

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

Original edition copyright © 2013 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2014 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

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/srip.zip>

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

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wprowadzenie	9
Podziękowania	11
O Spring	13
O książce	15

Rozdział 1. Przedstawiamy Spring — kontener wstrzykiwania zależności 19

1.1.	Czym jest Spring i dlaczego z niego korzystamy?	19
1.1.1.	Główne elementy biblioteki	20
1.1.2.	Dlaczego warto korzystać ze Spring?	22
1.2.	Elastyczna konfiguracja z użyciem wstrzykiwania zależności	23
1.2.1.	Konfigurowanie zależności w stary sposób	23
1.2.2.	Wstrzykiwanie zależności	24
1.2.3.	Odwrócenie kontroli	26
1.3.	Przykład prostej konfiguracji ziarna	28
1.3.1.	Tworzenie obiektów domeny	28
1.3.2.	Tworzenie interfejsu DAO konta oraz jego implementacji	29
1.3.3.	Konfigurowanie CsvAccountDao za pomocą Spring	31
1.3.4.	Tworzenie usługi wyszukującej konta niesolidnych dłużników	32
1.3.5.	Podłączanie AccountService do CsvAccountDao	33
1.4.	Łączenie ziaren za pomocą XML	35
1.4.1.	Przegląd przestrzeni nazw beans	35
1.4.2.	Zakres ziaren	39
1.4.3.	Przestrzeń nazw p	43
1.4.4.	Przestrzeń nazw c	44
1.5.	Automatyczne łączenie oraz skanowanie komponentów z użyciem adnotacji	46
1.5.1.	Adnotacja @Autowired	46
1.5.2.	Adnotacje stereotypów	49
1.5.3.	Skanowanie komponentów	50
1.5.4.	XML czy adnotacje? Co jest lepsze?	51
1.6.	Podsumowanie	52

Rozdział 2. Trwałość danych, ORM i transakcje 53

2.1.	Dostęp do danych za pomocą JDBC	55
2.2.	Wyszukiwanie DataSource za pomocą JNDI	60
2.3.	Odwzorowanie obiektowo-relacyjne i transakcje z użyciem Hibernate	62
2.4.	Tworzenie warstwy dostępu do danych	73
2.5.	Użycie JPA (opcjonalnie)	79
2.6.	Przegląd Spring Data JPA (opcjonalnie)	83
2.7.	Podsumowanie	85

Rozdział 3. Budowanie aplikacji WWW przy użyciu Spring Web MVC 87

- 3.1. Podstawy Spring Web MVC 88
 - 3.1.1. Wzorzec „model-widok-kontroler” (MVC) 88
 - 3.1.2. Czym jest Spring Web MVC? 89
 - 3.1.3. Przegląd architektury Spring Web MVC 90
- 3.2. Tworzenie pierwszej aplikacji Spring Web MVC 91
 - 3.2.1. Konfigurowanie aplikacji 91
 - 3.2.2. Prosty obiekt domeny 93
 - 3.2.3. Tworzenie prostego kontrolera 94
 - 3.2.4. Tworzenie widoku nadrzędnego i podrzędnego 96
- 3.3. Obsługa i przetwarzanie formularzy 97
 - 3.3.1. Użycie obiektów domeny jako ziaren formularza 97
 - 3.3.2. Dodawanie kontrolera 98
 - 3.3.3. Dodawanie plików JSP formularza i podziękowania 100
 - 3.3.4. Aktualizowanie kontekstu aplikacji 102
 - 3.3.5. Dodanie mechanizmu przekierowania po przestaniu 102
 - 3.3.6. Dodawanie białych list wiązania formularza 102
 - 3.3.7. Dodanie kontroli poprawności danych formularza 104
- 3.4. Konfigurowanie Spring Web MVC — web.xml 104
- 3.5. Konfigurowanie Spring Web MVC — kontekst aplikacji 106
 - 3.5.1. Konfiguracja HandlerMapping 107
 - 3.5.2. Konfigurowanie obiektów HandlerAdapter 112
 - 3.5.3. Konfigurowanie obiektów HandlerExceptionResolver 112
 - 3.5.4. Konfigurowanie obiektów ViewResolver 112
 - 3.5.5. Konfigurowanie obiektu RequestToViewNameTranslator 115
 - 3.5.6. Konfigurowanie innych obiektów rozpoznających 116
- 3.6. Przegląd technologii Spring Mobile 116
 - 3.6.1. Anatomia żądania HTTP 117
 - 3.6.2. Wykrywanie urządzenia mobilnego za pomocą Spring Mobile 118
 - 3.6.3. Konfigurowanie Spring Mobile 119
 - 3.6.4. Obsługa właściwości witryny 122
 - 3.6.5. Użycie bibliotek JavaScript do poprawiania wyglądu i działania aplikacji 124
 - 3.6.6. Przelączenie do osobnej witryny mobilnej 126
- 3.7. Technologie pokrewne 127
 - 3.7.1. Spring Web Flow 127
 - 3.7.2. Spring JavaScript 127
 - 3.7.3. Spring Faces 127
 - 3.7.4. Spring Security 127
 - 3.7.5. Usługi sieciowe w stylu REST 128
- 3.8. Podsumowanie 128

Rozdział 4. Proste formularze WWW 129

- 4.1. Wyświetlanie formularzy 129
- 4.2. Wyodrębnianie napisów z widoku 137
- 4.3. Kontrola danych formularza 139
- 4.4. Zapisywanie danych formularza 149
- 4.5. Podsumowanie 159

Rozdział 5. Rozszerzanie aplikacji Spring MVC za pomocą Web Flow 161

- 5.1. Czy Spring Web Flow jest właściwym narzędziem? 162
- 5.2. Przegląd Spring Web Flow 162
 - 5.2.1. Definiowanie przepływu 163
 - 5.2.2. Pięć typów stanów 164
 - 5.2.3. Przejścia pomiędzy stanami 167
 - 5.2.4. Dane przepływu 168
- 5.3. Aplikacja demonstracyjna Klub piłkarski 171
 - 5.3.1. Instalacja i konfigurowanie SWF 172
 - 5.3.2. Tworzenie przepływów z różnymi typami stanów 178
- 5.4. Użycie klas akcji 185
- 5.5. Wiązanie danych formularza 189
- 5.6. Kontrola poprawności formularzy 191
- 5.7. Dziedziczenia przepływów i stanu 194
- 5.8. Zabezpieczanie przepływów WWW 197
- 5.9. Podsumowanie 201

Rozdział 6. Uwierzytelnianie użytkowników 203

- 6.1. Implementacja funkcji logowania, wylogowywania oraz zapamiętywania użytkownika 203
- 6.2. Dostosowywanie strony logowania 212
- 6.3. Implementacja zawsze widocznego formularza logowania 216
- 6.4. Pobieranie danych użytkowników z bazy danych 219
- 6.5. Modyfikowanie schematu bazy danych użytkowników 223
- 6.6. Zastosowanie własnej usługi użytkowników oraz obiektu użytkownika 225
- 6.7. Zabezpieczanie haseł użytkowników w bazie danych 233
- 6.8. Automatyczne uwierzytelnianie użytkownika po rejestracji 239
- 6.9. Podsumowanie 241

Rozdział 7. Autoryzacja użytkowników 243

- 7.1. Autoryzacja metod Java z użyciem poziomów autoryzacji, ról oraz uprawnień 245
- 7.2. Autoryzacja widoków JSP z użyciem poziomów autoryzacji, ról oraz uprawnień 252
- 7.3. Autoryzacja zasobów WWW z użyciem poziomów autoryzacji, ról oraz uprawnień 255
- 7.4. Autoryzacja wywołań metod bazująca na ACL 258
- 7.5. Wyświetlanie elementów nawigacyjnych oraz zawartości na podstawie ACL 277
- 7.6. Podsumowanie 280

Rozdział 8. Komunikacja z użytkownikami i klientami 281

- 8.1. Tworzenie formularza kontaktowego 282
- 8.2. Automatyczne generowanie odpowiedzi oraz powiadomień pocztowych 289
- 8.3. Przyspieszanie automatycznego generowania wiadomości e-mail 296
- 8.4. Subskrybowanie listy wysyłkowej przez użytkowników 301
- 8.5. Publikowanie strumieni RSS z nowościami 311
- 8.6. Podsumowanie 315

