Willie Wheeler, Joshua White

Spring
w praktyce

Lektura obowiązkowa
dla programistów Javy!

Manning Publication

Helion

Tytuł oryginału: Spring in Practice

Tłumaczenie: Paweł Gonera

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-246-8184-6

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/sprip.zip

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

Printed in Poland.

Spis treści

1 Przedstawiamy Spring — kontener wstrzykiwania zależności

W tym rozdziale:

- Główne obszary funkcjonalne Spring Framework
- Elastyczna konfiguracja z użyciem wstrzykiwania zależności
- Łączenie ziaren za pomocą XML
- Automatyczne łączenie oraz skanowanie komponentów z użyciem adnotacji

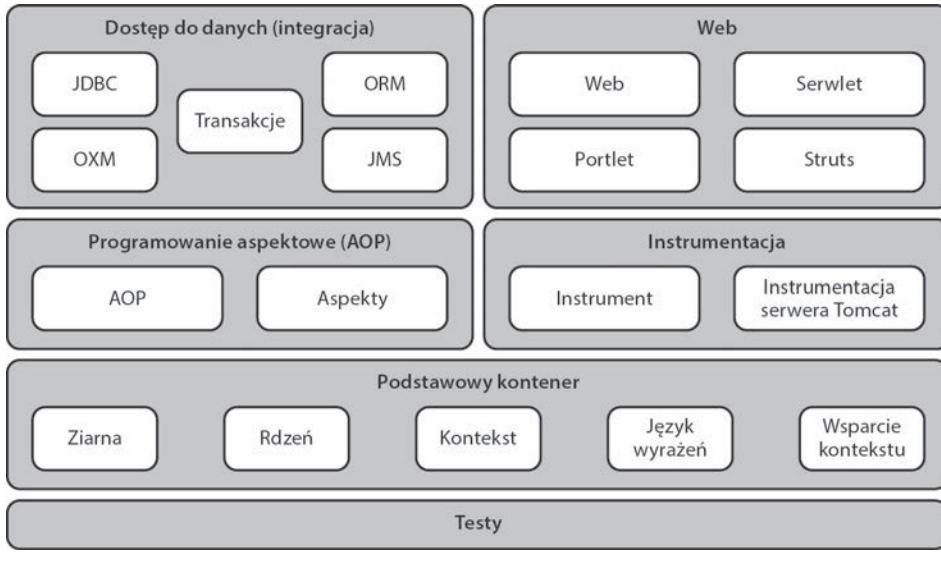
W tym rozdziale przedstawimy krótko Spring Framework, zaczynając od wyjaśnienia, czym on jest, i omówimy jego główne elementy. Następnie zajmiemy się podstawowymi zasadami wykorzystywanymi w Spring Framework oraz mechanizmem odwrócenia kontroli i jego wpływem na wstrzykiwanie zależności. Na koniec pokażemy niewielki przykład, który ilustruje, jak korzystać z Spring Core Container. Zaczynamy!

1.1. Czym jest Spring i dlaczego z niego korzystamy?

Spring Framework jest biblioteką tworzenia aplikacji dostępną na zasadach open source, która umożliwia uproszczenie procesu tworzenia oprogramowania biznesowego za pomocą języka Java. Biblioteka ta pozwala osiągnąć ten cel przez

udostępnienie programistom modelu komponentów oraz zbioru prostych i spójnych API, które w efektywny sposób izolują ich od złożonego kodu podstawowego, wymaganego w skomplikowanych aplikacjach.

W ostatnich dziewięciu latach zakres tej biblioteki znacznie się zwiększył, ale pomimo tego pozostała prosta i łatwa w użyciu. Obecnie składa się ona z około 20 modułów, które mogą być podzielone na 6 podstawowych obszarów funkcjonalnych. Jak pokazano na rysunku 1.1, obszarami tymi są: dostęp do danych (integracja), sieć WWW, programowanie aspektowe (AOP), instrumentacja, podstawowy kontener oraz testy.



Rysunek 1.1. Diagram blokowy przedstawiający sześć podstawowych obszarów funkcjonalnych Spring

Modularność ta daje programistom możliwość swobodnego wyboru elementów biblioteki wykorzystywanych w ich aplikacjach, bez potrzeby dołączania do niej całej biblioteki. Zacznijmy od przedstawienia tych obszarów funkcjonalnych.

1.1.1. Główne elementy biblioteki

W kolejnych punktach przedstawimy krótkie wprowadzenie do każdego z sześciu podstawowych obszarów funkcjonalnych Spring. Każdy z tych tematów zostanie omówiony dokładnie w dalszej części książki.

PODSTAWOWY KONTENER SPRING

Więcej informacji na temat wstrzykiwania zależności znajduje się w podrozdziale 1.2. Na razie wystarczy wiedzieć, że kontener DI jest głównym elementem Spring Framework i zapewnia podstawowe funkcje wykorzystywane przez wszystkie inne moduły. Kontener zapewnia funkcje oddzielania tworzenia, konfiguracji oraz zarządzania ziarnami (będą przedstawione później) od kodu aplikacji.

PROGRAMOWANIE ASPEKTOWE (AOP)

Spring Framework obsługuje również programowanie aspektowe, zarówno w prostszej wersji, nazywanej Spring AOP, jak i rozbudowanej, AspectJ. Przedstawiony dokładniej w dalszej części książki mechanizm AOP pozwala hermetyzować zadania przekrojowe (bezpieczeństwo, logowanie, zarządzanie transakcjami) w formie aspektów, dzięki czemu można zachować modularność i możliwość wielokrotnego użytku kodu. Zadania takie często nie mogą być elegancko oddzielone od reszty systemu, co powoduje powielanie kodu, powstanie zależności pomiędzy systemami lub oba te efekty jednocześnie. Podobnie jak kontener DI, obsługa AOP jest niezwykle użyteczna dla programistów i używana do tworzenia różnych części samej biblioteki. Spring implementuje na przykład deklaratywne zarządzanie transakcjami przez AOP, ponieważ transakcje są zadaniami przekrojowymi.

DOSTĘP DO DANYCH (INTEGRACJA)

Moduł dostępu do danych oraz integracji zapewnia obsługę API Java Database Connectivity (JDBC), odwzorowania relacyjno-obiektowego (ORM), odwzorowania Object/XML (OXM), Java Message Service (JMS) oraz obsługę transakcji.

Moduł JDBC zapewnia warstwę abstrakcji uwalniającą programistów od konieczności pisania nudnego i narażonego na błędy kodu podstawowego, ponieważ w sposób automatyczny zarządza połączeniami bazy danych oraz pulami połączeń, jak również przez odwzorowanie błędów definiowanych w produktach dostawców na standardową hierarchię wyjątków. Ułatwia też odwzorowanie obiektu `java.sql.ResultSet` na listę obiektów domeny i wykonywanie procedur składowanych.

Jeżeli wolisz korzystać z ORM zamiast prostego kodu JDBC obsługującego bazę danych, to masz szczęście. Moduł ORM wspiera najlepsze dostępne produkty, takie jak Hibernate, iBATIS, Java Data Objects (JDO) oraz Java Persistence API (JPA).

Informacja na temat iBATIS

Apache iBATIS został w roku 2010 wycofany i zastąpiony przez MyBatis (*mybatis.org*). Choć iBATIS 2 był obsługiwany od Spring 2, to z powodu problemów z obsługą czasu Spring 3 nie zawiera jeszcze oficjalnego wsparcia tej biblioteki. Zapoznaj się z modułem MyBatis-Spring dostępnym na stronie *http://www.mybatis.org/spring*.

Moduł OXM zapewnia warstwę abstrakcji oferującą prostą i spójną obsługę popularnych narzędzi odwzorowania Object/XML, takich jak Castor, Java Architecture for XML Binding (JAXB), JiBX, XMLBeans oraz XStream.

Moduł JMS oferuje uproszczone API do tworzenia i konsumowania komunikatów. Ostatni z modułów — transakcji — zapewnia zarówno programową, jak i deklaratywną obsługę transakcji.

SIEĆ WWW

Moduł sieci WWW w Spring oferuje standardowe mechanizmy do integrowania klas Spring z aplikacjami WWW, wieloczęściowego przesyłania plików oraz wywołań zdalnych korzystających z WWW. Moduł ten ma własną bibliotekę „model-widok-kontroler" (MVC), korzystającą z serwletów lub portletów, a także pozwala na integrowanie wielu popularnych bibliotek oraz technologii takich jak Struts, JavaServer Faces (JSF), Velocity, FreeMarker oraz JavaServer Pages (JSP).

TESTOWANIE

Spring wspiera również testowanie aplikacji. Dostępny jest moduł zapewniający obsługę zarówno biblioteki JUnit, jak i TestNG.

Po ogólnym przedstawieniu Spring Framework czas na pokazanie zalet korzystania z tej biblioteki.

1.1.2. Dlaczego warto korzystać ze Spring?

Być może korzystałeś z innych bibliotek lub API (a może nawet je tworzyłeś), które obsługują przynajmniej jeden z mechanizmów dostępnych w Spring. Dlaczego miałbyś przeznaczać czas na nauczenie się innej biblioteki? Oprócz modelu komponentowego oraz prostego i spójnego API, które izoluje programistów od złożonego i wrażliwego na błędy kodu podstawowego, istnieje jeszcze kilka innych przyczyn:

- *Jakość* — od projektu modułów, pakietów, struktury klas oraz API aż do implementacji i pokrycia kodu testami Spring Framework jest doskonałym przykładem kodu open source o wysokiej jakości.
- *Modularność* — jak wcześniej wspomnieliśmy, biblioteka posiada blisko 20 modułów, co daje programistom możliwość wybrania jej fragmentów wykorzystywanych w aplikacji bez konieczności dołączania całej biblioteki.
- *Promocja najlepszych praktyk* — model programowania Spring bazujący na zwykłych obiektach Java (POJO) promuje rozłączanie komponentów, testowanie jednostkowe oraz inne najlepsze praktyki.
- *Nieskomplikowany proces nauki* — dzięki spójności i prostocie API Spring nie jest zbyt trudny w nauce. Gdy będziemy pokazywać kolejne składniki tej biblioteki, zauważysz często występujące wzorce. Dodatkowo dostępne są setki źródeł drukowanych oraz internetowych, w tym również fora dyskusyjne, na których często publikują informacje główni programiści biblioteki.
- *Popularność* — jak łatwo wywnioskować z mnogości publikacji, witryn WWW oraz ofert pracy, biblioteka Spring Framework jest niemal wszechobecna.

Doskonałą książką uzupełniającą tę, którą właśnie czytasz, jest *Spring in Action*, wydanie 3. (Manning Publications 2011), której autorem jest Craig Walls.

Spring oferuje wiele możliwości, ale trzeba nieco czasu, aby je docenić. Możesz być pewien, że jest tego wart. Ucząc się korzystania z biblioteki Spring i stosując ją do rozwiązywania problemów, przekonasz się, że łączy ona ze sobą różne technologie,

dzięki czemu umożliwia tworzenie spójnych aplikacji. Zauważysz, że parametry konfiguracyjne nie są zapisywane na stałe w kodzie, lecz centralnie w standardowych lokalizacjach. Będziesz projektował zależności między klasami wykorzystujące interfejsy, co lepiej nadaje się do obsługi zmieniających się wymagań. Ostatecznie wykonasz więcej przy mniejszym nakładzie pracy, ponieważ Spring Framework obsługuje mechanizmy podstawowe; będziesz więc mógł w większym stopniu skupić się na rozwiązywaniu problemów biznesowych.

