



Twoja przepustka do świata
Spring Framework!

J Sharma, Ashish Sarin

SPRING FRAMEWORK

Wprowadzenie do tworzenia aplikacji

Wydanie II

Helion 

Tytuł oryginału: Getting started with Spring Framework

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-0932-6

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

© Jyotsna Sharma, Ashish Sarin 2015.

Polish edition copyright © 2015 by Helion S.A.
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 biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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)

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

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

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

Podstawy tworzenia aplikacji we frameworku Spring

Niniejsza książka stanowi praktyczny przewodnik umożliwiający Czytelnikowi rozpoczęcie tworzenia aplikacji za pomocą frameworku Spring. Jest przeznaczona dla programistów Javy, którzy mają niewielkie lub wręcz żadne doświadczenie w pracy ze Springiem. Wydanie drugie książki zawiera nowe rozdziały, których tematem są **Spring Web MVC**, **usługi sieciowe typu RESTful** oraz **Spring Security**.

Dołączone do książki przykładowe fragmenty kodu zostały oparte na **Spring 4.0**. Archiwum zawierające ponad **60 przykładowych projektów** omówionych w książce możesz pobrać ze strony <http://helion.pl/ksiazki/spfrwp.htm>.

- Rozdział 1.** Wprowadzenie do frameworku Spring
- Rozdział 2.** Podstawy frameworku Spring
- Rozdział 3.** Konfiguracja komponentów
- Rozdział 4.** Wstrzykiwanie zależności
- Rozdział 5.** Dostosowanie komponentów i ich definicji
- Rozdział 6.** Programowanie oparte na adnotacjach
- Rozdział 7.** Współpraca z bazą danych
- Rozdział 8.** Komunikaty, wiadomości e-mail, asynchroniczne wykonywanie metod i buforowanie w Springu
- Rozdział 9.** Programowanie zorientowane aspektowo
- Rozdział 10.** Podstawy modułu Spring Web MVC
- Rozdział 11.** Weryfikacja i dołączanie danych w Spring Web MVC
- Rozdział 12.** Tworzenie usług sieciowych typu RESTful za pomocą Spring Web MVC
- Rozdział 13.** Więcej o Spring Web MVC: internacjonalizacja, przesyłanie plików i przetwarzanie żądań asynchronicznych
- Rozdział 14.** Zabezpieczanie aplikacji za pomocą Spring Security

Ashish Sarin
J. Sharma

Wprowadzenie	15
Rozdział 1. Wprowadzenie do frameworku Spring	17
1.1. Wprowadzenie	17
1.2. Moduły frameworku Spring	17
1.3. Kontener IoC	18
1.4. Zalety stosowania frameworku Spring	21
Spójne podejście w zakresie zarządzania transakcjami lokalnymi i globalnymi	21
Deklaracyjne zarządzanie transakcjami	23
Zapewnienie bezpieczeństwa	24
JMX (Java Management Extensions)	24
JMS (Java Message Service)	26
Buforowanie	27
1.5. Prosta aplikacja Springa	28
Identyfikacja obiektów aplikacji i ich zależności	28
Utworzenie klas POJO odpowiadających zidentyfikowanym obiektom aplikacji	29
Utworzenie metadanych konfiguracyjnych	30
Utworzenie egzemplarza kontenera Springa	35
Uzyskanie dostępu do obiektów aplikacji z poziomu egzemplarza kontenera Springa	36
1.6. Frameworki zbudowane na bazie Springa	37
1.7. Podsumowanie	38
Rozdział 2. Podstawy frameworku Spring	39
2.1. Wprowadzenie	39
2.2. Reguła projektowa o nazwie „podejście oparte na programowaniu interfejsów”	39
Scenariusz: klasa zależna zawiera odwołanie do konkretnej klasy zależności	39
Scenariusz: klasa zależna zawiera odwołanie do interfejsu implementowanego przez zależność	40
Oferowana przez Spring obsługa podejścia opartego na programowaniu interfejsu	42
2.3. Różne podejścia podczas tworzenia komponentów Springa	44
Utworzenie komponentu za pomocą statycznej metody fabryki	44
Tworzenie komponentu za pomocą metody fabryki egzemplarza	46

2.4. Techniki wstrzykiwania zależności	47
Wstrzykiwanie zależności za pomocą metody typu setter	47
Wstrzykiwanie zależności za pomocą argumentów konstruktora	51
2.5. Zasięg komponentu	54
Singleton	55
Prototyp	63
Wybór właściwego zasięgu dla komponentu	64
2.6. Podsumowanie	64
Rozdział 3. Konfiguracja komponentów	65
3.1. Wprowadzenie	65
3.2. Dziedziczenie definicji komponentu	65
Przykład dziedziczenia definicji komponentu w aplikacji MyBank	65
Co jest dziedziczone?	68
3.3. Dopasowanie argumentu konstruktora	73
Przekazywanie wartości i odwołań do komponentów za pomocą elementu <constructor-arg>	73
Dopasowanie argumentu konstruktora na podstawie typu	74
Dopasowanie argumentu konstruktora na podstawie nazwy argumentu	79
3.4. Konfiguracja różnego typu właściwości komponentu i argumentów konstruktora	82
Wbudowane edytory właściwości	83
Określanie wartości dla różnych typów kolekcji	86
Definiowanie wartości dla tablic	92
Domyślne implementacje kolekcji dla elementów <list>, <set> i <map>	92
3.5. Wbudowane edytory właściwości	93
Edytor CustomCollectionEditor	93
Edytor CustomMapEditor	96
Edytor CustomDateEditor	96
3.6. Rejestracja edytorów właściwości w kontenerze Springa	97
Utworzenie implementacji PropertyEditorRegistrar	97
Konfiguracja klasy CustomEditorConfigurer	98
3.7. Zwięzłe definicje komponentów dzięki użyciu przestrzeni nazw p oraz c	99
Przestrzeń nazw p	99
Przestrzeń nazw c	100

3.8. Schemat util	103
Element <list>	104
Element <map>	105
Element <set>	107
Element <properties>	109
Element <constant>	110
Element <property-path>	111
3.9. Interfejs FactoryBean	113
Aplikacja MyBank — przechowywanie zdarzeń w bazie danych	113
Aplikacja MyBank — przykład użycia interfejsu FactoryBean	115
Uzyskanie dostępu do egzemplarza FactoryBean	118
3.10. Podsumowanie	120

Rozdział 4. Wstrzykiwanie zależności 121

4.1. Wprowadzenie	121
4.2. Komponenty wewnętrzne	121
4.3. Wyrażna kontrola kolejności inicjalizacji procesu za pomocą atrybutu depends-on	123
Aplikacja MyBank — zasugerowane zależności między komponentami	123
Problem niejawnej zależności	124
4.4. Zależności w postaci komponentów o zasięgu singleton i prototypu	129
Zależności komponentu o zasięgu singleton	129
Zależności komponentu o zasięgu prototypu	132
4.5. Pobieranie w komponencie singleton nowego egzemplarza komponentu o zasięgu prototypu	135
Interfejs ApplicationContextAware	136
Element <lookup-method>	138
Element <replaced-method>	142
4.6. Automatyczne wiązanie zależności	145
Wartość byType	146
Wartość constructor	147
Wartość byName	148
Wartości default i no	150
Uniedostępnienie komponentu dla funkcji automatycznego wiązania	150
Ograniczenia automatycznego wiązania	152
4.7. Podsumowanie	153

Rozdział 5. Dostosowanie komponentów i ich definicji	155
5.1. Wprowadzenie	155
5.2. Dostosowanie logiki inicjalizacji i usuwania komponentu	155
Wywołanie przez Spring metody czyszczącej wskazanej	
przez atrybut destroy-method	158
Metoda czyszcząca i komponent o zasięgu prototypu	160
Określenie domyślnych metod inicjalizacji i usuwania obiektu	
dla wszystkich komponentów	160
Interfejsy cyklu życiowego InitializingBean i DisposableBean	161
Adnotacje @PostConstruct i @PreDestroy wprowadzone w specyfikacji JSR 250	161
5.3. Użycie interfejsu BeanPostProcessor do pracy z nowo utworzonym	
egzemplarzem komponentu	163
Przykład użycia interfejsu BeanPostProcessor	
— weryfikacja egzemplarza komponentu	165
Przykład BeanPostProcessor — rozwiązywanie zależności komponentu	168
Zachowanie BeanPostProcessor w przypadku	
komponentów implementujących FactoryBeans	172
RequiredAnnotationBeanPostProcessor	174
DestructionAwareBeanPostProcessor	175
5.4. Modyfikacja definicji komponentu za pomocą BeanFactoryPostProcessor	176
Przykład użycia BeanFactoryPostProcessor	177
Komponent PropertySourcesPlaceholderConfigurer	182
Komponent PropertyOverrideConfigurer	187
5.5. Podsumowanie	190
Rozdział 6. Programowanie oparte na adnotacjach	191
6.1. Wprowadzenie	191
6.2. Identyfikacja komponentów Springa za pomocą adnotacji @Component	191
6.3. Adnotacja @Autowired, czyli automatyczne wiązanie zależności według ich typu	194
6.4. Adnotacja @Qualifier pozwalająca na automatyczne	
wiązanie zależności według ich nazwy	198
6.5. Adnotacje @Inject i @Named zdefiniowane przez specyfikację JSR 330	199
6.6. Zdefiniowana przez JSR 250 adnotacja @Resource	201
6.7. Adnotacje @Scope, @Lazy, @DependsOn i @Primary	202

6.8. Ułatwienie konfiguracji komponentu za pomocą adnotacji @Value	203
6.9. Weryfikacja obiektów za pomocą interfejsu Validator	207
6.10. Określanie ograniczeń za pomocą adnotacji JSR 303	211
Obsługa specyfikacji JSR 303 w Springu	212
6.11. Programowa konfiguracja komponentów Springa za pomocą adnotacji @Configuration i @Bean	216
6.12. Podsumowanie	219
Rozdział 7. Współpraca z bazą danych	221
7.1. Wprowadzenie	221
7.2. Wymagania aplikacji MyBank	221
7.3. Utworzenie aplikacji MyBank opartej na module JDBC	222
Konfiguracja źródła danych	223
Tworzenie obiektów DAO używających klas modułu JDBC	224
7.4. Utworzenie aplikacji MyBank z użyciem Hibernate	231
Konfiguracja SessionFactory	231
Utworzenie klas DAO używających do pracy z bazą danych API Hibernate	232
7.5. Zarządzanie transakcjami za pomocą Springa	233
Wymagania aplikacji MyBank w zakresie zarządzania transakcjami	233
Programowe zarządzanie transakcjami	234
Deklaracyjne zarządzanie transakcjami	238
Oferowana przez Spring obsługa JTA	241
7.6. Podsumowanie	243
Rozdział 8. Komunikaty, wiadomości e-mail, asynchroniczne wykonywanie metod i buforowanie w Springu	245
8.1. Wprowadzenie	245
8.2. Wymagania aplikacji MyBank	245
8.3. Wysyłanie komunikatów JMS	246
Konfiguracja brokera ActiveMQ do działania w trybie osadzonym	247
Konfiguracja JMS ConnectionFactory	247
Wysyłanie komunikatów JMS za pomocą klasy JmsTemplate	248
Wysyłanie komunikatów JMS w transakcji	249

Dynamiczni odbiorcy komunikatów JMS i konfiguracja klasy JmsTemplate	253
Klasa JmsTemplate i konwersja komunikatu	254
8.4. Odbieranie komunikatów JMS	255
Synchroniczne odbieranie komunikatów JMS za pomocą klasy JmsTemplate	255
Asynchroniczne odbieranie komunikatów JMS za pomocą kontenerów odbiorców komunikatów	255
8.5. Wysyłanie wiadomości e-mail	258
8.6. Harmonogram zadań i wykonywanie asynchroniczne	263
Interfejs TaskExecutor	263
Interfejs TaskScheduler	265
Adnotacje @Async i @Scheduled	267
8.7. Buforowanie	269
Konfiguracja interfejsu CacheManager	271
Adnotacje buforowania @Cacheable, @CacheEvict i @CachePut	271
8.8. Aplikacja MyBank	275
8.9. Podsumowanie	277
Rozdział 9. Programowanie zorientowane aspektowo	279
9.1. Wprowadzenie	279
9.2. Prosty przykład AOP	279
9.3. Oferowany przez Spring framework AOP	282
Tworzenie proxy	283
Atrybut expose-proxy	284
9.4. Wyrażenia punktu przecięcia	286
Adnotacja @Pointcut	286
Desygatory execution i args	287
Desygator bean	291
Desygatory oparte na adnotacjach	292
9.5. Typy rad	293
Rada before	293
Rada after returning	294
Rada after throwing	295
Rada after	296
Rada around	297

9.6. Spring AOP, czyli styl schematu XML	298
Konfiguracja aspektu AOP	299
Konfiguracja rady	300
Powiązanie wyrażenia punktu przecięcia z radą	301
9.7. Podsumowanie	302
Rozdział 10. Podstawy modułu Spring Web MVC	303
10.1. Wprowadzenie	303
10.2. Struktura katalogów przykładowych projektów	303
10.3. Poznajemy przykładową aplikację	304
Plik HelloWorldController.java, czyli klasa kontrolera aplikacji sieciowej	
Hello World	305
Plik helloworld.js, czyli strona JSP wyświetlająca komunikat	306
Plik myapp-config.xml, czyli plik XML kontekstu aplikacji sieciowej	307
Plik web.xml, czyli deskryptor wdrożenia aplikacji sieciowej	308
10.4. Serwlet DispatcherServlet	311
Uzyskanie dostępu do obiektów ServletContext i ServletConfig	313
10.5. Opracowanie kontrolerów za pomocą adnotacji @Controller i @RequestMapping	314
Opracowanie aplikacji sieciowej za pomocą kontrolera oznaczonego adnotacją	314
10.6. Wymagania aplikacji sieciowej MyBank	317
10.7. Adnotacje Spring Web MVC: @RequestMapping i @RequestParam	318
Mapowanie żądań na kontrolery lub metody kontrolerów	
za pomocą adnotacji @RequestMapping	318
Argumenty metod oznaczonych adnotacją @RequestMapping	324
Typy wartości zwrotnych metod oznaczonych adnotacją @RequestMapping	325
Przekazanie parametrów żądania do metod kontrolera z użyciem adnotacji	
@RequestParam	326
10.8. Weryfikacja	330
10.9. Obsługa wyjątków za pomocą adnotacji @ExceptionHandler	333
10.10. Wczytanie głównego pliku XML kontekstu aplikacji sieciowej	335
10.11. Podsumowanie	335

Rozdział 11. Weryfikacja i dołączanie danych w Spring Web MVC	337
11.1. Wprowadzenie	337
11.2. Dodawanie i pobieranie atrybutów modelu za pomocą adnotacji @ModelAttribute	337
Dodanie atrybutów modelu za pomocą adnotacji @ModelAttribute stosowane na poziomie metody	338
Pobieranie atrybutów modelu za pomocą adnotacji @ModelAttribute	342
Przetwarzanie żądania i metody oznaczone adnotacją @ModelAttribute	343
Zachowanie argumentów metody oznaczonych adnotacją @ModelAttribute	345
Obiekt RequestToViewNameTranslator	346
11.3. Buforowanie atrybutów modelu za pomocą adnotacji @SessionAttributes	347
11.4. Dołączanie danych	350
WebDataBinder, czyli dołączanie danych dla parametrów żądania sieciowego	352
Konfiguracja egzemplarza WebDataBinder	353
Włączenie lub wyłączenie poszczególnych właściwości w procesie dołączania danych	358
Analiza błędów dołączania danych i weryfikacji przechowywanych przez obiekt BindingResult	361
11.5. Weryfikacja w Springu	362
Weryfikacja atrybutów modelu za pomocą oferowanego przez Spring interfejsu Validator	363
Zdefiniowanie ograniczeń za pomocą adnotacji JSR 303	366
Weryfikacja obiektu używającego adnotacji JSR 303	367
11.6. Biblioteka znaczników form	371
Obsługa HTML5 przez bibliotekę form	372
11.7. Podsumowanie	374
Rozdział 12. Tworzenie usług sieciowych typu RESTful za pomocą Spring Web MVC	375
12.1. Wprowadzenie	375
12.2. Usługa sieciowa do obsługi lokat	376
12.3. Implementacja usługi sieciowej w stylu RESTful za pomocą Spring Web MVC	376
Format JSON	377
Implementacja usługi sieciowej FixedDepositWS	379

12.4. Uzyskanie dostępu do usługi sieciowej w stylu RESTful za pomocą klasy RestTemplate	386
12.5. Użycie HttpMessageConverter do konwersji obiektów Javy na żądania lub odpowiedzi HTTP i na odwrót	394
12.6. Adnotacje @PathVariable i @MatrixVariable	395
12.7. Podsumowanie	399

Rozdział 13. *Więcej o Spring Web MVC: internacjonalizacja, przesyłanie plików i przetwarzanie żądań asynchronicznych 401*

13.1. Wprowadzenie	401
13.2. Przetwarzanie żądań za pomocą procedur obsługi interceptorów	401
Implementacja i konfiguracja procedury obsługi interceptorów	401
13.3. Internacjonalizacja za pomocą paczek zasobów	404
Wymagania aplikacji sieciowej MyBank	404
Internacjonalizacja i lokalizacja aplikacji sieciowej MyBank	404
13.4. Asynchroniczne przetwarzanie żądań	407
Konfiguracja asynchronicznego przetwarzania żądań	407
Zwrot obiektu Callable przez metodę oznaczoną adnotacją @RequestMapping	408
Zwrot obiektu DeferredResult przez metodę oznaczoną adnotacją @RequestMapping	409
Ustawienie domyślnego limitu czasu	416
Przechwytywanie żądań asynchronicznych	417
13.5. Obsługa konwersji typu i formatowania	417
Utworzenie własnej implementacji interfejsu Converter	418
Konfiguracja i użycie własnej implementacji interfejsu Converter	418
Tworzenie własnej implementacji Formatter	420
Konfiguracja własnej implementacji Formatter	422
Utworzenie implementacji AnnotationFormatterFactory do formatowania jedynie właściwości oznaczonych adnotacją @AmountFormat	422
Konfiguracja implementacji AnnotationFormatterFactory	423
13.6. Obsługa przekazywania plików w Spring Web MVC	425
Przekazywanie plików za pomocą implementacji CommonsMultipartResolver	425
Przekazywanie plików za pomocą StandardServletMultipartResolver	427
13.7. Podsumowanie	428

Rozdział 14. Zabezpieczanie aplikacji za pomocą Spring Security	429
14.1. Wprowadzenie	429
14.2. Wymagania aplikacji sieciowej MyBank w zakresie bezpieczeństwa	429
14.3. Zabezpieczanie aplikacji MyBank za pomocą frameworku Spring Security	431
Konfiguracja zabezpieczeń dla żądań sieciowych	431
Konfiguracja uwierzytelniania	434
Zabezpieczenie zawartości strony JSP przez użycie oferowanej przez Spring Security biblioteki znaczników	435
Zabezpieczanie metod	436
14.4. Zabezpieczanie egzemplarzy FixedDepositDetails za pomocą modułu ACL	440
Wdrożenie i użycie projektu ch14-bankapp-db-security	440
Tabele bazy danych przeznaczone do przechowywania informacji o listach ACL i użytkownikach	442
Uwierzytelnianie użytkownika	446
Zabezpieczenie żądań sieciowych	447
Konfiguracja JdbcMutableAclService	449
Konfiguracja zabezpieczeń na poziomie metody	452
Bezpieczeństwo egzemplarza obiektu domeny	453
Programowe zarządzanie informacjami ACL	455
Egzemplarz MutableAcl i bezpieczeństwo	458
14.5. Podsumowanie	459
 Dodatek A Import i wdrożenie przykładowych projektów w Eclipse IDE (lub IntelliJ IDEA)	 461
A.1. Przygotowanie środowiska pracy	461
A.2. Import przykładowego projektu do Eclipse IDE (lub IntelliJ IDEA)	461
Import przykładowego projektu	462
Konfiguracja zmiennej M2_REPO w Eclipse IDE	463
A.3. Konfiguracja Eclipse IDE do pracy z serwerem Tomcat 7	463
A.4. Wdrożenie projektu sieciowego w serwerze Tomcat 7	465
Uruchomienie serwera Tomcat 7 w trybie osadzonym	465
 Skorowidz	 467

Wprowadzenie

Jak korzystać z tej książki?

Pobranie przykładowych fragmentów kodu

Do książki dołączono wiele przykładowych projektów, które możesz pobrać ze strony <http://helion.pl/ksiazki/spfrwp.htm>. Znajdziesz tam archiwum w formacie ZIP zawierające projekty omówione w książce.

Import przykładowych projektów do środowiska IDE Eclipse lub IntelliJ IDEA

Jeżeli w trakcie lektury natkniesz się na polecenie `import Rozdział<numer-rozdziału>/<nazwa->projektu`, wówczas wskazany projekt powinienś zimportować do Eclipse lub IntelliJ IDEA (bądź też innego używanego przez Ciebie środowiska IDE). Do utworzenia przykładowych projektów wykorzystano narzędzie Maven 3.x, plik *pom.xml* znajdziesz wewnątrz każdego projektu. Odpowiedni plik *pom.xml* znajduje się także w katalogu głównym archiwum kodu źródłowego i pozwala na kompilację wszystkich projektów.

Informacje szczegółowe dotyczące importu i uruchamiania przykładowych projektów znajdziesz w dodatku A.

Odwwołanie do przykładowego fragmentu kodu

Praktycznie każdy listing zawiera nazwę przykładowego projektu, z którego pochodzi (do tego użyto etykiety Projekt) oraz nazwę pliku źródłowego (etykieta Położenie pliku). Jeżeli listing nie zawiera wymienionych etykiet, wtedy można przyjąć założenie, że przedstawiony kod nie znajduje się w dołączonym archiwum przykładowych projektów. Tego rodzaju kod jest użyty jedynie w celu wyjaśnienia omawianego zagadnienia.

Konwencje typograficzne zastosowane w książce

Pochylenie użyto w celu podkreślenia ważnych pojęć.

Czcionka **LettrGoth12EU** została użyta w listingach i kodzie Javy, a także do przedstawienia szczegółów konfiguracji w plikach XML i właściwości.

Pogrubiona czcionka **LettrGoth12EU** została użyta w listingach, aby podkreślić wagę pewnych fragmentów kodu bądź konfiguracji.



Taka ramka zawiera ważną uwagę lub koncepcję.

Opinie i pytania

Oficjalnym miejscem do wyrażania opinii dotyczących książki lub zadawania pytań autorom jest forum grupy Google dostępne pod adresem <https://groups.google.com/forum/#!forum/getting-started-with-spring-framework>.

O autorach

Ashish Sarin zdobył tytuł Sun Certified Enterprise Architect i posiada ponad 14-letnie doświadczenie w tworzeniu aplikacji. Napisał książki *Spring Roo 1.1 Cookbook* (wydana przez Pack Publishing) oraz *Portlets in Action* (wydana przez Manning Publications).

J. Sharma to freelancer, programista Javy z ogromnym doświadczeniem w tworzeniu aplikacji Spring.

Rozdział 1. Wprowadzenie do frameworku Spring

1.1. Wprowadzenie

W trakcie tradycyjnego procesu tworzenia aplikacji korporacyjnej w języku programowania Java to programista jest odpowiedzialny za opracowanie aplikacji o doskonałej strukturze oraz łatwej do konserwacji i przetestowania. Programiści wykorzystują więc niezliczone ilości wzorców projektowych, aby sprostać niebiznesowym wymaganiom aplikacji. Skutkiem jest nie tylko mniejsza produktywność programistów, ale niestety także niższa jakość tworzonych aplikacji.