Rozdział 9. Tworzenie silnika komentarzy z tekstem sformatowanym	317
9.1. Tworzenie prostego silnika komentarzy	318
9.2. Integracja silnika komentarzy z usługą dostarczania artykułów	327
9.3. Dodanie obsługi tekstu sformatowanego do silnika komentarzy	336
9.4. Testowanie filtra HTML	343
9.5. Podsumowanie	346
Rozdział 10. Testy integracyjne	347
10.1. Konfigurowanie programu Maven dla testów integracyjnych	348
10.2. Pisanie transakcyjnych testów ścieżki pozytywnej	354
10.3. Sprawdzanie, czy testowany kod zgłasza wyjątek	368
10.4. Tworzenie testów integracyjnych do weryfikowania wydajności	370
10.5. Ignorowanie testu	374
10.6. Uruchamianie testów integracyjnych na wbudowanej bazie danych	376
10.7. Podsumowanie	381
Rozdział 11. Budowanie bazy danych zarządzania konfiguracją	383
11.1. Tworzenie prostego elementu konfiguracji	387
11.2. Tworzenie związanych ze sobą elementów konfiguracji	393
11.3. Dodawanie usługi sieciowej w stylu REST	405
11.4. Modyfikowanie CMDB po udanej kompilacji	417
11.5. Pozyskiwanie publicznych danych GitHub	422
11.6. Pozyskiwanie prywatnych danych GitHub	427
11.7. Szyfrowanie żetonów dostępu do zastosowań produkcyjnych	437
11.8. Podsumowanie	439
Rozdział 12. Budowanie silnika dostarczania artykułów	441
12.1. Przechowywanie artykułów w repozytorium treści	442
12.2. Tworzenie silnika dostarczania artykułów działającego w środowisku WWW	454
12.3. Przechowywanie artykułów w repozytorium dokumentów	466
12.4. Podsumowanie	471
Rozdział 13. Integracja w przedsiębiorstwie	473
13.1. Integracja aplikacji poprzez wspólną bazę danych	477
13.2. Rozłączanie aplikacji za pomocą usług sieciowych w stylu REST	482
13.3. Implementacja szyny komunikatów z użyciem RabbitMQ i Spring Integration	491
13.4. Tworzenie zgłoszeń na podstawie magazynu IMAP	510
13.5. Wysyłanie potwierdzeń po SMTP	516
13.6. Podsumowanie	520
Rozdział 14. Tworzenie biblioteki aktywności witryny bazującej na Spring	521
14.1. Tworzenie szablonu bezpiecznika i wywołania zwrotnego	523
14.2. Udostępnianie bezpieczników jako JMX MBean	537
14.3. Obsługa konfiguracji bazującej na AOP	542

- 14.4. Obsługa własnej przestrzeni nazw 549
- 14.5. Obsługa konfiguracji korzystającej z adnotacji 557
- 14.6. Podsumowanie 568

Dodatek. Korzystanie z kodu przykładów 569

- A.1. Konfiguracja IDE i środowiska 569
- A.2. Organizacja kodu 569
- A.3. Dostęp do kodu 570
- A.4. Budowanie kodu 571
- A.5. Konfigurowanie aplikacji 571
- A.6. Uruchamianie aplikacji 573

Skorowidz 575

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

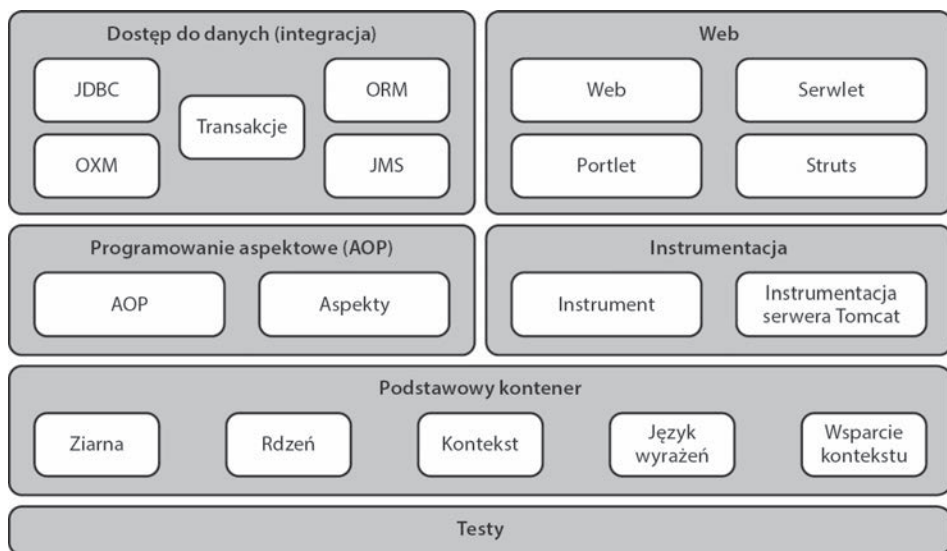
W tym rozdziale przedstawimy krótko Spring Framework, zaczynając od wyjaśnienia, czym on jest, i omówimy jego główne elementy. Następnie zajmiemy się podstawowymi zasadami wykorzystywanymi w Spring Framework oraz mechanizmem odwrócenia kontroli i jego wpływem na wstrzykiwanie zależności. Na koniec pokażemy 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. Zaczniemy 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-objektowego (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 modulem 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**

Przeanalizujemy 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 zewnątrz 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` zewnątrz, 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 `BasicDataSource`. 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;

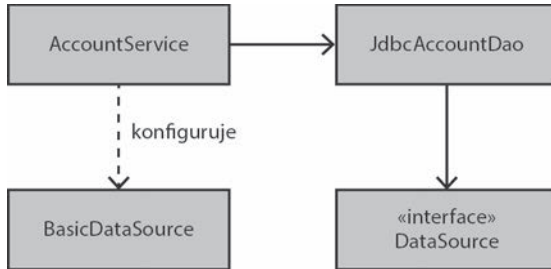
public class AccountService {
    private JdbcAccountDao accountDao;

    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 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>

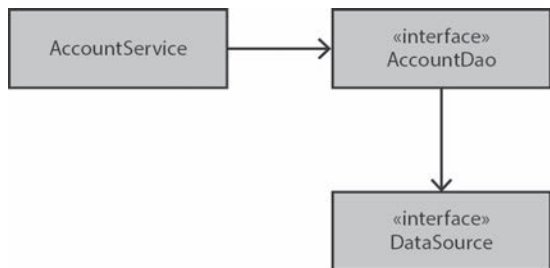
```

Obiekt DataSource wraz ze skonfigurowanymi parametrami 1

Obiekt DataSource wstrzyknięty do JdbcAccountDao 2

Obiekt JdbcAccountDao wstrzyknięty do AccountService 3

Jeżeli zaczynasz pracę z biblioteką Spring, taka konfiguracja może być dla Ciebie nieznana, ale jej znaczenie powinno być wystarczająco jasne. We fragmencie 1 deklarujemy DataSource i ustawiamy jego właściwości za pomocą parametrów konfiguracyjnych. We fragmencie 2 deklarujemy JdbcAccountDao i wstrzykujemy go do DataSource. Podobnie we fragmencie 3 wstrzykujemy JdbcAccountDao do AccountService. Ostatecznym wynikiem jest uzyskanie usługi, która obsługuje cały łańcuch zależności, ale konfiguracja jest dla niej zupełnie przezroczysta. Nowe, wyczystzone 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, utworzymy 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("MMdyyyy");
        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 **1**. Aby Spring mógł je dla nas wypełnić, wymagane jest zdefiniowanie publicznej metody ustawiającej **2**. Metoda `findAll()` z punktu **3** 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 kodu¹.

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 konfiguruje 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" ← 1 Schemat ziarna Spring
  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>
```

2 Definicja ziarna
CsvAccountDao

3 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 1. 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 2. Element bean posiada dwa atrybuty: *id* oraz *class*. Za pomocą atrybutu *id* będziemy obsługiwać zależności pomiędzy zianami. 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> findDelinquentAccounts() 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>

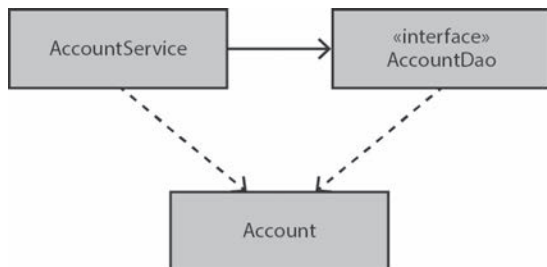
    <bean id="accountService"
        class="com.springinpractice.ch01.service.AccountService">
        <property name="accountDao" ref="accountDao"/>
    </bean>
</beans>

```

← Definicja ziarna `AccountService`
 ← Wstrzykiwanie `AccountDao`

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 =
            new ClassPathXmlApplicationContext("applicationContext.xml");
        AccountService accountService =
            (AccountService)appCtx.getBean("accountService");
        List<Account> delinquentAccounts = accountService
            .findDelinquentAccounts();
        for (Account a : delinquentAccounts) {
            System.out.println(a.getAccountNo());
        }
    }
}

```

← ❶ Kontener wstrzykiwania zależności

← ❷ Odczytanie AccountService z kontenera

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, niejawnny 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

Przeźnię 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 `ApplicationContext`.

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 `AccountService` 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 `JdbcAccountDao` 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 pulę 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 własnych².