Wiesz już, co może Ci zaoferować Spring. Zapoznamy się teraz z możliwościami kontenera podstawowego, pokazanymi na rysunku 1.2. Kontener podstawowy obsługuje funkcje odwrócenia kontroli (IoC) oraz DI, na bazie których są zbudowane pozostałe moduły.

Rysunek 1.2. Kontener podstawowy

1.2. Elastyczna konfiguracja z użyciem wstrzykiwania zależności

Dzięki kontenerom takim jak Spring mechanizm IoC stał się od kilku lat bardzo popularny. Choć w erze internetu wydaje się to co najmniej wiekiem, dla wielu programistów jest to nadal nowa i mało znana koncepcja. W tym podrozdziale wyjaśnimy, czym jest IoC, oraz przedstawimy mechanizmy, z których korzysta. Będziesz miał również możliwość samodzielnego skonfigurowania kontenera Spring.

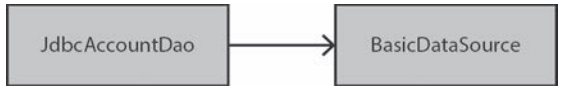
1.2.1. Konfigurowanie zależności w stary sposób

Przeanalizujmy relacje pomiędzy obiektem dostępu do danych (DAO) oraz `DataSource`, które to obiekty są przedstawione w poniższym kodzie. Aby obiekty DAO mogły współpracować z `DataSource`, należy w klasie `JdbcAccountDao` utworzyć i zainicjować obiekt `DataSource` odpowiednimi parametrami połączenia:

```
// Projekt źródłowy: sip01, gałąź: 01 (projekt Maven)
package com.springinpractice.ch01.dao.jdbc;

import org.apache.commons.dbcp.BasicDataSource;
import com.springinpractice.ch01.dao.AccountDao;

public class JdbcAccountDao implements AccountDao {
   private BasicDataSource dataSource;

   public JdbcAccountDao() {
      dataSource = new BasicDataSource();
      dataSource.setDriverClassName("com.mysql.jdbc.Driver");
      dataSource.setUrl("jdbc:mysql://localhost:3306/springbook" +
         "?autoReconnect=true");
      dataSource.setUsername("root");
```

```
        dataSource.setPassword("");
    }
}
```

W kodzie tym klasa `JdbcAccountDao` jest *zależna* od klasy JDBC `DataSource`. Korzystanie z interfejsów jest zdecydowanie najlepszą praktyką. Jak pokazano na rysunku 1.3, kod zawiera również zależność od `BasicDataSource` — specyficznej implementacji `DataSource` z projektu Apache Commons Database Connection Pool (DBCP).



Rysunek 1.3. Klasa JdbcAccountDao jest zależna od BasicDataSource z Apache DBCP — konkretnej implementacji DataSource

Głównym problemem jest oczywiście to, że klasa `JdbcAccountDao` zawiera informacje na temat implementacji, tworzenia i konfiguracji interfejsu `DataSource`. Innym potencjalnym problemem jest konieczność współdzielenia danych o połączeniu z wieloma obiektami DAO. W wyniku zastosowania przedstawionego projektu zmiana implementacji `DataSource` lub jej konfiguracji może powodować konieczność modyfikowania kodu w wielu miejscach oraz ponownej instalacji po zmianie implementacji `DataSource` lub jego konfiguracji.

Jakość kodu można poprawić przez wydzielenie z niego parametrów połączenia za pomocą `java.util.Properties`. Pozostanie jednak subtelniejszy problem. W poprzednim przykładzie kodu to specyfikacja klasy powoduje powstanie zależności. Spójrzmy teraz, w jaki sposób możemy odwrócić kontrolę poprzez wstrzyknięcie naszych zależności.

1.2.2. Wstrzykiwanie zależności

Jednym ze sposobów eliminowania konkretnych zależności od klasy `BasicDataSource` jest zdefiniowanie jej zewnętrznie i wstrzyknięcie jej do klasy `JdbcAccountDao` jako `DataSource`. Daje to nam sporą elastyczność, ponieważ możemy łatwo zmieniać konfigurację w jednym miejscu. Jeżeli chcesz użyć proxy dla `DataSource` przed operacją wstrzyknięcia, możesz to zrobić. W przypadku uruchamiania testów jednostkowych możesz zastąpić `DataSource` imitacją — to również da się zrobić. Mechanizm DI zapewnia dużą elastyczność, którą tracimy, gdy tworzenie zależności jest na stałe zapisane w korzystających z nich komponentach.

Aby mechanizm DI mógł działać, musimy utworzyć obiekt `DataSource` zewnętrznie, a następnie albo utworzyć obiekt DAO z jego użyciem, albo ustawić go za pomocą metody ustawiającej, jak pokazano poniżej:

```
// Projekt źródłowy: sip01, gałąź: 02 (projekt Maven)
package com.springinpractice.ch01.dao.jdbc;

import javax.sql.DataSource;
import com.springinpractice.ch01.dao.AccountDao;
```

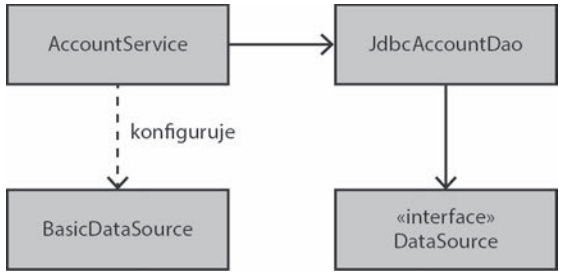
```
public class JdbcAccountDao implements AccountDao {
   private DataSource dataSource;

   public JdbcAccountDao() {}

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
   }
}
```

Zwróć uwagę, że DAO nie posiada już na stałe zapisanej zależności od `BasicData➥Source`. Zauważ, że wiersz z importem klasy `BasicDataSource` został usunięty. Ponieważ zależność jest inicjowana poprzez metodę ustawiającą, nie jest konieczne definiowanie konstruktora, za pomocą którego był inicjowany obiekt `DataSource`. Innym sposobem refaktoryzacji tej klasy jest przekazanie implementacji `DataSource` jako argumentu konstruktora. Oba podejścia są znacznym usprawnieniem kodu. Możesz jednak argumentować, że udało się nam jedynie przenieść tworzenie zależności w inne miejsce kodu. Spójrzmy na usługę tworzącą DAO:

```
//Projekt źródłowy: sip01, gałąź: 02 (projekt Maven)
package com.springinpractice.ch01.service;
import java.util.Properties;
import java.io.InputStream;
import org.apache.commons.dbcp.BasicDataSource;
import com.springinpractice.ch01.dao.jdbc.JdbcAccountDao;

public class AccountService {
   private JdbcAccountDao accountDao;

   public AccountService() {
   try {
      Properties props = new Properties();
      InputStream inputStream = this.getClass().getClassLoader()
         .getResourceAsStream("dataSource.properties");
      props.load(inputStream);

      BasicDataSource dataSource = new BasicDataSource();
      dataSource.setDriverClassName(
         props.getProperty("driverClassName"));
      dataSource.setUrl(props.getProperty("url"));
      dataSource.setUsername(props.getProperty("username"));
      dataSource.setPassword(props.getProperty("password"));
      accountDao = new JdbcAccountDao();
      accountDao.setDataSource(dataSource);
   } catch (Exception e) {
      throw new RuntimeException(e);
      }
   }
}
```

W jednym aspekcie pogorszyliśmy nasz kod: wprowadziliśmy zależności pomiędzy `AccountService` a `BasicDataSource` — relacje, których oczywiście nie chcemy. Mamy również zależność pomiędzy `AccountService` a `JdbcAccountDao` (klasa konkretna), więc

nadal jesteśmy w punkcie wyjścia (patrz rysunek 1.4)! Bardzo łatwo pokazać, że cały graf zależności dla określonego systemu może stać się skomplikowany i mało elastyczny z powodu węzłów, które trudno jest usunąć.



Rysunek 1.4. Teraz JdbcAccountDao posiada oczekiwaną zależność od DataSource, ale AccountService posiada odwołania do dwóch klas konkretnych

Nie oznacza to, że DI było nieudanym eksperymentem. Poszliśmy we właściwym kierunku. Aby ulepszyć nasz mechanizm, musimy zająć się tym, *co* realizuje wstrzykiwanie.

1.2.3. Odwrócenie kontroli

Dobrym pomysłem jest usunięcie DI z kodu klienckiego i przekazanie go do Spring. W takim scenariuszu kod klienta nie zawiera żądania ani wyszukiwania klasy `AccountService`. Zamiast tego implementacja `AccountService` jest w sposób przezroczysty wstrzykiwana do kodu klienta w momencie jego inicjalizacji. W poniższym fragmencie kodu przedstawiona jest klasa `AccountService` z zależnością od interfejsu `AccountDao`:

```
//Projekt źródłowy: sip01, gałąź: 03 (projekt Maven)
package com.springinpractice.ch01.service;
import com.springinpractice.ch01.dao.AccountDao;

public class AccountService {
   private AccountDao accountDao;

   public AccountService() {}

   public void setAccountDao(AccountDao accountDao) {
      this.accountDao = accountDao;
   }
}
```

W jaki sposób zdefiniować łańcuch zależności? W przypadku Spring jedną z możliwości jest podejście deklaratywne z wykorzystaniem XML, jak pokazano na poniższym listingu.

Listing 1.1. Plik konfiguracyjny Spring definiujący relacje między obiektami

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Projekt źródłowy: sip01, gałąź: 03 (projekt Maven) -->

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

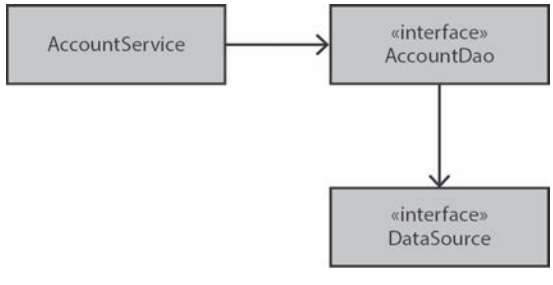
```
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url"
      value="jdbc:mysql://localhost:3306/springbook?autoReconnect=true"/>
      <property name="username" value="someusername"/>
      <property name="password" value="somepassword"/>
   </bean>

   <bean id="accountDao"
      class="com.springinpractice.ch01.dao.jdbc.JdbcAccountDao">
      <property name="dataSource" ref="dataSource"/>
   </bean>

   <bean id="accountService"
      class="com.springinpractice.ch01.service.AccountService">
      <property name="accountDao" ref="accountDao"/>
   </bean>
</beans>
```

❶ Obiekt DataSource wraz ze skonfigurowanymi parametrami

❷ Obiekt DataSource wstrzyknięty do JdbcAccountDao

❸ Obiekt JdbcAccountDao wstrzyknięty do AccountService

Jeżeli zaczynasz pracę z biblioteką Spring, taka konfiguracja może być dla Ciebie nieznana, ale jej znaczenie powinno być wystarczająco jasne. We fragmencie ❶ deklarujemy `DataSource` i ustawiamy jego właściwości za pomocą parametrów konfiguracyjnych. We fragmencie ❷ deklarujemy `JdbcAccountDao` i wstrzykujemy go do `DataSource`. Podobnie we fragmencie ❸ wstrzykujemy `JdbcAccountDao` do `AccountSer`
`↪vice`. Ostatecznym wynikiem jest uzyskanie usługi, która obsługuje cały łańcuch zależności, ale konfiguracja jest dla niej zupełnie przezroczysta. Nowe, wyczyszczone relacje są pokazane na diagramie klas na rysunku 1.5.