Spring Framework (lub w skrócie Spring) to opracowany przez SpringSource (<http://spring.io/>) i udostępniony jako open source framework aplikacji, który ułatwia tworzenie aplikacji korporacyjnych w Javie. Ten framework dostarcza infrastrukturę pomagającą w tworzeniu aplikacji o doskonałej strukturze oraz łatwych do późniejszej konserwacji i do przetestowania. Korzystając ze Springa, programista musi koncentrować się jedynie na opracowaniu logiki biznesowej aplikacji, co bezpośrednio przekłada się na jego produktywność. Frameworku Spring można używać do tworzenia samodzielnych aplikacji Javy, aplikacji sieciowych, appletów, a także wszelkich innych rodzajów aplikacji Javy.

Na początku rozdziału znajdziesz krótkie wprowadzenie do modułów Springa, a także poznasz ich zalety. Sercem Springa jest kontener IoC (ang. *Inversion of Control* — odwrócenie sterowania), który oferuje funkcję wstrzykiwania zależności (ang. *dependency injection*, DI). W tym rozdziale omówimy oferowane przez Spring funkcje DI i IoC oraz pokażemy, jak za pomocą Springa opracować samodzielną aplikację Javy. Pod koniec rozdziału przyjrzymy się pewnym projektom SpringSource opartym na Springu. Warto w tym miejscu dodać, że materiał przedstawiony w tym rozdziale dostarcza solidnych podstaw niezbędnych w kolejnych rozdziałach, w których bardziej zagłębimy się we frameworku Spring.



W tej książce funkcje frameworku Spring są omawiane na przykładzie aplikacji o nazwie *MyBank*.

1.2. Moduły frameworku Spring

Spring składa się z wielu modułów pogrupowanych na podstawie ich przeznaczenia w trakcie procesu tworzenia aplikacji. W tabeli 1.1 zostały wymienione różne grupy modułów, które znajdziesz w Springu.

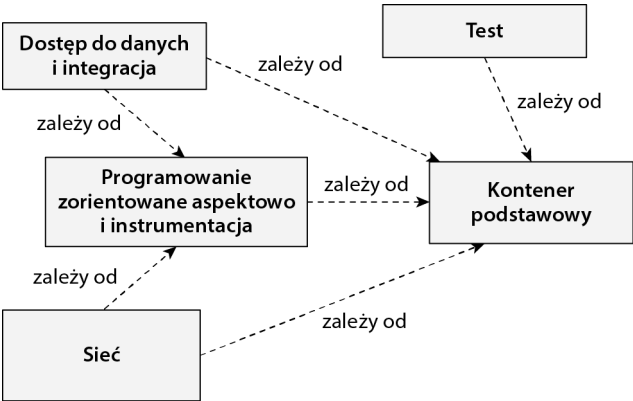
Przeglądając zawartość powyższej tabeli, możesz się przekonać, że Spring obsługuje każdy aspekt tworzenia aplikacji. Dlatego też można go wykorzystać do budowy aplikacji sieciowych, uzyskania dostępu do baz danych, zarządzania transakcjami, tworzenia testów jednostkowych i testów integracji itd. Moduły Springa zostały zaprojektowane w taki sposób, aby programista musiał dołączać w aplikacji *jedynie* niezbędne moduły. Na przykład wykorzystanie oferowanej przez Spring funkcji wstrzykiwania zależności wymaga jedynie dodania modułów znajdujących się w grupie *Kontener podstawowy*. W trakcie lektury książki poznasz szczegóły niektórych modułów dostarczanych przez Spring, a także przykłady ich użycia w tworzonych aplikacjach.

Tabela 1.1. Grupy modułów oferowane przez framework Spring

Grupa modułów	Opis
Kontener podstawowy	Zawiera moduły stanowiące podstawę dla Springu. Moduły znajdujące się w tej grupie dostarczają implementacje funkcji DI i kontenera IoC.
Programowanie zorientowane aspektowo i instrumentacja	Ta grupa zawiera moduły zapewniające obsługę programowania zorientowanego aspektowo (ang. <i>aspect-oriented programming</i>) oraz instrumentacji klas.
Dostęp do danych i integracja	Tutaj znajdziesz moduły ułatwiające współpracę z bazami danych, a także dostawców komunikatów. Ponadto omawiana grupa zawiera moduły przeznaczone do obsługi pragmatycznego i deklaratywnego zarządzania transakcjami oraz implementacje mapowania obiektowego i XML, takie jak JAXB i Castor.
Sieć	Ta grupa zawiera moduły ułatwiające tworzenie aplikacji sieciowych i komponentów portlet.
Test	W tej grupie znajduje się pojedynczy moduł, którego celem jest ułatwienie tworzenia testów jednostkowych i integracji.

Rysunek 1.1 przedstawia zależności zachodzące między poszczególnymi modułami Springa.

Rysunek 1.1.
Zależności zachodzące między poszczególnymi modułami frameworku Spring



Na podstawie powyższego rysunku można stwierdzić, że moduły z grupy *Kontener podstawowy* stanowią serce Springu, natomiast pozostałe moduły są zależne od wymienionej grupy. Bardzo ważne są również moduły dostarczane przez grupę *Programowanie zorientowane aspektowo i instrumentacja*, ponieważ innym modułom Springa zapewniają możliwość stosowania programowania zorientowanego aspektowo.

Skoro już wiesz, które obszary procesu tworzenia aplikacji są obsługiwane przez Spring, możemy przejść do omówienia kontenera IoC.

1.3. Kontener IoC

Aplikacja Javy składa się ze współpracujących ze sobą obiektów w celu dostarczenia odpowiednich funkcjonalności programu. Obiekty, z którymi współdziałają inne obiekty, są określane mianem ich *zależności*. Na przykład jeśli obiekt X współpracuje z obiektami Y i Z, wówczas obiekty Y i Z stają

się zależnościami obiektu X. Wstrzykiwanie zależności (ang. *dependency injection*) to wzorec projektowy, w którym zależności obiektu są najczęściej podawane jako argumenty jego konstruktora i metod typu setter. Wspomniane zależności są wstrzykiwane do obiektu w chwili jego tworzenia.

W aplikacji Springa kontener Spring IoC (określany także mianem kontenera Springa) jest odpowiedzialny za tworzenie obiektów aplikacji oraz wstrzykiwanie ich zależności. Obiekty tworzone przez kontener Springa i zarządzane przez niego są nazywane *komponentami* (ang. *bean*). Kontener Springa jest odpowiedzialny za połączenie ze sobą obiektów aplikacji, a więc tworząc aplikację, nie musisz implementować wzorców projektowych, takich jak fabryka lub lokalizator usługi. Mechanizm wstrzykiwania zależności jest także określany jako odwrócenie sterowania (ang. *inversion of control*, IoC), ponieważ odpowiedzialność za tworzenie i wstrzykiwanie zależności *nie* spoczywa na obiekcie aplikacji, ale na kontenerze Springa.

Przyjmujemy założenie, że MyBank, nasza przykładowa aplikacja, zawiera dwa obiekty — `FixedDepositController` i `FixedDepositService`. Na podstawie kodu przedstawionego w listingu 1.1 można powiedzieć, że obiekt `FixedDepositController` jest zależny od obiektu `FixedDepositService`.

Listing 1.1. Klasa `FixedDepositController`

```
public class FixedDepositController {  
    private FixedDepositService fixedDepositService;  
  
    public FixedDepositController() {  
        fixedDepositService = new FixedDepositService();  
    }  
  
    public boolean submit() {  
        // Zachowanie informacji o lokacie.  
        fixedDepositService.save(....);  
    }  
}
```

W powyższym listingu konstruktor `FixedDepositController` tworzy egzemplarz `FixedDepositService`, który następnie jest używany w metodzie `submit()` obiektu `FixedDepositController`. Ponieważ `FixedDepositController` współdziela z `FixedDepositService`, to `FixedDepositService` stanowi zależność dla `FixedDepositController`.

W celu skonfigurowania `FixedDepositController` w charakterze komponentu Springa należy w pierwszej kolejności zmodyfikować klasę `FixedDepositController` przedstawioną w listingu 1.1, aby akceptowała zależność `FixedDepositService` jako argument konstruktora lub metody typu setter. Zmodyfikowana wersja klasy `FixedDepositController` została zaprezentowana w listingu 1.2.

Listing 1.2. Klasa `FixedDepositController`, w której egzemplarz `FixedDepositService` jest przekazywany jako argument konstruktora

```
public class FixedDepositController {  
    private FixedDepositService fixedDepositService;  
  
    public FixedDepositController(FixedDepositService fixedDepositService) {  
        this.fixedDepositService = fixedDepositService;  
    }  
}
```

```
public boolean submit() {  
    // Zachowanie informacji o lokacie.  
    fixedDepositService.save(.....);  
}  
}
```

Jak możesz zobaczyć w listingu 1.2, egzemplarz `FixedDepositService` jest przekazywany jako argument konstruktora egzemplarza `FixedDepositController`. W tym momencie klasę `FixedDepositService` można skonfigurować jako komponent Springa. Zwróć uwagę na fakt, że klasa `FixedDepositController` nie implementuje ani nie rozszerza żadnego interfejsu lub klasy Springa.

W danej aplikacji informacje o używanych przez nią obiektach i zachodzących między nimi zależnościach można zdefiniować za pomocą tak zwanych *metadanych konfiguracyjnych*. Kontener Springa IoC odczytuje metadane konfiguracyjne aplikacji w celu utworzenia obiektów aplikacji oraz wstrzyknięcia ich zależności. W listingu 1.3 zostały przedstawione metadane konfiguracyjne (w formacie XML) przeznaczone dla aplikacji zawierającej klasy `MyController` i `MyService`.

Listing 1.3. Metadane konfiguracyjne

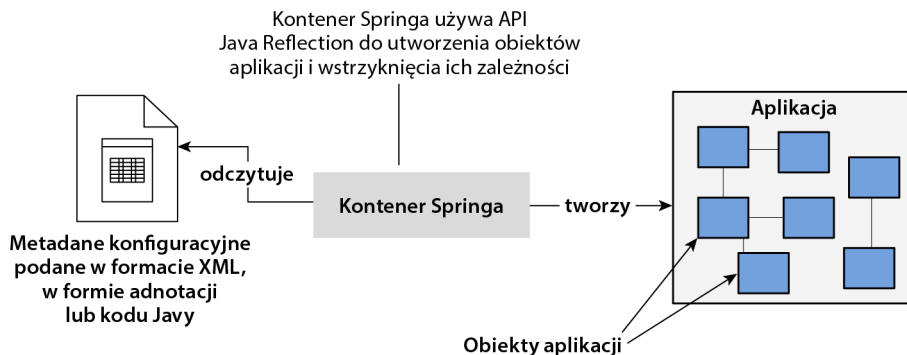
```
<beans .....
```

W powyższym listingu każdy element `<bean>` definiuje obiekt aplikacji zarządzany przez kontener Springa, natomiast element `<constructor-arg>` wskazuje, że egzemplarz `MyService` jest przekazywany jako argument do konstruktora `MyController`. Element `<bean>` zostanie szczegółowo omówiony w dalszej części rozdziału, a do elementu `<constructor-arg>` powrócimy w rozdziale 2.

Kontener Springa odczytuje metadane konfiguracyjne (takie jak przedstawione w listingu 1.3) aplikacji, a następnie tworzy obiekty zdefiniowane w elementach `<bean>` i wstrzykuje ich zależności. W trakcie tego procesu kontener Springa wykorzystuje przy tym API Java Reflection (<http://docs.oracle.com/javase/tutorial/reflect/index.html>). Podsumowanie sposobu działania kontenera Springa zostało pokazane na rysunku 1.2.

Metadane konfiguracyjne mogą być dostarczone kontenerowi Springa w formacie XML (takie rozwiązanie przedstawia w listing 1.3), w postaci adnotacji Javy (patrz: rozdział 6.) lub też za pomocą kodu Javy (patrz: rozdział 6.).

Ponieważ kontener Springa jest odpowiedzialny za utworzenie obiektów aplikacji i zarządzanie nimi, usługi korporacyjne (takie jak zarządzanie transakcjami, zapewnienie bezpieczeństwa, zdalny dostęp itd.) mogą być za pomocą kontenera Springa w sposób oczywisty stosowane w obiektach. Oferowana przez kontener Springa możliwość wzbogacenia obiektów aplikacji o dodatkowe funkcje pozwala nam na tworzenie tych obiektów w postaci prostych obiektów Javy (tak zwanych POJO, czyli *plain old Java objects*). Klasy Javy odpowiadające POJO są nazywane *klasami POJO*. Są to zwykłe klasy Javy, które nie implementują i nie rozszerzają interfejsów lub klas charakterystycznych dla frameworku. Wspomniane wcześniej usługi korporacyjne wymagane przez POJO są w sposób oczywisty dostarczane przez kontener Springa.



Rysunek 1.2. Kontener Springa odczytuje metadane konfiguracyjne aplikacji i na ich podstawie tworzy w pełni skonfigurowaną aplikację

Skoro już poznałeś sposób działania kontenera Springa, to możemy przejść do przykładów pokazujących zalety tworzenia aplikacji za pomocą frameworku Spring.

1.4. Zalety stosowania frameworku Spring

W poprzednim podrozdziale poznałeś następujące zalety stosowania Springa:

- Spring ułatwia tworzenie aplikacji Javy, ponieważ automatycznie zajmuje się tworzeniem obiektów aplikacji i wstrzykiwaniem ich zależności.
- Spring promuje tworzenie aplikacji w postaci POJO.

Ponadto Spring ułatwia współpracę z dostawcami JMS, JNDI, serwerami MBean, serwerami poczty, bazami danych itd., dostarczając warstwę abstrakcji odpowiedzialną za wygenerowanie niezbędnego kodu.

Warto zapoznać się teraz z kilkoma przykładami, które jeszcze bardziej pomogą zrozumieć zalety, jakie niesie ze sobą tworzenie aplikacji za pomocą Springa.

Spójne podejście w zakresie zarządzania transakcjami lokalnymi i globalnymi

Jeżeli używasz Springa do opracowania aplikacji *transakcyjnych*, wówczas do obsługi zarządzania transakcjami można wykorzystać oferowane przez Spring *deklaracyjne zarządzanie transakcjami*.

W listingu 1.4 została przedstawiona klasa `FixedDepositService` naszej aplikacji `MyBank`.

Listing 1.4. Klasa `FixedDepositService`

```
public class FixedDepositService {  
    public FixedDepositDetails getFixedDepositDetails( ..... ) { ..... }  
    public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) { ..... }  
}
```

Klasa `FixedDepositService` to klasa typu POJO definiująca metody przeznaczone do tworzenia i pobierania informacji szczegółowych o lokatach. Na rysunku 1.3 został pokazany formularz, za pomocą którego tworzymy nową lokatę.

Rysunek 1.3.

Formularz HTML
przeznaczony
do utworzenia
nowej lokaty

The image shows a web form with a title bar that says 'Załącz nową lokatę'. Inside the form, there are three labels with corresponding text input fields: 'Wysokość:', 'Czas trwania:', and 'E-mail:'. At the bottom left of the form is a button labeled 'Zapisz'.

W wyświetlonym formularzu klient podaje wysokość lokaty, czas jej trwania oraz swój adres e-mail. Po kliknięciu przycisku *Zapisz* nastąpi utworzenie nowej lokaty o podanej wartości. W celu faktycznego utworzenia lokaty następuje wywołanie metody `createFixedDeposit()` klasy `FixedDepositService` (patrz: listing 1.1). Działanie metody `createFixedDeposit()` polega na pobraniu z konta klienta podanej kwoty i utworzeniu lokaty na taką samą kwotę.

Przyjmujemy założenie, że informacje o aktualnym saldzie kont klientów są przechowywane w tabeli bazy danych o nazwie `BANK_ACCOUNT_DETAILS`, natomiast informacje szczegółowe o lokatach — w tabeli `FIXED_DEPOSIT_DETAILS`. Jeżeli klient tworzy lokatę na kwotę *x*, wówczas ta wartość zostaje odjęta od wartości przechowywanej w tabeli `BANK_ACCOUNT_DETAILS`, a ponadto wstawiany jest nowy rekord do tabeli `FIXED_DEPOSIT_DETAILS` odzwierciedlający nowo utworzoną lokatę. Jeżeli tabela `BANK_ACCOUNT_DETAILS` *nie* zostanie uaktualniona lub nowy rekord *nie* zostanie wstawiony do tabeli `FIXED_DEPOSIT_DETAILS`, wówczas system pozostanie w niespójnym stanie. Oznacza to konieczność wykonania metody `createFixedDeposit()` w ramach transakcji.

Baza danych używana przez aplikację *MyBank* stanowi *zasób transakcyjny*. W tradycyjnym podejściu polegającym na przeprowadzaniu zestawu operacji modyfikacji w bazie danych jako pojedynczego zadania najpierw wyłączany jest tryb automatycznego zatwierdzania w połączeniu JDBC, następnie wykonywane są polecenia SQL, a na końcu następuje zatwierdzenie (lub wycofanie) transakcji. W listingu 1.5 przedstawiamy sposób zarządzania transakcjami bazy danych w metodzie `createFixedDeposit()` z zastosowaniem tradycyjnego podejścia.

Listing 1.5. Programowe zarządzanie transakcjami bazy danych za pomocą obiektu *JDBC Connection*

```
import java.sql.Connection;
import java.sql.SQLException;

public class FixedDepositService {
    public FixedDepositDetails getFixedDepositDetails( ..... ) { ..... }

    public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) {
        Connection con = ..... ;
        try {
            con.setAutoCommit(false);
            // Wykonanie poleceń SQL modyfikujących tabele bazy danych.
            con.commit();
        } catch (SQLException sqle) {
            if (con != null) {
                con.rollback();
            }
        }
        .....
    }
}
```

W powyższym listingu wyraźnie widać, że metoda `createFixedDeposit()` w programowy sposób zarządza transakcją bazy danych, używając do tego obiektu `JDBC` o nazwie `Connection`. Tego rodzaju podejście jest odpowiednie w aplikacjach, w których wykorzystywana jest pojedyncza baza danych. Transakcje charakterystyczne dla zasobu, na przykład powiązane z `JDBC Connection`, są określane mianem *transakcji lokalnych*.

Kiedy używanych jest wiele zasobów transakcyjnych, wówczas ich zarządzanie odbywa się za pomocą `JTA` (ang. *Java Transaction API*). Na przykład jeśli chcesz wysłać komunikat `JMS` do pośrednika komunikatów (zasób transakcyjny) i w tej samej transakcji uaktualnić jeszcze bazę danych (to kolejny zasób transakcyjny), wtedy konieczne jest użycie menedżera transakcji `JTA` w celu zarządzania transakcjami. Transakcje `JTA` są określane mianem *transakcji globalnych* (lub *rozproszonych*). Aby użyć `JTA`, należy z `JNDI` pobrać obiekt `UserTransaction` (stanowi część `API JTA`), a następnie programowo rozpocząć i zatwierdzić (lub wycofać) transakcję.

Jak możesz zobaczyć, do programowego zarządzania transakcjami można wykorzystać obiekt `JDBC Connection` (transakcje lokalne) lub `UserTransaction` (transakcje globalne). Trzeba koniecznie pamiętać, że transakcje lokalne *nie mogą* być wykonywane w ramach transakcji globalnych. Oznacza to, że jeśli operacja uaktualnienia bazy danych w przedstawionej w listingu 1.5 metodzie `createFixedDeposit()` ma być częścią transakcji `JTA`, wówczas konieczne jest zmodyfikowanie metody `createFixedDeposit()` w taki sposób, aby do zarządzania transakcją był używany obiekt `UserTransaction`.

Spring ułatwia zarządzanie transakcjami, ponieważ dostarcza warstwę abstrakcji zapewniającą *spójne* podejście w zakresie zarządzania transakcjami zarówno lokalnymi, jak i globalnymi. Oznacza to, że w przypadku zdefiniowania metody `createFixedDeposit()` — jak przedstawia listing 1.5 — za pomocą abstrakcji Springa nie ma konieczności jej modyfikacji po zmianie transakcji z lokalnej na globalną bądź na odwrót. Dokładne omówienie warstwy abstrakcji transakcji w Springu znajdziesz w rozdziale 7.

Deklaracyjne zarządzanie transakcjami

Spring pozwala na użycie *deklaracyjnego zarządzania transakcjami*. Po oznaczeniu metody adnotacją `@Transactional` (patrz: listing 1.6) Spring zajmuje się obsługą transakcji.

Listing 1.6. Przykład użycia adnotacji `@Transactional`

```
import org.springframework.transaction.annotation.Transactional;

public class FixedDepositService {
    public FixedDepositDetails getFixedDepositDetails( ..... ) { ..... }

    @Transactional
    public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) { ..... }
}
```

Powyższy listing pokazuje, że klasa `FixedDepositService` nie implementuje ani nie rozszerza żadnego charakterystycznego dla Springa interfejsu bądź klasy, aby móc wykorzystać oferowane przez Spring możliwości w zakresie zarządzania transakcjami. Spring w sposób oczywisty zapewnia funkcję

zarządzania transakcjami metodzie `createFixedDeposit()` oznaczoną adnotacją `@Transactional`. Jak widzisz, Spring to rodzaj *nieinwazyjnego* frameworku, ponieważ nie wymaga, aby obiekty aplikacji były zależne od charakterystycznych dla Springa interfejsów lub klas. Ponadto w celu zarządzania transakcjami nie ma potrzeby bezpośredniej pracy z API odpowiedzialnym za zarządzanie transakcjami.

Zapewnienie bezpieczeństwa

Zapewnienie bezpieczeństwa to bardzo ważny aspekt każdej aplikacji Javy. Framework Spring Security (<http://projects.spring.io/spring-security/>) to projekt SpringSource zbudowany na podstawie frameworku Spring. Spring Security dostarcza funkcje uwierzytelniania i autoryzacji, za pomocą których można zabezpieczać aplikacje Javy.

Przyjmujemy założenie, że w aplikacji MyBank zostały zidentyfikowane trzy następujące role użytkownika: `LOAN_CUSTOMER`, `SAVINGS_ACCOUNT_CUSTOMER` i `APPLICATION_ADMIN`. Klient *musi* mieć przypisaną rolę `SAVINGS_ACCOUNT_CUSTOMER` lub `APPLICATION_ADMIN`, aby mógł wywołać metodę `createFixedDeposit()` klasy `FixedDepositService` (została przedstawiona w listingu 1.6). Używając frameworku Spring Security, można bardzo łatwo spełnić to wymaganie przez dodanie adnotacji `@Secured` do metody `createFixedDeposit()`, jak przedstawia listing 1.7.

Listing 1.7. Zabezpieczona metoda `createFixedDeposit()`

```
import org.springframework.transaction.annotation.Transactional;
import org.springframework.security.access.annotation.Secured;

public class FixedDepositService {
    public FixedDepositDetails getFixedDepositDetails( ..... ) { ..... }

    @Transactional
    @Secured({ "SAVINGS_ACCOUNT_CUSTOMER", "APPLICATION_ADMIN" })
    public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) { ..... }
}
```

Po oznaczeniu metody za pomocą adnotacji `@Secured` oferowanej przez Spring Security funkcje zabezpieczeń będą przez ten framework w sposób oczywisty zastosowane w metodzie. W powyższym listingu widać, że implementując zabezpieczenia na *poziomie metody*, nie trzeba rozszerzać lub implementować jakichkolwiek klas bądź interfejsów charakterystycznych dla Springa. Ponadto nie ma konieczności tworzenia w metodach logiki biznesowej żadnego kodu związanego z zapewnianiem bezpieczeństwa.

Framework Spring Security omówimy dokładnie w rozdziale 14.

JMX (Java Management Extensions)

Oferowana przez Spring obsługa JMX znacznie ułatwia wykorzystanie technologii JMX w tworzonych aplikacjach.

Przyjmujemy założenie, że możliwość tworzenia lokat za pomocą aplikacji MyBank ma być dostępna dla klientów jedynie w godzinach od 9.00 do 18.00. W celu spełnienia takiego wymagania

do klasy `FixedDepositService` dodajemy zmienną działającą w charakterze opcji wskazującej, czy usługa tworzenia lokat jest aktywna. Zmodyfikowana wersja klasy `FixedDepositService` zawierająca wspomnianą zmienną została przedstawiona w listingu 1.8.