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

1 Deklaracja `PropertyPlaceholderConfigurer`

2 Lokalizacja korzysta z systemu zmiennych środowiska Java

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>

```

3 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 `PropertyPlaceholderConfigurer` 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

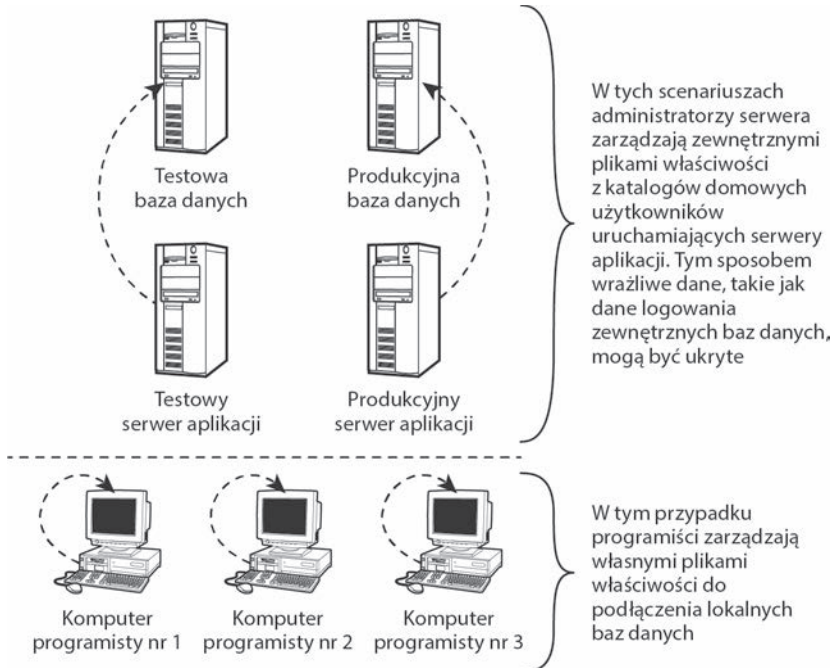
```
<property name="location" value="file:${user.home}/springbook.properties"/>
```

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

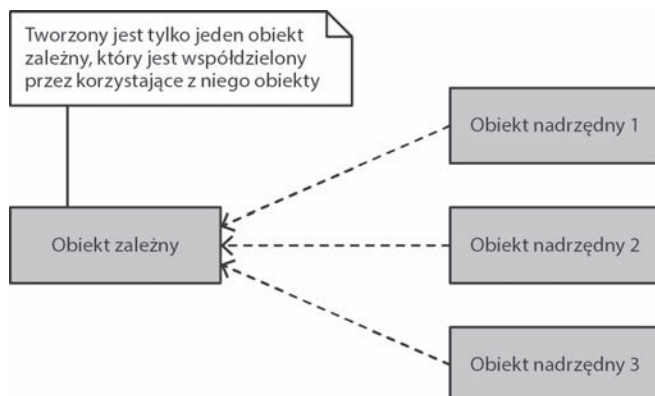
Zakres jest konfigurowany za pomocą atrybutu `scope`:

```
<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 serwetami). 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, niezmiennie, 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.

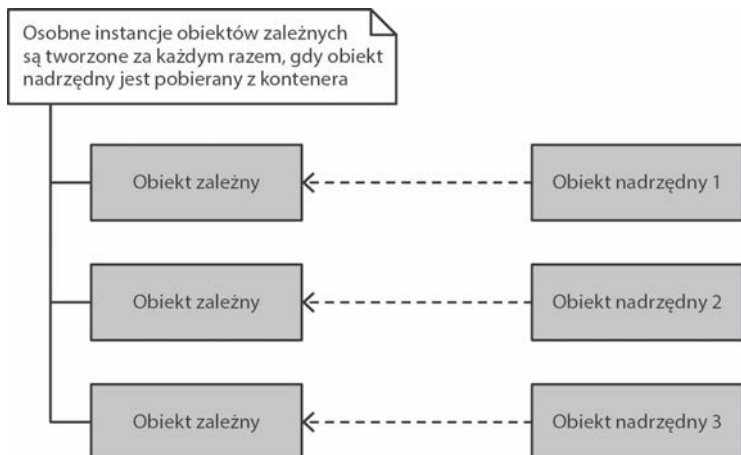
ZAKRES PROTOTYPE

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 deskrypcorze 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 `XmlWebApplicationContext`. 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>

```

2 Deklaracja właściwości jako atrybutów elementu bean

Przestrzeń nazw `p` zapewnia nam udogodnienia składniowe oraz pozwala na tworzenie spójniejszej konfiguracji. W punkcie **1** deklarujemy przestrzeń nazw, ale nie należy zwracać uwagi na podaną lokalizację schematu. Aby dowiedzieć się, dlaczego, spójrzmy na użycie tej przestrzeni w punkcie **2**. 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 `konstruktor-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" ← ❶ Deklaracja przestrzeni nazw 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>
```

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 opisanu 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:accountDao-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 właściwość. 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 `@AutoWired`. 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 accountDao` w `AccountService` zostanie wypełnione automatycznie instancją `JdbcAccountDao`. 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 `component-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 litery⁴. 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

- AbstractHandlerMapping, 110
- abstrakcja
 - NamedParameterJdbcOperations, 55, 58
 - RowMapper, 55
- AccountForm, 131
- ACE, 259
- Acegi, 127
- ACL, 128, 245, 259, 261
 - manipulowanie programowe, 273
 - optymalizacja definicji, 272
 - właściciel, 276
 - wyszukiwanie, 271
- adapter, 226, 230
- administratora uprawnienia, *Patrz:* witryna
 - uprawnienia administratora
- adnotacja, 44, 46, 51, 52, 174, 246, 558, 559
 - @Async, 297, 299, 300, 309, 323
 - @Autowired, 46, 47, 48
 - @Column, 66, 152
 - @Component, 46, 49, 300
 - @Controller, 46, 49, 50, 94, 108, 109, 132, 135, 414
 - @DirtyContext, 370
 - @Entity, 66
 - @ExpectedException, 362, 370
 - @Fetch, 398
 - @GeneratedValue, 152
 - @GenerationStrategy, 366
 - @GraphId, 390, 391
 - @Id, 66, 152
 - @IfProfileValue, 372
 - @Ignore, 362, 375
 - @Indexed, 397, 398
 - @InitBinder, 142, 355
 - @ManagedAttribute, 538
 - @ManagedOperation, 539
 - @ManagedResource, 538
 - @ModelAttribute, 100, 142
 - @NamedQuery, 66
 - @NodeEntity, 390, 391
 - @NotEmpty, 194
 - @PathVariable, 355
 - @PersistenceContext, 81
 - @PostAuthorize, 248, 267, 268
 - @PostConstruct, 515
 - @PostFilter, 268
 - @PreAuthorize, 247, 248, 251, 267, 268
 - @PreFilter, 268
 - @Qualifier, 240
 - @RelatedTo, 394
 - @RelatedToVia, 395
 - @RelationshipEntity, 390, 399
 - @Repeat, 362, 372
 - @Repository, 46, 49, 51, 77, 154, 449
 - @RequestBody, 419
 - @RequestMapping, 50, 109, 128, 132, 355, 412, 414
 - @ResponseBody, 412, 413
 - @RestResource, 485
 - @RunWith, 345, 361
 - @ScriptAssert, 143
 - @Secured, 251
 - @Service, 46, 49, 51
 - @Test, 362, 370
 - @Timed, 362, 372
 - @Transaction, 363
 - @Transactional, 68, 70, 71, 323, 403, 449, 450
 - @TransactionConfiguration, 363
 - @Transient, 66, 152
 - @Valid, 142, 193, 285, 355
 - @Value, 419, 519
 - @XmlAttribute, 407

