDAO.setDoc3(getDocumentFromMap("doc3"));
                documentDAO.setDoc4(getDocumentFromMap("doc4"));
                return documentDAO;
        }

        private Document getDocumentFromMap(String documentKey){
                return documents.get(documentKey);
        }

        private Type getTypeFromMap(String typeKey){
                return types.get(typeKey);
        }
}
```

Na początku kodu tej klasy dodaliśmy adnotację `@Configuration`, a metodom przypisaliśmy adnotację `@Bean`. Użycie adnotacji `@Configuration` jest jak przekazanie kontenerowi Spring następującej informacji: „Tutaj znajdują się definicje moich ziaren". Natomiast adnotacja `@Bean` przed metodą jest równoznaczna z utworzeniem elementu `<bean/>` i ustawieniem jego własności. A zatem powyższa klasa poinformuje kontener Spring o tym, jakie są ziarna i jak będą powiązane.

Teraz wykorzystamy tę nową konfigurację w teście jednostkowym (listing 3.12).

Listing 3.12. *Zawartość pliku MyDocumentsBeanConfigurationTest.java*

```
package com.apress.isf.spring.test;

import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertEquals;

import java.util.List;

import org.junit.Before;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.apress.isf.java.model.Document;
import com.apress.isf.java.model.Type;
import com.apress.isf.java.service.SearchEngine;
import com.apress.isf.spring.config.MyDocumentsContext;

public class MyDocumentsBeanConfigurationTest {

        private ApplicationContext context;
        private SearchEngine engine;
        private Type webType;

        @Before
        public void setup(){
                context = new AnnotationConfigApplicationContext(MyDocumentsContext.class);
                engine = context.getBean(SearchEngine.class);
                webType = context.getBean(Type.class);
        }

        @Test
        public void testWithBeanConfigurationFindByType() {
                List<Document> documents = engine.findByType(webType);
                assertNotNull(documents);
                assertTrue(documents.size() == 1);
                assertEquals(webType.getName(),documents.get(0).getType().getName());
                assertEquals(webType.getDesc(),documents.get(0).getType().getDesc());
                assertEquals(webType.getExtension(),documents.get(0).getType().getExtension());
        }

        @Test
        public void testWithBeanConfigurationListAll() {
                List<Document> documents = engine.listAll();
                assertNotNull(documents);
                assertTrue(documents.size() == 4);
        }
}
```

W tym kodzie do załadowania konfiguracji użyliśmy klasy Java `AnnotationConfigApplicationContext`. W ten sposób informujemy kontener Spring, że stosowana jest konfiguracja klasowa, a nie w postaci pliku XML.

Aby przeprowadzić ten test jednostkowy, należy wykonać polecenie `gradle`:

```
gradle test
```

Użycie klasy GroovyBeanDefinitionReader

W wersji 4. systemu Spring wprowadzono nowy sposób konfigurowania ziaren przy użyciu języka programowania Groovy. Nowość tę ściągnięto z systemu Grails, w którym do tworzenia ziaren zaczęto używać języka specjalistycznego (ang. *domain-specific language* — DSL).

Spójrz na przykładowy kod przedstawiony na listingu 3.13.

Listing 3.13. *Zawartość pliku mydocuments.groovy*

```
import com.apress.isf.spring.service.ServiceSearchEngine
import com.apress.isf.spring.data.DocumentRepository
import com.apress.isf.java.model.Document

beans {

        engine(ServiceSearchEngine){
                documentDAO = ref("documentDAO")
        }

        documentDAO(DocumentRepository){
                doc1 = ref("doc1")
                doc2 = ref("doc2")
                doc3 = ref("doc3")
                doc4 = ref("doc4")
        }

        doc1(Document){
                name = "Szablon książki"
                type = ref("pdfType")
                location = "/Users/felipeg/Documents/Random/Book Template.pdf"
        }

        doc2(Document){
                name = "Przykładowa umowa"
                type = ref("pdfType")
                location = "/Users/felipeg/Documents/Contracts/Sample Contract.pdf"
        }

        doc3(Document){
                name = "Clustering with RabbitMQ"
                type = ref("noteType")
                location = "/Users/felipeg/Documents/Random/Clustering with RabbitMQ.txt"
        }

        doc4(Document){
                name = "Pro Spring Security Book"
                type = ref("webType")
                location = "http://www.apress.com/9781430248187"
        }
```