Listing 1.8. Klasa `FixedDepositService` wraz ze zmienną `active`

```
public class FixedDepositService {
    private boolean active;

    public FixedDepositDetails getFixedDepositDetails( .... ) {
        if(active) { ..... }
    }
    public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) {
        if(active) { ..... }
    }
    public void activateService() {
        active = true;
    }
    public void deactivateService() {
        active = false;
    }
}
```

Jak widzisz w powyższym listingu, do klasy `FixedDepositService` dodano zmienną o nazwie `active`. Jeżeli wartością tej zmiennej będzie `true`, wtedy metody `getFixedDepositDetails()` i `createFixedDeposit()` działają zgodnie z oczekiwaniami. Natomiast w przypadku przypisania zmiennej `active` wartości `false` wywołanie metody `getFixedDepositDetails()` lub `createFixedDeposit()` spowoduje zgłoszenie wyjątku wskazującego na niedostępność usługi zakładania lokat. Metody `activateService()` i `deactivateService()` przypisują zmiennej `active` wartości odpowiednio `true` i `false`.

W tym momencie powstaje pytanie, kto wywołuje metody `activateService()` i `deactivateService()`. Przyjmujemy założenie, że istnieje oddzielna aplikacja harmonogramu o nazwie `Bank App Scheduler` uruchamiana o godzinach 9.00 i 18.00, której zadaniem jest wywoływanie metody odpowiednio `activateService()` i `deactivateService()`. Ta aplikacja harmonogramu używa API JMX w celu zdalnej współpracy z egzemplarzem `FixedDepositService`.



Więcej informacji na temat technologii JMX znajdziesz w artykule dostępnym na stronie <http://docs.oracle.com/javase/tutorial/jmx/index.html>.

Ponieważ aplikacja `Bank App Scheduler` używa technologii JMX do zmiany wartości zmiennej `active` egzemplarza `FixedDepositService`, konieczne jest zarejestrowanie w serwerze MBean egzemplarza `FixedDepositService` jako *komponentu zarządzanego* (ang. *managed bean*, MBean), a także udostępnienie metod `activateService()` i `deactivateService()` klasy `FixedDepositService` jako operacji JMX. W Springu rejestracja egzemplarzy klas w serwerze MBean odbywa się przez dodanie do klasy adnotacji `@ManagedResource`, natomiast udostępnienie metody klasy jako operacji JMX wymaga dodania do niej adnotacji `@ManagedOperation`.

W listingu 1.9 pokazujemy przykład użycia adnotacji `@ManagedResource` i `@ManagedOperation` w celu rejestracji w serwerze MBean egzemplarzy klasy `FixedDepositService` oraz udostępnienia jej metod `activateService()` i `deactivateService()`.

Listing 1.9. Klasa `FixedDepositService` wykorzystująca oferowaną przez Spring obsługę JMX

```
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;

@ManagedResource(objectName = "fixed_deposit_service:name=FixedDepositService")
public class FixedDepositService {
    private boolean active;

    public FixedDepositDetails getFixedDepositDetails( .... ) {
        if(active) { .... }
    }
    public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) {
        if(active) { .... }
    }
    @ManagedOperation
    public void activateService() {
        active = true;
    }
    @ManagedOperation
    public void deactivateService() {
        active = false;
    }
}
```

W powyższym listingu widać, że klasa `FixedDepositService` nie używa bezpośrednio API JMX do zarejestrowania jej egzemplarzy w serwerze MBean oraz udostępnienia metod jako operacji JMX.

JMS (Java Message Service)

Oferowana przez Spring obsługa JMS ułatwia wysyłanie i otrzymywanie komunikatów od dostawców JMS.

W naszej przykładowej aplikacji `MyBank`, gdy klient wykona żądanie mające na celu otrzymanie przez e-mail informacji szczegółowych o lokatach, wówczas klasa `FixedDepositService` wysyła szczegóły żądania do pośrednika JMS (takiego jak `ActiveMQ`). Następnie żądanie jest przetwarzane przez komponent nasłuchujący komunikatów. Spring upraszcza współpracę z dostawcami JMS, zapewniając programistom warstwę abstrakcji. W listingu 1.10 pokazujemy, jak klasa `FixedDepositService` wysyła do dostawcy JMS informacje szczegółowe o żądaniu, wykorzystując w tym celu `JmsTemplate`.

Listing 1.10. Klasa `FixedDepositService` pozwala na wysyłanie komunikatów JMS

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;

public class FixedDepositService {
    @Autowired
    private transient JmsTemplate jmsTemplate;
```

```
.....
public boolean submitRequest(Request request) {
    jmsTemplate.convertAndSend(request);
}
}
```

Powyższy fragment kodu pokazuje, że klasa `FixedDepositService` definiuje zmienną typu `JmsTemplate` oraz została oznaczona adnotacją `@Autowired`. W tym momencie wystarczy, jeśli przyjmiesz do wiadomości, że adnotacja ta zapewnia dostęp do egzemplarza `JmsTemplate`. Egzemplarz `JmsTemplate` zna miejsce docelowe JMS, do którego zostanie wysłany komunikat JMS. Szczegółowe informacje dotyczące konfiguracji `JmsTemplate` przedstawimy w rozdziale 8. Metoda `submitRequest()` klasy `FixedDepositService` wywołuje metodę `convertAndSend()` egzemplarza `JmsTemplate` w celu wysłania szczegółów żądania (są przedstawione przez argument `Request` metody `submitRequest()`) jako komunikatu JMS do dostawcy JMS.

Warto przypomnieć ponownie o jednym: powyższy listing pokazuje, że dzięki użyciu Springa nie musisz bezpośrednio wysyłać komunikatów do dostawców JMS, a także nie musisz bezpośrednio pracować z API JMS.

Buforowanie

Spring oferuje także warstwę abstrakcji, dzięki której w aplikacji można wykorzystać spójne podejście w zakresie buforowania.

Bardzo często używa się buforowania, aby w ten sposób zwiększyć wydajność działania aplikacji. W naszej przykładowej aplikacji `MyBank` buforowanie produktu wykorzystano w celu poprawienia wydajności operacji *odczytu* informacji szczegółowych o lokatach. Spring upraszcza pracę z różnymi rozwiązaniami z zakresu buforowania przez dostarczenie abstrakcji logiki związanej z buforowaniem.

W listingu 1.11 możesz zobaczyć, jak metoda `getFixedDepositDetails()` klasy `FixedDepositService` używa oferowanej przez Spring warstwy abstrakcji do buforowania informacji szczegółowych o lokatach.

Listing 1.11. Klasa `FixedDepositService` buforująca informacje szczegółowe o lokatach

```
import org.springframework.cache.annotation.Cacheable;

public class FixedDepositService {

    @Cacheable("FixedDeposits")
    public FixedDepositDetails getFixedDepositDetails( ..... ) { ..... }

    public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) { ..... }
}
```

W powyższym listingu adnotacja `@Cacheable` wskazuje, że informacje szczegółowe zwracane przez metodę `getFixedDepositDetails()` są buforowane. Jeżeli metoda `getFixedDepositDetails()` zostanie ponownie wywołana z takimi samymi wartościami argumentów, wówczas *nie* nastąpi faktyczne wykonanie metody `getFixedDepositDetails()`, a odpowiednie dane o lokatach zostaną pobrane

z bufora. Dlatego też używając Springa, nie musisz w klasach tworzyć logiki związanej z buforowaniem. Szczegółowe omówienie oferowanej przez Spring warstwy abstrakcji buforowania znajdziesz w rozdziale 8.

W tym podrozdziale przekonałeś się, że Spring ułatwia tworzenie aplikacji korporacyjnych dzięki transparentnemu dostarczaniu usług do POJO, a tym samym pozwala programistom uniknąć konieczności pracy z API działającym na niskim poziomie. Ponadto Spring zapewnia łatwą integrację ze standardowymi frameworkami, takimi jak Hibernate, iBATIS, Quartz, JSF, Struts, EJB itd. Dzięki tym możliwościom Spring stanowi idealny wybór podczas tworzenia aplikacji korporacyjnych.

Skoro poznałeś pewne zalety użycia Springa, warto teraz zobaczyć, jak przebiega proces tworzenia prostej aplikacji za pomocą tego frameworku.

1.5. Prosta aplikacja Springa

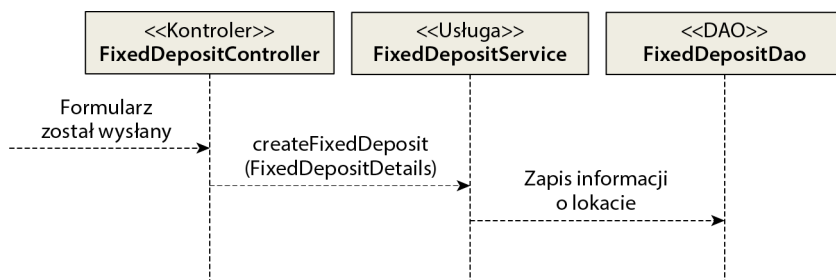
W tym podrozdziale zobaczysz prostą aplikację Springa używającą funkcji wstrzykiwania zależności. Wykorzystanie tej funkcji wymaga wykonania następujących kroków:

1. Identyfikacja obiektów aplikacji i ich zależności.
2. Utworzenie klas POJO odpowiadających obiektom aplikacji zidentyfikowanym w kroku 1.
3. Utworzenie *metadanych konfiguracyjnych*, które opisują obiekty aplikacji i ich zależności.
4. Utworzenie egzemplarza kontenera Springa IoC i przekazanie mu metadanych konfiguracyjnych.
5. Uzyskanie dostępu do obiektów aplikacji z poziomu egzemplarza kontenera Springa IoC.

Przeanalizujmy teraz powyższe kroki, ale w kontekście przykładowej aplikacji MyBank.

Identyfikacja obiektów aplikacji i ich zależności

Wcześniej wspomnieliśmy, że aplikacja MyBank wyświetla formularz (patrz: rysunek 1.3) umożliwiający klientowi założenie nowej lokaty na wskazaną kwotę. Diagram zaprezentowany na rysunku 1.4 pokazuje obiekty aplikacji (i ich interakcje) pojawiające się, gdy użytkownik wysła formularz.



Rysunek 1.4. Obiekty i ich zależności w aplikacji MyBank

W powyższym diagramie FixedDepositController przedstawia kontroler otrzymujący żądanie po wysłaniu formularza przez klienta. Informacje szczegółowe o lokacie znajdują się w obiekcie FixedDepositDetails. Klasa FixedDepositController wywołuje metodę createFixedDeposit() klasy

`FixedDepositService` (obiekt warstwy usługi). Następnie `FixedDepositService` wywołuje obiekt `FixedDepositDao` (czyli obiekt dostępu do danych) w celu zapisania informacji szczegółowych o lokacie w magazynie danych aplikacji.

Na podstawie powyższego diagramu możemy stwierdzić, że egzemplarz `FixedDepositService` jest *zależnością* obiektu `FixedDepositController`, natomiast egzemplarz `FixedDepositDao` to *zależność* obiektu `FixedDepositService`.

IMPORT **Rozdział01/ch01-bankapp-xml**. (W tym projekcie znajduje się prosta aplikacja Springa używająca funkcji wstrzykiwania zależności. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `MyBankApp` projektu).

Utworzenie klas POJO odpowiadających zidentyfikowanym obiektom aplikacji

Po zidentyfikowaniu obiektów aplikacji kolejnym krokiem jest utworzenie klas POJO odpowiadających tym obiektom. Klasy POJO odpowiadające obiektom aplikacji `FixedDepositController`, `FixedDepositService` i `FixedDepositDao` znajdują się w projekcie `ch01-bankapp-xml`. Projekt ten przedstawia uproszczoną wersję aplikacji `MyBank` używającą funkcji wstrzykiwania zależności. Powinieneś zaimportować projekt `ch01-bankapp-xml` do używanego środowiska IDE, ponieważ w następnych krokach będziemy korzystać z plików znajdujących się w tym projekcie.

Z podrozdziału 1.3 dowiedziałeś się, że zależność jest przekazywana do obiektu aplikacji w postaci argumentu konstruktora lub też argumentu metody typu setter. W listingu 1.12 możesz zobaczyć, jak egzemplarz klasy `FixedDepositService` (zależność dla `FixedDepositController`) jest przekazywany do obiektu `FixedDepositController` jako argument metody typu setter.

Listing 1.12. Klasa `FixedDespositController`

Projekt: `ch01-bankapp-xml`

Położenie pliku: `src/main/java/sample/spring/chapter01/bankapp`

```
package sample.spring.chapter01.bankapp;
.....
public class FixedDepositController {
    .....
    private FixedDepositService fixedDepositService;
    .....
    public void setFixedDepositService(FixedDepositService fixedDepositService) {
        logger.info("Ustawienie właściwości fixedDepositService.");
        this.fixedDepositService = fixedDepositService;
    }
    .....
    public void submit() {
        fixedDepositService.createFixedDeposit(new FixedDepositDetails( 1, 10000,
            365, "nazwa-uzytkownika@nazwa-domeny.pl"));
    }
    .....
}
```

W powyższym listingu zależność w postaci egzemplarza `FixedDepositService` jest przekazywana do `FixedDepositController` za pomocą metody `setFixedDepositService()`. Wkrótce przekonasz się, że metoda `setFixedDepositService()` jest wywoływana przez Spring.



Jeżeli przyjrzyj się klasom `FixedDepositController`, `FixedDepositService` i `FixedDepositDao`, zauważysz, że żadna z nich nie implementuje jakiegokolwiek charakterystycznego dla Springa interfejsu, a także nie rozszerza żadnych klas Springa.

Teraz przechodzimy do sposobu, w jaki obiekty aplikacji i ich zależności są określane w metadanych konfiguracyjnych.

Utworzenie metadanych konfiguracyjnych

Z podrozdziału 1.3 dowiedziałeś się, że metadane konfiguracyjne określają obiekty aplikacji i ich zależności. Te metadane są odczytywane przez kontener Springa w celu utworzenia obiektów aplikacji i wstrzyknięcia ich zależności. W tym podrozdziale najpierw zobaczymy, jakie jeszcze inne informacje znajdują się w metadanych konfiguracyjnych, a następnie dokładnie zajmiemy się omówieniem tematu określania metadanych konfiguracyjnych w formacie XML.

Metadane konfiguracyjne dostarczają informacje o usługach korporacyjnych (takich jak zarządzanie transakcjami, zapewnienie bezpieczeństwa i zdalny dostęp) wymaganych przez aplikację. Na przykład jeżeli Spring ma zajmować się zarządzaniem transakcjami, wówczas w metadanych konfiguracyjnych należy skonfigurować implementację interfejsu `PlatformTransactionManager`. Implementacja `PlatformTransactionManager` jest odpowiedzialna za zarządzanie transakcjami (w rozdziale 7. znajdziesz więcej informacji o oferowanej przez Spring funkcji zarządzania transakcjami).

Jeżeli aplikacja współdziała z pośrednikami zajmującymi się komunikatami (takimi jak ActiveMQ), bazami danych (takimi jak MySQL), serwerami e-mail itd., wtedy charakterystyczne dla Springa obiekty upraszczające pracę ze wspomnianymi systemami zewnętrznymi również są skonfigurowane w metadanych konfiguracyjnych. Na przykład jeśli aplikacja wysyła lub otrzymuje komunikaty JMS z ActiveMQ, to w metadanych konfiguracyjnych można skonfigurować klasę Springa `JmsTemplate` i tym samym ułatwić współdziałanie aplikacji z ActiveMQ. W listingu 1.10 widziałeś, że w przypadku wykorzystania `JmsTemplate` w celu wysyłania komunikatów do dostawcy JMS nie musisz używać działającego na niskim poziomie API JMS. (W rozdziale 8. znajdziesz więcej informacji o oferowanej przez Spring obsłudze współpracy z dostawcami JMS).

Metadane konfiguracyjne można dostarczyć kontenerowi Springa za pomocą pliku w formacie XML bądź też przez adnotacje w klasach POJO. Począwszy od wersji 3.0 biblioteki Springa, istnieje również możliwość dostarczania metadanych konfiguracyjnych do kontenera Springa za pomocą klas Javy oznaczonych adnotacją `@Configuration`. W tym podrozdziale zobaczysz, jak metadane konfiguracyjne są określane w formacie XML. Natomiast w rozdziale 6. zajmiemy się dostarczaniem metadanych konfiguracyjnych za pomocą adnotacji w klasach POJO oraz przez klasy Javy oznaczone adnotacją `@Configuration`.

Metadane konfiguracyjne w formacie XML są dostarczane aplikacji przez utworzenie tak zwanego *pliku XML kontekstu aplikacji*, który zawiera informacje o obiektach aplikacji i ich zależnościach. Przykład tego rodzaju pliku XML przedstawiamy w listingu 1.13. Plik o nazwie *applicationContext.xml* pokazuje kontekst aplikacji MyBank składającej się z obiektów *FixedDepositController*, *FixedDepositService* i *FixedDepositDao* (na rysunku 1.4 możesz zobaczyć, jak wymienione obiekty współdziałają ze sobą).

Listing 1.13. *Plik applicationContext.xml, czyli plik XML kontekstu aplikacji znajdujący się w aplikacji MyBank*

Projekt: *ch01-bankapp-xml*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<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-4.0.xsd">

  <bean id="controller"
    class="sample.spring.chapter01.bankapp.FixedDepositController">
    <property name="fixedDepositService" ref="service" />
  </bean>

  <bean id="service" class="sample.spring.chapter01.bankapp.FixedDepositService">
    <property name="fixedDepositDao" ref="dao" />
  </bean>

  <bean id="dao" class="sample.spring.chapter01.bankapp.FixedDepositDao"/>
</beans>
```

Oto najważniejsze kwestie, na które trzeba zwrócić uwagę w pliku XML kontekstu aplikacji przedstawionym w listingu 1.13:

- Element `<beans>` jest elementem głównym pliku XML kontekstu aplikacji i został zdefiniowany w schemacie *spring-beans-4.0.xsd* (nazywanym także schematem beans w Springu). Schemat *spring-beans-4.0.xsd* znajduje się w pliku JAR o nazwie *spring-beans-4.0.0.RELEASE.jar*, który jest dostarczany wraz ze Springiem.
- Poszczególne elementy `<bean>` zawierają konfigurację obiektu aplikacji zarządzanego przez kontener Springa. W terminologii Springa element `<bean>` przedstawia *definicję komponentu*. Obiekt tworzony przez kontener Springa na podstawie definicji komponentu nosi nazwę *komponentu*. Atrybut `id` wskazuje unikalną nazwę komponentu, natomiast atrybut `class` zawiera w pełni kwalifikowaną nazwę komponentu. Istnieje również możliwość użycia atrybutu `name` elementu `<bean>` do wskazania *aliasu* dla komponentu. W przykładowej aplikacji MyBank obiekty aplikacji to *FixedDepositController*, *FixedDepositService* i *FixedDepositDao*. Dlatego mamy trzy elementy `<bean>`, po jednym dla każdego z obiektów aplikacji. Ponieważ obiekty aplikacji konfigurowane przez elementy `<bean>` są zarządzane przez kontener Springa, odpowiedzialność za ich tworzenie oraz wstrzykiwanie zależności spada na kontener Springa. Zamiast bezpośrednio tworzyć obiekty aplikacji definiowane przez elementy `<bean>` powinien je pobierać z kontenera Springa. W dalszej części podrozdziału zobaczysz, jak pobierać obiekty aplikacji zarządzane przez kontener Springa.

- Nie został zdefiniowany żaden element `<bean>` odpowiadający obiektowi domeny `FixedDeposit` ↪ `Details` aplikacji `MyBank`. Wynika to z faktu, że obiekty domeny zwykle *nie* są zarządzane przez kontener Springa. Za ich utworzenie odpowiada framework ORM (taki jak Hibernate) używany przez aplikację lub też są one tworzone programowo za pomocą operatora `new`.
- Element `<property>` określa zależność (lub właściwość konfiguracyjną) komponentu konfigurowanego przez element `<bean>`. Element `<property>` odpowiada *metodzie typu setter w stylu `JavaBean`* w klasie komponentu, w której jest wywoływana przez kontener Springa, aby ustawić zależność (bądź właściwość konfiguracyjną) komponentu.

Zobaczmy teraz, jak zależności są wstrzykiwane za pomocą metod typu setter.

Wstrzykiwanie zależności za pomocą metod typu setter

Aby zrozumieć, jak przebiega proces wstrzykiwania zależności za pomocą metod typu setter zdefiniowanych w klasie komponentu, raz jeszcze spójrz na zaprezentowaną w listingu 1.14 klasę `FixedDepositController` w przykładowej aplikacji `MyBank`.

Listing 1.14. Klasa `FixedDepositController`

Projekt: `ch01-bankapp.xml`

Położenie pliku: `src/main/java/sample/spring/chapter01/bankapp`

```
package sample.spring.chapter01.bankapp;

import org.apache.log4j.Logger;

public class FixedDepositController {
    private static Logger logger = Logger.getLogger(FixedDepositController.class);

    private FixedDepositService fixedDepositService;

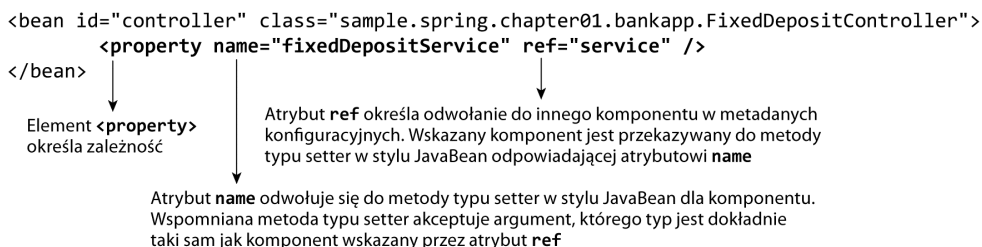
    public FixedDepositController() {
        logger.info("Inicjacja.");
    }

    public void setFixedDepositService(FixedDepositService fixedDepositService) {
        logger.info("Ustawienie właściwości fixedDepositService.");
        this.fixedDepositService = fixedDepositService;
    }

    ....
}
```

Jak widać w powyższym listingu, klasa `FixedDepositController` deklaruje zmienną egzemplarza o nazwie `fixedDepositService` i typie `FixedDepositService`. Wartość zmiennej `fixedDepositService` jest przypisywana za pomocą metody `setFixedDepositService()`, czyli dostępnej w stylu *JavaBean metody typu setter* dla `fixedDepositService`. To jest przykład metody typu setter opartej na wstrzykiwaniu zależności, a jej zadaniem jest spełnienie zależności.

Na rysunku 1.5 pokazano znajdującą się w pliku `applicationContext.xml` (patrz: listing 1.13) definicję komponentu dla klasy `FixedDepositController`.



Rysunek 1.5. Definiowanie zależności za pomocą elementów `<property>`

W pokazanej na rysunku 1.5 definicji komponentu można zauważyć, że za pomocą elementu `<property>` komponent `FixedDepositController` definiuje jego zależność od komponentu `FixedDepositService`. Atrybut `name` elementu `<property>` odpowiada metodzie typu setter w stylu JavaBean w klasie komponentu wywoływanej przez kontener Springa podczas tworzenia komponentu. Z kolei atrybut `ref` elementu `<property>` identyfikuje komponent Springa, którego egzemplarz musi być utworzony i przekazany metodzie typu setter w stylu JavaBean. Wartość atrybutu `ref` musi odpowiadać wartości atrybutu `id` (lub jednej z nazw określonej przez atrybut `name`) elementu `<bean>` w metadanych konfiguracyjnych.