- adnotacja
 - @XmlElement, 407, 408
 - @XmlRootElement, 408
 - @XmlType, 408
 - Hibernate, 51
 - Hibernate Validator, 51
 - Jackson, 406
 - JAXB, 406, 415
 - JPA, 51
 - JSR-250, 251
 - JSR-303, 192
 - kontroli dostępu, 266, 267
 - SDR, 485
 - Spring Data Neo4j, 390
 - stereotypu, 46, 49
 - zabezpieczeń, 251
- adres
 - e-mail, 68, 84, 282, 293, 474
 - IP, 99, 304, 311, 385
 - URI, 487
 - URL, 50, 95, 108, 115, 214, 255, 310, 419
 - filtr, 199
 - ochrona, 277
- Advanced Message Queuing Protocol, *Patrz:* AMQP
- akcja, 88, 265
- Amazon EC2, 385
- AMQP, 491, 492, 498, 506
- AOP, 20, 21, 31, 569
 - do zabezpieczania metod Java, 246
- Apache Commons Database Connection Pool,
 - Patrz:* DBCP
- Apache iBATIS, *Patrz:* iBATIS
- API
 - JPA, 79
 - trwałości, 54
 - trwałości z Hibernate, 79
- aplikacja
 - dekompozycja, 396
 - konfiguracja, 134, *Patrz:* konfiguracja aplikacji
 - konsolowa, 34
 - kontekst, 102, 105, 106
 - mobilna, 116
 - obsługi treści, 444
 - portletowa, 42
 - praca z wieloma bazami danych, 478, 481
 - rozwłączanie, 491
 - separacja, 483
 - skórka, 116
 - stos, 355
 - strukturalizacja konfiguracji, 355
 - uruchamianie, 573
 - wielowątkowa, 40
 - zarządzająca kontaktami, 348
- Artifactory, 385, 417
- artykuł, 441
 - importowanie, 443, 444, 459
 - metadane, 446
 - przechowywanie, 466
 - strona, 446
 - tworzenie, 445
- AspectJ, 72
- aspekt, 21
- atak
 - międzywitrynowy, 327
 - słownikowy, 236, 307
 - XSS, 336, 344
- Atom, 115, 311, 312
- atrybut
 - access, 253, 254
 - authentication-failure-url, 215
 - auto-config, 205
 - base-package, 50
 - class, 31
 - destroy-method, 59
 - driverClassName, 59
 - elementu bean, 43
 - expired, 306
 - id, 31, 49, 61
 - ifAllGranted, 254
 - ifAnyGranted, 254
 - ifNotGranted, 254
 - ignore-failures, 358
 - jndi-name, 61
 - login-page, 215
 - member, 99
 - method, 254
 - model, 286
 - name, 36
 - p:accountDao-ref, 46
 - password, 59
 - path-type, 256
 - property, 37
 - ref, 33, 36
 - resource-ref, 61
 - scope, 40

url, 59, 254
 username, 59
 value, 33
 view-name, 215
 Attacklab, 336
 autoryzacja, 128, 205, 243, 427, 428, *Patrz też:*
 uwierzytelnianie
 bazująca na
 rolach, 128
 uwierzytelnianiu, rolach i uprawnieniach,
 245, 246, 248, 249, 252, 255, 257
 cel, 243
 lista, *Patrz:* lista
 OAuth, 428
 obiektów domeny, 245
 styl, 243, 245

B

Bamboo, 385, 417
 BasicDataSource, 24
 baza danych, 63, 219, 221, 282, 301
 kont użytkownika, 151
 modyfikowanie schematu, 223, 224
 NoSQL, 389, 444
 odwołanie w kontekście transakcji, 69
 oparta na grafach, 389
 przywrócenie, 358
 relacji, *Patrz:* relacja
 relacyjna, 55, 444
 aktualizacja, 55
 schemat, 389
 udostępnianie JNDI, 221
 wbudowana, 376
 wspólna, 477, 478, 482, 483
 zarządzania konfiguracją, 387
 Bean Validation, 104
 BeanNameUrlHandlerMapping, 107, 109
 BeanNameViewResolver, 314
 bezpieczeństwo, 21, 44, 102, 128, 197, 243, 306,
 318, 327, 336, 389, 437
 obiektów domeny, 245
 plik konfiguracyjny, 31
 punkt przekroju, 246
 bezpiecznik, 525, 542, 548, 553
 fizyczny, 522
 JMX MBean, 537
 konfigurowanie, 535

korzystający z AOP, 542, 549
 otwarty, 522, 529
 półotwarty, 522, 530
 programowy, 522
 stan, 522, 529
 ulotny, 529
 zamknięty, 522, 529
 biblioteka
 aktywności, 521
 Commons Lang, 131
 DAO, 83
 Hibernate, *Patrz:* Hibernate
 mobilna JavaScript, 124
 model-widok-kontroler, *Patrz:* MVC
 MVC, 161
 ORM, *Patrz:* ORM
 Spring 3.1, *Patrz:* Spring 3.1
 Spring Framework, *Patrz:* Spring
 Spring Web MVC, *Patrz:* Spring Web MVC
 TestContext, 343, 344, 347, 354, 362, 368, 376
 znaczników, 128, 137, 216, 277
 form, 134, 140, 147
 formularzy, 101, 189

błąd

globalny, 286
 kod, 145, 193
 kontrola, 30
 brama, 488, 489, 490, 492, 496, 504, 505
 interfejs, 511
 wchodząca, 507, 509
 wychodząca, 506

C

CAPTCHA, 97, 282
 CAS, 207
 Castor, 21
 CDM, 493
 CI, 387, 396
 abstrakcja, 389
 aplikacji, 397
 implementacja, 389
 modułu, 393, 395, 396, 409
 pakietu, 393, 397, 405, 408
 relacje, 393, 394, 399
 zespołu, 393, 396, 398
 Clover, 376
 CMDB, 385, 387, 405, 417, 422

CMS, 441, 444
 Cobertura, 376
 ContactDao, 80
 ControllerClassNameHandlerMapping, 108, 109
 CouchDB, 444
 CRUD, 67, 75, 265, 391, 401, 411, 413, 450

D

dane

baza relacyjna, *Patrz:* baza danych relacyjna
 dostęp, *Patrz:* integracja
 dynamiczne, 291
 instalacji, 422
 kompilacji, 422
 model kanoniczny, *Patrz:* CDM
 operacyjne, 422
 prywatne GitHub, 427
 publiczne GitHub, 422, 423
 repozytorium, 422, 423
 użytkowników, 422

DAO, 23, 24, 36, 74, 206, 207, 228, 391

Hibernate, 447
 implementacja, 74
 interfejs, 29
 modułu, 400
 pakietu, 400
 zakres singleton, 40

DaoAuthenticationProvider, 206, 207, 208

data access object, *Patrz:* DAO

DataSource, 23, 24, 222

konfigurowanie, 58, 60, 62
 proxy, 24
 wyszukiwanie JNDI, 62

DBCP, 24, 59

DBMS, 354

DDD, 150

DDL, 430

debuger, 267

DefaultAnnotationHandlerMapping, 109, 136

DefaultRequestToViewNameTranslator, 115

DelegatingFilterProxy, 209

Derby, 376

DeviceResolverHandlerInterceptor, 121, 123

DispatcherServlet, 90, 104, 106, 109, 119, 136,
 173, 355

konfiguracja, 106

domena, 88

integracji, 475

logika, *Patrz:* logika domeny

mobi, 126

obiekt, *Patrz:* obiekt:domeny

dostawca, 206, 207

usługi obliczeniowej, 385

dostęp do danych, *Patrz:* integracja

DTO HATEOAS, 493

dziedziczenie, 63, 74, 110

przepływu, 194, 195, 197

stanów, 195, 197

w interfejsie podrzędnym, 75

E

EBR, 353

Eclipse, 569

EclipseLink, 80

EclipseLink JPA, 82

edytor PageDown, 336, 337

Ehcache, 270, 443

EJB, 88

element

<authentication-manager>, 207

<authenticationprovider>, 206

<evaluate>, 169, 170

<flow:flow-executor>, 177

<form-login>, 215, 218

<http>, 205

<jackrabbit:repository>, 453

<jdbc:initialize-database>, 358

<mvc:annotation-driven>, 416

<rabbit:admin/>, 498

<secured>, 200

<security:authorize>, 212

<set>, 169, 170

<var>, 170

annotation-config, 47

konstruktor-arg, 44

context:annotation-config, 50

context:component-scan, 50

filterTarget, 268

flow-builder-services, 192

flowexecution-listener, 200

nawigacyjny ukrywanie, 277

przepływu pracy, 161

transition, 167

view-state, 164, 190

EMMA, 376
 encja, 63
 jako węzeł, 390
 łączenie, 395
 przekształcenie w obiekt, 63
 relacji, *Patrz:* relacja encja
 trwałości, 140
 Enterprise JavaBeans, *Patrz:* EJB
 EntityManager, 81

F

Facebook, 317, 423, 425
 Failsafe, 351, 352, 362, 372, *Patrz:* Maven Failsafe
 FeedBurner, 302, 315
 formater, 416
 formularz, 97, 129, 137
 bezpieczeństwo, 102
 biała lista wiązania, 98, 102
 filtrowanie pól, 140
 HTML, 286
 kontaktowy, 282, 289, 290, 297
 logowania, 212, 213, 216
 poprawność, 104, 140, 142, 155, 191
 przekierowanie, 136
 przekierowanie po przesłaniu, 102
 przesyłania artykułu, 460
 przesyłanie danych, 132
 rejestracji, 97
 subskrypcji, 302, 304, 305
 wiązaną danych, 140, 190
 WWW, 130
 wyświetlanie, 149
 FreeMaker, 519
 FreeMarker, 296
 funkcja
 przełączania witryn, 116
 sterowana adnotacjami, 174
 zakresu, 43