Rysunek 1.5. Teraz zależności bazują na interfejsach, a klasy konkretne są konfigurowane w sposób przezroczysty przez Spring

Jak widzisz, struktura klas ma teraz bardziej warstwową budowę. Zauważ, że w porównaniu z rysunkiem 1.4 warstwa górna jest zależna wyłącznie od poniższych warstw, a wszystkie zależności są definiowane za pomocą interfejsów. Upraszcza to graf zależności i ułatwia wymianę poszczególnych węzłów. W następnym podrozdziale użyjemy przykładu ilustrującego, w jaki sposób można użyć biblioteki Spring do zarządzania i wstrzykiwania konkretnych implementacji naszych zależności.

1.3. Przykład prostej konfiguracji ziarna

Gdy mamy już za sobą odpowiedzi na pytania, *co* i *dlaczego*, spróbujmy użyć DI ze Spring Framework w małej przykładowej aplikacji. Aplikacja nie będzie robiła nic poważnego — pozwoli nam tylko nauczyć się podstaw zapisywania zależności i zarządzania nimi z użyciem Spring. Zbudujemy obiekt domeny, DAO odczytujące jego zawartość z pliku CSV, usługę, a następnie połączymy wszystkie te elementy ze sobą.

Kod ten pozwala na uzyskanie listy kont niesolidnych dłużników w hipotetycznej firmie usługowej. Konto niesolidnego dłużnika jest definiowane jako zawierające niezapłacony rachunek, na które nie wpłynęły pieniądze od co najmniej 30 dni. Usługa będzie odpowiedzialna za sprawdzenie, czy konto należy do niesolidnego dłużnika, ale listę potencjalnych kont uzyskamy z DAO. Przed utworzeniem DAO zdefiniujemy obiekty domeny, na których będziemy pracować.

1.3.1. Tworzenie obiektów domeny

Przykładowy obiekt domeny `Account`, przedstawiony na poniższym listingu, posiada tylko te pola i metody, które są potrzebne do zademonstrowania mechanizmu DI w pozostałych częściach przykładowej aplikacji.

Listing 1.2. Proste ziarno konta — Account.java

```
// Projekt źródłowy: sip01, gałąź: 04 (projekt Maven)
package com.springinpractice.ch01.model;

import java.math.BigDecimal;
import java.util.Date;

public class Account {

   private String accountNo;
   private BigDecimal balance;
   private Date lastPaidOn;

   public Account(String accountNo, BigDecimal balance, Date lastPaidOn) {
      this.accountNo = accountNo;
      this.balance = balance;
      this.lastPaidOn = lastPaidOn;
   }
   public String getAccountNo() {
      return accountNo;
   }

   public BigDecimal getBalance() {
      return balance;
   }

   public Date getLastPaidOn() {
      return lastPaidOn;
   }
}
```

W rzeczywistej aplikacji konta nie pojawiają się z powietrza. Mamy bazy danych, pliki i inne systemy, które przechowują informacje, i możemy je odczytywać. W tym przykładzie utworzymy obiekt DAO analizujący plik z wartościami rozdzielanymi przecinkami (CSV), zawierający następujące dane:

```
100,0,09012008
200,100,08012008
300,-100,09012008
```

W pliku CSV o nazwie *accounts.csv* pierwsze pole zawiera numer konta, drugie jego stan (dodatni lub ujemny), a trzecie datę ostatniego wpływu na konto w formacie MMDDRRRR. Jak wspominaliśmy w podrozdziale 1.2, zależności wykorzystujące interfejsy zapewniają elastyczność, tworząc uniwersalne i modyfikowalne implementacje. Zanim utworzymy DAO odpowiedzialne za wczytywanie plików, utwórzmy interfejs dla niego.

1.3.2. Tworzenie interfejsu DAO konta oraz jego implementacji

Poniższy interfejs posiada tylko jedną operację odczytu pozwalającą na pobranie wszystkich kont z magazynu zaplecza, z którego określona implementacja może skorzystać. Nie zamieszczamy tu pozostałych operacji CRUD, ponieważ w tym przykładzie ich nie potrzebujemy.

```
// Projekt źródłowy: sip01, gałąź: 04 (projekt Maven)
package com.springinpractice.ch01.dao;

import java.util.List;

import com.springinpractice.ch01.model.Account;

public interface AccountDao {
   List<Account> findAll() throws Exception;
}
```

Teraz utworzymy konkretną implementację `AccountDao`, która odczytuje obiekty `Account` z pliku CSV. Zakładamy, że nazwa pliku CSV może się z czasem zmienić, więc nie wpisujemy jej na sztywno. Uzasadnione jest przeniesienie tej danej do pliku właściwości i użycie klasy `java.util.Properties` do jej odczytania, ale zamiast to robić, skonfigurujemy ją za pomocą Spring. Kod jest następujący.

Listing 1.3. Odczyt kont z pliku CSV — CsvAccountDao.java

```
// Projekt źródłowy: sip01, gałąź: 04 (projekt Maven)
package com.springinpractice.ch01.dao.csv;

import java.io.BufferedReader;
import java.io.FileReader;
import java.math.BigDecimal;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```

```
import org.springframework.core.io.Resource;
import com.springinpractice.ch01.dao.AccountDao;
import com.springinpractice.ch01.model.Account;

public class CsvAccountDao implements AccountDao {

   private Resource csvResource;

   public void setCsvResource(Resource csvFile) {
   this.csvResource = csvFile;
   }

   public List<Account> findAll() throws Exception {
      List<Account> results = new ArrayList<Account>();

      DateFormat fmt = new SimpleDateFormat("MMddyyyy");
      BufferedReader br = new BufferedReader(

      new FileReader(csvResource.getFile()));
      String line;
      while ((line = br.readLine()) != null) {
         String[] fields = line.split(",");

         String accountNo = fields[0];
         BigDecimal balance = new BigDecimal(fields[1]);
         Date lastPaidOn = fmt.parse(fields[2]);
         Account account =
            new Account(accountNo, balance, lastPaidOn);
         results.add(account);
      }
      br.close();
      return results;
   }
}
```

1 Plik CSV konfigurowany przez Spring

2 Wymagany do wstrzykiwania za pomocą metody ustawiającej

3 Implementuje interfejs AccountDao

W klasie `CsvAccountDao` zadeklarowane jest pole `csvFile` wypełniane przez Spring ❶. Aby Spring mógł je dla nas wypełnić, wymagane jest zdefiniowanie publicznej metody ustawiającej ❷. Metoda *findAll()* z punktu ❸ przegląda plik wiersz po wierszu, dzieląc tekst na elementy i zapisując je w obiektach domeny `Account`, które są następnie zbierane w `ArrayList` i zwracane do wywołującego kodu[1].

Brak kontroli błędów i asercji

Aby uprościć przykłady, pominąłem kontrolę błędów, która musi znaleźć się w kodzie produkcyjnym. Na przykład w poprzednim przykładzie kod produkcyjny powinien zawierać sprawdzenie, czy wartość `String` przekazana do `csvFile` nie jest pusta oraz czy plik wskazywany przez nią faktycznie istnieje.

[1] Użyta tu metoda konfiguracji jest nazywana *wstrzykiwaniem za pomocą settera*. Spring zawiera również inne metody konfiguracji ziaren oraz dołączania zależności, w tym *wstrzykiwanie z użyciem konstruktora* oraz *wstrzykiwanie metod fabryki*. W książce tej używamy najczęściej tej metody, ponieważ jest to najpopularniejsze podejście. Wady i zalety strategii wstrzykiwania są opisane w punkcie „Constructor versus Setter Injection" artykułu Martina Fowlera „Inversion of Control Containers and the Dependency Injection pattern" ze stycznia 2004 roku, dostępnego pod adresem *http://mng.bz/xvk5*.

1.3.3. Konfigurowanie CsvAccountDao za pomocą Spring

Mamy już wystarczająco dużo kodu, aby skorzystać z możliwości, jakie daje nam Spring. W Spring dostępne jest kilka sposobów konfigurowania obiektów i ich zależności, a dwoma najpopularniejszymi są pliki XML oraz adnotacje. W tym punkcie skorzystamy z XML; podczas przedstawiania w podrozdziale 1.5 konfiguracji na podstawie adnotacji będziemy odwoływać się do zaprezentowanych tu koncepcji. Plik XML zamieszczony na poniższym listingu zawiera znaczniki definiujące i konfigurujące ziarna w Spring.

Listing 1.4. Plik konfiguracyjny Spring — applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Projekt źródłowy: sip01, gałąź: 04 (projekt Maven) -->

<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-3.1.xsd">

   <bean id="accountDao"
   class="com.springinpractice.ch01.dao.csv.CsvAccountDao">
      <property name="csvResource" value="accounts.csv"/>
   </bean>

   <bean id="accountService"
   class="com.springinpractice.ch01.service.AccountService">
      <property name="accountDao" ref="accountDao"/>
   </bean>
</beans>
```

❶ Schemat ziarna Spring

❷ Definicja ziarna CsvAccountDao

❸ Konfiguracja właściwości csvResource

Zgodnie z konwencją programiści zwykle nadają temu plikowi nazwę *application Context.xml*, ale może on mieć dowolną nazwę. W rzeczywistych aplikacjach zwykle sensowny jest podział konfiguracji Spring na wiele plików, szczególnie jeżeli są one duże i zawierają wiele definicji ziaren. W takim przypadku konfiguracja jest dzielona zazwyczaj na części architektoniczne, a nie funkcjonalne. Warto więc tworzyć pliki konfiguracyjne dla DAO, usług, serwletów i bezpieczeństwa. W dalszych częściach tej książki pokażemy przykłady tego podejścia. Ponieważ teraz chcemy zachować prostotę, wystarczy nam jeden plik.

Spring jest dostarczany z wieloma schematami konfiguracji różnych elementów funkcyjnych, takich jak AOP czy zarządzanie transakcjami. Na razie zadeklarujemy schemat `beans` w punkcie ❶. Jest on najbardziej podstawowym schematem, ponieważ funkcje dostarczane przez pozostałe schematy dają się zazwyczaj wyrazić (choć w bardziej rozbudowanej formie) jako definicje ze schematu `beans`. Schemat `beans` zawiera wszystko, co jest potrzebne do definiowania ziaren każdego rodzaju, ich konfiguracji oraz łączenia ze sobą.

Konfiguracja Twojego pierwszego ziarna znajduje się w punkcie ❷. Element `bean` posiada dwa atrybuty: `id` oraz `class`. Za pomocą atrybutu `id` będziemy obsługiwać zależności pomiędzy ziarnami. W atrybucie `class` umieszczamy w pełni kwalifikowaną

nazwę klasy `CsvAccountDao`. Spring korzysta z refleksji do tworzenia klasy w czasie obsługi żądania z kontenera (lub żądania innej klasy, która od niej zależy). W punkcie ❸ zadeklarowany jest element `property` dla właściwości `csvResource`. Również w tym przypadku Spring korzysta z refleksji do ustawienia wartości elementu zdefiniowanego w atrybucie `value`. Jeżeli analizujesz dołączany do książki przykład kodu, znajdziesz ten plik w katalogu *src/main/resources*. Spring korzysta z modelu programowania JavaBeans do określenia kolejności ustawiania właściwości, dlatego zadeklarowaliśmy metodę ustawiającą w obiekcie domeny `Account`.