Na rysunku 1.5 wartość atrybutu `name` elementu `<property>` wynosi `fixedDepositService`. Oznacza to, że element `<property>` odpowiada metodzie typu setter `setFixedDepositService()` klasy `FixedDepositController` (patrz: listing 1.14). Ponieważ wartością atrybutu `ref` elementu `<property>` jest `service`, element `<property>` odwołuje się do elementu `<bean>`, którego wartością atrybutu `id` jest `service`. Teraz element `<bean>`, którego wartość atrybutu `id` wynosi `service`, to komponent `FixedDepositService` (patrz: listing 1.13). Kontener Springa tworzy egzemplarz klasy `FixedDepositService` (zależność) i wywołuje metodę `setFixedDepositService()` (metoda typu setter w stylu JavaBean dla zmiennej `fixedDepositService`) obiektu `FixedDepositController` (obiekt zależności), przekazując egzemplarz `FixedDepositService`.

W kontekście obiektu aplikacji `FixedDepositController` zamieszczone na rysunku 1.6 podsumowanie pokazuje przeznaczenie atrybutów `name` i `ref` elementu `<property>`.

Z powyższego rysunku wynika, że wartość atrybutu `name` wnosi `fixedDepositService` i odpowiada metodzie `setFixedDepositService()` klasy `FixedDepositController`. Natomiast wartość atrybutu `ref` wynosi `service` i odpowiada komponentowi, którego wartością atrybutu `id` jest `service`.



Bardzo często do definicji komponentu odwołujemy się poprzez jego nazwę (wartość atrybutu `id`), typ (wartość atrybutu `class`) lub interfejs implementowany przez klasę komponentu. Na przykład do komponentu `FixedDepositController` możemy się odwoływać za pomocą nazwy „komponent kontrolera”. Jeżeli klasa `FixedDepositController` implementuje interfejs `FixedDepositControllerIntf`, wówczas do komponentu `FixedDepositController` możemy się odwoływać za pomocą nazwy „komponent `FixedDepositControllerIntf`”.

Przedstawiony na rysunku 1.7 diagram podsumowuje sposób, w jaki kontener Springa tworzy komponenty i wstrzykuje ich zależności na podstawie metadanych konfiguracyjnych dostarczanych przez plik `applicationContext.xml` (patrz: listing 1.13) aplikacji `MyBank`.

```

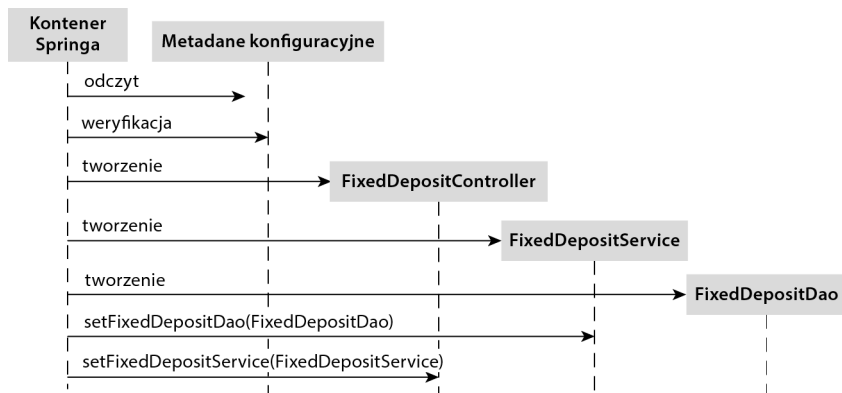
<bean id="controller"
      class="sample.spring.chapter01.bankapp.FixedDepositController">
  <property name="fixedDepositService" ref="service" ?>
</bean>

<bean id="service"
      class="sample.spring.chapter01.bankapp.FixedDepositService">
  <property name="fixedDepositDao" ref="dao" />
</bean>

public class FixedDepositController {
    ...
    private FixedDepositService fixedDepositService
    ...
    public void setFixedDepositService(FixedDepositService
                                     fixedDepositService) {
        logger.info("Setting fixedDepositService property");
        this.fixedDepositService = fixedDepositService;
    }
    ...
}

```

Rysunek 1.6. Atrybut `name` elementu `<property>` odpowiada metodzie typu `setter` w stylu `JavaBean` odpowiedzialnej za spełnienie zależności, natomiast atrybut `ref` odwołuje się do innego komponentu



Rysunek 1.7. Sekwencja, w jakiej kontener Springa IoC tworzy komponenty i wstrzykuje ich zależności

Na rysunku 1.7 została pokazana sekwencja kroków wykonywanych przez kontener Springa IoC w celu utworzenia komponentów `FixedDepositController`, `FixedDepositService` i `FixedDepositDao` oraz wstrzyknięcia ich zależności. Przed podjęciem próby utworzenia komponentów kontener Springa odczytuje i weryfikuje metadane konfiguracyjne dostarczane przez plik `applicationContext.xml`. Kolejność, w jakiej komponenty są tworzone przez kontener Springa odpowiada kolejności, w jakiej zostały zdefiniowane w pliku `applicationContext.xml`. Kontener Springa gwarantuje, że zależności komponentu zostaną w pełni skonfigurowane, zanim nastąpi wywołanie metody typu `setter`. Na przykład komponent `FixedDepositController` zależy od `FixedDepositService` i dlatego kontener Springa przeprowadza konfigurację komponentu `FixedDepositService`, zanim wywoła metodę `setFixedDepositService()` komponentu `FixedDepositController`.

Przedstawione dotąd definicje komponentów nakazują kontenerowi Springa utworzenie egzemplarzy komponentów poprzez wywołanie *bezargumentowego* konstruktora klasy komponentu i wstrzyknięcie zależności za pomocą mechanizmu wstrzykiwania zależności opartego na metodzie typu setter. W rozdziale 2. zapoznasz się z definicjami komponentów, które nakazują kontenerowi Springa tworzenie egzemplarzy komponentów za pomocą zdefiniowanej w klasie *metody fabryki*. Ponadto dowiesz się, jak wstrzykiwać zależności za pomocą argumentów konstruktora (tak zwany mechanizm wstrzykiwania zależności *oparty na konstruktorze*), za pomocą argumentów metody fabryki tworzącej egzemplarz komponentu, a także przez użycie mechanizmu wstrzykiwania zależności opartego na metodzie typu setter egzemplarza zwróconego przez metodę fabryki.

Teraz przechodzimy do zagadnienia tworzenia egzemplarza kontenera Springa i przekazywania mu metadanych konfiguracyjnych.

Utworzenie egzemplarza kontenera Springa

Oferowany przez Spring obiekt `ApplicationContext` przedstawia egzemplarz kontenera Springa. Framework Spring udostępnia kilka wbudowanych implementacji interfejsu `ApplicationContext`, na przykład `ClassPathXmlApplicationContext`, `FileSystemXmlApplicationContext`, `XmlWebApplicationContext`, `WebApplicationContext`, `XmlPortletApplicationContext` itd. Wybór implementacji `ApplicationContext` zależy od sposobu zdefiniowania metadanych konfiguracyjnych (za pomocą formatu XML, adnotacji lub kodu Javy), a także rodzaju samej aplikacji (samodzielna, sieciowa lub portlet). Dlatego też klasy `ClassPathXmlApplicationContext` i `FileSystemXmlApplicationContext` są odpowiednie dla *samodzielnych* aplikacji, w których metadane konfiguracyjne dostarczone w formacie XML. Natomiast `XmlWebApplicationContext` jest odpowiednia dla aplikacji *sieciowych*, w których metadane konfiguracyjne dostarczone w formacie XML. Z kolei klasa `AnnotationConfigWebApplicationContext` najlepiej sprawdza się dla aplikacji *sieciowych*, w których metadane konfiguracyjne dostarczone za pomocą kodu Javy itd.

Ponieważ `MyBank` przedstawia samodzielną aplikację, do utworzenia egzemplarza kontenera Springa możemy wykorzystać klasę `ClassPathXmlApplicationContext` lub `FileSystemXmlApplicationContext`. Warto zwrócić uwagę na fakt, że klasa `ClassPathXmlApplicationContext` wczytuje kontekst aplikacji z pliku XML wskazanego przez zmienną środowiskową `CLASSPATH`, natomiast klasa `FileSystemXmlApplicationContext` wczytuje kontekst aplikacji z pliku XML wskazanego przez zmienną środowiskową `FILESYSTEM`.

Przedstawiona w listingu 1.15 klasa `BankApp` aplikacji `MyBank` pokazuje, że egzemplarz kontenera Springa został utworzony za pomocą klasy `ClassPathXmlApplicationContext`.

Listing 1.15. Klasa `BankApp`

Projekt: `ch01-bankapp-xml`

Położenie pliku: `src/main/java/sample/spring/chapter01/bankapp`

```
package sample.spring.chapter01.bankapp;

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

```
public class BankApp {  
    .....  
    public static void main(String args[]) {  
        ApplicationContext context = new ClassPathXmlApplicationContext(  
            "classpath:META-INF/spring/applicationContext.xml");  
        .....  
    }  
}
```

Z powyższego listingu jasno wynika, że metoda `main()` klasy `BankApp` jest odpowiedzialna za przygotowanie kontenera Springa. Wartość zmiennej środowiskowej `CLASSPATH` wskazuje położenie pliku XML kontekstu aplikacji przekazywanego konstruktorowi klasy `ClassPathXmlApplicationContext` → `Context`. Utworzenie egzemplarza `ClassPathXmlApplicationContext` skutkuje utworzeniem tych komponentów, które w pliku XML kontekstu aplikacji są zdefiniowane jako *singleton* i określone jako *preinicjowane*. W rozdziale 2. poruszymy temat *zasięgu komponentów*. Ponadto dowiesz się, co oznacza *wstępna inicjacja* i *opóźniona inicjacja* kontenera Springa. Teraz możesz przyjąć założenie, że komponenty zdefiniowane w pliku `applicationContext.xml` aplikacji `MyBank` mają zasięg *singleton* i zostały określone jako *preinicjowane*. Oznacza to, że komponenty zdefiniowane w pliku `applicationContext.xml` są tworzone wraz z egzemplarzem klasy `ClassPathXmlApplicationContext`.

Skoro dowiedziałeś się, jak utworzyć egzemplarz kontenera Springa, możemy przejść do tematu pobierania egzemplarzy komponentów z kontenera Springa.

Uzyskanie dostępu do obiektów aplikacji z poziomu egzemplarza kontenera Springa

Obiekty aplikacji definiowane za pomocą elementów `<bean>` są tworzone i zarządzane przez kontener Springa. Do wspomnianych obiektów aplikacji możesz uzyskać dostęp przez wywołanie jednej z metod `getBean()` interfejsu `ApplicationContext`.

W listingu 1.16 zostało pokazane, że metoda `main()` klasy `BankApp` pobiera z kontenera Springa egzemplarz komponentu `FixedDepositController` i wywołuje jego metody.

Listing 1.16. Klasa `BankApp`

Projekt: `ch01-bankapp-xml`

Położenie pliku: `src/main/java/sample/spring/chapter01/bankapp`

```
package sample.spring.chapter01.bankapp;  
  
import org.apache.log4j.Logger;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class BankApp {  
    private static Logger logger = Logger.getLogger(BankApp.class);  
  
    public static void main(String args[]) {  
        ApplicationContext context = new ClassPathXmlApplicationContext(  
            "classpath:META-INF/spring/applicationContext.xml");  
  
        FixedDepositController fixedDepositController =  
            (FixedDepositController) context.getBean("controller");
```

```
        logger.info("Stan wysłania żądania informacji o lokacie : " +
                    fixedDepositController.submit());
        logger.info("Otrzymane informacje o lokacie : " +
                    fixedDepositController.get());
    }
}
```

Na początku metoda `getBean()` interfejsu `ApplicationContext` jest wywoływana w celu pobrania z kontenera Springa egzemplarza `FixedDepositController`. Następnie widzimy wywołania metod `submit()` i `get()` komponentu `FixedDepositController`. Argumentem przekazywanym metodzie `getBean()` jest nazwa komponentu, którego egzemplarz ma zostać pobrany z kontenera Springa. Nazwa komponentu przekazywana metodzie `getBean()` *musi* być wartością atrybutu `id` lub `name` komponentu przeznaczonego do pobrania. Jeżeli w kontenerze Springa nie jest zarejestrowany żaden komponent o podanej nazwie, wówczas metoda `getBean()` spowoduje zgłoszenie wyjątku.

W listingu 1.16, w celu przeprowadzenia konfiguracji egzemplarza `FixedDepositController` w sposób programowy, nie tworzymy egzemplarza `FixedDepositService` i nie przypisujemy go egzemplarzowi `FixedDepositController`. Ponadto nie tworzymy egzemplarza `FixedDepositDao` i nie przypisujemy go egzemplarzowi `FixedDepositService`. Wynika to z faktu, że zadanie utworzenia zależności i wstrzyknięcia ich do obiektów zależnych jest wykonywane przez kontener Springa.

Jeżeli przejdziesz do projektu *ch01-bankapp-xml* i wykonasz metodę `main()` klasy `BankApp`, wówczas w konsoli otrzymasz następujące dane wyjściowe:

```
INFO sample.spring.chapter01.bankapp.FixedDepositController - Inicjacja.
INFO sample.spring.chapter01.bankapp.FixedDepositService - Inicjacja.
INFO sample.spring.chapter01.bankapp.FixedDepositDao - Inicjacja.
INFO sample.spring.chapter01.bankapp.FixedDepositService - Ustawienie właściwości
↳fixedDepositDao.
INFO sample.spring.chapter01.bankapp.FixedDepositController - Ustawienie właściwości
↳fixedDepositService.
INFO sample.spring.chapter01.bankapp.BankApp - Stan wysłania żądania informacji o lokacie : true
INFO sample.spring.chapter01.bankapp.BankApp - Otrzymane informacje o lokacie : id :1, deposit
↳amount : 10000.0, tenure : 365, email : nazwa-uzytkownika@nazwa-domeny.pl
```

Powyższe dane wyjściowe pokazują, że kontener Springa utworzył egzemplarze wszystkich komponentów zdefiniowanych w pliku *applicationContext.xml* aplikacji *MyBank*. Ponadto kontener Springa używa opartego na metodzie typu setter mechanizmu wstrzykiwania zależności, aby wstrzyknąć egzemplarz `FixedDepositService` do egzemplarza `FixedDepositController` oraz egzemplarz `FixedDepositDao` do egzemplarza `FixedDepositService`.

Spójrzmy teraz na wybrane frameworki, które zostały zbudowane na bazie Springa.

1.6. Frameworki zbudowane na bazie Springa

Wprawdzie istnieje wiele frameworków opracowanych przez SpringSource na podstawie Springa, tutaj jednak wymienimy jedynie najpopularniejsze. Pełną listę dostępnych frameworków wraz informacjami szczegółowymi o każdym z nich znajdziesz w witrynie SpringSource (<http://spring.io/>).

W tabeli 1.2 przedstawiamy ogólny opis frameworków opracowanych przez SpringSource na podstawie Springa.

Tabela 1.2. Frameworki opracowane przez SpringSource na podstawie Springa

Nazwa	Opis
Spring Security	Framework uwierzytelniania i autoryzacji przeznaczony dla aplikacji korporacyjnych. W celu zaimplementowania funkcji uwierzytelniania i autoryzacji będziesz musiał w pliku XML kontekstu aplikacji skonfigurować kilka komponentów.
Spring Data	Ten framework zapewnia spójny model programowania przeznaczony do współpracy z różnymi rodzajami baz danych. Na przykład umożliwia pracę z nierelacyjnymi bazami danych, takimi jak MongoDB lub Neo4j. Można wykorzystać go także w celu uzyskania dostępu do relacyjnych baz danych za pomocą JPA.
Spring Batch	Jeżeli w aplikacji musisz zastosować przetwarzanie wsadowe, ten framework pomoże w implementacji tego rodzaju funkcjonalności.
Spring Integration	Dostarcza aplikacji możliwości w zakresie integracji aplikacji korporacyjnych (ang. <i>Enterprise Application Integration</i> , EAI).
Spring Social	Jeżeli aplikacja musi współdziałać z witrynami portali społecznościowych, takich jak Facebook lub Twitter, wówczas ten framework może okazać się niezwykle użyteczny.
Spring BlazeDS Integration	Jeżeli tworzysz aplikację opartą na technologii Adobe Flex, ten framework możesz wykorzystać w celu połączenia warstwy Flex z opartą na Springu warstwą biznesową.

Ponieważ frameworki wymienione w powyższej tabeli zostały zbudowane na bazie Springa, zanim spróbujesz użyć któregośkolwiek z nich, najpierw upewnij się, że jest zgodny z wykorzystywaną przez Ciebie wersją Springa.

1.7. Podsumowanie

W tym rozdziale poznałeś zalety użycia frameworku Spring. Przedstawiliśmy prostą aplikację Springa, na podstawie której dowiedziałeś się, jak zdefiniować metadane konfiguracji w formacie XML, utworzyć egzemplarz kontenera Springa oraz pobrać z niego komponenty. W kolejnym rozdziale zajmiemy się podstawowymi koncepcjami frameworku Spring.

Rozdział 2. Podstawy frameworku Spring

2.1. Wprowadzenie

Z poprzedniego rozdziału dowiedziałeś się, że kontener Springa wywołuje pozbawiony argumentów konstruktor klasy komponentu, aby utworzyć egzemplarz tego komponentu. Natomiast oparty na metodzie typu setter mechanizm wstrzykiwania zależności jest używany do spełnienia zależności komponentu. W tym rozdziale przejdziemy o krok dalej i poznamy kolejne zagadnienia:

- oferowana przez Spring obsługą reguły projektowej o nazwie „podejście oparte na programowaniu interfejsów”;
- różne podejścia w zakresie tworzenia egzemplarzy komponentów Springa;
- oparty na konstruktorze mechanizm wstrzykiwania zależności przeznaczony do przekazywania zależności komponentu jako argumentów konstruktora;
- oparte na konstruktorze i metodzie typu setter mechanizmy wstrzykiwania zależności przeznaczone do przekazywania komponentom prostych wartości typu String;
- zakresy komponentów.

Na początek przekonasz się, jak Spring poprawia możliwości w zakresie testowania aplikacji dzięki zaoferowaniu obsługi reguły projektowej o nazwie „podejście oparte na programowaniu interfejsów”.

2.2. Reguła projektowa o nazwie „podejście oparte na programowaniu interfejsów”

Z podrozdziału 1.5 dowiedziałeś się, że klasa zależna POJO zawiera odwołanie do konkretnej klasy zależności. Na przykład klasa `FixedDepositController` zawiera odwołanie do klasy `FixedDepositService`, natomiast `FixedDepositService` zawiera odwołanie do `FixedDepositDao`. Jeżeli klasa zależna ma bezpośrednie odwołanie do konkretnej klasy zależności, skutkiem będzie powstanie ścisłego powiązania między klasami. W tym przypadku chęć zastosowania innej implementacji zależności będzie wymagała zmiany klasy zależnej.

Spójrzmy teraz z na scenariusz, w którym klasa zależna zawiera bezpośrednie odwołanie do konkretnej klasy zależności.

Scenariusz: klasa zależna zawiera odwołanie do konkretnej klasy zależności

Przyjmujemy założenie, że klasa `FixedDepositDao` używa zwykłego interfejsu JDBC do współpracy z bazą danych. W celu ułatwienia pracy z bazą danych stworzymy kolejną implementację DAO o nazwie `FixedDepositHibernateDao`, która wykorzystuje Hibernate ORM do przeprowadzania operacji w bazie danych. Teraz przejście z rozwiązania opartego na zwykłym interfejsie JDBC do rozwiązania używającego implementacji Hibernate ORM wymaga zmiany klasy `FixedDepositService`, aby stosowana była klasa `FixedDepositHibernateDao` zamiast `FixedDepositDao`, jak to przedstawia listing 2.1.

Listing 2.1. *Klasa FixedDepositService*

```
public class FixedDepositService {  
    private FixedDepositHibernateDao fixedDepositDao;  
  
    public void setFixedDepositDao(FixedDepositHibernateDao fixedDepositDao) {  
        this.fixedDepositDao = fixedDepositDao;  
    }  
  
    public FixedDepositDetails getFixedDepositDetails(long id) {  
        return fixedDepositDao.getFixedDepositDetails(id);  
    }  
  