G

Git, 385, 569
 GitHub, 422, 423, 424, 425, 569
 zaczep, 427, 432
 Google Analytics, 302
 graf, 389
 krawędź skierowana, 395
 stanów, 127

węzeł, 389
 końcowy, 395
 początkowy, 395

H

H2, 376
 handler
 HttpServletRequest, 90
 odzworowania, 108, 109, 110
 przestrzeni nazw, 553, 554
 uprawnień, 269
 wyrażeń, 269, 270
 wywołania POST, 100
 zadań, 90
 HandlerInterceptor, 110
 HandlerMapping, 110
 hasło, 97, 128, 131, 153, 154, 157, 358
 odcisk, 235, 236, 238
 pobieranie, 228
 stare, 238
 wartość początkowa, 236
 HATEOAS, 482, 483, 487
 łącza, 487
 HbnUserMessageDao, 288
 Hibernate, 21, 53, 54, 62, 67, 73, 79, 154, 282,
 301, 332, 333, 365, 447, 477
 API Session, 63, 67
 Hibernate 3, 149
 Hibernate Core, 318, 327
 Hibernate Validator, 104, 140, 142, 191, 193, 318
 komunikat, 146
 HibernateTemplate, 524
 HSQL, 376
 HTTPS, 205
 Hudson, 417
 Hypermedia as the Engine of Application State,
 Patrz: HATEOAS

I

iBatis, 21
 identyfikator zabezpieczeń, *Patrz:* SID
 IMAP, 510, 512, 513
 SSL, *Patrz:* IMAPS
 IMAP IDLE, 513
 IMAPS, 514
 imitacja zależności, 354
 instancja, 63, 385, 386

instrukcja SQL
 select, 57
 update, 57

instrumentacja, 20

integracja, 20, 21, 55, 475, 476, 483
 brama, 488, 489, 490, 492, 496, *Patrz:* brama
 punkt, 522, 524, 531
 punkt-punkt, 491

interfejs

- AccountService, 229
- Action, 185, 186
- ApplicationContextAware, 345
- ApplicationRepository, 392
- AuthenticationManager, 206
- BindingResult, 141
- bramy, 495, 511
- CommentService, 322
- DAO, 29, 74, 75
- DataSource, 24
- Device, 118
- DeviceResolver, 118
- FactoryBean, 295
- GuardListSource, 546
- HandlerAdapter, 112
- HandlerInterceptor, 110
- HandlerMapping, 107
- InitializingBean, 187
- javax.validation.Validator, 192
- javax.validation.ValidatorFactory, 192
- JPA EntityManager, 80, 81
- kontrolera, 89
- org.springframework.security.acls.model.Acl, 266
- org.springframework.validation.Validator, 192
- PortalGateway, 495
- programowy programowania, 423
- RowMapper, 57
- Runner, 361
- Spring BeanNameGenerator, 51
- TicketGateway, 495, 496
- UserDetails, 226
- UserDetailsService, 207, 220, 229
- użytkownika, 122, 474
- WWW, 441
- wywołania zwrotnego, 525

internacjonalizacja, 116, 137

InternalResourceViewResolver, 174

IoC, 23

J

JAAS, 207

Jackrabbit, 441, 442, 444, 446, 453

Jackson, 405, 415

Java Architecture for XML Binding, *Patrz:* JAXB

Java Context Repository, *Patrz:* JCR

Java Database Connectivity, *Patrz:* JDBC

Java Message Service, *Patrz:* JMS

Java Naming and Directory Interface, *Patrz:* JNDI

Java Persistence API, *Patrz:* JPA

Java Persistence Query Language, *Patrz:* JPQL

Java VisualVM, 537, 540

java.util.Properties, 24

JavaMail, 61, 290, 296, 301, 510, 516

JavaMailSender, 294

JavaScript, 336

JavaScript Rhino, 144

JavaServer Faces, 127

JAXB, 21

JAXB 2, 405

JBoss EL, 170

JCA, 61

JConsole, 540

JCR, 441, 443, 444

JCR 2.0, 443

JDBC, 21, 53, 54, 55, 60, 62, 154, 229
 sterownik, 37
 szablon, 157

JdbcDaoImpl, 222, 223

JdbcMutableAclService, 274

JdbcTemplate, 54, 524

JDO, 21

Jenkins, 385, 417

język
 definicji przepływów, 163, 164
 SpEL, 246
 wyrażeń, 170
 zapytań, 63

JiBX, 21

JMS, 21, 61, 492

JMX, 537, 540

JNDI, 54, 60, 157

JPA, 21, 54, 79, 83, 149, 318, 327, 391

JPQL, 63

jQuery Mobile, 124

JSF, 163, 172

JSON, 405, 406, 410, 412

JSP, 113, 165
 bazujący na JSTL, 113
 wyłączanie fragmentów, 252
 JSR 283, 443
 JSR-223 Java Scripting, 140
 JSR-303, 104, 142, 191
 implementacja, 191
 komunikat, 146
 JSR-303 Bean Validation, 140, 191, 282, 321
 JSTL, 165
 JUnit, 343, 348, 354, 370

K

klasa, 63

AbstracHibernateDao, 331
 AbstractAction, 187
 AbstractAtomFeedView, 312
 AbstractContextLoader#modifyLocations, 362
 AbstractFeedView, 312
 AbstractHbnDao, 77, 79, 447
 AbstractRssFeedView, 312
 AccessControlListTag, AccessControlListTag, 279
 Account, 226
 AccountForm, 142
 akcji, 185, 187
 Article, 329
 ArticleController, 455, 468
 ArticleConverter, 463
 ArticleMapper, 450
 BasePermission, 265
 BasicDataSource, 24
 BeanDefinitionParser, 554
 CircuitBreakerTemplate, 543
 CircuitOpenException, 530
 Comment, 320, 321
 CommentTarget, 322
 ContactController, 283
 ContactServiceImpl, 67, 291, 297, 300
 DaoAuthenticationProvider, 220
 domeny, 396
 DTO, 493
 dynamicznych pośredników, 83
 FormAction, 188
 ForumServiceImpl, 267
 identyfikator jako typu parametru, 83
 import, 50

jako kontroler, 94
 JcrArticleDao, 449
 JcrDaoSupport, 449
 JDBC DataSource, 24
 JdbcAccountDao, 23, 24
 JsRuntimeSupport, 343
 LocalValidatorFactoryBean, 192
 MultiAction, 187, 188
 OpenSessionInViewFilter, 333
 OpenSessionInViewInterceptor, 333
 oznaczona za pomocą stereotypu, 50
 POJO, 52, 63, 94, 131, 150, 447
 PropertyPlaceholderConfigurer, 38, 39
 proxy, 297, 298
 Query, 80
 refaktoryzacja, 25
 RequestContextListener, 43
 Resources, 488
 ResourceSupport, 487
 RestTemplate, 424, 426, 483
 Session, 80
 SessionFactory, 80
 SitePreferenceHandlerInterceptor, 122
 SpringJUnit4ClassRunner, 361
 TicketTransformer, 514, 518
 UserMessage, 287, 288
 uwierzytelniania, 207
 View, 312
 wywołania zwrotnego, 524
 klucz, 57, 307
 kod DDL, 430
 kolekcja, 63
 kolumna, 63
 komentarz
 silnik, *Patrz:* silnik komentarzy
 tekstowy, 318
 wysyłanie, 325
 komponent
 obsługujący, 88
 rozłączanie, 22
 skanowanie, 46, 51, 300
 wielokrotnego użytku, 163
 komunikacja
 asynchroniczna, 502, 503
 punkt-punkt, 518
 pytanie-odpowiedź, 502, 503
 wyslij-subskrybuj, 518