1.3.4. Tworzenie usługi wyszukującej konta niesolidnych dłużników

Gdy mamy już obiekty domeny oraz DAO, możemy zbudować usługę, która jest odpowiedzialna za przeglądanie wszystkich kont i wyszukiwanie niesolidnych dłużników. Kod zamieszczony na następnym listingu jest dosyć prosty.

Listing 1.5. AccountService.java — usługa odpowiedzialna za wyszukiwanie niesolidnych dłużników

```
//Projekt źródłowy: sip01, gałąź: 04 (projekt Maven)
package com.springinpractice.ch01.service;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.List;

import com.springinpractice.ch01.dao.AccountDao;
import com.springinpractice.ch01.model.Account;

public class AccountService {
   private AccountDao accountDao;

   public AccountService() {}

   public void setAccountDao(AccountDao accountDao) {
      this.accountDao = accountDao;
   }

   public List<Account> findDeliquentAccounts() throws Exception {
      List<Account> delinquentAccounts = new ArrayList<Account>();
      List<Account> accounts = accountDao.findAll();

      Date thirtyDaysAgo = daysAgo(30);
      for (Account account : accounts) {
         boolean owesMoney = account.getBalance()
            .compareTo(BigDecimal.ZERO) > 0;
         boolean thirtyDaysLate = account.getLastPaidOn()
            .compareTo(thirtyDaysAgo) <= 0;
         if (owesMoney && thirtyDaysLate) {
            delinquentAccounts.add(account);
         }
      }
```

❶ **Obiekt `AccountDao` podstawiany przez Spring**

❷ **Metoda ustawiająca wymagana do wstrzykiwania**

❸ **Wyszukiwanie niesolidnych dłużników**

```
        return delinquentAccounts;
    }

    private static Date daysAgo(int days) {
        GregorianCalendar gc = new GregorianCalendar();
        gc.add(Calendar.DATE, -days);
        return gc.getTime();
    }
}
```

Zależność od `AccountDao` jest deklarowana za pomocą interfejsu w punkcie ❶. Wymagana metoda ustawiająca znajduje się w punkcie ❷. W punkcie ❸ przeglądamy wszystkie obiekty `Account` zwracane przez DAO i sprawdzamy, czy należy do niesolidnego dłużnika. Jeżeli jest to takie konto, możemy je dodać do listy, która następnie jest zwracana przez metodę.

1.3.5. Podłączanie AccountService do CsvAccountDao

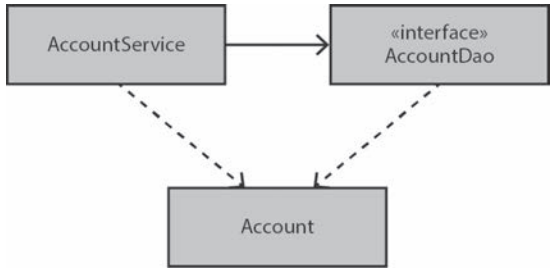
Teraz połączymy `AccountService` z `CsvAccountDao`. W tym celu wystarczy dodać prostą definicję ziarna do pliku konfiguracyjnego *applicationContext.xml*, pokazanego na poniższym listingu.

Listing 1.6. Kompletny plik konfiguracyjny Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Projekt źródłowy: sip01, gałąź: 04 (projekt Maven) -->

<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-3.1.xsd">

    <bean id="accountDao"
    class="com.springinpractice.ch01.dao.csv.CsvAccountDao">
    <property name="csvResource" value="accounts.csv"/>
    </bean>

    <bean id="accountService"                          <-- Definicja ziarna AccountService
    class="com.springinpractice.ch01.service.AccountService">
    <property name="accountDao" ref="accountDao"/>     <-- Wstrzykiwanie AccountDao
    </bean>
</beans>
```

Podobnie jak w przypadku definicji dla klasy `CsvAccountDao`, umieszczonej na listingu 1.4, określamy wartość atrybutów `id` oraz `class`. Ważną różnicą jest tutaj określenie sposobu wstrzykiwania `AccountDao` do usługi. W tym przypadku korzystamy z atrybutu `ref` zamiast `value`. Atrybut `ref` jest używany do wstrzykiwania innych ziaren, które zdefiniowaliśmy w Spring. Atrybut `value` pozwala na wstrzyknięcie prostych wartości oraz wartości właściwości obiektów.

Na rysunku 1.6 pokazany jest diagram klas dla obiektów.



Rysunek 1.6. AccountService posiada połączenie z AccountDao bazujące na interfejsie, a obie te klasy zależą od Account

Teraz pozostało nam napisać aplikację konsolową do uruchomienia naszego kodu. Wśród trzech zdefiniowanych kont tylko jedno — o numerze 200 — należy do niesolidnego dłużnika. Poniższy kod wyświetla numery kont niesolidnych dłużników.

Listing 1.7. Klasa ConsoleApp.java korzystająca z AccountService ze Spring

```
// Projekt źródłowy: sip01, gałąź: 04 (projekt Maven)
package com.springinpractice.ch01;

import java.util.List;

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

import com.springinpractice.ch01.model.Account;
import com.springinpractice.ch01.service.AccountService;

public class ConsoleApp {
   public static void main(String[] args) throws Exception {
      ApplicationContext appCtx =                          <- 1 Kontener wstrzykiwania zależności
         new ClassPathXmlApplicationContext("applicationContext.xml");
      AccountService accountService =
          (AccountService)appCtx.getBean("accountService");   <- Odczytanie AccountService z kontenera (2)
      List<Account> delinquentAccounts = accountService
         .findDeliquentAccounts();
      for (Account a : delinquentAccounts) {
         System.out.println(a.getAccountNo());
      }
   }
}
```

W punkcie ❶ tworzymy obiekt `ClassPathXmlApplicationContext` i przekazujemy do niego lokalizację pliku konfiguracyjnego określoną względem zawartości zmiennej `classpath`. Za pomocą tej klasy możemy uzyskać referencję dowolnego ziarna zdefiniowanego w pliku konfiguracyjnym Spring, podając tylko jego identyfikator określony przy definicji ziarna. Interfejs `ApplicationContext` i jego implementacja jest naszym pomostem do ziaren Spring. W gruncie rzeczy tworzy on złożoną implementację wzorca fabryki. Aby fabryka mogła tworzyć ziarna, muszą mieć one konstruktor bezargumentowy (domyślny, niejawny konstruktor bezargumentowy jest wystarczający). Spring posiada również obsługę tworzenia ziaren przy użyciu konstruktora

z argumentami, którą przedstawimy w następnym podrozdziale. W punkcie ❷ otrzymujemy referencję do obiektu `AccountService` wraz z zależnym obiektem `AccountDao`, który ma skonfigurowaną wartość właściwości `csvFile`.

W tym podrozdziale zbudowałeś prostą aplikację z użyciem Spring Framework. Można zauważyć, że pełna aplikacja z wieloma obiektami DAO, usługami i innymi komponentami skonfigurowanymi i połączonymi przez Spring może być łatwo podzielona na logiczne elementy — w przeciwieństwie do aplikacji pisanej w standardowy sposób.

Teraz, gdy przedstawiliśmy już podstawy, w następnym podrozdziale zajmiemy się dokładniej możliwościami mechanizmów DI dostępnych w bibliotece. Po omówieniu przestrzeni nazw `beans` przedstawimy inne sposoby wstrzykiwania zależności, konfigurowania i wyodrębniania właściwości ziaren, zakres ziaren oraz zagadnienia składniowe, pozwalające na tworzenie czytelniejszych i spójniejszych konfiguracji.

1.4. Łączenie ziaren za pomocą XML

Jak już pokazaliśmy, ziarno Spring jest reprezentowane przez komponent POJO. Ponieważ pozostałe pięć obszarów funkcjonalnych Spring Framework (dostęp do danych i integracja, sieć WWW, AOP, instrumentacja oraz testowanie) bazuje na funkcjach udostępnianych przez kontener podstawowy, to zapoznanie się ze sposobami łączenia ziaren jest niezbędne do zrozumienia Spring Framework i efektywnego korzystania z niego. W tym podrozdziale zapoznamy się z konfigurowaniem biblioteki Spring za pomocą plików XML. W następnych punktach przedstawimy dwie przestrzenie nazw XML, z których będziemy korzystać w tej książce: podstawową przestrzeń nazw `beans` oraz przydatną przestrzeń nazw `p`.

1.4.1. Przegląd przestrzeni nazw beans

Przestrzeń nazw `beans` jest przestrzenią podstawową i obsługuje mechanizm DI; zapewnia elementy do definiowania ziaren oraz relacji pomiędzy nimi. Aby utworzyć plik konfiguracyjny korzystający z przestrzeni nazw `beans`, utwórz dokument XML i umieść w nim odwołanie do schematu:

```
<?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-3.1.xsd">
</beans>
```

Plik taki stanowi pustą konfigurację. Teraz możesz dodać definicję obiektu w wewnętrznym elemencie `bean`:

```
<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-3.1.xsd">
```

```
   <bean id="accountService"
      class="com.springinpractice.ch01.service.AccountService"/>
</beans>
```

Zdefiniowaliśmy tu `AccountService` przez utworzenie elementu `bean` z atrybutami `id` oraz `class`. Wartość atrybutu `id` pozwala nam w prosty sposób odwoływać się do ziarna. Jak pisaliśmy w podrozdziale 1.3, Spring korzysta z refleksji przy tworzeniu nowych instancji klasy zwracanych za pośrednictwem interfejsu `Application ↪Context`.

ŁĄCZENIE ZIAREN

Obiekty `AccountService` można tworzyć za pomocą słowa kluczowego `new`, ale w przypadku warstwy usług rzadko jest to tak proste. Często zależą one od obiektów DAO, obsługi poczty, SOAP i innych. Możliwe jest tworzenie każdej z tych zależności programowo w konstruktorze `AccountService` (lub za pomocą statycznych metod inicjujących), ale prowadzi to do powstania sztywnych zależności, a w przypadku konieczności wymiany jednej z klas — do kaskadowych zmian. Można również tworzyć zależności na zewnątrz i dodawać je do `AccountService` za pomocą metod ustawiających lub argumentów konstruktora. W ten sposób można wyeliminować sztywne zależności wewnętrzne (o ile będą zadeklarowane poprzez interfejsy), ale zachodzi konieczność powielania kodu inicjującego.

Korzystając ze Spring, można utworzyć obiekt DAO i podłączyć go do `Account ↪Service` w następujący sposób:

```
<bean id="accountDao"
   class="com.springinpractice.ch01.dao.jdbc.JdbcAccountDao"/>

<bean id="accountService"
   class="com.springinpractice.ch01.service.AccountService">
   <property name="accountDao" ref="accountDao"/>
</bean>
```

Deklarując jedynie element `property` dla zależności, wstrzyknęliśmy obiekt `JdbcAc ↪countDao` do `AccountService`. Element `property` posiada atrybut `name`, który odpowiada nazwie właściwości do ustawienia, jak również atrybut `ref`, do którego należy przypisać wartość `id` ziarna do wstrzyknięcia.

Inne usługi i klasy mogą zależeć od ziarna `accountDao`. Jeżeli jego implementacja ulegnie zmianie, na przykład z JDBC na Hibernate, to wystarczy zaktualizować atrybut `class` w konfiguracji, bez konieczności sprawdzania wszystkich zależnych klas i ręcznej wymiany zależności. Spring wspiera ten typ łączenia, zapewniając jego pracę nawet ze złożonymi, wielopoziomowymi grafami obiektów.

WSTRZYKIWANIE Z UŻYCIEM KONSTRUKTORA I METOD USTAWIAJĄCYCH

Podobnie jak `AccountService` potrzebuje bezargumentowego konstruktora, Spring musi posiadać metodę ustawiającą `setAccountDao()`, odpowiadającą właściwości `accountDao`, aby mógł użyć refleksji do utworzenia obiektu:

```
public class AccountService {
   private AccountDao accountDao;

   public void setAccountDao(AccountDao accountDao) {
      this.accountDao = accountDao;
   }
   //...
}
```

Taki typ wstrzykiwania jest nazywany *wstrzykiwaniem z użyciem metody ustawiającej*. Jak wcześniej wspominaliśmy, Spring pozwala również na tworzenie obiektów za pomocą konstruktorów z argumentami. Można więc pominąć metodę ustawiającą i zadeklarować odpowiedni konstruktor:

```
public AccountService(AccountDao accountDao) {
   this.accountDao = accountDao;
}
```

Zależności takie są rozwiązywane w następujący sposób:

```
<bean id="accountService"
   class="com.springinpractice.ch01.service.AccountService">
   <constructor-arg ref="accountDao"/>
</bean>
```

Jak możesz się domyślać, jest to nazywane *wstrzykiwaniem z użyciem konstruktora*. W książce tej w większości przypadków będziemy korzystać ze wstrzykiwania z użyciem metody ustawiającej.

KONFIGUROWANIE PROSTYCH WŁAŚCIWOŚCI ZIARNA

Konfigurowana przez nas klasa `JdbcAccountDao` może potrzebować dodatkowej inicjalizacji.

W tym przykładzie zarejestrujemy sterownik JDBC oraz skonfigurujemy dane połączenia. W przedstawionym poniżej kodzie obiekt `BasicDataSource` z prostymi właściwościami jest konfigurowany za pomocą elementów `property`, a nie jak wcześniej `ref`:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
   destroy-method="close">
   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
   <property name="url"
   value="jdbc:mysql://localhost:3306/springbook?autoReconnect=true"/>
   <property name="username" value="someusername"/>
   <property name="password" value="somepassword"/>
</bean>

<bean id="accountDao"
   class="com.springinpractice.ch01.dao.jdbc.JdbcAccountDao">
   <property name="dataSource" ref="dataSource"/>
</bean>
```

Oczywiście w systemie produkcyjnym najprawdopodobniej wykorzystywany będzie obiekt `javax.sql.DataSource` obsługujący pule połączeń (w rozdziale 2. pokażemy, jak z niego skorzystać). W tym przykładzie ustawiamy proste właściwości znakowe

w `BasicDataSource`. A gdyby to były właściwości numeryczne lub typu `java.util.Date`? Spring podejmie w takiej sytuacji próbę konwersji ciągu znaków z atrybutu `value` na odpowiedni typ, korzystając z implementacji `java.beans.PropertyEditor`. Spring ma zdefiniowane kilka takich implementacji, a w razie potrzeby pozwala na definiowanie własnych[2].

WYODRĘBNIANIE WARTOŚCI PROSTYCH WŁAŚCIWOŚCI ZA POMOCĄ KLASY PROPERTYPLACEHOLDERCONFIGURER

Konfigurowanie właściwości `JdbcAccountDao` na zewnątrz klasy jest korzystne. Zmiana parametrów konfiguracji nie wymaga ponownej kompilacji kodu — wystarczy zmienić XML. Dodatkowo centralizacja konfiguracji pozwala na jej modyfikowanie w jednym miejscu (lub kilku logicznie powiązanych miejscach, jeżeli korzystasz z wielu plików konfiguracyjnych). Jednak w większości środowisk nie podłączamy się bezpośrednio do produkcyjnego serwera bazy danych i nie wykonujemy na nim nieprzetestowanego kodu. Mamy jedno środowisko testowe lub większą ich liczbę. Na szczęście Spring pozwala obsłużyć takie scenariusze za pomocą klasy API: `PropertyPlaceholderConfigurer`.

Aby skorzystać z `PropertyPlaceholderConfigurer`, należy utworzyć plik właściwości (nazwijmy go *springbook.properties*):

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql://localhost:3306/springbook?autoReconnect=true
dataSource.username=root
dataSource.password=secret
```

Następnie definiujemy w konfiguracji Spring ziarno `PropertyPlaceholderConfigurer`. Na koniec przy definiowaniu ziaren korzystamy z konstrukcji `${}` do oznaczania wartości właściwości, dzięki czemu kontener może je rozpoznać w czasie działania aplikacji.

Listing 1.8. Deklarowanie PropertyPlaceholderConfigurer do podstawiania właściwości

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Projekt źródłowy: sip01, gałąź: 05 (projekt Maven) -->
<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-3.1.xsd">
   <bean class="org.springframework.beans.factory.config.
   ↪PropertyPlaceholderConfigurer">          <-- 1 Deklaracja PropertyPlaceholderConfigurer
      <property name="location" value="springbook.properties"/>
   </bean>                                   <-- Lokalizacja korzysta z systemu zmiennych środowiska Java 2

   <bean id="accountService"
      class="com.springinpractice.ch01.service.AccountService">
      <property name="accountDao" ref="accountDao"/>
```

[2] Więcej informacji na temat dostępnych implementacji `PropertyEditor` można znaleźć pod adresem *http://mng.bz/7CO9*.

```
    </bean>

    <bean id="accountDao"
        class="com.springinpractice.ch01.dao.jdbc.JdbcAccountDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
            <property name="driverClassName"
                value="${dataSource.driverClassName}"/>
            <property name="url" value="${dataSource.url}"/>
            <property name="username" value="${dataSource.username}"/>
            <property name="password" value="${dataSource.password}"/>
    </bean>
</beans>
```

❸ **Symbole podstawiane w czasie działania aplikacji**

Definicja `PropertyPlaceholderConfigurer` ❶ nie zawiera atrybutu `id`. Nie jest to potrzebne, ponieważ kontener Spring automatycznie wykrywa jego istnienie i włącza dla nas odpowiednią funkcję. We właściwości `location` ❷ klasy `PropertyPlaceholder`
`↪Configurer` określamy plik umieszczony w głównej gałęzi `classpath`. Jeżeli analizujesz towarzyszący książce przykład kodu, plik *springbook.properties* znajdziesz w katalogu *src/main/resources*, który jest dołączany do zmiennej `classpath`. Możesz również zmienić konfigurację w punkcie ❷ na

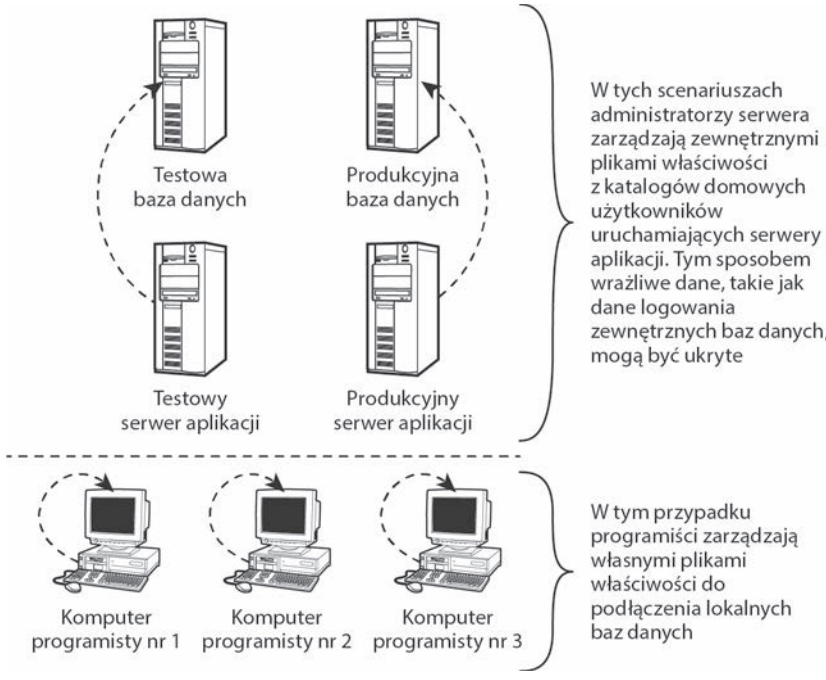
```
<property name="location" value="file:${user.home}/springbook.properties"/>
```

W powyższej konfiguracji wskazujemy na plik umieszczony w katalogu domowym użytkownika uruchamiającego aplikację, definiowanym poprzez systemową zmienną środowiskową Java (jeżeli symbol nie zostanie znaleziony w pliku właściwości, Spring próbuje podstawiać systemowe zmienne środowiskowe). Jest to wygodne, ponieważ administrator serwera może przechowywać w tych plikach wrażliwe informacje, do których nie muszą mieć dostęp programiści. Pozwala to również na jednoczesną pracę nad projektem przez wielu programistów, korzystających z własnych baz danych. Na rysunku 1.7 przedstawiony jest sposób działania takiego scenariusza. Wpisane w punkcie ❸ wartości są zmieniane na właściwe dla danego środowiska.

Po przeczytaniu tego podrozdziału książki powinieneś już wiedzieć, w jaki sposób można konfigurować ziarna Spring. Choć zagadnienie to jest znacznie obszerniejsze, poświęcimy teraz nieco miejsca na przedstawienie ważnego tematu zakresu ziaren.

1.4.2. Zakres ziaren

Definiując ziarno za pomocą Spring, można określić sposób tworzenia obiektów otrzymywanych z kontenera i zarządzania nimi. Jest to nazywane *zakresem ziaren*. Możemy wyróżnić pięć takich zakresów: `singleton`, `prototype`, `request`, `session` oraz `globalSession`.



Rysunek 1.7. Użycie zewnętrznych plików właściwości do zarządzania konfiguracją w poszczególnych środowiskach

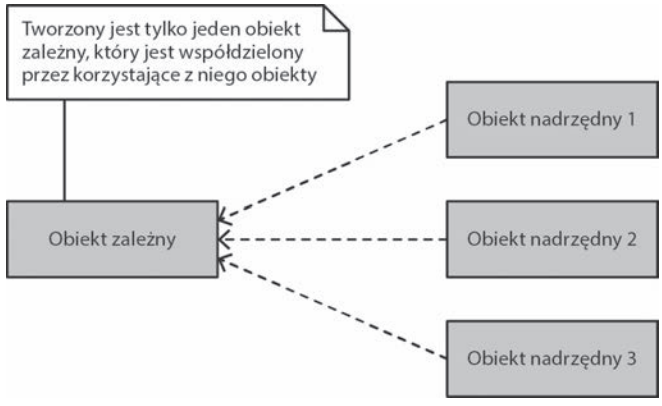
Zakres jest konfigurowany za pomocą atrybutu `scope`:

```
<bean id="accountDao"
   class="com.springinpractice.ch01.dao.jdbc.JdbcAccountDao"
   scope="singleton|prototype|request|session|globalSession"/>
```

ZAKRES SINGLETON

Zakres `singleton` jest domyślnym zakresem ziaren deklarowanych w Spring. Nazwa tego zakresu oznacza coś innego niż klasę Java implementującą wzorzec singleton. Zadeklarowanie ziarna o zakresie `singleton` w Spring powoduje, że w kontenerze będzie istniała tylko jedna instancja, natomiast klasa implementująca wzorzec projektowy singleton będzie miała jedną instancję w ramach danego obiektu ładującego klasy.