    public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) {  
        return fixedDepositDao.createFixedDeposit(fixedDepositDetails);  
    }  
}
```

W listingu 2.1 widzimy, że odwołanie do klasy FixedDepositDao zostało zastąpione odwołaniem do klasy FixedDepositHibernateDao, aby współpraca z bazą danych odbywała się za pomocą Hibernate ORM. Ten przykład pokazuje, że jeżeli klasa zależna odwołuje się do konkretnej klasy implementacji zależności, to użycie innej implementacji wymaga wprowadzenia zmian w klasie zależnej.

Spójrzmy teraz na inny scenariusz, w którym klasa zależna zawiera odwołanie do interfejsu implementowanego przez zależność.

Scenariusz: klasa zależna zawiera odwołanie do interfejsu implementowanego przez zależność

Wiemy, że interfejs Javy definiuje schemat, z którym implementacja klasy zachowuje zgodność. Dlatego też jeżeli klasa zależy od interfejsu implementowanego przez zależność, wówczas w klasie nie trzeba wprowadzać żadnych modyfikacji, gdy nastąpi zmiana implementacji zależności. Podejście do projektowania aplikacji w sposób polegający na tym, że klasa zależy od interfejsu implementowanego przez zależność, nosi nazwę *podejścia opartego na programowaniu interfejsu*. Interfejs implementowany przez klasę zależności jest określany mianem *interfejsu zależności*.

Wprowadzenie *podejścia opartego na programowaniu interfejsu* jest lepszą praktyką programistyczną niż *oparte na programowaniu klas*, ale diagram na rysunku 2.1 pokazuje, że rozwiązanie okazuje się dobre, gdy klasa ABean zależy od interfejsu BBean, a *nie* od klasy BBeanImpl implementującej interfejs BBean.

Rysunek 2.1.

Podejście oparte na programowaniu interfejsu jest lepsze niż podejście oparte na programowaniu klas

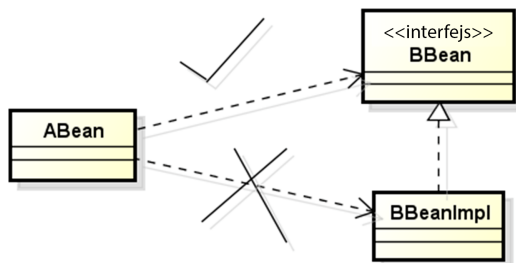
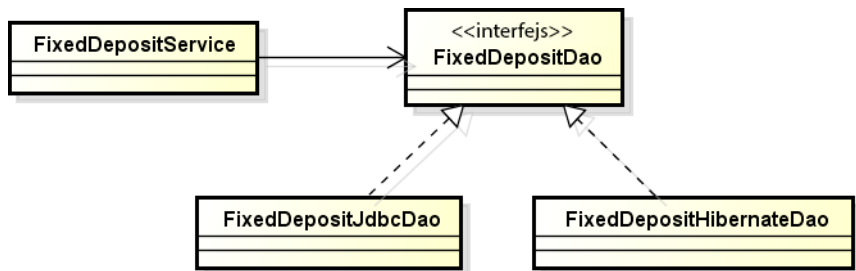


Diagram na rysunku 2.2 pokazuje, jak klasa `FixedDepositService` może wykorzystać podejście *oparte na programowaniu klas* w celu łatwej zmiany strategii przyjętej podczas współpracy z bazą danych.



Rysunek 2.2. *FixedDepositService* zależy od interfejsu *FixedDepositDao*, który został zaimplementowany przez klasy *FixedDepositJdbcDao* i *FixedDepositHibernateDao*

Na rysunku 2.2 widać, że klasa `FixedDepositService` nie jest bezpośrednio zależna od klas `FixedDepositJdbcDao` i `FixedDepositHibernateDao`. Zamiast tego klasa `FixedDepositService` zależy od interfejsu `FixedDepositDao` (czyli od interfejsu zależności) implementowanego przez klasy `FixedDepositJdbcDao` i `FixedDepositHibernateDao`. Teraz w zależności od rozwiązania, które chcesz zastosować — zwykle JDBC czy framework ORM, na przykład Hibernate — egzemplarzowi `FixedDepositService` musisz dostarczyć egzemplarz `FixedDepositJdbcDao` lub `FixedDepositHibernateDao`.

Ponieważ `FixedDepositService` zależy od interfejsu `FixedDepositDao`, zachowujesz możliwość zastosowania w przyszłości innych rozwiązań w zakresie współpracy z bazą danych. Przyjmujemy założenie, że do pracy z bazą danych zdecydowałeś się na użycie frameworku iBATIS (teraz o nazwie zmienionej na MyBatis). Framework iBATIS możesz zastosować bez wprowadzania jakichkolwiek zmian w klasie `FixedDepositService`. Wystarczy jedynie utworzyć nową klasę `FixedDepositIbatisDao` implementującą interfejs `FixedDepositDao`, a następnie egzemplarzowi `FixedDepositService` dostarczyć egzemplarz `FixedDepositIbatisDao`.

Jak dotąd zobaczyłeś, podejście *oparte na programowaniu interfejsu* skutkuje zmniejszeniem stopnia powiązania zachodzącego między klasą zależną i jej zależnościami. Przekonamy się teraz, jak to podejście projektowe przyczynia się do poprawy możliwości testowania klasy zależnej.

Większe możliwości w zakresie testowania klasy zależnej

Na rysunku 2.2 można zobaczyć, że klasa `FixedDepositService` zawiera odwołanie do interfejsu `FixedDepositDao`. `FixedDepositJdbcDao` i `FixedDepositHibernateDao` to konkretne implementacje klas interfejsu `FixedDepositDao`. Teraz w celu ułatwienia przeprowadzenia testów jednostkowych klasy `FixedDepositService` interfejs `FixedDepositDao` możesz zastąpić makietą, która nie wymaga bazy danych.

Jeżeli klasa `FixedDepositService` zawierałaby bezpośrednie odwołanie do klasy `FixedDepositJdbcDao` lub `FixedDepositHibernateDao`, przetestowanie `FixedDepositService` wymagałoby skonfigurowania bazy danych na potrzeby testu. Dlatego też korzystając z implementacji makiety interfejsu, możesz zaoszczędzić czas i uniknąć konieczności przygotowania infrastruktury niezbędnej do przeprowadzenia testów jednostkowych klas zależnych.

Zobaczmy teraz, jak Spring obsługuje w aplikacjach podejście *oparte na programowaniu interfejsu*.

Oferowana przez Spring obsługa podejścia opartego na programowaniu interfejsu

W celu użycia w aplikacji Springa podejścia *opartego na programowaniu interfejsu* musisz zwrócić uwagę na następujące kwestie:

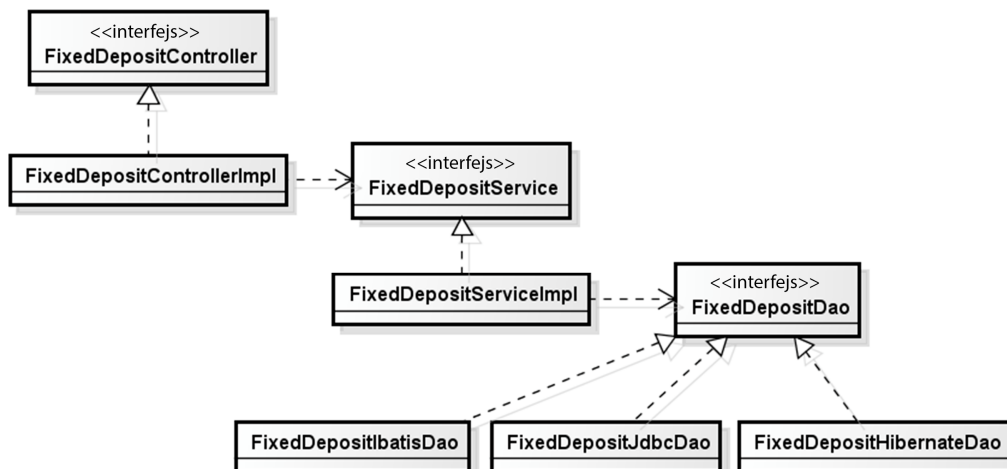
- Elementy `<bean>` w metadanych konfiguracyjnych powinny wskazywać konkretne klasy zależności.
- Klasy komponentu zależnego odwołują się do interfejsu zależności zamiast do konkretnej klasy zależności.

Przekonajmy się teraz, jak zmodyfikowana wersja naszej aplikacji MyBank używa podejścia *opartego na programowaniu interfejsu*.

IMPORT **Rozdział 102/ch02-bankapp-interfaces.** (Celem tego projektu jest pokazanie, jak można wykorzystać podejście *oparte na programowaniu interfejsu* podczas tworzenia aplikacji Springa. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `MyBankApp` projektu).

Aplikacja MyBank używająca podejścia opartego na programowaniu interfejsu

Diagram na rysunku 2.3 pokazuje, jak zmodyfikowana wersja naszej aplikacji MyBank używa podejścia *opartego na programowaniu interfejsu*.



Rysunek 2.3. Aplikacja MyBank używająca podejścia opartego na programowaniu interfejsu

Na rysunku 2.3 widać, że klasa zależna posiada zależność od interfejsu implementowanego przez zależność, a *nie* od konkretnej implementacji klasy zależności. Na przykład klasa `FixedDepositControllerImpl` zależy od interfejsu `FixedDepositService`, natomiast klasa `FixedDepositServiceImpl` zależy od interfejsu `FixedDepositDao`.

Analizując kod w listingu 2.2, z łatwością dostrzeżesz, że klasa `FixedDepositServiceImpl` została oparta na projekcie pokazanym na rysunku 2.3.

Listing 2.2. *Klasa FixedDepositService*Projekt: *ch02-bankapp-interfaces*Położenie pliku: *src/main/java/sample/spring/chapter02/bankapp*

```
package sample.spring.chapter02.bankapp;

public class FixedDepositServiceImpl implements FixedDepositService {
    private FixedDepositDao fixedDepositDao;
    ....
    public void setFixedDepositDao(FixedDepositDao fixedDepositDao) {
        this.fixedDepositDao = fixedDepositDao;
    }

    public FixedDepositDetails getFixedDepositDetails(long id) {
        return fixedDepositDao.getFixedDepositDetails(id);
    }

    public boolean createFixedDeposit(FixedDepositDetails fdd) {
        return fixedDepositDao.createFixedDeposit(fdd);
    }
}
```

Z powyższego listingu wynika, że klasa `FixedDepositServiceImpl` zawiera odwołanie do interfejsu `FixedDepositDao`. Implementacja `FixedDepositDao`, która ma zostać wstrzyknięta do egzemplarza `FixedDepositServiceImpl`, jest wskazana w pliku XML kontekstu aplikacji. Jak możesz się przekonać, patrząc na rysunek 2.3, masz możliwość wstrzyknięcia dowolnej z konkretnych implementacji interfejsu `FixedDepositDao`: `FixedDepositIbatisDao`, `FixedDepositJdbcDao` lub `FixedDeposit` ↪ `HibernateDao`.

Zawartość pliku *applicationContext.xml* definiującego projekt pokazany na rysunku 2.3 została przedstawiona w listingu 2.3.

Listing 2.3. *Zawartość pliku applicationContext.xml, czyli pliku XML kontekstu aplikacji MyBank*Projekt: *ch02-bankapp-interfaces*Położenie pliku: *src/main/resources/META-INF/spring*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans .....>

    <bean id="controller"
        class="sample.spring.chapter02.bankapp.controller.FixedDepositControllerImpl">
        <property name="fixedDepositService" ref="service" />
    </bean>

    <bean id="service"
        class="sample.spring.chapter02.bankapp.service.FixedDepositServiceImpl">
        <property name="fixedDepositDao" ref="dao" />
    </bean>

    <bean id="dao"
        class="sample.spring.chapter02.bankapp.dao.FixedDepositHibernateDao"/>
</beans>
```

Plik `applicationContext.xml`, którego kod przedstawiamy w listingu 2.3, pokazuje, że egzemplarz `FixedDepositHibernateDao` (implementacja interfejsu `FixedDepositDao`) jest wstrzykiwany do `FixedDepositServiceImpl`. Jeżeli teraz zdecydujesz się na użycie frameworku iBATIS zamiast Hibernate do trwałego przechowywania danych, wówczas musisz jedynie zmienić wartość atrybutu `class` definicji komponentu dao w pliku `applicationContext.xml`. Nowa wartość powinna odwoływać się do w pełni kwalifikowanej nazwy `FixedDepositIbatisDao`.

Spójrzmy teraz na różne sposoby, na jakie kontener Springa może tworzyć komponenty.

2.3. Różne podejścia podczas tworzenia komponentów Springa

Jak dotąd widziałeś przykłady definicji komponentów, które nakazywały kontenerowi Springa utworzenie egzemplarza komponentu przez wywołanie *bezargumentowego* konstruktora klasy komponentu. Spójrz na następującą definicję komponentu:

```
<bean id="myBean" class="mypackage.MyBean" />
```

W powyższej definicji komponentu klasa `MyBean` przedstawia klasę POJO definiującą bezargumentowy konstruktor. Klasa `MyBean` nie implementuje żadnego charakterystycznego dla Springa interfejsu ani nie rozszerza żadnej z klas Springa. W praktyce oznacza to, że kontener Springa może utworzyć egzemplarz dowolnej klasy dostarczającej bezargumentowy konstruktor i nim zarządzać.



Trzeba koniecznie zwrócić uwagę na fakt, że kontener Springa ma możliwość utworzenia egzemplarza dowolnej klasy i zarządzania nim, niezależnie od tego, czy dostarcza ona bezargumentowy konstruktor. W podrozdziale 2.4 przyjrzymy się definicjom komponentów, w których konstruktor akceptuje jeden lub więcej argumentów.

Jeżeli masz istniejący projekt używający klas fabryki do tworzenia egzemplarzy obiektu, nadal możesz wykorzystać kontener Springa do zarządzania obiektami utworzonymi przez wspomniane metody fabryki. Dowiesz się teraz, jak kontener Springa wywołuje *statyczną* metodę fabryki w celu zarządzania otrzymanym egzemplarzem obiektu.

Utworzenie komponentu za pomocą statycznej metody fabryki

Na rysunku 2.3 zostało pokazane, że interfejs `FixedDepositDao` jest implementowany przez klasy `FixedDepositHibernateDao`, `FixedDepositIbatisDao` i `FixedDepositJdbcDao`. W listingu 2.4 możesz zobaczyć, że klasa `FixedDepositDaoFactory` posiada *statyczną* metodę fabryki odpowiedzialną za tworzenie i zwrot egzemplarza `FixedDepositDao` na podstawie argumentu przekazanego metodzie *statycznej*.

Listing 2.4. Klasa `FixedDepositDaoFactory`

```
public class FixedDepositDaoFactory {  
    private FixedDepositDaoFactory() { }  
  
    public static FixedDepositDao getFixedDepositDao(String daoType) {  
        FixedDepositDao fixedDepositDao = null;  
    }  
}
```

```

        if("jdbc".equalsIgnoreCase(daoType)) {
            fixedDepositDao = new FixedDepositJdbcDao();
        }
        if("hibernate".equalsIgnoreCase(daoType)) {
            fixedDepositDao = new FixedDepositHibernateDao();
        }
        ....
        return fixedDepositDao;
    }
}

```

Z listingu 2.4 wynika, że klasa `FixedDepositDaoFactory` definiuje metodę *statyczną* `getFixedDepositDao()` odpowiedzialną za utworzenie i zwrot egzemplarza klasy `FixedDepositJdbcDao`, `FixedDepositHibernateDao` lub `FixedDepositIbatisDao`, w zależności od wartości argumentu `daoType`.

Przedstawiona w listingu 2.5 definicja klasy `FixedDepositDaoFactory` nakazuje kontenerowi Spring wywołanie metod `getFixedDepositDao()` klasy `FixedDepositDaoFactory` w celu otrzymania egzemplarza klasy `FixedDepositJdbcDao`.

Listing 2.5. Definicja komponentu dla klasy `FixedDepositDaoFactory`

```

<bean id="dao" class="sample.spring.FixedDepositDaoFactory"
      factory-method="getFixedDepositDao">
    <constructor-arg index="0" value="jdbc"/>
</bean>

```

W definicji przedstawionej w listingu 2.5 atrybut `class` wskazuje w pełni kwalifikowaną nazwę klasy definiującą *statyczną* metodę fabryki. Atrybut `factory-method` określa nazwę *statycznej* metody fabryki, która przez kontener Springa będzie wywołana w celu pobrania egzemplarza obiektu `FixedDepositDao`. Element `<constructor-arg>` jest definiowany w schemacie beans i używany w celu przekazywania argumentów do konstruktorów, a także metodom fabryki zarówno *statycznym*, jak i *egzemplarza*. Atrybut `index` odwołuje się do położenia argumentu w konstruktorze bądź też metodzie fabryki zarówno *statycznej*, jak i *egzemplarza*. W omawianej definicji komponentu wartość 0 atrybutu `index` oznacza, że element `<constructor-arg>` dostarcza wartość dla pierwszego argumentu (tutaj `daoType`) metody fabryki `getFixedDepositDao()`. Atrybut `value` określa wartość argumentu. Jeżeli metoda fabryki akceptuje wiele argumentów, konieczne jest zdefiniowanie elementu `<constructor-arg>` dla każdego z nich.

W tym miejscu trzeba koniecznie wspomnieć, że wywołanie metody `getBean()` klasy `ApplicationContext` w celu uzyskania komponentu `dao` (patrz: listing 2.5) spowoduje wywołanie metody fabryki `getFixedDepositDao()` klasy `FixedDepositDaoFactory`. Dlatego wartością zwrótną wywołania `getBean("dao")` będzie egzemplarz `FixedDepositDao` utworzony przez metodę fabryki `getFixedDepositDao()`, a nie egzemplarz klasy `FixedDepositDaoFactory`.

Skoro poznałeś już konfigurację klasy fabryki odpowiedzialnej za tworzenie egzemplarza `FixedDepositDao`, teraz możemy przejść do kolejnego fragmentu kodu, pokazującego, jak wstrzyknąć egzemplarz `FixedDepositDao` do `FixedDepositServiceImpl`.

W listingu 2.6 egzemplarz `FixedDepositDao` zwrócony przez metodę fabryki `getFixedDepositDao()` klasy `FixedDepositDaoFactory` zostaje przez element `<property>` wstrzyknięty do egzemplarza

Listing 2.6. Wstrzyknięcie egzemplarzy obiektów utworzonych przez statyczną metodę fabryki

```
<bean id="service" class="sample.spring.chapter02.bankapp.FixedDepositServiceImpl">
  <property name="fixedDepositDao" ref="dao" />
</bean>

<bean id="dao" class="sample.spring.chapter02.basicapp.FixedDepositDaoFactory"
      factory-method="getFixedDepositDao">
  <constructor-arg index="0" value="jdbc"/>
</bean>
```

FixedDepositServiceImpl. Jeżeli definicję komponentu dla FixedDepositServiceImpl porównasz z definicją przedstawioną w listingu 2.3, zauważysz, że są one takie same. To pokazuje, że zależności komponentu są wyrażone w taki sam sposób niezależnie od sposobu (pozbawiony argumentów konstruktor lub *statyczna* metoda fabryki), w jaki kontener Springa tworzy egzemplarze komponentu.

Zobaczmy teraz, jak kontener Springa tworzy komponenty za pomocą wywołania metody fabryki *egzemplarza*.

Tworzenie komponentu za pomocą metody fabryki egzemplarza

Kod w listingu 2.7 pokazuje klasę FixedDepositDaoFactory zawierającą metodę fabryki *egzemplarza* przeznaczoną do tworzenia i przekazywania egzemplarza FixedDepositDao.

Listing 2.7. Klasa FixedDepositDaoFactory

```
public class FixedDepositDaoFactory {
    public FixedDepositDaoFactory() {
    }

    public FixedDepositDao getFixedDepositDao(String daoType) {
        FixedDepositDao fixedDepositDao = null;

        if("jdbc".equalsIgnoreCase(daoType)) {
            fixedDepositDao = new FixedDepositJdbcDao();
        }
        if("hibernate".equalsIgnoreCase(daoType)) {
            fixedDepositDao = new FixedDepositHibernateDao();
        }
        ....
        return fixedDepositDao;
    }
}
```

Jeżeli klasa definiuje metodę fabryki *egzemplarza*, wówczas klasa musi zawierać publiczny (public) konstruktor, aby kontener Springa mógł utworzyć egzemplarz tej klasy. W listingu 2.7 klasa FixedDepositDaoFactory ma zdefiniowany publiczny konstruktor bezargumentowy. Metoda getFixedDepositDao() klasy FixedDepositDaoFactory to metoda fabryki *egzemplarza*, która jest odpowiedzialna za utworzenie i zwrot egzemplarza FixedDepositDao.

W listingu 2.8 pokazujemy, jak nakazać kontenerowi Springa wywołanie metody getFixedDepositDao() klasy FixedDepositDaoFactory w celu otrzymania egzemplarza FixedDepositDao.

Listing 2.8. Konfiguracja pozwalająca na wywołanie metody `getFixedDepositDao()` klasy `FixedDepositDaoFactory`

```
<bean id="daoFactory" class="sample.spring.chapter02.basicapp.FixedDepositDaoFactory" />

<bean id="dao" factory-bean="daoFactory" factory-method="getFixedDepositDao">
  <constructor-arg index="0" value="jdbc"/>
</bean>

<bean id="service" class="sample.spring.chapter02.bankapp.FixedDepositServiceImpl">
  <property name="fixedDepositDao" ref="dao" />
</bean>
```

W listingu 2.8 pokazujemy, że klasa `FixedDepositDaoFactory` (ta klasa zawiera metodę fabryki *egzemplarza*) została skonfigurowana podobnie jak zwykły komponent Springa. Oddzielny element `<bean>` został wykorzystany do konfiguracji szczegółowej metody fabryki *egzemplarza*. Aby podać szczegóły konfiguracji metody fabryki *egzemplarza*, użyto atrybutów `factory-bean` i `factory-method` elementu `<bean>`. Atrybut `factory-bean` odwołuje się do komponentu definiującego metodę fabryki *egzemplarza*, natomiast atrybut `factory-method` określa nazwę tej metody. W omawianym listingu egzemplarz `FixedDepositDao` zwrócony przez metodę fabryki `getFixedDepositDao()` klasy `FixedDepositDaoFactory` jest przez element `<property>` wstrzykiwany do egzemplarza `FixedDepositServiceImpl`.

Podobnie jak w przypadku *statycznej* metody fabryki, istnieje możliwość przekazania argumentów metodzie fabryki *egzemplarza* za pomocą elementu `<constructor-arg>`. Trzeba w tym miejscu koniecznie zwrócić uwagę na fakt, że wywołanie w listingu 2.8 metody `getBean()` klasy `ApplicationContext` w celu uzyskania komponentu `dao` skutkuje wywołaniem metody fabryki `getFixedDepositDao()` klasy `FixedDepositDaoFactory`.

Dotychczas poznałeś przykłady definicji komponentów, w których zależności są wstrzykiwane za pomocą metod typu setter. Przejdziemy teraz do różnych mechanizmów wstrzykiwania zależności dostępnych do użycia w celu spełnienia zależności.

2.4. Techniki wstrzykiwania zależności

W Springu wstrzykiwanie zależności odbywa się przez przekazanie argumentów do konstruktora komponentu i metod typu setter. Jeżeli do utworzenia egzemplarza komponentu używana jest metoda fabryki (zarówno *statyczna*, jak i *egzemplarza*), wówczas zależności komponentu można przekazać metodzie fabryki lub też ustawić je w egzemplarzu zwróconym przez metodę fabryki.

Przejdziemy teraz do przykładów demonstrujących różne techniki wstrzykiwania zależności.

Wstrzykiwanie zależności za pomocą metody typu setter

Dotychczas przedstawiliśmy w tej książce przykłady wstrzykiwania zależności za pomocą metod typu setter. W tego rodzaju wstrzykiwaniu zależności element `<property>` jest używany do określenia zależności komponentu. Element `<property>` jest ponadto używany do przekazania *informacji konfiguracyjnych*, o ile są one wymagane przez komponent.

Przyjmujemy założenie, że aplikacja MyBank zawiera usługę `PersonalBankingService` umożliwiającą klientom pobieranie wyciągu z konta bankowego, sprawdzanie informacji szczegółowych o koncie, uaktualnianie informacji o posiadaczu konta, zmianę hasła dostępu, a także kontakt z pomocą techniczną. W celu dostarczenia wymaganej funkcjonalności klasa `PersonalBankingService` używa `JmsMessageSender` (do wysyłania komunikatów JMS), `EmailMessageSender` (do wysyłania wiadomości e-mail) i obiektów `WebServiceInvoker` (do wywoływania zewnętrznych usług sieciowych). W listingu 2.9 przedstawiamy kod klasy `PersonalBankingService`.

Listing 2.9. Klasa `PersonalBankingService`

```
public class PersonalBankingService {
    private JmsMessageSender jmsMessageSender;
    private EmailMessageSender emailMessageSender;
    private WebServiceInvoker webServiceInvoker;
    .....
    public void setJmsMessageSender(JmsMessageSender jmsMessageSender) {
        this.jmsMessageSender = jmsMessageSender;
    }

    public void setEmailMessageSender(EmailMessageSender emailMessageSender) {
        this.emailMessageSender = emailMessageSender;
    }

    public void setWebServiceInvoker(WebServiceInvoker webServiceInvoker) {
        this.webServiceInvoker = webServiceInvoker;
    }
    .....
}
```

Jak możesz zobaczyć w listingu 2.9, metoda typu setter jest zdefiniowana dla zależności `JmsMessageSender`, `EmailMessageSender` i `WebServiceInvoker` klasy `PersonalBankingService`.

Wstrzykiwanie zależności za pomocą metody typu setter możemy więc wykorzystać do spełnienia zależności klasy `PersonalBankingService`, jak zostało przedstawione w listingu 2.10.

Listing 2.10. Definicje komponentów dla klasy `PersonalBankingService` i jej zależności

```
<bean id="personalBankingService" class="PersonalBankingService">
    <property name="emailMessageSender" ref="emailMessageSender" />
    <property name="jmsMessageSender" ref="jmsMessageSender" />
    <property name="webServiceInvoker" ref="webServiceInvoker" />
</bean>

<bean id="jmsMessageSender" class="JmsMessageSender">
    .....
</bean>
<bean id="webServiceInvoker" class="WebServiceInvoker" />
    .....
</bean>
<bean id="emailMessageSender" class="EmailMessageSender" />
    .....
</bean>
```


Definicja komponentu `PersonalBankingService` pokazuje, że dla każdej zależności klasy `PersonalBankingService` został przygotowany element `<property>`.

Klasa `PersonalBankingService` używa komponentu `EmailMessageSender` w celu wysyłania na adres e-mail podany przez klienta powiadomień, jeśli zmieni on numer kontaktowy. Klasa `EmailMessageSender` wymaga adresu serwera poczty, a także nazwy użytkownika i hasła do uwierzytelnienia w serwerze poczty. W przedstawionym poniżej listingu 2.11 pokazujemy, że element `<property>` można również użyć do ustawienia właściwości komponentu typu `String`.

Listing 2.11. Klasa `EmailMessageSender` i odpowiadająca jej definicja komponentu

```
public class EmailMessageSender {
    private String host;
    private String username;
    private String password;
    ....
    public void setHost(String host) {
        this.host = host;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }
    ....
}

<bean id="emailMessageSender" class="EmailMessageSender">
    <property name="host" value="smtp.gmail.com"/>
    <property name="username" value="myusername"/>
    <property name="password" value="mypassword"/>
</bean>
```

W listingu 2.11 elementy `<property>` zostały użyte do ustawienia wartości właściwości `host`, `username` i `password` komponentu `EmailMessageSender`. Atrybut `value` określa wartość typu `String`, która będzie ustawiona dla właściwości komponentu wskazanej przez atrybut `name`. Właściwości `host`, `username` i `password` przedstawiają informacje konfiguracyjne wymagane przez komponent `EmailMessageSender`. W rozdziale 3. zobaczysz, jak element `<property>` jest wykorzystywany do ustawienia wartości typu podstawowego (np. `int`, `long`), kolekcji (np. `java.util.List`, `java.util.Map`), a także niestandardowego typu właściwości (np. `Address`).