```
        webType(com.apress.isf.java.model.Type){
                name = "WEB"
                desc = "Łącze sieciowe"
                extension = ".url"
        }

        pdfType(com.apress.isf.java.model.Type){
                name = "PDF"
                desc = "Portable Document Format"
                extension = ".url"
        }

        noteType(com.apress.isf.java.model.Type){
                name = "NOTE"
                desc = "Notatki tekstowe"
                extension = ".txt"
        }
}
```

Na listingu tym przedstawiono nowy sposób definiowania konfiguracji za pomocą języka Groovy zamiast XML (listing 3.6), adnotacji (listing 3.9) lub klasy Java (listing 3.10). Na razie nie przejmuj się składnią, bo jej opis znajduje się w rozdziale 18.

Teraz utworzymy test jednostkowy (listing 3.14), w którym użyjemy nowej klasy `GroovyBeanDefinitionReader` do załadowania definicji ziaren. Klasa ta będzie ładowała plik *mydocuments.groovy*, którego zawartość pokazano na listingu 3.13.

Listing 3.14. *Zawartość pliku MyDocumentsBeanDefinitionReaderTest.java*

```
package com.apress.isf.spring.test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

import java.util.List;

import org.junit.Before;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericGroovyApplicationContext;

import com.apress.isf.java.model.Document;
import com.apress.isf.java.model.Type;
import com.apress.isf.java.service.SearchEngine;

public class MyDocumentsBeanDefinitionReaderTest {

        private ApplicationContext context;
        private SearchEngine engine;
        private Type webType;

        @Before
        public void setup(){
               context = new GenericGroovyApplicationContext("META-INF/spring/mydocuments.groovy");
                engine = context.getBean(SearchEngine.class);
                webType = context.getBean("webType",Type.class);
        }
```

```
        @Test
        public void testWithGroovyFindByType() {
                List<Document> documents = engine.findByType(webType);
                assertNotNull(documents);
                assertTrue(documents.size() == 1);
                assertEquals(webType.getName(),documents.get(0).getType().getName());
                assertEquals(webType.getDesc(),documents.get(0).getType().getDesc());
                assertEquals(webType.getExtension(),documents.get(0).getType().getExtension());
        }

        @Test
        public void testWithGroovyListAll() {
                List<Document> documents = engine.listAll();
                assertNotNull(documents);
                assertTrue(documents.size() == 4);
        }
}
```

Za pomocą klasy `GroovyBeanDefinitionReader` załadowaliśmy skrypt w języku Groovy, podając ścieżkę do niego (`META-INF/spring/mydocuments.groovy`). Klasa ta uruchomi kontener Spring, tworząc wszystkie potrzebne egzemplarze, wiążąc ze sobą nasze klasy oraz przygotowując je do użytku, gdy będą potrzebne. Pamiętaj, że tym razem do konfiguracji kontenera Spring wykorzystaliśmy język programowania Groovy!

Teraz możesz uruchomić test za pomocą poniższego polecenia:

```
gradle test
```

Który rodzaj konfiguracji wybrać

Każdy typ konfiguracji różni się czymś od pozostałych. Różnice te zostały opisane w tabeli 3.3.

Tabela 3.3. *Różnice w zastosowaniu różnych rodzajów konfiguracji*

Typ konfiguracji	Zastosowanie
XML	Nadaje się do użycia z zewnętrznymi bibliotekami i różnymi środowiskami programistycznymi. Jest czytelna, ale może być bardzo rozwlekła. Wprawdzie można ją podzielić na kilka plików, jednak trzeba nad nimi zapanować.
Adnotacje	W tym typie konfiguracji wiąże się kontekst Spring z aplikacją. Można tego uniknąć przy użyciu własnych dziedzinowych adnotacji.
Ziarno Java	Jest to jedna z aktualnie zalecanych metod dla programistów, którzy nie lubią formatu XML. Może być stosowana w przypadku ziaren i składników, które niewiele się zmieniają.
Konfiguracja w języku Groovy	Nowa technika umożliwiająca wykorzystanie języka programowania Groovy do definiowania konfiguracji. Jest prosta i oszczędna, jeśli chodzi o kod źródłowy.

Wybór typu konfiguracji zależy od potrzeb i sposobu zarządzania cyklem programistycznym. Czasami dyktują go też preferencje zespołu programistycznego, któremu łatwiej może być zarządzać projektem i osiągać cele przy użyciu określonej technologii.

Podsumowanie

W tym rozdziale poznałeś różne sposoby tworzenia ziaren w celu uzyskania tego samego efektu. Poznałeś też różnorodne klasy do ładowanie ziaren, takie jak `ClassPathXmlApplicationContext`, `AnnotationConfigApplicationContext` oraz nowa klasa `GenericGroovyApplicationContext`.

Nauczyłeś się oznaczać klasy adnotacjami w taki sposób, by poinformować system Spring o tym, jakich ziaren ma używać, jakie są ich zależności oraz jakie relacje je łączą. Dowiedziałeś się, że adnotacja `@Repository` oznacza klasę jako zarządzany przez Spring składnik pełniący funkcję obiektu dostępowego do danych.

Ponadto dowiedziałeś się, jak używać konfiguracyjnej klasy w Javie oraz jak za pomocą jej adnotacji `@Configuration` i `@Bean` tworzyć ziarna i relacje między nimi. Poznałeś też technikę tworzenia konfiguracji przy użyciu języka programowania Groovy.

W następnym rozdziale dodamy nowe funkcje do aplikacji *Moje dokumenty*. Dowiesz się, w jaki sposób system Spring inicjuje klasy, oraz poznasz różne metody tworzenia różnych egzemplarzy tej samej klasy.

Skorowidz

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Ś

T

U

W

Z

ODMIEŃ SWOJE PROGRAMISTYCZNE ŻYCIE DZIĘKI SPRING FRAMEWORK!

Wprowadzenie do Spring Framework dla programistów Java

Pojawienie się Springa na zawsze zmieniło świat Javy. Ten przemyślany, dobrze zaprojektowany, dający programistom mnóstwo możliwości framework został przez nich wyjątkowo ciepło przyjęty. Obecnie jest on prawdopodobnie najczęściej wybieranym narzędziem do tworzenia aplikacji — niezależnie od stopnia ich skomplikowania. Za Springiem stoją ogromna społeczność oraz przepastne zasoby dokumentacji, przykładów i instrukcji. Już teraz dowiedz się, dlaczego Spring jest numerem 1!

Książka ta wprowadzi Cię w świat Springa oraz zagadnień z nim związanych. Sięgnij po nią, by skonfigurować swój pierwszy projekt i przetestować aplikację. W kolejnych rozdziałach odkryjesz sposoby wykorzystania programowania aspektowego i wzorca MVC oraz integrowania aplikacji Spring z systemami zewnętrznymi. Ponadto przygotujesz swoje pierwsze API typu REST oraz zapiszesz dane w bazie MongoDB. Książka ta jest doskonałą lekturą dla osób znających język Java, chcących poznać legendarne możliwości Spring Framework.

Dzięki Spring Framework:

- błyskawicznie przygotujesz rozbudowaną aplikację
- bezproblemowo wykorzystasz JMS do integracji aplikacji
- z łatwością skorzystasz z baz danych NoSQL
- zbudujesz przejrzyste REST API

ISBN 978-83-283-0439-0
9 788328 304390
cena: 57,00 zł

Apress