- komunikat, 21, 137
 - broker, 492, 512
 - Hibernate Validator, 146
 - HTTP konwerter, 416
 - JSR-303, 146
 - szyna, *Patrz:* szyna komunikatów
 - złożoność translacji, 493
 - komunikator, 282
 - konfiguracja, 204
 - ACL, 270, 271
 - aplikacji, 91, 134, 385, 387
 - bazująca na adnotacjach, 135, 558, 568
 - DataSource, 158
 - deklaratywna, 558
 - DispatcherServlet, 105, 106, 136, 355
 - dryfowanie, 384, 388
 - element, *Patrz:* CI
 - HandlerMapping, 107
 - handlerów, 270
 - Hibernate, 157, 332
 - InternalResourceViewResolver, 113
 - kontekstu aplikacji, 156, 158, 338, 339
 - kontroli poprawności, 148
 - kopia główna, 388
 - Maven, 246, 348, 354, 424
 - Jetty, 157, 282
 - programowa, 558
 - przestrzeni nazw, 206
 - serwletu, 106
 - Spring, 192, 392, 414, 434
 - Spring MVC, 163, 173
 - Spring Security, 201, 205
 - Spring Web MVC, 104, 116
 - SWF, 176
 - środowiska, 387
 - uwierzytelniania, 206
 - wtyczki, 353
 - zabezpieczeń, 250
 - zabezpieczeń serwera, 301
 - zarządzanie, 383, 385, 387
 - danymi, 405
 - ziarna, 326
 - konsola JMX, 540
 - konstruktor, 46
 - AclAuthorizationStrategyImpl, 271
 - argument, 36, 37, 44, 45
 - kontekst
 - aplikacji, *Patrz:* aplikacja kontekst
 - zarządzanie, 67
 - kontener
 - DI, 20
 - java.util.Collection, 268
 - podstawowy, 20, 23
 - serwletów, 221
 - kontrola
 - dostępu, 243, 244, 245, 246, 258
 - pozycja, *Patrz:* ACE
 - zasady, 246, 247, 256, 257, 259, 266
 - poprawności, 44, 78, 99
 - ról, 249
 - uprawnień, 249
 - zabrudzenia, 67
 - kontroler, 88, 185, 313, 455
 - brzegowy, 90, 91
 - ContactController, 285
 - CRUD, 413
 - dodawanie, 98
 - MVC, 132
 - POJO, 89, 97
 - Spring Web MVC, 333
 - tworzenie, 94, 132
 - widoku, 214
 - konwerter, 416
 - komunikatów HTTP, 416
 - kryptografia, 233, 301
- ## L
- LDAP, 207
 - LinkedIn, 317, 423, 425
 - lista
 - ACL, 245
 - biała, 98, 102, 142, 248, 256, 285
 - czarna, 142, 248
 - JSON, 410, 412
 - kontroli dostępu, 259
 - obiektów domeny, 21
 - strażników, 545, 546, 559
 - wysyłkowa, 301, 302, 307, 311
 - XML, 410, 413
 - logika
 - domeny, 73, 78
 - trwałości, 73
 - logowanie, 21, 204, 212, 219, 245, 252, 258
 - strona, 213, 216
 - użytkownika anonimowego, 212
 - z użyciem formularzy, 205
 - Lucene, 443

Ł

łańcuch, 499, 501, 509
 zależności, 26
 łącze mailto, 282, 290
 łączenie automatyczne, 46

M

magazyn, 442
 IMAP, 510, 516
 mapowanie, 64
 Markdown, 336, 337
 maszyna wirtualna, 384
 Maven, 172, 192, 246, 348, 349, 385, 417, 569
 Jetty, 282, 290, 328, 571, 572, 573
 konfiguracja, *Patrz:* konfiguracja Maven
 test integracyjny, 368, 370, 374
 zależności, 353
 Maven 3, 417
 Maven Build Helper, 348, 349
 Maven Failsafe, 348, 351
 Maven Surefire, 351
 MD5, 235
 mechanizm DI, 24, 28, 35
 usunięcie z kodu klienckiego, 26
 media, 387
 metauprawnienia, 272
 metoda
 @InitBinder, 103, 141, 150, 304
 AclImpl.isGranted, 266
 addArticleNode, 452
 addPageNode, 452
 addVolunteer, 166
 ArticleServiceImpl.postComment, 332
 atomowa, 529
 attemptResetAfter, 529
 authenticate, 206
 bind, 189
 bindAndValidate, 189
 convert, 464
 create, 75, 77, 153, 401, 450
 createEmail, 293
 createOrUpdate, 448
 createPackage, 420
 createTicket, 496
 CRUD, 75, 392, 450
 DELETE, 256
 doPostExecute, 187

doPostExecute, 187
 dotMobi, 126
 encodePassword, 235
 execute, 187, 525, 529
 fabryki, 126, 187
 findOpenTicketStatus, 505
 GET, 256
 GET HTTP, 118
 getAll, 75
 getAllArticles, 333
 getBean, 41
 getCurrentSession, 68
 getForums, 248
 getFullName, 66
 getId, 152
 getListAsJson, 412
 getModule, 408
 getMostSpecificMethod, 560
 getPackages, 420
 getPage, 448
 getState, 529
 getUserInfo, 169
 gistOperations, 425
 HTTP, 256
 invoke, 546
 Java ochrona przed nieautoryzowanym
 dostępem, 244, 245
 JavaMailSender.send, 293
 JcrDaoSupport.setSessionFactory, 450
 list, 95
 load, 341
 loadUserByUsername, 207, 230
 MapSqlParameterSource.addValue, 57
 matches, 562
 mDot, 126
 member, 95
 mostkowana, 560
 mostkująca, 560
 nazwa przekształcanie na zapytanie, 84
 obsługująca żądania, 95
 odczytująca, 395
 parse, 564
 parseAnnotation, 561
 password, 228
 POST, 256
 postRegistrationForm, 140
 preHandle, 121
 print, 341

metoda

PUT, 256
 ReflectionTestUtils.setField, 370
 registerAccount, 155
 reject, 145
 rejectValue, 145
 repoOperations, 425
 reset, 529
 resetForm, 189
 sendEmail, 298
 sendNotificationEmail, 323
 Session.save, 67
 setAccountDao, 36
 setupForm, 188
 sygnatura, 95
 szablon, 450
 szablonu, 524
 toString, 131
 transakcyjna, 68
 update, 401
 username, 228
 userOperations, 425
 ustawiająca, 24, 36
 validate, 189
 wyszukująca, 84, 392, 401, 502
 zapytania, 83
 Mockito, 354
 model, 90
 artykułów, 446
 domeny, 97
 REST, 128
 modularność, 21, 22
 moduł, 395, 396, 397, 406
 domeny, 396
 klienta, 396
 Markdown.Sanitizer.Modified, 341
 WWW, 396
 MongoDB, 442, 444, 466, 467
 MVC, 22, 88, 161
 MyBatis, 21
 MySQL, 149, 157, 359, 378

N

nagłówek

AMQP, 506
 komunikatu, 506
 SI, 506

User-Agent, *Patrz:* User-Agent
 żądania, 118, 119, 311
 Neo4j, 387, 389, 393
 Nexus, 385, 417
 niedopasowanie
 strukturalne, 66
 zachowania, 66
 Nygard Michael, 521

O

OAuth 2, 428
 obiekt
 AccountController, 135
 AccountService, 36, 41
 BeanNameViewResolver, 114, 115
 BindingResult, 145
 buforowanie, 41
 DataSource, 23, 24, 58, 59, 222, 358
 deszeregujący, 415
 DispatcherServlet, *Patrz:* DispatcherServlet
 domeny, 21, 28, 149, 320, 447
 Account, 151
 autoryzacja, 245
 bezpieczeństwo, 245
 jako ziarno formularza, 97, 98
 zabezpieczenia, 267, 269
 dostawcy, 206
 dostępu do danych, 23, 153, *Patrz też:* DAO
 error, 187, 188
 Errors, 154
 Event, 187
 form, 188
 formularza, 189
 HandlerAdapter, 112
 HandlerExceptionResolver, 112
 HandlerInterceptor, 110, 121
 InternalResourceView, 113
 InternalResourceViewResolver, 113, 174
 Java, 63
 odwzorowanie na graf, 391
 java.sql.ResultSet, 21
 javax.sql.DataSource, 37
 JdbcAccountDao, 41
 JdbcMutableAclService, 274
 klucza, 57
 konfigurowanie, 31
 LocaleResolver, 116

MongoTemplate, 469
 MultipartResolver, 116
 nietrwały, 67
 odłączony, 67
 org.springframework.security.core.userdetails.
 ↳ User, 225
 osłonowy, 406
 POJO, 90, 185, 283
 połączenie, 63
 ProviderManager, 206
 przechwytyjący, 110, 553, 566
 porada, 543
 przekształcamie w encje bazy danych, 63
 RequestContext, 186
 RequestToViewNameTranslator, 115
 rozpoznający, 116
 RSS Channel, 313
 Session, 68, 333
 SessionFactory, 72, 158
 SessionFactory Hibernate, 40
 success, 187
 szeregujący JAXB2, 415
 ThemeResolver, 116
 trwały, 67
 UserMessage, 285, 287
 usunięty, 67
 użytkownika, 233
 ViewResolver, 112, 115
 WebDataBinder, 141
 wykonujący przepływ, 177
 XmlViewResolver, 114, 115
 zaślepkowy, 448
 odwzorowanie, 63
 obiektowo-XML, *Patrz:* OXM
 Object/XML, *Patrz:* OXM
 relacyjno-obiektowe, *Patrz:* ORM
 OGNL, 170
 ogranicznik wywołań, 545
 OID, 264
 OpenID, 207
 OpenJPA, 80, 82
 operacja wsadowa, 83
 org.springframework.web.servlet.mvc.Controller,
 90
 ORM, 21, 53, 62, 79
 OSCache, 443
 OXM, 21, 498