Gdy zażądamy ziarna o zakresie `singleton` z kontenera, gdy nie był on wcześniej utworzony, Spring utworzy obiekt i zapisze go w buforze (patrz rysunek 1.8), a w przeciwnym razie zwróci utworzoną wcześniej instancję zapisaną w buforze. Z tego powodu ziarna `singleton` zwykle nie posiadają stanu, ponieważ są współdzielone przez wiele wątków (na przykład w środowiskach z serwletami). Na przykład usługi o zakresie `singleton` często mają powiązania z obiektami DAO o zakresie `singleton`, a z kolei te obiekty mogą zawierać referencje do obiektów `SessionFactory` Hibernate, które można bezpiecznie wykorzystywać w aplikacjach wielowątkowych.



Rysunek 1.8. Ziarna o zakresie singleton są współdzielone przez nadrzędne instancje klas

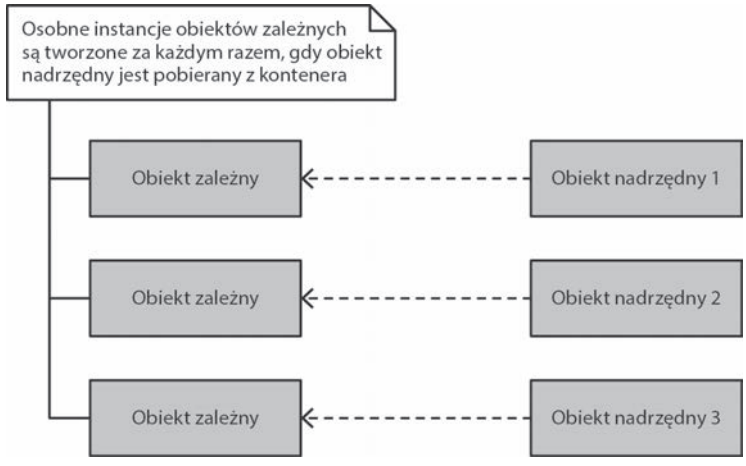
Dopóki zasoby są bezpieczne dla wątków — co oznacza, że są synchronizowane, niezmienne, nie mają stanu ani pól, w których nie są spełnione powyższe kryteria — można je bezpiecznie deklarować w zasięgu `singleton`, aby uniknąć narzutu na ich tworzenie.

ZAKRES PROTOTYPE

Ziarna o zakresie `prototype` są tworzone za każdym razem, gdy zostanie wywołana metoda `getBean()` z obiektu `ApplicationContext`, oraz za każdym razem, gdy są wstrzykiwane do innych ziaren. Przeanalizujmy ten drugi przypadek. Za pomocą poniższego kodu wstrzykujemy ziarno o zakresie `prototype` do innego ziarna o domyślnym zakresie `singleton` (zwróć uwagę, że zwykle nie deklarujemy DAO z użyciem zakresu `prototype`, ponieważ chcemy zapewnić możliwość ich bezpiecznego stosowania w wątkach — korzystamy tu z DAO dla zachowania spójności z poprzednim przykładem).

```
<bean id="accountDao"
   class="com.springinpractice.ch01.dao.jdbc.JdbcAccountDao"
   scope="prototype"/>

<bean id="accountService"
   class="com.springinpractice.ch01.service.AccountService">
   <property name="accountDao" ref="accountDao"/>
</bean>
```

W tym scenariuszu obiekt `AccountService` jest tworzony tylko raz i jest zapisywany w buforze. Tym razem obiekt `JdbcAccountDao` jest tworzony i wstrzykiwany, ale nigdy nie jest buforowany. Kolejne żądania obiektu `AccountService`, zarówno poprzez `getBean()` z `ApplicationContext`, jak i poprzez wstrzykiwanie, spowodują utworzenie nowej instancji `AccountService` wraz z jej odwołaniem do `JdbcAccountDao`. Ziarno `singleton` odwołujące się do `prototype` powoduje, że ziarno `prototype` ma efektywnie zakres `singleton`. Jeżeli jednak jednocześnie wstrzykniemy `JdbcAccountDao` do innego ziarna o zakresie `singleton`, otrzyma ono referencję do osobnej instancji. Zakres `prototype` jest przedstawiony na rysunku 1.9.



Rysunek 1.9. Ziarna o zakresie prototype są tworzone za każdym razem, gdy ich klasy nadrzędne są pobierane z kontenera. Jeżeli jednak klasa nadrzędna ma zakres singleton, kolejne jej pobrania spowodują zwrócenie buforowanych obiektów, w których zależności o zakresie prototype są również buforowane (i nie są ponownie tworzone)

Ziarna o zakresie `prototype` mają inny cykl życia niż ziarna o zakresie `singleton`. Spring może zarządzać pełnym cyklem życia, razem z tworzeniem i usuwaniem ziaren o zakresie `singleton`, ale dotyczy to (konkretyzacja, konfiguracja i dekoracja za pośrednictwem dynamicznych pośredników) jedynie ziaren o zakresie `prototype`. Do kodu klienta należą czyszczenie i zwalnianie zasobów czy inne operacje zarządzania cyklem życia ziarna. Powoduje to, że ziarna o zakresie `prototype` są podobne do obiektów tworzonych za pomocą słowa kluczowego `new`, choć mogą pojawić się trudności w podstawianiu jednych do drugich — z powodu skomplikowanej inicjalizacji ziaren ze stanem, których zarządzanie jest znacznie uproszczone w Spring.

ZAKRES REQUEST, SESSION I GLOBALSESSION

Trzy ostatnie zakresy — `request`, `session` i `globalSession` — są przydatne wyłącznie w kontekście aplikacji WWW. Nie ma znaczenia to, jakiej biblioteki WWW używasz — działają one identycznie w każdej z nich.

Ziarna o zakresie `request` są tworzone za każdym razem, gdy do serwletu, do którego jest ono wstrzyknięte, trafia żądanie HTTP. Podobnie jak w przypadku zmiennych o zakresie `request` w serwletach, ziarna te są bezpieczne w użyciu, ponieważ specyfikacja serwletów narzuca tylko jeden wątek na żądanie HTTP.

Ziarna o zakresie `session` są ograniczone do zakresu standardowych zmiennych o zakresie `session`. Są one, podobnie jak ziarna `request`, bezpieczne w użyciu i modyfikacji, ponieważ są ograniczone do jednego wątku jednocześnie, związanego z klientem wykonującym żądanie.

Ziarna o zakresie `globalSession` nadają się do wykorzystania wyłącznie w aplikacjach portletowych. Podobnie jak ziarna o zakresie `session`, istnieją one przez całą sesję, ale są współdzielone przez wszystkie portlety w aplikacji WWW, natomiast ziarna `session` są tworzone i zarządzane dla każdego portletu osobno.

Istnieje kilka wymagań dotyczących użycia ziaren o tych zakresach. Jeżeli korzystamy z innej biblioteki niż Spring Web MVC, konieczne jest zarejestrowanie klasy `RequestContextListener` w deskryptorze instalacji serwletu (*web.xml*):

```
<web-app>
   ...
   <listener>
      <listener-class>
         org.springframework.web.context.request.RequestContextListener
      </listener-class>
   </listener>
   ...
</web-app>
```

Możesz użyć klasy `RequestContextListener` w kontenerze serwletów implementujących specyfikację w wersji 2.4 lub nowszej. W przypadku kontenera 2.3 dostępna jest osobna implementacja `RequestContextFilter`. Te obiekty nasłuchu i filtry pozwalają na integrację pomiędzy Spring a używanym kontenerem serwletów. Mówiąc najprościej, Spring przechwytuje żądania i dekoruje je za pomocą funkcji zakresu.

Jeżeli pobierasz ziarna o tych zakresach z `ApplicationContext`, to implementacja `ApplicationContext` musi obsługiwać żądania WWW, tak jak na przykład `XmlWebAp` ➥`plicationContext`. W przeciwnym razie otrzymasz wyjątek `IllegalStateException` z komunikatem o nieznanym zakresie.

Poświęciliśmy sporo miejsca na zapoznanie się z przestrzenią nazw `beans`. Jak wcześniej wspomnieliśmy, rozdział ten nie wyczerpuje tematu i jeżeli zdecydujesz się poznać to zagadnienie dokładniej, to masz sporo do nauki. Teraz skierujemy naszą uwagę na przestrzeń nazw `p`, której zadaniem jest uproszczenie konfiguracji XML w stosunku do tej, którą możemy zbudować wyłącznie z użyciem przestrzeni nazw `beans`.

1.4.3. Przestrzeń nazw p

Przestrzeń nazw `p` jest rozszerzeniem przestrzeni `beans` o alternatywną składnię deklaracji właściwości. Zamiast konfigurować właściwości jako elementy XML, tak jak pokazywaliśmy to na przykładzie `JdbcAccountDao`, można zadeklarować je jako atrybuty elementu `bean`. Jak pokazano na poniższym listingu, właściwości są definiowane jako atrybuty w formacie `p:[`*`nazwaWłaściwości`*`]="`*`wartość`*`"`.

Listing 1.9. Deklarowanie PropertyPlaceholderConfigurer do podstawiania właściwości

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Projekt źródłowy: sip01, gałąź: 06 (projekt Maven) -->

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

   ...
```

1 **Deklaracja przestrzeni nazw** `p`

```
    <bean id="dataSource"
       class="org.apache.commons.dbcp.BasicDataSource"
       destroy-method="close"
          p:driverClassName="${dataSource.driverClassName}"
       p:url="${dataSource.url}"
       p:username="${dataSource.username}"
       p:password="${dataSource.password}"/>

    ...
</beans>
```

❷ **Deklaracja właściwości jako atrybutów elementu** `bean`

Przestrzeń nazw `p` zapewnia nam udogodnienia składniowe oraz pozwala na tworzenie spójniejszej konfiguracji. W punkcie ❶ deklarujemy przestrzeń nazw, ale nie należy zwracać uwagi na podaną lokalizację schematu. Aby dowiedzieć się, dlaczego, spójrzmy na użycie tej przestrzeni w punkcie ❷. Nie ma tu zdefiniowanych żadnych elementów ani atrybutów. Przestrzeń nazw `p` nie jest zdefiniowana w pliku XSD, jak przestrzeń `beans`. Jej funkcje są udostępniane przez Spring. Oprócz tego, że można definiować proste właściwości, można również wstrzykiwać pełne ziarna:

```
<bean
    id="accountService"
    class="com.springinpractice.ch01.service.AccountService"
    p:accountDao-ref="accountDao"/>
```

Jedyną różnicą pomiędzy tą deklaracją atrybutu a pozostałymi, używanymi dla prostych właściwości, jest końcówka nazwy atrybutu `-ref`.

W następnym podrozdziale przedstawimy konfigurację bazującą na adnotacjach. Wygoda i czytelność adnotacji powoduje, że faworyzujemy je przy wielu zadaniach przekrojowych (kontrola poprawności, trwałość, transakcje, bezpieczeństwo, usługi sieciowe, odwzorowania żądań itp.), więc są one powszechnie stosowane (w punkcie 1.5.4 przedstawione jest zestawienie zalet konfiguracji korzystających z XML i adnotacji). Kolejna technika jest nadal użyteczna z kilku powodów — nie wszystkie mechanizmy Spring mogą być konfigurowane wyłącznie za pomocą adnotacji i możesz preferować oddzielenie obiektów POJO od zadań konfiguracji.

1.4.4. Przestrzeń nazw c

W Spring 3.1 wprowadzono przestrzeń nazw `c`, mającą dwa zadania. Pierwszym jest ulepszenie istniejącej składni wstrzykiwania z użyciem konstruktora oraz usprawnienie określania argumentów konstruktora do ustawienia. Drugim celem było zapewnienie podobnych rozszerzeń składni XML dla użytkowników wykorzystujących wstrzykiwanie z użyciem konstruktora. Nie trzeba już podawać wielu elementów `construktor-arg` definiujących wstrzykiwanie z użyciem konstruktora.

Zacznijmy od bardzo prostego przykładu. Spójrz na poniższą klasę `Person`.

Zauważ, że konstruktor posiada dwa argumenty typu `String`:

```
public Person(String firstName, String lastName) {
   this.firstName = firstName;
   this.lastName = lastName;
}
```

W wersjach wcześniejszych niż Spring 3.1 ziarno to należało konfigurować w następujący sposób:

```
<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-3.1.xsd">

   <bean id="janAsPerson" class="foo.Person">
      <constructor-arg value="Jan"/>
      <constructor-arg value="Kowalski"/>
   </bean>

</beans>
```

W przypadku Spring 3.1 mamy dwie opcje. Są one przedstawione w poniższej konfiguracji:

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:c="http://www.springframework.org/schema/c"
   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-3.1.xsd">

   <bean id="janAsPerson" class="foo.Person"
      c:firstName="Jan"
      c:lastName="Kowalski"/>

   <bean id="wojtekAsPerson" class="foo.Person"
      c:_0="Wojciech"
      c:_1="Wojciechowski"/>
</beans>
```

❶ **Deklaracja przestrzeni nazw** `c`

❷ **Deklaracja argumentów konstruktora z użyciem nazw zmiennych**

❸ **Deklaracja argumentów konstruktora z użyciem kolejności zmiennych**

Podobnie jak przestrzeń nazw `p`, przestrzeń `c` zapewnia nam udogodnienia składniowe oraz pozwala na tworzenie spójniejszej i czytelniejszej konfiguracji. W przykładowej konfiguracji zadeklarowaliśmy wymaganą przestrzeń nazw w punkcie ❶. Ponownie zwróć uwagę, że nie podajemy lokalizacji schematu. Podobnie jak przestrzeń nazw `p`, przestrzeń `c` nie jest zdefiniowana w pliku XSD. Jej funkcje są udostępniane przez Spring.

W punkcie ❷ zademonstrowana jest pierwsza z funkcji nowej składni. Dzięki wykorzystaniu wyrażenia `c:[nazwaZmiennej]="wartość"` jasne jest, która wartość jest kojarzona z którym argumentem konstruktora. Możliwość kojarzenia nazw zapisanych w XML z nazwami argumentów konstruktora jest uzależniona od dostępności kodu skompilowanego z danymi debugera. Jeżeli korzystasz z biblioteki zewnętrznej, która pozostaje poza Twoją kontrolą, możesz użyć formatu `c:_[indeks ➥Zmiennej]="wartość"`, pokazanego w punkcie ❸. W tym przypadku zamiast korzystać z nazw zmiennych zapisanych w kodzie bajtowym, możemy dzięki tej konfiguracji podać numer kolejnego argumentu konstruktora. Podobnie jak w przypadku przestrzeni nazw `p`, możesz dodać końcówkę `-ref`, aby odwoływać się do innych ziaren.

Do tej pory w rozdziale tym przedstawiliśmy dokładniej sposoby łączenia ziaren z użyciem kodu XML. Opisaliśmy przestrzeń nazw `beans` oraz metody łączenia ziaren ze sobą. Następnie przeszliśmy do wstrzykiwania z użyciem konstruktora oraz metody ustawiającej i opisaliśmy konfigurowanie i wyodrębnianie właściwości ziaren. Na koniec przedstawiliśmy zakresy ziaren oraz zademonstrowaliśmy sposób użycia przestrzeni nazw `p` oraz `c`. W następnym podrozdziale pokażemy, w jaki sposób korzystać z adnotacji języka Java do definiowania komponentów Spring oraz ich zależności, aby wyeliminować potrzebę używania plików XML.

1.5. Automatyczne łączenie oraz skanowanie komponentów z użyciem adnotacji

Po opisaniu podstawowych informacji na temat łączenia elementów ze sobą przedstawimy konfigurację wykorzystującą adnotacje, która została wprowadzona w Spring 2.0 i rozwinięta w Spring 2.5. Adnotacja `@Autowired` pozwala na łączenie relacji za pomocą typu i może być ona stosowana do konstruktorów i pól bez potrzeby ustawiania ich w konfiguracji XML. Następnie przedstawimy adnotacje stereotypów dostępne w bibliotece: `@Component`, `@Repository`, `@Service` oraz `@Controller`. Dzięki adnotacjom stereotypów oraz `@Autowired` można użyć skanowania komponentów w poszukiwaniu zależności przy minimalnym wykorzystaniu kodu XML. W książce tej często wykorzystujemy skanowanie komponentów i automatyczne łączenie, ponieważ pozwala to na zmniejszenie kodu (choć staje się on wtedy mniej jednoznaczny)[3].

1.5.1. Adnotacja @Autowired

We wcześniejszym podrozdziale wstrzykiwaliśmy DAO do usługi za pomocą deklaracji `bean` oraz przestrzeni nazw `p`:

```
<bean id="accountDao"
   class="com.springinpractice.ch01.dao.jdbc.JdbcAccountDao"/>

<bean id="accountService"
   class="com.springinpractice.ch01.service.AccountService"
   p:accountDao-ref="accountDao"/>
```

W przypadku użycia adnotacji `@Autowired` można wyeliminować atrybut `p:account-ref`. Na początek przyjrzymy się adnotacjom, a następnie zmodyfikujemy konfigurację XML, aby je wspierała.

```
import org.springframework.beans.factory.annotation.Autowired;

   ... other imports omitted ...
```

[3] Automatyczne łączenie ma swoje wady i zalety. Pozwala na uproszczenie konfiguracji, ale powoduje, że zależności pomiędzy komponentami są mniej widoczne — niektórzy mogą powiedzieć „magiczne". Im bardziej skomplikowana jest nasza konfiguracja, tym mniej sensowne jest automatyczne łączenie komponentów, choć twierdzenie to nie zawsze się sprawdza.

```
public class AccountService {

   @Autowired
   private AccountDao accountDao;

   ...
}
```

PRZESTRZEŃ NAZW CONTEXT

Adnotacja `@Autowired` nie ma żadnego znaczenia, jeżeli nie zostanie zmieniona konfiguracja XML. Musimy dołączyć do niej schemat `context` i zadeklarować element `annotation-config` w sposób pokazany na następnym listingu. Z ziaren `accountDao` oraz `accountService` usunęliśmy również metody ustawiające.

Listing 1.10. Zmodyfikowana konfiguracja Spring obsługująca adnotacje

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Projekt źródłowy: sip01, gałąź: 07 (projekt Maven) -->

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.1.xsd">

   <context:annotation-config/>

   <bean
      class="org.springframework.beans.factory.config.
      PropertyPlaceholderConfigurer"
      p:location="springbook.properties"/>

   <bean
      id="accountService"
      class="com.springinpractice.ch01.service.AccountService"/>

   <bean
      id="accountDao"
      class="com.springinpractice.ch01.dao.jdbc.JdbcAccountDao"/>

   <bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${dataSource.driverClassName}"
      p:url="${dataSource.url}"
      p:username="${dataSource.username}"
      p:password="${dataSource.password}"/>

</beans>
```

1. **Odwołanie do schematu** context
2. **Aktywuje konfigurację bazującą na adnotacjach**
3. **Jawne wstrzykiwanie nie jest już potrzebne**
4. **Konfiguracja parametrów znakowych jest nadal wykorzystywana**

Elementy przestrzeni nazw `context` umożliwiają obsługę konfiguracji. Jeden z nich pozwala włączyć konfigurację bazującą na adnotacjach. Schemat `context` deklarujemy w punkcie ❶. Następnie włączamy adnotacje poprzez zadeklarowanie elementu `annotation-config` ❷. Jak można zauważyć w punkcie ❸, atrybut `p:accountDao-ref` nie jest już potrzebny. Zwróć uwagę, że wydzieliliśmy konfigurację `DataSource` ❹ do osobnego ziarna i że parametry konfiguracji prostych właściwości znakowych są nadal wykorzystywane.

Jeżeli będziesz tworzyć aplikację korzystającą z tych ziaren i tej konfiguracji, zauważysz, że Spring wstrzyknie `JdbcAccountDao` do `AccountService`.

Gdy pole jest oznaczone jako `@Autowired`, Spring wyszukuje w kontenerze ziarna o pasującym typie. Ponieważ klasa `JdbcAccountDao` implementuje interfejs `AccountDao`, a `AccountService` zawiera deklarację pola `AccountDao`, Spring automatycznie podłączy te właściwości. Co się stanie, gdy zdefiniowane są co najmniej dwa ziarna o pasującym typie?

```
...

<bean id="jdbcAccountDao"
   class="com.springinpractice.ch01.dao.jdbc.JdbcAccountDao"/>
<bean id="hibernateAccountDao"
   class="com.springinpractice.ch01.dao.hibernate.HibernateAccountDao"/
<bean id="accountService"
   class="com.springinpractice.ch01.service.AccountService"/>

...
```

Spring zgłasza w takim przypadku wyjątek `BeanCreationException`:

```
Exception in thread "main"
org.springframework.beans.factory.BeanCreationException: Error creating bean with name
'accountService': Autowiring of fields failed; nested exception is
org.springframework.beans.factory.BeanCreationException: Could not autowire field:
private springinpractice.ch01.dao.AccountDao
springinpractice.ch01.service.AccountService.accountDao; nested exception is
org.springframework.beans.factory.NoSuchBeanDefinitionException: No unique bean of type
[springinpractice.ch01.dao.AccountDao] is defined:
expected single matching bean but found 2: [jdbcAccountDao,hibernateAccountDao]
```

Jeżeli zadeklarujemy pole jako tablicę `AccountDao` lub jakąkolwiek kolekcję bazującą na typie `AccountDao`, Spring umieści w tablicy lub kolekcji wszystkie typy DAO:

```
@Autowired
private AccountDao[] accountDaos;
```

Oczywiście w większości przypadków nie ma sensu posiadanie tablicy obiektów DAO, ale dosyć często mamy wiele implementacji jednego interfejsu i musimy określić tę właściwą. Adnotacja `@Autowired` pozwala na określenie właściwego ziarna:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class AccountService {

   @Autowired
```

```
    @Qualifier("hibernateAccountDao")
    private AccountDao accountDao;
}
```

Kwalifikatorem jest atrybut `id` z deklaracji `bean`. Nie jest to jedyny sposób; w przestrzeni nazw `context` zdefiniowany jest element `qualifier`, który może być użyty w deklaracji `AccountDao`. Element ten jest wygodny, jeżeli musisz pozostawić atrybuty `id` bez zmian (aby na przykład zachować konwencję) albo potrzebujesz innych zasad kwalifikacji. Możesz na przykład odróżniać zwykłe DAO bazujące na Hibernate od wydajnego DAO JDBC z ręcznie dostrajanymi zapytaniami SQL, nad którymi DBA spędził długie godziny.