Wstrzykiwanie zależności oparte na metodach typu `setter` jest używane także w celu wstrzykiwania zależności do komponentów tworzonych za pomocą metody fabryki zarówno *statycznej*, jak i *egzemplarza*. Zobaczymy więc, jak wstrzykiwania zależności opartego na metodach typu `setter` używać w połączeniu z metodami fabryki zarówno *statycznymi*, jak i *egzemplarza*.

Wstrzykiwanie zależności do egzemplarza komponentu utworzonego przez metodę fabryki

Wstrzykiwania zależności opartego na metodach typu setter możemy użyć w celu wstrzykiwania zależności do egzemplarza komponentu utworzonego za pomocą metody fabryki zarówno *statycznej*, jak i *egzemplarza*.

Spójrz na klasę `FixedDepositJdbcDao` w listingu 2.12 definiującą właściwość `databaseInfo`.

Listing 2.12. Klasa `FixedDepositJdbcDao`

```
public class FixedDepositJdbcDao {  
    private DatabaseInfo databaseInfo;  
    .....  
    public FixedDepositJdbcDao() { }  
  
    public void setDatabaseInfo(DatabaseInfo databaseInfo) {  
        this.databaseInfo = databaseInfo;  
    }  
    .....  
}
```

W listingu 2.12 atrybut `databaseInfo` przedstawia zależność klasy `FixedDepositJdbcDao` spełnioną przez metodę `setDatabaseInfo()`.

Z kolei klasa `FixedDepositDaoFactory` przedstawiona w listingu 2.13 definiuje metodę fabryki odpowiedzialną za utworzenie i zwrócenie egzemplarza klasy `FixedDepositDaoJdbc`.

Listing 2.13. Klasa `FixedDepositDaoFactory`

```
public class FixedDepositDaoFactory {  
    public FixedDepositDaoFactory() {  
    }  
  
    public FixedDepositDao getFixedDepositDao(String daoType) {  
        FixedDepositDao fixedDepositDao = null;  
  
        if("jdbc".equalsIgnoreCase(daoType)) {  
            FixedDepositDao = new FixedDepositJdbcDao();  
        }  
        if("hibernate".equalsIgnoreCase(daoType)) {  
            FixedDepositDao = new FixedDepositHibernateDao();  
        }  
        .....  
        return fixedDepositDao;  
    }  
}
```

W listingu 2.13 metoda `getFixedDepositDao()` jest metodą fabryki *egzemplarza* przeznaczoną do tworzenia egzemplarzy klasy `FixedDepositDao`. Działanie metody `getFixedDepositDao()` polega na utworzeniu egzemplarza klasy `FixedDepositJdbcDao`, o ile wartością argumentu `daoType` jest `jdbc`. Trzeba koniecznie zwrócić uwagę na fakt, że metoda `getFixedDepositDao()` nie przypisuje właściwości `databaseInfo` wartości `FixedDepositJdbcDao`.

Jak mogłeś zobaczyć w listingu 2.8, przedstawione w listingu 2.14 definicje komponentów nakazują kontenerowi Springa utworzenie egzemplarza `FixedDepositJdbcDao` przez wywołanie metody fabryki `getFixedDepositDao()` klasy `FixedDepositDaoFactory`.

Listing 2.14. Konfiguracja powodująca wywołanie metody `getFixedDepositDao()` klasy `FixedDepositDaoFactory`

```
<bean id="daoFactory" class="FixedDepositDaoFactory" />
<bean id="dao" factory-bean="daoFactory" factory-method="getFixedDepositDao">
  <constructor-arg index="0" value="jdbc"/>
</bean>
```

Definicja komponentu dao skutkuje wywołaniem metody `getFixedDepositDao()` klasy `FixedDepositDaoFactory`, która tworzy i zwraca egzemplarz `FixedDepositJdbcDao`. Jednak właściwość `databaseInfo` klasy `FixedDepositJdbcDao` nie jest ustawiana. W celu spełnienia zależności `databaseInfo` można przeprowadzić oparte na metodzie typu setter wstrzyknięcie zależności w egzemplarzu `FixedDepositJdbcDao` zwróconym przez metodę `getFixedDepositDao()`, jak przedstawia listing 2.15.

Listing 2.15. Konfiguracja pozwalająca na wywołanie metody `getFixedDepositDao()` klasy `FixedDepositDaoFactory` oraz ustawienie właściwości `databaseInfo` zwróconego egzemplarza `FixedDepositJdbcDao`

```
<bean id="daoFactory" class="FixedDepositDaoFactory" />

<bean id="dao" factory-bean="daoFactory" factory-method="getFixedDepositDao">
  <constructor-arg index="0" value="jdbc"/>
  <property name="databaseInfo" ref="databaseInfo"/>
</bean>

<bean id="databaseInfo" class="DatabaseInfo" />
```

Definicja komponentu w listingu 2.15 pokazuje, że element `<property>` jest używany do ustawienia właściwości `databaseInfo` egzemplarza `FixedDepositJdbcDao` zwróconego przez metodę fabryki `getFixedDepositDao()`. Ponieważ to jest metoda fabryki *egzemplarza*, elementu `<property>` można użyć w celu wstrzyknięcia zależności do egzemplarza komponentu zwróconego przez *statyczną* metodę fabryki.

Spójrzmy teraz, jak można wstrzykiwać zależności komponentów za pomocą argumentów konstruktora.

Wstrzykiwanie zależności za pomocą argumentów konstruktora

W omawianym tutaj przypadku wstrzykiwania zależności wspomniane zależności komponentu są przekazywane w postaci argumentów konstruktora klasy. Na przykład listing 2.16 pokazuje klasę `PersonalBankingService`, której konstruktor akceptuje obiekty `JmsMessageSender`, `EmailMessageSender` i `WebServiceInvoker`.

Listing 2.16. Klasa `PersonalBankingService`

```
public class PersonalBankingService {
    private JmsMessageSender jmsMessageSender;
    private EmailMessageSender emailMessageSender;
    private WebServiceInvoker webServiceInvoker;
```

```

.....
public PersonalBankingService(JmsMessageSender jmsMessageSender,
    EmailMessageSender emailMessageSender,
    WebServiceInvoker webServiceInvoker) {

    this.jmsMessageSender = jmsMessageSender;
    this.emailMessageSender = emailMessageSender;
    this.webServiceInvoker = webServiceInvoker;
}
.....
}

```

Argumenty konstruktora klasy `PersonalBankingService` przedstawiają zależności wymienionej klasy. W listingu 2.17 pokazujemy, jak zależności egzemplarza `PersonalBankingService` są dostarczane za pomocą elementów `<constructor-arg>`.

Listing 2.17. Definicja komponentu *PersonalBankingService*

```

<bean id="personalBankingService" class="PersonalBankingService">
  <constructor-arg index="0" ref="jmsMessageSender" />
  <constructor-arg index="1" ref="emailMessageSender" />
  <constructor-arg index="2" ref="webServiceInvoker" />
</bean>

<bean id="jmsMessageSender" class="JmsMessageSender">
  .....
</bean>
<bean id="webServiceInvoker" class="WebServiceInvoker" />
  .....
</bean>
<bean id="emailMessageSender" class="EmailMessageSender" />
  .....
</bean>

```

W listingu 2.17 elementy `<constructor-arg>` zawierają informacje szczegółowe o argumentach konstruktora przekazywanych egzemplarzowi `PersonalBankingService`. Atrybut `index` określa indeks argumentu konstruktora. Jeżeli wartość atrybutu `index` wynosi 0, oznacza to, że element `<constructor-arg>` odpowiada pierwszemu argumentowi konstruktora. Wartość 1 oznacza, że element `<constructor-arg>` odpowiada drugiemu argumentowi konstruktora, itd. Wcześniej dowiedziałeś się, że atrybut `ref` elementu `<property>` jest używany w celu przekazania odwołania do komponentu. Podobnie atrybut `ref` elementu `<constructor-arg>` jest używany w celu przekazania informacji konfiguracyjnych, o ile są one wymagane przez komponent.

Warto zwrócić uwagę na fakt, że element `<constructor-arg>` jest używany także w celu przekazywania argumentów do metod fabryki zarówno *statycznych*, jak i *egzemplarza*, które są odpowiedzialne za tworzenie egzemplarzy komponentów (patrz: podrozdział 2.3).



Aby ustawić odwołania do komponentów, można, zamiast używać atrybutu `ref` elementów `<property>` i `<constructor-arg>`, wykorzystać element `<ref>` wewnątrz elementów `<property>` i `<constructor-arg>`. Preferowane jest stosowanie atrybutu `ref`, ponieważ pozwala on na uproszczenie kodu XML.

W listingu 2.18 została przedstawiona klasa `EmailMessageSender` i odpowiadająca jej definicja komponentu, która pokazuje użycie elementów `<constructor-arg>` do dostarczenia wartości typu `String` argumentom konstruktora.

Listing 2.18. Klasa `EmailMessageSender` i odpowiadająca jej definicja komponentu

```
public class EmailMessageSender {
    private String host;
    private String username;
    private String password;
    ....
    public EmailMessageSender(String host, String username, String password) {
        this.host = host;
        this.username = username;
        this.password = password;
    }
    ....
}

<bean id="emailMessageSender" class="EmailMessageSender">
    <constructor-arg index="0" value="smtp.gmail.com"/>
    <constructor-arg index="1" value="myusername"/>
    <constructor-arg index="2" value="mypassword"/>
</bean>
```

Dotychczas poznałeś sposób użycia elementu `<constructor-arg>` w celu wstrzykiwania zależności komponentu i przekazywania wartości dla argumentów konstruktora typu `String`. Z rozdziału 3. dowiesz się, jak element `<constructor-arg>` można wykorzystać do ustawiania wartości typów podstawowych (np. `int`, `long`), kolekcji (np. `java.util.List`, `java.util.Map`), a także właściwości niestandardowego typu (np. `Address`).

Spójrz teraz, jak wstrzykiwanie zależności oparte na konstruktorze można wykorzystać wraz ze wstrzykiwaniem zależności opartym na metodach typu `setter`.

Połączenie mechanizmów wstrzykiwania zależności opartych na konstruktorze i metodach typu `setter`

Jeżeli klasa komponentu wymaga mechanizmu wstrzykiwania zależności zarówno opartego na konstruktorze, jak i na metodach typu `setter`, wówczas w celu wstrzyknięcia zależności możesz skorzystać z połączenia elementów `<constructor-arg>` i `<property>`.

W listingu 2.19 została przedstawiona klasa komponentu, której zależności są wstrzykiwane jako argumenty konstruktora i metody typu `setter`.

Listing 2.19. Klasa `PersonalBankingService`

```
public class PersonalBankingService {
    private JmsMessageSender jmsMessageSender;
    private EmailMessageSender emailMessageSender;
    private WebServiceInvoker webServiceInvoker;
    ....
    public PersonalBankingService(JmsMessageSender jmsMessageSender,
        EmailMessageSender emailMessageSender) {
        this.jmsMessageSender = jmsMessageSender;
    }
}
```

```

        this.emailMessageSender = emailMessageSender;
    }

    public void setWebServiceInvoker(WebServiceInvoker webServiceInvoker) {
        this.webServiceInvoker = webServiceInvoker;
    }

    .....
}

```

W klasie `PersonalBankingService` zależności `jmsMessageSender` i `emailMessageSender` są wstrzykiwane za pomocą argumentów konstruktora, natomiast zależność `webServiceInvoker` jest wstrzykiwana za pomocą metody typu setter o nazwie `setWebServiceInvoker()`. Definicja komponentu w listingu 2.20 pokazuje, że elementy `<constructor-arg>` i `<property>` są używane do wstrzykiwania zależności klasy `PersonalBankingService`.

Listing 2.20. Połączenie w kodzie mechanizmów wstrzykiwania zależności za pomocą konstruktora i metod typu setter

```

<bean id="dataSource" class="PersonalBankingService">
    <constructor-arg index="0" ref="jmsMessageSender" />
    <constructor-arg index="1" ref="emailMessageSender" />
    <property name="webServiceInvoker" ref="webServiceInvoker" />
</bean>

```

Skoro dowiedziałeś się, jak nakazać kontenerowi Springa utworzenie komponentu i wstrzyknięcie jego zależności, możemy przejść do poznania różnych *zasięgów*, które są definiowane dla komponentów.

2.5. Zasięg komponentu

Być może chcesz określić zasięg komponentu w celu zachowania kontroli nad sposobem tworzenia egzemplarza komponentu: współdzielony (zasięg singleton) lub zupełnie nowy egzemplarz, gdy z kontenera Springa pojawia się żądanie danego komponentu (zasięg prototype). Do zdefiniowania zasięgu komponentu używamy atrybutu `scope` elementu `<bean>`. Jeżeli atrybut `scope` nie zostanie podany, wówczas oznacza to utworzenie komponentu o zasięgu singleton.



W aplikacjach sieciowych Spring pozwala na użycie jeszcze innych zasięgów: `request`, `session` i `globalSession`. Zasięgi te określają *czas życia* egzemplarza komponentu. Na przykład czas życia komponentu o zasięgu `request` pozostaje ograniczony do pojedynczego żądania HTTP. Ponieważ w tym rozdziale *nie* będziemy omawiać Spring Web MVC i Spring Portlet MVC, analizę zasięgów ograniczymy do singleton i prototype. Omówienie zasięgów `request`, `session` i `globalSession` znajdziesz w rozdziale 10.

IMPORT **Rozdział 102/ch02-bankapp-scopes.** (Ten projekt pokazuje użycie zasięgu singleton i prototype dla komponentu. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `MyBankApp` projektu. Ponadto w projekcie znajdują się dwa testy JUnit — `PrototypeTest` i `SingletonTest` — które możesz wykonać).

Singleton

Zasięg singleton jest zasięgiem *domyślnym* dla wszystkich komponentów zdefiniowanych w pliku XML kontekstu aplikacji. Egzemplarze komponentów o zasięgu singleton są tworzone podczas tworzenia kontenera Springa, a także w chwili jego usuwania. Kontener Springa tworzy *pojedynczy* egzemplarz komponentu o zasięgu singleton, który następnie jest współdzielony przez *wszystkie* komponenty od niego zależne.

W listingu 2.21 przedstawiamy plik *applicationContext.xml* projektu *ch02-bankapp-scopes*, w którym wszystkie komponenty mają zasięg singleton.

Listing 2.21. Plik *applicationContext.xml* zawierający komponenty o zasięgu singleton

```
Projekt: ch02-bankapp-scopes
Polozenie pliku: src/main/resources/META-INF/spring
<beans ..... >
  <bean id="controller"
    class="sample.spring.chapter02.bankapp.controller.FixedDepositControllerImpl">
    <property name="fixedDepositService" ref="service" />
  </bean>

  <bean id="service"
    class="sample.spring.chapter02.bankapp.service.FixedDepositServiceImpl">
    <property name="fixedDepositDao" ref="dao" />
  </bean>

  <bean id="dao" class="sample.spring.chapter02.bankapp.dao.FixedDepositDaoImpl" />
  .....
</beans>
```

W przedstawionym powyżej pliku *applicationContext.xml* komponenty controller, service i dao mają zasięg singleton, ponieważ elementy `<bean>` nie zawierają atrybutu `scope`. Oznacza to, że kontener Springa tworzy tylko po jednym egzemplarzu klas `FixedDepositControllerImpl`, `FixedDepositServiceImpl` i `FixedDepositDaoImpl`. Skoro wymienione komponenty mają zasięg singleton, kontener Springa zwraca ten sam egzemplarz komponentu za każdym razem, gdy dowolny z tych komponentów będzie zwracany przez metodę `getBean()` klasy `ApplicationContext`.



Jeżeli atrybut `scope` nie zostanie zdefiniowany lub jego wartością będzie `singleton`, oznacza to, że komponent ma zasięg singleton.

W listingu 2.22 przedstawiamy metodę `testInstances()` klasy `SingletonTest` (klasa testu JUnit) projektu *ch02-bankapp-scopes*. Metoda `testInstances()` sprawdza, czy kilka wywołań metody `getBean()` klasy `ApplicationContext` zwraca ten sam czy zupełnie inny egzemplarz komponentu controller.

W powyższym listingu oferowana przez JUnit adnotacja `@BeforeClass` wskazuje, że metoda `init()` będzie wywołana przed jakąkolwiek inną metodą testową (czyli oznaczoną oferowaną przez JUnit adnotacją `@Test`) klasy. Oznacza to, że metoda wraz z adnotacją `@BeforeClass` jest wywoływana tylko *jednokrotnie*, natomiast metody wraz z adnotacją `@Test` są wykonywane tylko *po* wykonaniu metody oznaczonej adnotacją `@BeforeClass`. Zwróć uwagę na fakt, że metoda `init()` jest metodą

Listing 2.22. *Klasa testowa SingletonTest z JUnit**Projekt: ch02-bankapp-scopes**Położenie pliku: src/test/java/sample/spring/chapter02/bankapp*

```
package sample.spring.chapter02.bankapp;

import static org.junit.Assert.assertSame;
import org.junit.BeforeClass;
import org.junit.Test;

import sample.spring.chapter02.bankapp.controller.FixedDepositController;

public class SingletonTest {
    private static ApplicationContext context;

    @BeforeClass
    public static void init() {
        context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");
    }

    @Test
    public void testInstances() {
        FixedDepositController controller1 = (FixedDepositController)
            context.getBean("controller");
        FixedDepositController controller2 = (FixedDepositController)
            context.getBean("controller");
        assertSame("Różne egzemplarze FixedDepositController", controller1,
            controller2);
    }
    .....
}
```

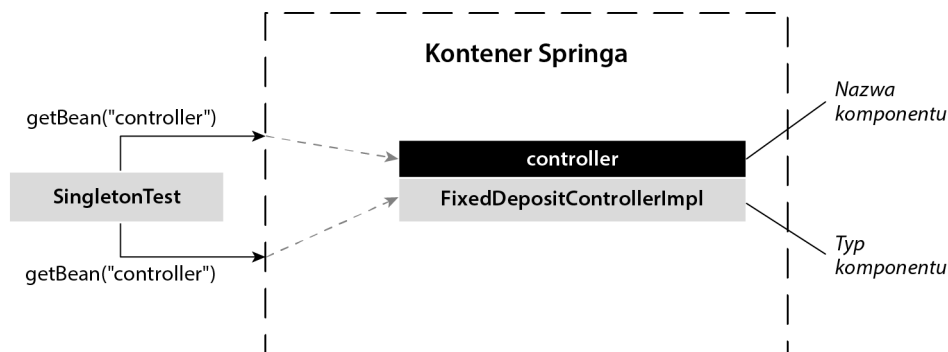
statyczną. Metoda `init()` tworzy egzemplarz obiektu `ApplicationContext`, przekazując metadane konfiguracyjne (patrz: listing 2.21) konstruktorowi `ClassPathXmlApplicationContext`. Metoda `testInstances()` pobiera dwa egzemplarze komponentu `controller` i za pomocą oferowanej przez JUnit asercji `assertSame` sprawdza, czy oba egzemplarze są dokładnie takie same. Ponieważ komponent `controller` ma zasięg `singleton`, egzemplarze `controller1` i `controller2` oznaczają ten sam obiekt. Dlatego też wykonanie metody `testInstances()` klasy `SingletonTest` kończy się bez wystąpienia żadnych błędów asercji.

Na rysunku 2.4 widać, że kontener Springa zwraca ten sam egzemplarz komponentu `controller` pomimo wielokrotnego wywołania metody `getBean()` klasy `ApplicationContext`.

Z rysunku 2.4 wynika, że wiele wywołań otrzymania komponentu `controller` kończy się zwrotem dokładnie *tego samego* egzemplarza komponentu `controller`.

Egzemplarz komponentu o zasięgu `singleton` jest współdzielony przez wszystkie komponenty, które są od niego zależne. W listingu 2.23 przedstawiamy metodę `testReference()` klasy JUnit `SingletonTest` sprawdzającą, czy egzemplarz `FixedDepositDao`, do którego odwołujemy się w egzemplarzu `FixedDepositController`, jest tym samym co pobrany bezpośrednio za pomocą wywołania metody `getBean()` klasy `ApplicationContext`.

W listingu 2.23 metoda `testReference()` najpierw pobiera egzemplarz `FixedDepositDao` (patrz: zmienna `fixedDepositDao1` w omawianym listingu), do którego odwołujemy się w komponentcie



Rysunek 2.4. W przypadku wielu żądań komponentu o zasięgu singleton kontener Springa zwraca wciąż ten sam egzemplarz komponentu

➔ Na rysunku 2.4 egzemplarz komponentu `controller` jest przedstawiony za pomocą dwuczęściowego prostokąta. Część górna pokazuje *nazwę* komponentu (to znaczy wartość atrybutu `id` elementu `<bean>`), natomiast część dolna pokazuje *typ* komponentu (to znaczy wartość atrybutu `class` elementu `<bean>`). W pozostałej części książki będziemy korzystali z tej konwencji w celu pokazania egzemplarza komponentu w kontenerze Springa.

Listing 2.23. Metoda `testReference` klasy testowej JUnit `SingletonTest`

Projekt: `ch02-bankapp-scopes`

Położenie pliku: `src/test/java/sample/spring/chapter02/bankapp`

```
package sample.spring.chapter02.bankapp;

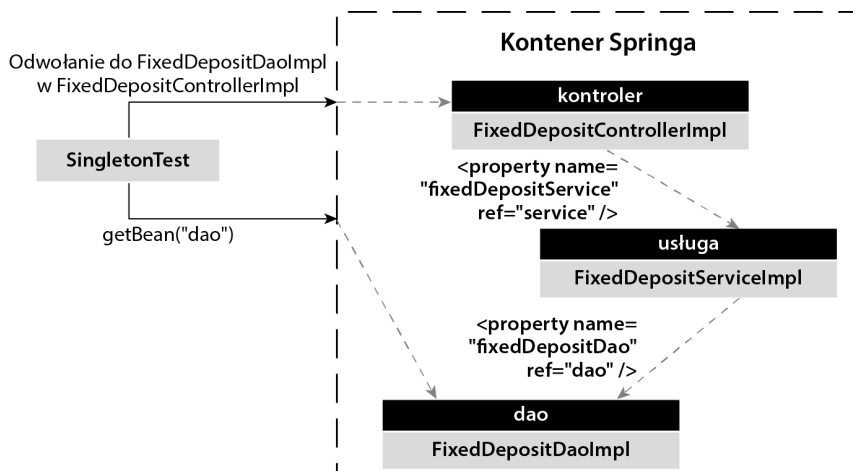
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class SingletonTest {
    private static ApplicationContext context;
    ....
    @Test
    public void testReference() {
        FixedDepositController controller = (FixedDepositController)
            context.getBean("controller");

        FixedDepositDao fixedDepositDao1 =
            controller.getFixedDepositService().getFixedDepositDao();
        FixedDepositDao fixedDepositDao2 = (FixedDepositDao) context.getBean("dao");
        assertEquals("Różne egzemplarze FixedDepositDao", fixedDepositDao1,
            fixedDepositDao2);
    }
}
```

`FixedDepositController`. Następnie za pomocą wywołania metody `getBean()` klasy `ApplicationContext` bezpośrednio pobierany jest inny egzemplarz komponentu `FixedDepositDao` (patrz: zmienna `fixedDepositDao2` w omawianym listingu). Po wykonaniu metody testowej `testReference()` zobaczysz, że test kończy się powodzeniem, ponieważ egzemplarze `fixedDepositDao1` i `fixedDepositDao2` oznaczają ten sam obiekt.