P

PageDown, 336, 337
 pakiet, 397, 405, 419
 duplikowanie, 419
 JSON, 406
 XML, 406
 parametry
 konfiguracyjne, 23
 połączenia, 24
 typu generycznego, 75
 pasek nawigacji, 213
 Pinterest, 317
 PlainTextFilter, 327
 plik
 applicationContext.xml, 33
 applicationContext.xml, 31
 applicationContext-security.xml, 199
 beans-datasource.xml, 362
 beans-datasource-it.xml, 362, 378, 379
 bean-service.xml, 82
 beans-integration.xml, 512
 beans-jcr.xml, 468
 beans-kite.xml, 535, 556
 beans-security.xml, 205, 210, 215, 218, 231,
 235, 258
 beans-security-acl.xml, 269
 beansservice.xml, 535
 beans-service.xml, 294, 300, 310, 339, 378
 beans-web.xml, 135, 138, 142, 213, 214, 218,
 314, 465, 535
 dispatcherServlet-context.xml, 176
 environment.properties, 453
 findExistingPlayer-flow.xml, 189
 JAR, 396
 jetty-env.xml, 246, 571
 JSP, 113, 211
 konfiguracyjny, 31, 33, 52, 569
 bezpieczeństwa, 31
 DAO, 31
 serwletów, 31
 usług, 31
 list.jsp, 96
 loginRequired.jsp, 217
 main-servlet.xml, 123
 member.jsp, 96
 modelu projektu, *Patrz:* POM
 przesyłanie, 116

plik

- registrationForm.jsp, 142, 147
- repository.xml, 446, 453
- spring.schemas, 550
- subhead.jspf, 211, 213, 215, 217, 218, 233, 257
- środowiska .properties, 59
- WAR, 396
- web.xml, 91, 104, 119, 134, 158, 173, 208, 209, 470, 485, 535
 - testowanie integracyjne, 359
- webflowContext.xml, 189, 192, 200
- XML, 35, 44, 51, 246
- XML Schema, 551
- XSD, 550
- poczta, 36
- poczta elektroniczna, 282, 510
- podprzepływ, 162, 167, 183, 194
- POJO, 22, 35, 52, 88, 89, 94, 131, 150, 185, 283, 447
- pole, 46
- POM, 172
- porada, 553
 - bazująca na obiekcie przechwytyjącym, 543
 - z listą strażników, 552, 553
- portlet, 42, 163, 170
- problem niedopasowania strukturalnego, 66
- procedura składowana, 21
- profil, 387
- programowanie
 - aspektowe
 - AOP, *Patrz:* AOP
 - AspectJ, 21
 - sterowane testami, *Patrz:* TDD
- projektowanie dziedziczne, 150
- PropertyPlaceholderConfiguraton, 62
- protokół
 - AMQP, *Patrz:* AMQP
 - Wireless Access Protocol, *Patrz:* WAP
- ProviderManager, 206
- proxy
 - automatyczne, 549, 561, 562, 564
 - bazujący
 - na interfejsach, 564
 - na klasie, 564
- przepływ, 127, 162, 163, 185
 - dane, 168
 - dziedziczenie, 194, 195, 197
 - implementowanie, 178

- rejestr, 176
- rozpoczęcie, 170
- stan, 164
 - action, 164, 166
 - decision, 164, 166
 - dynamiczny, 178
 - dziedziczenie, 195, 197
 - end, 164, 167
 - predefiniowany, 178
 - przejście, 167
 - subflow, 164, 167
 - view, 164, 170, 190, 194, 196
- wyszukiwania, 164
- zabezpieczanie, 197, 200
- zarządzanie, 168

przestrzeń nazw, 61

- AOP, 548
- beans, 35, 270
- c, 44, 45
- context, 47, 48
- handler, 553, 554
- jdbc, 358, 376
- jee, 61
- konstruktorów Spring, 59
- mvc, 135, 214
- MVC, 387
- p, 43, 44, 45
- security, 270
- Spring Modules jcr, 453
- Spring Security, 205
- task, 300, 327
- util, 72
 - własna, 549, 550, 556, 557
- punkt integracji, *Patrz:* integracja punkt

R

- RabbitMQ, 475, 491, 492, 493, 498, 510, 512
- rejestr przepływów, *Patrz:* przepływ rejestr
- relacja, 63, 396, 399
 - encja, 395, 400
 - jeden do wielu, 394
 - łączenie za pomocą typu, 46
 - typ, 395
 - wiele do wielu, 394, 398
 - właściwość, 395
- ReloadableResourceBundleMessageSource, 137
- repozytorium treści, 444, 445, 446, 453
- RequestContext, 186

RequestToViewNameTranslator, 116
 RequireJS, 336
 REST, 128, 405, 417, 422, 482, 483
 RestTemplate, 425, 426, 488, 489
 RFI, 99
 Rhino, 336
 ROME, 311
 router, 509
 routing, 110
 RSS, 115, 311, 312

S

schemat
 beans, 31
 context, 47
 konfiguracji, 31
 SDJ, 484
 SDR, 484, 487
 ServletContextAware, 464
 serwer
 ciągłej integracji, 417
 pocztowy Gmail, 290
 serwlet, 88, 91, 104, 113, 173, 333
 deskryptor instalacji, 43
 eksportera, 485
 filtr, 121
 filtra, 209, 210
 filtrów bezpieczeństwa, 205
 SHA-1, 235
 SHA-256, 235
 SI, 475, 491, 498, 510, 512, 516
 filtr, *Patrz:* SI punkt końcowy
 kanał, *Patrz:* kanał
 punkt końcowy, 475, 476, 501
 wstrzykiwanie zależności, 475
 SID, 262, 264
 sieć
 społecznościowa, 317, 423
 WWW, 20, 22
 zawodowa, 317
 silnik
 dostarczania artykułów, 441, 454, 466
 FreeMarker, 296
 komentarzy, 317, 318, 326, 327, 336
 przepływów pracy, 162
 SpEL, 205
 szablonów, 296, 519
 silnik JavaScript Rhino, 144
 SimpleUrlHandlerMapping, 109
 SitePreference, 122
 SitePreferenceHandler, 122
 SitePreferenceHandlerInterceptor, 123, 126
 SitePreferenceRepository, 122
 SiteSwitcherHandlerInterceptor, 126
 skrypt
 międzywytynowy, *Patrz:* XSS
 SQL testowanie integracyjne, 359
 Skybase, 385
 słowo kluczowe, 119
 new, 36
 volatile, 529
 SMTP, 516
 SOAP, 36, 483
 sortowanie, 83
 SpEL, 169, 170, 212, 246, 251, 256, 501, 505
 Spring 2, 21
 Spring 2.0, 46
 Spring 2.5, 46
 Spring 3, 21, 90
 Spring 3.0, 128
 Spring 3.1, 45, 387
 Spring AMQP, 475, 491, 498
 Spring AOP, 542
 Spring Data JPA, 54, 82, 480, 482, *Patrz:* SDJ
 Spring Data MongoDB, 466, 469
 Spring Data Neo4, 391
 Spring Data Neo4j, 387, 389, 393
 Spring Data REST, 475, 482, 483, *Patrz:* SDR
 Spring Expression Language, 252, *Patrz:* SpEL
 Spring Faces, 127
 Spring HATEOAS, 475, 482, 483, 487
 Spring Integration, *Patrz:* SI
 Spring JavaScript, 127
 Spring JDBC, 55
 Spring JdbcTemplate, 149
 Spring JMX, 537
 Spring Mobile, 116, 117
 konfigurowanie, 119
 Spring Modules, 443
 Spring Modules JCR, 441, 442, 443
 Spring MVC, 172, 173
 Spring Rabbit, 475, 491
 Spring RestTemplate, 417
 Spring Roo, 150