1.5.2. Adnotacje stereotypów

Do tej pory pokazaliśmy, w jaki sposób łączyć ziarna z użyciem adnotacji `@Auto`
`↪wired`. Jednak tam ten atrybut zapewnia wyłącznie łączenie. Nadal zachodzi konieczność definiowania ziaren, aby kontener miał o nich informacje i mógł je dla nas wstrzykiwać. Jednak przy użyciu adnotacji stereotypów Spring można opisać klasy i włączyć skanowanie komponentów (co pokażemy wkrótce), a Spring automatycznie zaimportuje ziarna do kontenera, więc nie będziemy musieli ich definiować w XML. Obecnie dostępne są cztery adnotacje stereotypów: `@Component`, `@Repository`, `@Service` oraz `@Controller`.

Adnotacja `@Component` oznacza ziarno, więc mechanizm skanowania komponentów może je rozpoznać i udostępnić w kontekście aplikacji. Jeżeli chcemy usunąć deklarację `JdbcAccountDao` z konfiguracji XML, wystarczy umieścić frazę `@Component` przed deklaracją klasy:

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

@Component
public class JdbcAccountDao implements AccountDao {
...
}
```

Teraz możesz całkowicie usunąć z XML deklarację ziarna, a pole `@Autowired acco`
`↪untDao` w `AccountService` zostanie wypełnione automatycznie instancją `JdbcAcco`
`↪untDao`. Choć jest to poprawna i przydatna adnotacja, to istnieje właściwsza, która zapewnia dodatkowe korzyści w przypadku DAO.

Adnotacja `@Repository` jest specjalizacją adnotacji `@Component`. Powoduje nie tylko zaimportowanie DAO do kontenera DI, ale również przekształcenie nieobsłużonych wyjątków na `DataAccessException` ze Spring (również nieobsłużonych).

Adnotacja `@Service` jest również specjalizacją adnotacji komponentu. Obecnie nie oferuje żadnych dodatkowych funkcji w stosunku do adnotacji `@Component`, ale dobrym pomysłem jest używanie `@Service` zamiast `@Component` w klasach warstwy usług, ponieważ lepiej określa ona nasze intencje. Ponadto w przyszłości może to być wykorzystane w narzędziach wspierających oraz dodatkowych funkcjach.

Ostatnia adnotacja, `@Controller`, oznacza klasę jako kontroler Spring Web MVC. Jest to także specjalizacja adnotacji `@Component`, więc ziarna oznaczone za jej pomocą

są również importowane do kontenera DI. Nie mówiliśmy zbyt wiele na temat MVC w Spring, ale w dalszej części książki wzorzec ten będzie intensywnie wykorzystywany. Mówiąc najprościej, gdy dodamy do klasy adnotację `@Controller`, możemy użyć kolejnej adnotacji, `@RequestMapping`, do powiązania adresu URL z metodą instancyjną tej klasy. Dzięki temu możemy poinformować Spring, że chcemy wywołać określoną metodę, gdy zostanie wykonane żądanie odpowiedniego adresu URL. Więcej informacji na ten temat znajduje się w dalszej części książki.

1.5.3. Skanowanie komponentów

Tak samo jak musimy zadeklarować `context:annotation-config` w celu włączenia automatycznego łączenia, musimy również zadeklarować `context:component-scan` do włączenia importu klas oznaczonych za pomocą stereotypów. Jest to pokazane na poniższym listingu.

Listing 1.11. Zmodyfikowana konfiguracja Spring obsługująca skanowanie komponentów

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Projekt źródłowy: sip01, gałąź: 08 (projekt Maven) -->

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">
   <context:component-scan base-package="com.springinpractice.ch01"/>
   …
</beans>
```

Element `context:component-scan` wymaga użycia atrybutu `base-package`, który pozwala określić punkt początkowy do rekurencyjnego przeszukiwania komponentów. W naszym przypadku ma on wartość `com.springinpractice.ch01`. Element `compo➥nent-scan` może ponadto być deklarowany wielokrotnie i wskazywać na wiele pakietów. Często struktura pakietów odpowiada warstwom aplikacji (konwencję tę stosujemy w przykładach w tej książce), więc możemy mieć pakiety dla DAO, usług i kontrolerów. W takiej sytuacji deklarujemy trzy elementy `component-scan`, których atrybut `package-base` wskazuje na różne pakiety.

Gdy zadeklarowany jest element `component-scan`, nie ma potrzeby deklarowania `context:annotation-config`, ponieważ automatyczne łączenie jest niejawnie włączane po włączeniu skanowania komponentów.

Nie potrzebujemy również deklarować ziaren w konfiguracji. Po włączeniu skanowania komponentów przy starcie aplikacji Spring przegląda wszystkie wskazane przez nas pakiety, wyszukuje klasy oznaczone stereotypami i importuje je do kontenera. Domyślnymi nazwami ziaren są niekwalifikowane nazwy klas zaczynające się

od małej litery[4]. Na przykład `springinpractice.ch01.dao.jdbc.JdbcAccountDao` jest zamieniane na `jdbcAccountDao`. Wynikowy plik konfiguracyjny wygląda następująco:

Listing 1.12. Plik konfiguracyjny Spring korzystający ze skanowania komponentów

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Projekt źródłowy: sip01, gałąź: 08 (projekt Maven) -->

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd">
   <context:component-scan base-package="com.springinpractice.ch01"/>
   <bean
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
      p:location="springbook.properties"/>

   <bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${dataSource.driverClassName}"
      p:url="${dataSource.url}"
      p:username="${dataSource.username}"
      p:password="${dataSource.password}"/>
</beans>
```

Ponieważ włączyliśmy skanowanie komponentów i w `accountService` korzystamy z adnotacji `@Service`, natomiast w `accountDao` z `@Repository`, nie musimy już definiować ich w konfiguracji XML. Używamy skompilowanych klas z zewnętrznej biblioteki (`BasicDataSource`), więc nie jesteśmy w stanie dodać do kodu adnotacji. Dlatego właśnie nadal musimy tutaj definiować ich konfigurację.

1.5.4. XML czy adnotacje? Co jest lepsze?

W książce tej przedstawimy wykorzystywaną powszechnie konfigurację bazującą na adnotacjach. W wielu z naszych przykładów pojawiają się adnotacje Hibernate, JPA oraz Hibernate Validator. Odzwierciedla to kierunek, w którym zmierza rozwój Spring, oraz fakt, że adnotacje są wygodne i łatwe w użyciu w wielu sytuacjach.

To, czy korzystać z XML, czy z adnotacji do tworzenia konfiguracji, jest przedmiotem dyskusji w środowisku. Konfiguracja XML jest scentralizowana w zbiorze plików, a każdy plik jest zazwyczaj dedykowany określonej jednostce lub warstwie

[4] Jeżeli domyślne nazwy ziaren są nieodpowiednie dla naszych celów, możesz zaimplementować interfejs Spring `BeanNameGenerator` i za jego pomocą zmienić strategię nazewnictwa. Następnie, przy deklarowaniu elementu `componentscan`, wykorzystujemy element `name-generator` i wskazujemy za jego pomocą nasz `BeanNameGenerator`. Więcej informacji znajdziesz w rozdziale „Naming autodetected components" z „The Spring Framework Reference" autorstwa Roda Johnsona. Rozdział jest dostępny pod adresem *http://mng.bz/23kk*.

architektonicznej. Jeżeli mamy na przykład jeden plik konfiguracyjny Spring dla DAO, a drugi dla usług i chcemy coś w nich zmienić, wystarczy pobrać odpowiedni plik, przejrzeć go i wprowadzić zmianę. Łatwo znaleźć ten plik, ale w zależności od jego wielkości trudne może być zapoznanie się z nim. Dodatkowo przechowywanie konfiguracji w XML zapewnia, że klasy POJO są czyste, co było jednym z argumentów przemawiających za używaniem Spring. Dystrybucja konfiguracji jest znacznie bardziej złożona, szczególnie gdy wymieniamy część technologii infrastruktury na inne.

Jeżeli korzystamy z adnotacji, nasza konfiguracja jest skonsolidowana w aplikacji w przekrojach pionowych. Wolimy taki podział, ponieważ zmiany funkcji zwykle przekraczają warstwy architektoniczne, więc przydatna jest możliwość zmian elementów w jednym miejscu. Jeżeli musimy na przykład dodać pole do klasy domeny, otwieramy tę klasę w IDE, dodajemy pole i dekorujemy je adnotacjami JPA oraz Bean Validation Framework w jednym kroku. Jest to łatwiejsze niż otwieranie każdego pliku konfiguracyjnego XML, przeglądanie go i wprowadzanie niezbędnych zmian. Inną zaletą używania adnotacji jest łatwość uzyskania pełnego widoku na poszczególne usługi domeny oraz ich obiekty.

Jakaś konfiguracja XML jest nadal potrzebna. Konieczne jest włączenie skanowania komponentów w XML — Spring posiada kilka klas, które można konfigurować wyłącznie w XML.

1.6. Podsumowanie

W tym rozdziale przedstawiliśmy krótko Spring Framework, jego główne elementy i podstawowe zasady, w tym odwrócenie kontroli oraz jego związek ze wstrzykiwaniem zależności. Na kilku przykładach zademonstrowaliśmy zalety Spring i zarządzania zależnościami aplikacji, jak również opisaliśmy sposób definiowania tych zależności za pomocą XML oraz adnotacji Java.

W następnym rozdziale zostawimy kontener i przejrzymy część z (bardzo) użytecznych komponentów dostarczanych przez Spring, które obsługują trwałość, odwzorowanie obiektowo-relacyjne oraz zarządzanie transakcjami. W wielu przypadkach możesz dołączyć te komponenty do swojej aplikacji w sposób całkowicie przezroczysty.

Skorowidz

A

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

T

U

V

W

X

Y

Z

Ź

Ż

SPRING FRAMEWORK zrewolucjonizował sposób wytwarzania aplikacji w języku Java. Rzeczy trudne do wykonania stały się łatwe, a te łatwe w zasadzie zaczęły robić się same. Od pierwszej wersji ten framework jest cały czas rozwijany, a obecna, trzecia wersja, to prawdopodobnie najczęściej wybierane narzędzie do tworzenia aplikacji. Co sprawiło, że Spring zyskał taką popularność? Ogrom możliwości, świetna architektura, duża społeczność oraz ciągły rozwój i wsparcie dla nowych technologii to jego główne atuty.

W tej książce znajdziesz najlepsze techniki pracy ze Spring Framework w wersji 3. Przekonasz się, jak skutecznie wykorzystać potencjał Spring MVC. Dowiesz się, jak uwierzytelniać użytkowników, korzystać ze Spring Web Flow oraz budować usługi sieciowe. Ponadto przekonasz się, jak proste może być pisanie testów integracyjnych oraz korzystanie z narzędzi do mapowania obiektowo-relacyjnego. Książka ta jest doskonałą lekturą dla wszystkich programistów korzystających z języka Java. Z pewnością wzbogaci ich warsztat oraz umiejętności.

DZIĘKI TEJ KSIĄŻCE:

- poznasz szkielet Spring w wersji 3
- wykorzystasz potencjał Spring MVC
- napiszesz skuteczne testy integracyjne
- stworzysz lepszą aplikację!

NAJLEPSZE TECHNIKI PRACY ZE SPRING 3!

helion.pl
księgarnia internetowa

Nr katalogowy: 18239

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

ISBN 978-83-246-8184-6

9 788324 681846

Cena: 89,00 zł