Na rysunku 2.5 zostało pokazane, że egzemplarz `FixedDepositDao`, do którego odwołujemy się w `FixedDepositController` to dokładnie ten sam obiekt co zwrócony przez wywołanie metody `getBean("dao")` klasy `ApplicationContext`.



Rysunek 2.5. Egzemplarz komponentu o zasięgu singleton jest współdzielony przez wszystkie komponenty od niego zależne

Na rysunku 2.5 widać, że egzemplarz `FixedDepositDao`, do którego odwołujemy się w egzemplarzu komponentu `FixedDepositController`, oraz egzemplarz pobrany bezpośrednio za pomocą wywołania metody `getBean()` klasy `ApplicationContext` to dokładnie ten sam obiekt. Jeżeli istnieje wiele komponentów zależnych od komponentu o zasięgu singleton, to wszystkie komponenty zależne współdzielą ten sam egzemplarz o zasięgu singleton.

Przekonajmy się teraz, czy ten sam egzemplarz komponentu o zasięgu singleton jest współdzielony przez wiele egzemplarzy kontenera Springa.

Komponenty o zasięgu singleton i wiele egzemplarzy kontenera Springa

Egzemplarz komponentu o zasięgu singleton jest ograniczony do egzemplarza kontenera Springa. Oznacza to, że po utworzeniu dwóch egzemplarzy kontenera Springa na podstawie tych samych metadanych konfiguracyjnych każdy kontener Springa będzie miał własne egzemplarze komponentów o zasięgu singleton.

W listingu 2.24 została przedstawiona metoda `testSingletonScope()` klasy `SingletonTest`. Zadaniem tej metody jest sprawdzenie, czy egzemplarze komponentu `FixedDepositController` pobrane z dwóch różnych kontenerów Springa są tym samym obiektem.

Listing 2.24. Metoda `testSingletonScope` klasy testowej `SingletonTest` JUnit

Projekt: `ch02-bankapp-scopes`

Położenie pliku: `src/test/java/sample/spring/chapter02/bankapp`

```
package sample.spring.chapter02.bankapp;

import static org.junit.Assert.assertNotSame;
```

```

public class SingletonTest {
    private static ApplicationContext context;
    .....
    @BeforeClass
    public static void init() {
        context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");
    }

    @Test
    public void testSingletonScope() {
        ApplicationContext anotherContext = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");

        FixedDepositController fixedDepositController1 = (FixedDepositController)
            anotherContext.getBean("controller");

        FixedDepositController fixedDepositController2 = (FixedDepositController)
            context.getBean("controller");

        assertNotSame("Ten sam egzemplarz FixedDepositController",
            fixedDepositController1, fixedDepositController2);
    }
}

```

Metoda `init()` klasy `SingletonTest` (oznaczona adnotacją `@BeforeClass` z JUnit) tworzy egzemplarz `ApplicationContext` (identyfikowany przez zmienną `context`), zanim nastąpi wykonanie jakiegokolwiek metody oznaczonej adnotacją `@Test`. Metoda `testSingletonScope()` tworzy kolejny egzemplarz kontenera Springa (identyfikowany przez zmienną `anotherContext`) na podstawie tego samego pliku *applicationContext.xml*. Egzemplarz komponentu `FixedDepositController` jest pobierany z poziomu obu kontenerów Springa, a następnie sprawdzamy, czy mamy do czynienia z *różnymi* egzemplarzami. Po wykonaniu metody `testSingletonScope()` zobaczysz, że test kończy się pomyślnie, ponieważ egzemplarz komponentu `FixedDepositController` pobrany ze zmiennej `context` jest zupełnie odmiennym obiektem niż obiekt wskazywany przez `anotherContext`.

Na rysunku 2.6 widać zachowanie metody `testSingleScope()`.

Z rysunku 2.6 wynika, że każdy kontener Springa tworzy własne egzemplarze komponentu `controller`. Dlatego też zmienne `context` i `anotherContext` wskazują różne egzemplarze komponentu `controller` po wywołaniu metody `getBean("controller")`.

Działanie metody `testSingletonScope()` ujawniło, że poszczególne kontenery Springa tworzą własne egzemplarze komponentu o zasięgu `singleton`. Trzeba koniecznie zapamiętać, że kontener Springa tworzy egzemplarz komponentu o zasięgu `singleton` dla każdej definicji komponentu. W listingu 2.25 przedstawiamy wiele definicji komponentu dla klasy `FixedDepositDaoImpl`.

Listing 2.25. Plik *applicationContext.xml* zawiera wiele definicji komponentu dla tej samej klasy

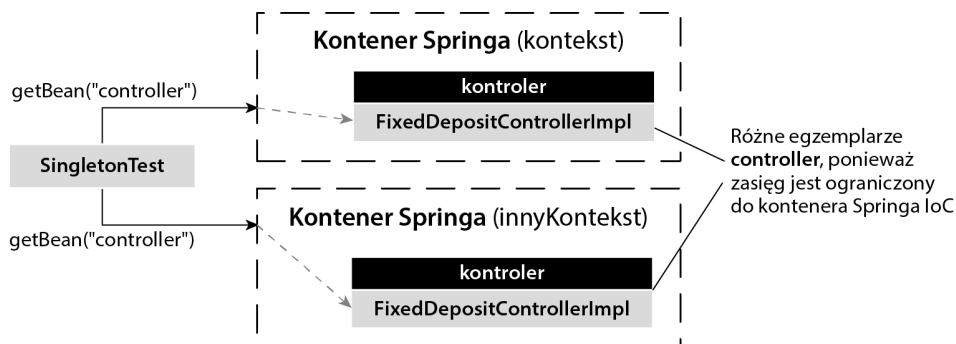
Projekt: *ch02-bankapp-scopes*

Położenie pliku: *src/main/resources/META-INF/spring*

```

<bean id="dao" class="sample.spring.chapter02.bankapp.dao.FixedDepositDaoImpl" />
<bean id="anotherDao"
    class="sample.spring.chapter02.bankapp.dao.FixedDepositDaoImpl" />

```



Rysunek 2.6. Każdy kontener Springa tworzy własny egzemplarz komponentów o zasięgu singleton

Definicje komponentów przedstawione w powyższym listingu są przeznaczone dla klasy `FixedDepositDaoImpl`. Ponieważ atrybut `scope` *nie* został określony, definicje w omawianym listingu przedstawiają komponenty o zasięgu singleton. Nawet w przypadku istnienia wielu definicji komponentów dla klasy kontener Springa tworzy egzemplarz komponentu dla każdej definicji. Dlatego ten konkretny kontener Springa utworzy oddzielne egzemplarze klasy `FixedDepositDaoImpl` odpowiadające definicjom komponentów `dao` i `anotherDao`. W listingu 2.26 przedstawiamy metodę `testSingletonScopePerBeanDef()` klasy `SingletonScope`. Metoda ta jest odpowiedzialna za sprawdzenie, czy egzemplarze `FixedDepositDaoImpl` odpowiadające komponentom `dao` i `anotherDao` to ten sam obiekt czy jednak różne obiekty.

Listing 2.26. Metoda `testSingletonScopePerBeanDef()` klasy testowej `SingletonTest JUnit`

Projekt: `ch02-bankapp-scopes`

Położenie pliku: `src/test/java/sample/spring/chapter02/bankapp`

```
package sample.spring.chapter02.bankapp;

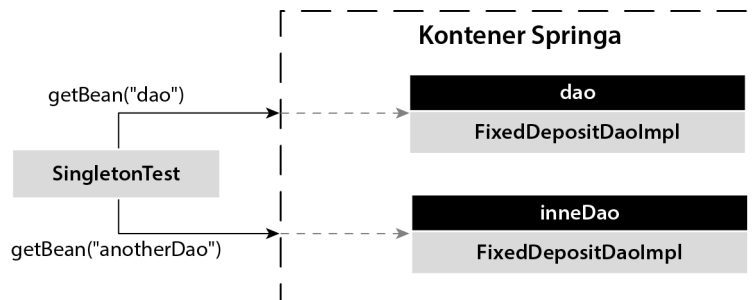
import static org.junit.Assert.assertNotSame;

public class SingletonTest {
    private static ApplicationContext context;
    .....
    @Test
    public void testSingletonScopePerBeanDef() {
        FixedDepositDao fixedDepositDao1 = (FixedDepositDao) context.getBean("dao");
        FixedDepositDao fixedDepositDao2 = (FixedDepositDao)
            context.getBean("anotherDao");
        assertNotSame("Ten sam egzemplarz FixedDepositDao", fixedDepositDao1,
            fixedDepositDao2);
    }
}
```

W powyższym listingu zmienne `fixedDepositDao1` i `fixedDepositDao2` przedstawiają egzemplarze klasy `FixedDepositDaoImpl`, które kontener Springa tworzy dla odpowiadających im definicji komponentów, odpowiednio `dao` i `anotherDao`. Jak się możesz przekonać, wykonanie metody testowej `testSingletonScopePerBeanDef()` kończy się powodzeniem, ponieważ egzemplarz `fixedDepositDao1` (odpowiadający definicji komponentu `dao`) i egzemplarz `fixedDepositDao2` (odpowiadający definicji komponentu `anotherDao`) są różne.

Na rysunku 2.7 pokazujemy podsumowanie potwierdzające, że komponent o zasięgu singleton jest tworzony dla poszczególnych definicji komponentów.

Rysunek 2.7.
Dla każdej definicji komponentu istnieje tylko jeden komponent o zasięgu singleton



Z powyższego rysunku wynika, że dla poszczególnych definicji komponentów w kontenerze Springa istnieje tylko jeden egzemplarz komponentu o zasięgu singleton.

Wspomnieliśmy wcześniej, że komponenty o zasięgu singleton są domyślnie *preinicjowane*. Oznacza to, że egzemplarz komponentu o zasięgu singleton jest tworzony wraz z utworzeniem kontenera Springa. Spójrzmy teraz, jak można z opóźnieniem *zainicjować* komponent o zasięgu singleton.

Opóźniona inicjacja komponentu o zasięgu singleton

Kontenerowi Springa można nakazać utworzenie komponentu o zasięgu singleton tylko wtedy, gdy jest żądany po raz *pierwszy*. W listingu 2.27 znajduje się definicja komponentu `lazyExample` pokazująca, jak nakazać kontenerowi Springa opóźnioną inicjację komponentu `lazyBean`.

Listing 2.27. Opóźniona inicjacja komponentu o zasięgu singleton

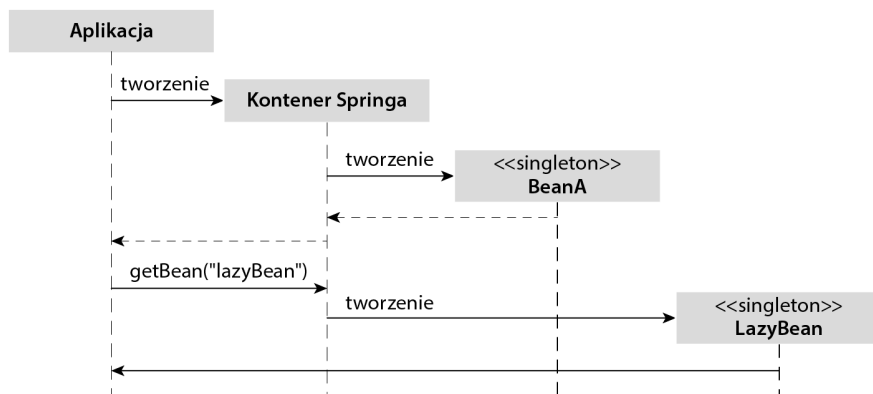
```
<bean id="lazyBean" class="example.LazyBean" lazy-init="true"/>
```

Atrybut `lazy-init` elementu `<bean>` określa, czy egzemplarz komponentu będzie tworzony z opóźnieniem. Jeżeli wartością wymienionego komponentu jest `true` (jak ma to miejsce w omawianej definicji), wówczas egzemplarz komponentu jest inicjowany przez kontener Springa po otrzymaniu *pierwszego* żądania danego komponentu.

Diagram na rysunku 2.8 pokazuje, jak atrybut `lazy-init` wpływa na tworzenie egzemplarza komponentu o zasięgu singleton.

W diagramie pokazanym na rysunku 2.8 `BeanA` przedstawia egzemplarz komponentu o zasięgu singleton, który *nie* jest inicjowany z opóźnieniem. Natomiast `LazyBean` przedstawia komponent o zasięgu singleton, który ma być zainicjowany z opóźnieniem. Podczas tworzenia egzemplarza kontenera Springa następuje utworzenie także komponentu `BeanA`, ponieważ *nie* ma on zdefiniowanej opóźnionej inicjacji. Z kolei `LazyBean` jest tworzony dopiero po pierwszym wywołaniu metody `getBean()` klasy `ApplicationContext` żądającym pobrania egzemplarza `LazyBean` z kontenera Springa.

Ponieważ komponent o zasięgu singleton może być inicjowany z opóźnieniem lub wraz z tworzeniem kontenera Springa, być może zastanawiasz się, kiedy powinieneś definiować komponent jako inicjowany z opóźnieniem, a kiedy jako inicjowany wraz z kontenerem Springa. W większości



Rysunek 2.8. Egzemplarz komponentu o zasięgu singleton inicjowany z opóźnieniem, czyli dopiero po jego pierwszym żądaniu przez aplikację



Istnieje możliwość użycia atrybutu `default-lazy-init` elementu `<beans>` w celu określenia domyślnej strategii inicjacji komponentów zdefiniowanych w pliku XML kontekstu aplikacji. Jeżeli atrybut `lazy-init` elementu `<bean>` zawiera inną wartość niż atrybut `default-lazy-init` elementu `<beans>`, wówczas dla komponentu zastosowana będzie wartość atrybutu `lazy-init`.

przypadków korzystne będzie inicjowanie komponentów wraz z kontenerem, ponieważ wszelkie problemy związane z konfiguracją zostaną wykryte w chwili tworzenia kontenera Springa. W listingu 2.28 został przedstawiony komponent `aBean` o zasięgu singleton, który ma określoną inicjację z opóźnieniem i jest zależny od komponentu `bBean`.

Listing 2.28. Inicjowany z opóźnieniem komponent o zasięgu singleton

```

public class ABean {
    private BBean bBean;

    public void setBBean(BBean bBean) {
        this.bBean = bBean;
    }
    ....
}

<bean id="aBean" class="ABean" lazy-init="true">
    <property name="bBean" value="bBean" />
</bean>

<bean id="bBean" class="BBean" />
  
```

W powyższym listingu właściwość `bBean` klasy `ABean` odwołuje się do komponentu `BBean`. Zwróć uwagę na fakt, że do ustawienia właściwości `bBean` w klasie `ABean` zamiast atrybutu `ref` został użyty atrybut `value` elementu `<property>`. Jeżeli utworzysz egzemplarz `ApplicationContext`, przekazując plik XML zawierający przedstawioną wcześniej definicję komponentu, to nie zostaną zgłoszone żadne błędy. Jednak próba pobrania `aBean` za pomocą wywołania metody `getBean()` klasy `ApplicationContext` zakończy się wygenerowaniem następującego komunikatu błędu:

Źródło błędu: java.lang.IllegalStateException: Nie można skonwertować wartości typu [java.lang.String] na typ wymagany [BBean] dla właściwości 'bBean: nie znaleziono żadnych dopasowanych edytorów lub strategii konwersji.

Powyższy komunikat błędu zostanie wyświetlony, ponieważ kontener Springa nie mógł skonwertować wartości typu String właściwości bBean klasy ABean na typ BBean. To wskazuje prosty problem związany z konfiguracją, gdy zamiast atrybutu ref elementu <bean> został użyty atrybut value. Jeżeli komponent aBean zostałby skonfigurowany do utworzenia wraz z kontenerem (zamiast do inicjacji opóźnionej), wówczas powyższy błąd mógłby zostać wychwycony w trakcie tworzenia egzemplarza ApplicationContext, a nie w chwili próby otrzymania egzemplarza aBean z poziomu ApplicationContext.

Przechodzimy teraz do komponentów o *zasięgu prototypu* w Springu.

Prototyp

Komponent o zasięgu prototypu jest odmienny od komponentu o zasięgu singleton pod tym względem, że kontener Springa zawsze zwraca *nowy* egzemplarz komponentu o zasięgu prototypu. Kolejna cecha odróżniająca komponent o zasięgu prototypu to *zawsze* stosowana inicjacja z opóźnieniem.

W listingu 2.29 komponent FixedDepositDetails w pliku *applicationContext.xml* pochodzącym z projektu *ch02-bankapp-scopes* przedstawia komponent o zasięgu prototypu.

Listing 2.29. Plik *applicationContext.xml* definiujący komponent o zasięgu prototypu

Projekt: *ch02-bankapp-scopes*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean id="FixedDepositDetails"
class="sample.spring.chapter02.bankapp.domain.FixedDepositDetails"
scope="prototype" />
```

W powyższym listingu wartość atrybutu scope elementu <bean> to prototype. Oznacza to, że komponent FixedDepositDetails ma zasięg prototypu.

W listingu 2.30 przedstawiamy metodę testInstances() klasy testowej PrototypeTest pochodzącej z frameworku JUnit. Działanie tej klasy potwierdza, że dwa egzemplarze FixedDepositDetails to zupełnie różne obiekty.

Listing 2.30. Metoda testInstances() klasy testowej PrototypeTest frameworku JUnit

Projekt: *ch02-bankapp-scopes*

Położenie pliku: *src/test/java/sample/spring/chapter02/bankapp*

```
package sample.spring.chapter02.bankapp;

import static org.junit.Assert.assertNotSame;

public class PrototypeTest {
    private static ApplicationContext context;
    ....
    @Test
    public void testInstances() {
```

```
FixedDepositDetails fixedDepositDetails1 =
    (FixedDepositDetails) context.getBean("fixedDepositDetails");
FixedDepositDetails fixedDepositDetails2 =
    (FixedDepositDetails) context.getBean("fixedDepositDetails");

    assertNotNull("Ten sam egzemplarz FixedDepositDetails",
        fixedDepositDetails1, fixedDepositDetails2);
}
```

Jeżeli uruchomisz metodę `testInstances()`, zostanie wykonana bez błędów, ponieważ dwa egzemplarze `FixedDepositDetails` (`fixedDepositDetails1` i `fixedDepositDetails2`) pobrane przez `ApplicationContext` to zupełnie różne obiekty.

Zobaczmy teraz, jak wybrać odpowiedni zasięg (singleton lub prototypu) dla komponentu.

Wybór właściwego zasięgu dla komponentu

Jeżeli komponent nie musi obsługiwać żadnego stanu (czyli jest bezstanowy), powinien zostać zdefiniowany jako komponent o zasięgu singleton. Natomiast jeśli komponent musi obsługiwać stan, powinien być zdefiniowany jako komponent o zasięgu prototypu. W naszej aplikacji `MyBank` komponenty `FixedDepositServiceImpl`, `FixedDepositDaoImpl` i `FixedDepositControllerImpl` są z natury bezstanowe i dlatego zostały zdefiniowane jako komponenty o zasięgu singleton. Z kolei komponent `FixedDepositDetails` (*obiekt domeny*) aplikacji `MyBank` obsługuje stan, stąd jego zdefiniowanie jako komponentu o zakresie prototypu.



Jeżeli w aplikacji używasz frameworku ORM (np. Hibernate lub iBATIS), obiekty domeny są tworzone przez framework ORM lub tworzysz je programowo w kodzie aplikacji za pomocą operatora `new`. Wynika to z faktu, że obiekty domeny nie są definiowane w pliku XML kontekstu aplikacji, jeżeli dla zapewnienia trwałego magazynu danych aplikacja korzysta z frameworku ORM.

2.6. Podsumowanie

W tym rozdziale zostały przedstawione pewne podstawy frameworku Spring. Poznałeś podejście projektowe o nazwie *podejście oparte na programowaniu interfejsu*, różne podejścia w zakresie tworzenia egzemplarzy komponentu, wstrzykiwanie zależności za pomocą konstruktora, a także zasięg komponentu. W następnym rozdziale przechodzimy do ustawiania różnych typów (np. `int`, `long`, `Map`, `Set`) właściwości komponentu i argumentów konstruktora.

Rozdział 3. Konfiguracja komponentów

3.1. Wprowadzenie

W poprzednich rozdziałach przedstawiliśmy pewne podstawowe koncepcje frameworku Spring. Zobaczyłeś, jak w pliku XML kontekstu aplikacji określane są komponenty Springa i ich zależności. Zajęliśmy się również komponentami o zasięgu singleton i prototypu, a także przeanalizowaliśmy implikacje wynikające z przypisania komponentom wymienionych zasięgów.

W tym rozdziale omówimy następujące zagadnienia:

- dziedziczenie definicji komponentu;
- obsługa argumentów konstruktora klasy komponentu;
- sposób konfiguracji właściwości i argumenty konstruktora, które są zaliczane do typu podstawowego (np. `int`, `float`), kolekcji (np. `java.util.List`, `java.util.Map`) lub typu niestandardowego (np. `Address`) itd.;
- sposoby zmniejszania wielkość pliku XML kontekstu aplikacji przez zastosowanie tak zwanych przestrzeni nazw `p` oraz `c` w celu określenia odpowiednio właściwości i argumentów konstruktora komponentu;
- użycie interfejsu `FactoryBean` pozwalającego na opracowywanie własnych klas fabryk przeznaczonych do tworzenia egzemplarzy komponentu.

3.2. Dziedziczenie definicji komponentu

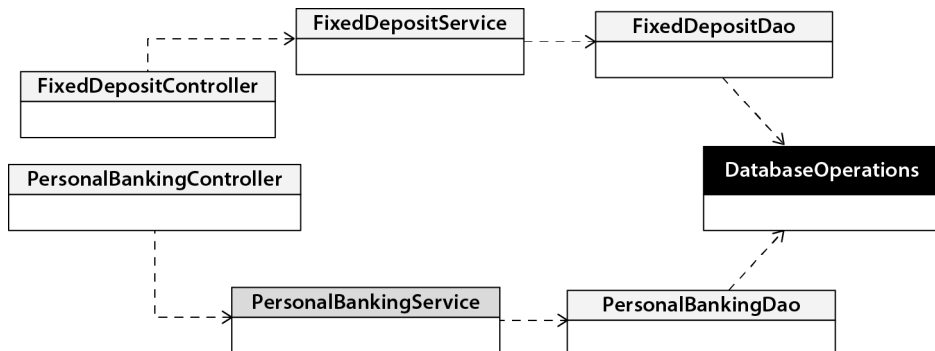
W rozdziałach 1. i 2. zobaczyłeś, że definicje komponentu w pliku XML kontekstu aplikacji określają w pełni kwalifikowaną nazwę klasy komponentu i jej zależności. W pewnych sytuacjach można zmniejszyć wielkość definicji komponentu przez *dziedziczenie* informacji konfiguracyjnych po innej definicji komponentu. Spójrzmy na przykład takiego rozwiązania w aplikacji `MyBank`.

IMPORT **Rozdział103/ch03-bankapp-inheritance.** (Celem tego projektu jest pokazanie, jak aplikacja korzysta z *dziedziczenia definicji komponentu*. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Przykład dziedziczenia definicji komponentu w aplikacji `MyBank`

W poprzednim rozdziale widzieliśmy, jak aplikacja `MyBank` uzyskuje dostęp do bazy danych za pomocą DAO. Przyjmujemy założenie, że aplikacja `MyBank` definiuje klasę `DatabaseOperations`, która ułatwia pracę z bazą danych. W takim przypadku wszystkie obiekty DAO w aplikacji `MyBank` są zależne od klasy `DatabaseOperations`, przynajmniej w kwestii przeprowadzania operacji w bazie danych. Tę sytuację przedstawia rysunek 3.1.

Na rysunku 3.1 widać, że klasy `FixedDepositDao` i `PersonalBankingDao` są zależne od klasy `DatabaseOperations`. W listingu 3.1 został przedstawiony plik XML kontekstu aplikacji zawierający definicje wymienionych klas.



Rysunek 3.1. Klasy DAO w aplikacji MyBank używają klasy DatabaseOperations do współpracy z bazą danych

Listing 3.1. Komponenty DAO zależne od komponentu DatabaseOperations