- Spring Security, 127, 197, 199, 277, 435, 437
 - konfigurowanie, 204
 - Spring Security 3, 203, 252, 255, 258
 - Spring Social, 422, 423, 424, 430
 - Spring Task Execution, 296, 297
 - Spring Tool Suite, 569
 - Spring Validation, 191
 - Spring Web Flow, *Patrz:* SWF
 - Spring Web MVC, 87, 89, 97, 98, 128, 140, 282, 283, 311, 327, 405
 - aplikacja, 91
 - architektura, 90
 - konfiguracja, 104, 106
 - springSoccer, 174
 - SpringSource Enterprise Bundle Repository, *Patrz:* EBR
 - Spring-Source Tool Suite, 549
 - SQL, 55
 - SSO, 128
 - Stack Overflow, 336
 - StandardSitePreferenceHandler, 122
 - strażnik, 525, 545, 553
 - strona podziękowania, 285
 - stronicowanie, 83
 - struktura domeny
 - bazy danych, 66
 - Java, 66
 - strumień, 312, 475
 - Atom, 115
 - elementy, 313
 - RSS, 115
 - Struts 1, 97, 98
 - subskrypcja, 302, 315
 - prywatność, 306, 310
 - Subversion, 385
 - SWF, 127, 162, 172, 178, 185
 - konfiguracja, 176
 - konfigurowanie, 189
 - zmienne, 169
 - conversationScope, 170
 - flashScope, 170
 - flowScope, 170
 - requestScope, 170
 - viewScope, 170
 - zakres, 170
 - system
 - bazujący na hipermediach, 483
 - integracja, 475
 - kontroli wersji, 385
 - obsługi działu wsparcia, 474, 479, 480, 499, 507, 510
 - pobierający, 417
 - zarządzania konfiguracją, *Patrz:* konfiguracja zarządzania
 - zarządzania treścią, *Patrz:* CMS
 - szyna komunikatów, 493, 503, 507, 510
- ## T
- tabela, 63
 - acl_object_identity, 263
 - acl_sid, 262
 - ról, 263
 - tablica AccountDao, 48
 - taniec OAuth 2, 428, 429
 - TDD, 343, 374
 - tekst
 - filtrowanie, 324, 327
 - sformatowany, 318, 340
 - test, 20
 - bezpieczeństwa, 318
 - ignorowanie, 374, 375
 - jednostkowy, 24
 - TestNG, 348
 - testowanie, 347, 384
 - filtra, 343
 - integracyjne, 343, 347, 349, 351, 353, 354, 355, 356
 - bazy danych, 376
 - konfiguracja ziarna, 358
 - o ograniczonym czasie wykonania, 371
 - obsługa sytuacji wyjątkowych, 368
 - plik web.xml, 359
 - skrypt SQL, 359
 - ścieżki pozytywnej, 360, 364, 366, 368, 374
 - weryfikowanie wydajności, 370
 - jednostkowe, 22, 343, 351
 - TransactionTemplate, 524
 - transakcja, 21, 44, 54, 62, 67, 127, 403, 453
 - DAO, 450
 - menedżer, 73, 479
 - obsługa, 21
 - rozproszona, 478
 - zarządzanie, 31, 68
 - deklaratywne, 70
 - programowe, 318

trwałość, 44, 54, 67, 73, 140, 151, 287, 331
 cykl życia, 67
 dostawca, 80
 logika, *Patrz:* logika trwałości
 Twitter, 423, 425

U

udostępnianie filmów i zdjęć, 317
 Unified EL, 170
 uprawnienia, 265, 266, *Patrz też:* użytkownik
 uprawnienia
 specjalne, 272
 zestaw, 266
 urządzenie
 mobilne, 116, 118
 wykrywanie, 116
 User Agent Switcher, 118
 User-Agent, 118
 UserDetailsService, 207
 userMessage, 286
 UserMessage, 285, 287
 usługa, 90
 aktywator, 476
 ArticleService, 332
 BlogPostService, 332
 budowania przepływu, 176
 CommentService, 332
 dostarczania artykułów, 327
 FeedBurner, 302
 InMemoryDaoImpl, 220
 integracja, 476
 JdbcDaoImpl, 220
 kont użytkownika, 154
 konwersji, 416
 obliczeniowa, 385
 pocztowa SMTP, 244
 powolna, 522
 ProductService, 332
 REST, *Patrz:* REST
 sieciowa, 44, 405, 482, 483, 503
 punkt końcowy, 410, 418, 420
 REST, 128, 484, 491
 zaczep, 427, 432
 uwierzytelnianie, 128, 205, 226, 243, 306, 453,
Patrz też: autoryzacja
 automatyczne, 239
 menedżer, 206
 OpenID, 128

użytkownik, 225
 autoryzacja, *Patrz:* autoryzacja
 dane rejestracji, 149
 hasło, 157
 identyfikator jako wartość początkowa, 236
 konta modelowanie, 131
 logowanie, *Patrz:* logowanie
 nazwa, 128, 154, 358
 rejestracja, 236
 rola, 248, 249, 279
 strona danych, 433
 uprawnienia, 248, 249, 265
 definiowanie, 261
 uwierzytelnianie, *Patrz:* uwierzytelnianie
 wylogowywanie, *Patrz:* wylogowywanie
 zapamiętanie, 206, 219
 zapamiętywanie, 245, 252, 258

V

Velocity, 165, 290, 519
 velocityEngine, 295
 VelocityEngine, 294
 ViewResolver, 136
 VTL, *Patrz:* Velocity

W

WAP, 119
 warstwa
 biznesowa, 88
 dostępu do danych, 54, 80
 integracji, 476
 usług, 36, 69, 401
 wątek, 296
 główny, 297
 widoczność, 529
 wysyłania wiadomości, 297
 WebContentInterceptor, 110
 wiadomość e-mail, 283, 290, 293, 294, 310, 323,
 510, 512, 516
 widok, 88, 90, 99
 forward:, 114
 FreeMarker, 115
 JSP, 456, 458
 kontroler, 214
 nadrzędny, 96
 nazwa, 95, 215
 logiczna, 165

widok

- podrzędny, 96
- prefiksów nazwy, 114
- redirect:, 114
- strategia rozpoznawania, 113
- strona
 - podziękowania, 135
 - rejestracji, 133
 - tworzenie, 133
- technologia, 165
- Velocity, 115
- wyodrębnianie napisów, 137, 139
- XSLT, 115
- wyświetlanie błędów, 147

wiersz, 63

Wikipedia, 317

Wireless Universal Resource FiLe, *Patrz:* WURFL

witryna, 281

- przełączanie, 116
- uprawnienia administratora, 259, 260, 261, 272
- w poddomenie, 126
- właściwości, 116
- właściwościami, 122

właściwość, 63

- acceptTerms, 131
- csvResource, 32
- email, 227
- enabled, 150
- firstName, 227
- fullName, 152, 227
- indeksowana, 391
- lastName, 227
- marketingOk, 131
- nazwa, 415
- password, 153
- relacji, *Patrz:* relacja właściwość
- repository.conf, 453
- repository.dir, 453

wrodzona niezgodność, 63

wstrzykiwanie

- metod fabryki, 30
- serwletów filtrów, 210
- z użyciem konstruktora, 30, 37, 44
- z użyciem metody ustawiającej, 37
- za pomocą settera, 30
- zależności, 20, 23, 24, 89, 102, 345, 347
 - pionowe, 475, 476
 - poziome, 475
 - SI, 475
- ziarna, *Patrz:* ziarno wstrzykiwanie

WURFL, 119

WurflDeviceResolver, 119, 121

wydajność, 371

wyjątek

- BeanCreationException, 48
- biznesowy, 185
- DuplicateCIEException, 403
- hierarchia standardowa, 21
- IllegalStateException, 43
- nieobsłużony, 49
- ResourceNotFoundException, 367
- translacja, 450

wylogowywanie, 204, 212, 219, 245, 252, 258

wyrażenie regularne, 146

wywołanie zwrotne, 524

wzorzec

- bezpiecznik, 521, 522
- data access object, 74
- singleton, 40
- szablon, 523
- szablonu, 524
- szablonu metody, 450

X

XML, 405, 406, 410

XML Schema, 551, *Patrz:* XSD

XMLBeans, 21

XSD, 549

XSLT, 165

XSS, 336

XStream, 21

Y

YouTube, 317

Z

zadanie

- hermetyzacja, 21
- przekrojowe, 21

zakładka społecznościowa, 317

zapisu opóźnianie transakcyjne, 67

zapytanie, 55, 57, 63

CRUD, 67

JPA, 63

odwzorowanie, 63

ogólnego przeznaczenia, 75

zasób, 61
 messages.properties, 142
 tworzenie zestawu na komunikaty, 137
 ValidationMessages.properties, 142

zdarzenie, 165
 error, 189
 ID, 165
 success, 188, 189

ziarno, 20, 106
 accountDao, 36
 cykl życia, 42
 definiowanie, 31
 dekoracja, 42
 diagram zależności, 569
 formularza, 97, 99, 131, 140, 149, 150, 288, 321
 implementacja, 36
 konfiguracja, 28, 42
 konkretyzacja, 42
 nazwa domyślna, 50
 PropertyPlaceholderConfigurer, 38
 testowanie integracyjne, 358
 UserMessage, 285
 usługowe, 88, 287, 331
 wstrzykiwanie, 41
 zakres, 39, 40
 globalSession, 39, 42
 prototype, 39, 41, 42
 request, 39, 42

session, 39, 42
 singleton, 39, 40, 41, 42
 Zkybase, 385, 389, 419, 427, 428, 430

Ź

źródło
 komunikatów, 138
 uwierzytelniania, 206

Ż

żądanie
 GET, 99
 handler, 90
 HTTP, 42, 88, 117, 170
 HTTP GET, 420
 informacji, *Patrz:* RFI
 JNDI, 60
 odwzorowanie, 44
 POST, 99, 132
 przekształcenie na nazwę, 215
 tworzenia zgłoszeń, 499
 utworzenia pakietu, 418
 uwierzytelniania, 206
 WWW, 43
 żeton dostępu, 428, 429, 430, 435, 437, 439

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

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 wykorzystywać 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>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-246-8184-6



9 788324 681846

Cena: 89,00 zł

Informatyka w najlepszym wydaniu