```

<bean id="databaseOperations"
      class="sample.spring.chapter01.bankapp.utils.DatabaseOperations" />

<bean id="personalBankingDao"
      class="sample.spring.chapter01.bankapp.dao.PersonalBankingDaoImpl">
  <property name="databaseOperations" ref="databaseOperations" />
</bean>

<bean id="FixedDepositDao"
      class="sample.spring.chapter01.bankapp.dao.FixedDepositDaoImpl">
  <property name="databaseOperations" ref="databaseOperations" />
</bean>
  
```

Definicje komponentów personalBankingDao i FixedDepositDao używają elementu <property> w celu przeprowadzenia wstrzyknięcia zależności w postaci egzemplarza DatabaseOperations. Ponieważ w obu definicjach komponentu nazwą właściwości odwołującej się do egzemplarza DatabaseOperations jest databaseOperations, oznacza to, że klasy PersonalBankingDaoImpl i FixedDepositDaoImpl definiują metodę setDatabaseOperations() pozwalającą kontenerowi Springa na wstrzyknięcie egzemplarza DatabaseOperations.

Jeżeli wiele komponentów w aplikacji współdzieli ten sam zbiór informacji konfiguracyjnych (właściwości, argumenty konstruktora itd.), wówczas można utworzyć definicję komponentu działającą w charakterze nadrzędnej dla innych definicji. W przypadku definicji komponentów personalBankingDao i fixedDepositDao łączącą je opcją konfiguracyjną jest właściwość databaseOperations. W listingu 3.2 pokazujemy, jak definicje komponentów personalBankingDao i fixedDepositDao mogą dziedziczyć definicję komponentu.

Listing 3.2. Plik applicationContext.xml, czyli plik w formacie XML przedstawiający kontekst aplikacji MyBank

Projekt: ch03-bankapp-inheritance

Położenie pliku: src/main/resources/META-INF/spring

```

<bean id="databaseOperations"
      class="sample.spring.chapter03.bankapp.utils.DatabaseOperations" />

<bean id="daoTemplate" abstract="true">
  <property name="databaseOperations" ref="databaseOperations" />
  <property name="personalBankingDao" ref="personalBankingDao" />
  <property name="fixedDepositDao" ref="fixedDepositDao" />
</bean>
  
```

```
<property name="databaseOperations" ref="databaseOperations" />
</bean>

<bean id="FixedDepositDao" parent="daoTemplate"
      class="sample.spring.chapter03.bankapp.dao.FixedDepositDaoImpl" />

<bean id="personalBankingDao" parent="daoTemplate"
      class="sample.spring.chapter03.bankapp.dao.PersonalBankingDaoImpl" />
```

W powyższym listingu definicja komponentu `daoTemplate` zawiera opcje konfiguracyjne współdzielone przez definicje komponentów `fixedDepositDao` i `personalBankingDao`. Ponieważ definicje `fixedDepositDao` i `personalBankingDao` wymagają zależności w postaci `databaseOperations` (patrz: listing 3.1), w definicji `daoTemplate` określono zależność od `databaseOperations`, używając w tym celu elementu `<property>`. Atrybut `parent` elementu `<bean>` wskazuje nazwę definicji komponentu, po której jest dziedziczona konfiguracja. W omawianym przykładzie wartością atrybutu `parent` jest `daoTemplate`, więc definicje komponentów `fixedDepositDao` i `personalBankingDao` dziedziczą właściwość `databaseOperations` po `daoTemplate`. Działanie kodów przedstawionych w listingach 3.1 i 3.2 jest takie samo, przy czym kod w listingu 3.2 korzysta z dziedziczenia definicji komponentu.

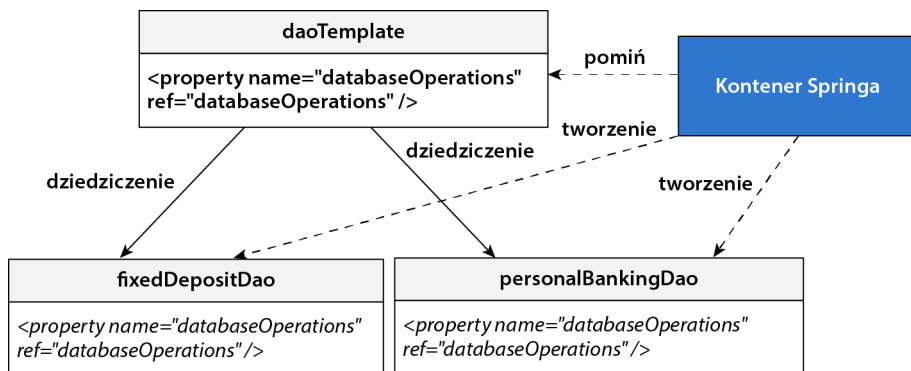
Jeżeli wartością atrybutu `abstract` elementu `<bean>` będzie `true`, oznacza to, że definicja komponentu jest *abstrakcyjna*. Trzeba w tym miejscu wspomnieć, że kontener Springa *nie* próbuje utworzyć komponentu odpowiadającego definicji *abstrakcyjnej*. Ponadto nie można zdefiniować komponentu zależnego od komponentu *abstrakcyjnego*. To oznacza brak możliwości użycia elementu `<property>` lub `<constructor-arg>` w celu odwołania się do komponentu *abstrakcyjnego*.

W listingu 3.2 definicja komponentu `daoTemplate` jest *abstrakcyjna*. Prawdopodobnie zauważyłeś, że definicja komponentu `daoTemplate` nie zawiera atrybutu `class`. Jeżeli definicja komponentu nadrzędnego nie zawiera atrybutu `class`, wówczas definicje komponentów potomnych (np. `fixedDepositDao` i `personalBankingDao`) określają atrybut `class`. Warto w tym miejscu wspomnieć, że jeżeli nie określisz atrybutu `class`, to musisz zdefiniować definicję komponentu nadrzędnego jako *abstrakcyjną*, aby kontener Springa nie próbował utworzyć egzemplarza odpowiadającego tej definicji.

W celu potwierdzenia, że definicje komponentów `fixedDepositDao` i `personalBankingDao` dziedziczą właściwość `databaseOperations` po definicji `daoTemplate`, wykonujemy metodę `main()` klasy `BankApp` projektu *ch03-bankapp-inheritance*. Metoda `main()` klasy `BankApp` wywołuje metody w komponentach `fixedDepositDao` i `personalBankingDao`, które z kolei wywołują metody w egzemplarzu `DatabaseOperations`. Jeżeli egzemplarz `DatabaseOperations` *nie* zostanie wstrzyknięty do komponentów `fixedDepositDao` i `personalBankingDao`, to nastąpi zgłoszenie wyjątku `java.lang.NullPointerException`.

Diagram na rysunku 3.2 przedstawia podsumowanie sposobu działania dziedziczenia definicji komponentu w przypadku definicji komponentów `FixedDepositDao` i `personalBankingDao`.

Na rysunku 3.2 zostało pokazane, że definicje komponentów `fixedDepositDao` i `personalBankingDao` dziedziczą właściwość `databaseOperations` (przedstawioną kursywą w ramkach zatytułowanych `fixedDepositDao` i `personalBankingDao`) po definicji komponentu `daoTemplate`. Z rysunku wynika



Rysunek 3.2. Dziedziczenie definicji komponentów w aplikacji MyBank

także, że kontener Springa nie próbuje utworzyć egzemplarza komponentu odpowiadającego definicji `daoTemplate`, ponieważ jest on oznaczony jako *abstrakcyjny*.

Zobaczmy teraz, jakie informacje konfiguracyjne są dziedziczone po definicji komponentu nadrzędnego.

Co jest dziedziczone?

Definicja komponentu potomnego dziedziczy po komponencie nadrzędnym następujące informacje konfiguracyjne:

- właściwości — określone za pomocą elementów `<property>`;
- argumenty konstruktora — określone za pomocą elementów `<constructor-arg>`;
- przypadki nadpisania metod (ten temat zostanie omówiony w podrozdziale 4.5);
- metody inicjalizacji i destrukcji (omówione w rozdziale 5.);
- metody fabryki — określone za pomocą atrybutu `factory-method` elementu `<bean>` (z podrozdziału 2.3 dowiedziałeś się, jak metody fabryki zarówno *statyczne*, jak i *egzemplarza* są używane podczas tworzenia komponentów).

IMPORT **Rozdział 103/ch03-bankapp-inheritance-example.** (Celem tego projektu jest pokazanie, jak aplikacja korzysta z dziedziczenia definicji komponentu. W tym projekcie mamy kilka przypadków zastosowania dziedziczenia komponentów. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Zobaczmy teraz kilka przykładów dziedziczenia definicji komponentu.

Przykład dziedziczenia definicji komponentu — definicja komponentu nadrzędnego nie jest abstrakcyjna

W listingu 3.3 przedstawiamy przykład dziedziczenia definicji komponentu, w którym definicja komponentu nadrzędnego *nie* jest abstrakcyjna, a definicje potomne zawierają określone dodatkowe zależności.

Listing 3.3. Przykład dziedziczenia definicji w pliku `applicationContext.xml` — definicja nadrzędna nie jest abstrakcyjnaProjekt: `ch03-bankapp-inheritance-examples`Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="serviceTemplate"
      class="sample.spring.chapter03.bankapp.base.ServiceTemplate">
  <property name="jmsMessageSender" ref="jmsMessageSender" />
  <property name="emailMessageSender" ref="emailMessageSender" />
  <property name="webServiceInvoker" ref="webServiceInvoker" />
</bean>

<bean id="fixedDepositService" class=".....FixedDepositServiceImpl"
      parent="serviceTemplate">
  <property name="fixedDepositDao" ref="fixedDepositDao" />
</bean>

<bean id="personalBankingService" class=".....PersonalBankingServiceImpl"
      parent="serviceTemplate">
  <property name="personalBankingDao" ref="personalBankingDao" />
</bean>

<bean id="userRequestController" class=".....UserRequestControllerImpl">
  <property name="serviceTemplate" ref="serviceTemplate" />
</bean>
```

Zanim zagłębimy się w szczegóły przedstawionej konfiguracji, warto najpierw zwrócić uwagę na pewną kwestię. Usługa w aplikacji MyBank może wysyłać komunikaty JMS do pośrednika zajmującego się obsługą komunikatów, wysyłać wiadomości e-mail do serwera poczty lub wywoływać zewnętrzną usługę sieciową. W omawianym listingu komponenty `jmsMessageSender`, `emailMessageSender` ➤ `Sender` i `webServiceInvoker` ułatwiają wymienione zadania, dostarczając warstwę abstrakcji. Komponent `serviceTemplate` zapewnia dostęp do komponentów `jmsMessageSender`, `emailMessageSender` i `webServiceInvoker`. Dlatego komponent `serviceTemplate` pozostaje zależny od komponentów `jmsMessageSender`, `emailMessageSender` i `webServiceInvoker`.

W listingu 3.3 zostało pokazane, że definicja komponentu `serviceTemplate` jest definicją nadrzędną dla `fixedDepositService` i `personalBankingService`. Zwróć uwagę na fakt, że definicja komponentu `serviceTemplate` nie jest *abstrakcyjna*, ponieważ atrybut `class` określa `ServiceTemplate` jako klasę. W poprzednim przykładzie dziedziczenia komponentu (patrz: listing 3.2) definicje komponentów potomnych nie zawierały żadnych właściwości. Zauważ, że w omawianym listingu definicje komponentów potomnych `fixedDepositService` i `personalBankingService` zawierają określone właściwości odpowiednio `fixedDepositDao` i `personalBankingDao`.

Ponieważ właściwości definicji komponentów nadrzędnych są dziedziczone przez definicje komponentów potomnych, klasy `FixedDepositServiceImpl` i `PersonalBankingServiceImpl` ➤ `BankingServiceImpl` muszą zawierać metody typu setter dla właściwości `jmsMessageSender`, `emailMessageSender` i `webServiceInvoker`. Masz możliwość zdefiniowania metod typu setter w klasach `FixedDepositServiceImpl` i `PersonalBankingServiceImpl` lub zdefiniować klasy `FixedDepositServiceImpl` i `PersonalBankingServiceImpl` jako podklasy klasy `ServiceTemplate`. W projekcie `ch03-bankapp-inheritance-examples` klasy `FixedDepositServiceImpl` i `PersonalBankingServiceImpl` są podklasami klasy `ServiceTemplate`.

W listingu 3.4 została przedstawiona klasa `PersonalBankingServiceImpl`.

Listing 3.4. Klasa `PersonalBankingServiceImpl`

Projekt: `ch03-bankapp-inheritance-examples`

Położenie pliku: `src/main/java/sample/spring/chapter03/bankapp/service`

```
package sample.spring.chapter03.bankapp.service;

public class PersonalBankingServiceImpl extends ServiceTemplate implements
    PersonalBankingService {

    private PersonalBankingDao personalBankingDao;

    public void setPersonalBankingDao(PersonalBankingDao personalBankingDao) {
        this.personalBankingDao = personalBankingDao;
    }

    @Override
    public BankStatement getMiniStatement() {
        return personalBankingDao.getMiniStatement();
    }
}
```

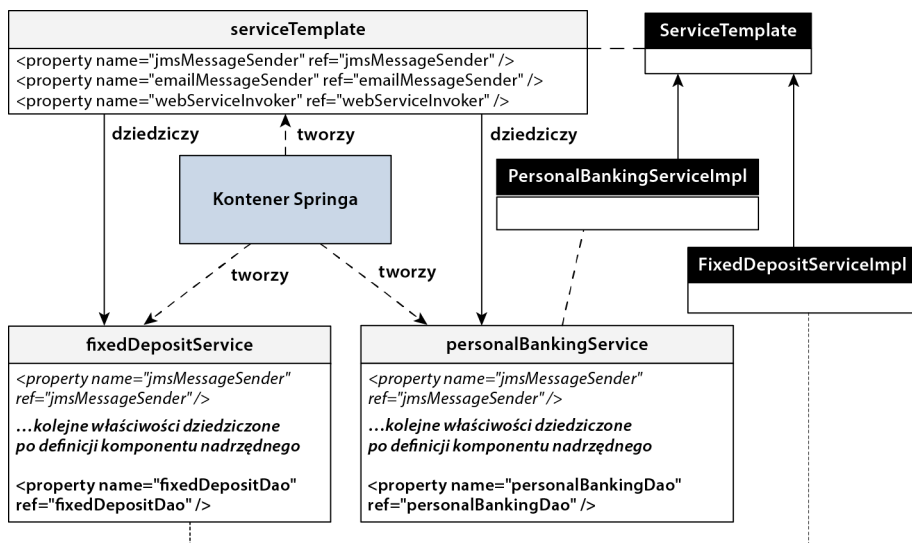
W listingu 3.3 widzieliśmy, że definicja komponentu `personalBankingService` wskazywała `personalBankingDao` jako zależność. Natomiast w listingu 3.4 metoda typu setter o nazwie `setPersonalBankingDao()` odpowiada zależności `personalBankingDao`. Ponadto zwróć uwagę, że klasa `PersonalBankingServiceImpl` jest podklasą klasy `ServiceTemplate`.

Diagram przedstawiony na rysunku 3.3 pokazuje, że definicja komponentu nadrzędnego (`serviceTemplate`) nie musi być *abstrakcyjna*, a definicje komponentów potomnych (`fixedDepositService` i `personalBankingService`) mogą definiować dodatkowe właściwości. Ponadto klasy przedstawiane przez definicje komponentów nadrzędnych (np. `ServiceTemplate`) i potomnych (np. `FixedDepositServiceImpl` i `PersonalBankingServiceImpl`) mogą być powiązane na skutek dziedziczenia.

Na podstawie rysunku 3.3 można wyciągnąć następujące wnioski:

- Kontener Springa tworzy egzemplarz komponentu `serviceTemplate`, ponieważ nie został on zdefiniowany jako *abstrakcyjny*.
- Klasy `FixedDepositServiceImpl` i `PersonalBankingServiceImpl` (odpowiadające definicjom komponentów potomnych) są podklasami klasy `ServiceTemplate`, która odpowiada definicji komponentu nadrzędnego `serviceTemplate`.
- Definicje komponentów `fixedDepositService` i `personalBankingService` zawierają dodatkowe właściwości, odpowiednio `fixedDepositDao` i `personalBankingDao`. Warto zwrócić uwagę, że wymienione definicje komponentów potomnych mogą zawierać także dodatkowe argumenty konstruktora oraz przypadki *nadpisania metod* (to zagadnienie omawiamy w podrozdziale 4.5).

Skoro definicja komponentu `serviceTemplate` nie jest *abstrakcyjna*, inne komponenty mogą wskazywać `serviceTemplate` jako swoją zależność. Na przykład w listingu 3.3 komponent `serviceTemplate` jest zależnością komponentu `userRequestController`. Na podstawie przedstawionych



Rysunek 3.3. Definicje komponentów potomnych zawierają dodatkowe właściwości, definicja komponentu nadrzędnego nie jest abstrakcyjna, a ponadto istnieją zawiązki nadrzędny – potomny między klasami przedstawiającymi definicje nadrzędne i potomne

tutaj informacji możesz wywnioskować, że jeśli definicja komponentu nadrzędnego nie jest *abstrakcyjna*, wówczas funkcjonalność oferowana przez ten komponent może być wykorzystana nie tylko przez komponenty potomne, ale również inne komponenty w kontekście aplikacji.

Przykład dziedziczenia definicji komponentu — dziedziczenie konfiguracji metody fabryki

Definicje komponentów potomnych mogą stosować dziedziczenie definicji komponentów w celu dziedziczenia konfiguracji metody fabryki po definicji komponentu nadrzędnego. Spójrz na przykład pokazujący, jak konfiguracja metody fabryki jest dziedziczona przez definicje komponentów potomnych.

W listingu 3.5 została przedstawiona klasa `ControllerFactory` definiująca metodę `getController()`, czyli metodę fabryki *egzemplarza*.

Listing 3.5. Klasa `ControllerFactory`

Projekt: `ch03-bankapp-inheritance-examples`

Położenie pliku: `src/main/java/sample/spring/chapter03/bankapp/controller`

```

package sample.spring.chapter03.bankapp.controller;

public class ControllerFactory {

    public Object getController(String controllerName) {
        Object controller = null;
        if ("fixedDepositController".equalsIgnoreCase(controllerName)) {
            controller = new FixedDepositControllerImpl();
        }
        if ("personalBankingController".equalsIgnoreCase(controllerName)) {
            controller = new PersonalBankingControllerImpl();
        }
    }
}

```

```

    }
    return controller;
}
}

```

W powyższym listingu widać, że metoda fabryki `getController()` tworzy egzemplarz klasy `FixedDepositControllerImpl` lub `PersonalBankingControllerImpl`, w zależności od wartości przekazanego jej argumentu `controllerName`. Jeżeli wartością argumentu `controllerName` jest `fixedDepositController`, wówczas metoda `getController()` utworzy egzemplarz klasy `FixedDepositControllerImpl`. Natomiast gdy wartością argumentu `controllerName` jest `personalBankingController`, metoda `getController()` utworzy egzemplarz klasy `PersonalBankingControllerImpl`.

W listingu 3.6 przedstawiamy definicje komponentów w pliku `applicationContext.xml` z projektu `ch03-bankapp-inheritance-example`. Jak widzisz, definicje komponentów potomnych dziedziczą konfigurację wspomnianej wcześniej metody fabryki *egzemplarza* po definicji komponentu nadrzędnego.

Listing 3.6. Plik `applicationContext.xml` zawierający definicje komponentów i przedstawiający dziedziczenie konfiguracji metody fabryki

Projekt: `ch03-bankapp-inheritance-examples`

Położenie pliku: `src/main/resources/META-INF/spring`

```

<bean id="controllerFactory"
      class="sample.spring.chapter03.bankapp.controller.ControllerFactory" />

<bean id="controllerTemplate" factory-bean="controllerFactory"
      factory-method="getController" abstract="true">
</bean>

<bean id="fixedDepositController" parent="controllerTemplate">
  <constructor-arg index="0" value="fixedDepositController" />
  <property name="fixedDepositService" ref="fixedDepositService" />
</bean>

<bean id="personalBankingController" parent="controllerTemplate">
  <constructor-arg index="0" value="personalBankingController" />
  <property name="personalBankingService" ref="personalBankingService" />
</bean>

```

W listingu 3.6 klasa `ControllerFactory` przedstawia klasę fabryki definiującą metodę `getController()`, która jest metodą fabryki *egzemplarza*. Definicja komponentu `controllerTemplate` określa, że metoda fabryki `getController()` klasy `ControllerFactory` będzie używana do tworzenia egzemplarzy komponentu. Przedstawiona w listingu 3.5 metoda `getController()` tworzy egzemplarz komponentu `FixedDepositControllerImpl` lub `PersonalBankingControllerImpl`, w zależności od argumentu przekazanego metodzie `getController()`.

Ponieważ definicja komponentu `controllerTemplate` została oznaczona jako *abstrakcyjna*, od definicji komponentów potomnych `fixedDepositController` i `personalBankingController` zależy użycie konfiguracji metody fabryki `getController()`. Definicja komponentu `fixedDepositController` będzie przekazywała argument metodzie fabryki `getController()` klasy `ControllerFactory` odpowiedzialnej za tworzenie egzemplarzy komponentu `FixedDepositControllerImpl`. Definicja kom-

ponentu `personalBankingController` będzie przekazywała argument metodzie fabryki `getController()` klasy `ControllerFactory` odpowiedzialnej za tworzenie egzemplarzy komponentu `PersonalBankingControllerImpl`. W podrozdziale 2.3 widziałeś, że element `<constructor-arg>` jest używany w celu przekazania argumentu metodzie fabryki *egzemplarza*. W listingu 3.6 element `<constructor-arg>` został użyty przez definicje komponentów potomnych `fixedDepositController` i `personalBankingController` do przekazania metodzie fabryki `getController()` wartości odpowiednio `fixedDepositService` i `personalBankingService`.

Teraz warto wykonać metodę `main()` klasy `BankApp` w projekcie *ch03-bankapp-inheritance-example*, aby zobaczyć, jak w praktyce działa omawiane w rozdziale dziedziczenie definicji komponentów.

Przechodzimy teraz do tematu dopasowania argumentu konstruktora.

3.3. Dopasowanie argumentu konstruktora

W poprzednim rozdziale widziałeś, że argumenty konstruktora są w definicji komponentu określane za pomocą elementu `<constructor-arg>`. W tym podrozdziale zobaczymy, jak kontener Springa dopasowuje określony przez element `<constructor-arg>` argument konstruktora do odpowiadającego mu argumentu konstruktora zdefiniowanego w konstruktorze klasy komponentu.

Zanim jednak przejdziemy do szczegółów dopasowywania argumentu konstruktora, przypomnimy sobie, jak można przekazywać argumenty do konstruktora klasy komponentu.

IMPORT **Rozdział103/ch03-bankapp-constructor-args-by-type.** (Ten projekt pokazuje aplikację `MyBank`, w której argumenty konstruktora klasy są dopasowywane *według typu*, co zostanie dokładnie omówione w dalszej części podrozdziału. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Przekazywanie wartości i odwołań do komponentów za pomocą elementu `<constructor-arg>`

Jeżeli argument konstruktora jest prostym typem Javy (np. `int`, `String`), wówczas atrybut `value` elementu `<constructor-arg>` będzie używany w celu określenia wartości argumentu konstruktora. Natomiast gdy argument konstruktora jest odwołaniem do komponentu, nazwę komponentu podajesz za pomocą atrybutu `ref` elementu `<constructor-arg>`.

W listingu 3.7 została przedstawiona klasa `UserRequestControllerImpl` z projektu *ch03-bankapp-constructor-args-by-type*, której konstruktor akceptuje argument typu `ServiceTemplate`.

Listing 3.7. Klasa `UserRequestControllerImpl`

Projekt: *ch03-bankapp-constructor-args-by-type*

Położenie pliku: *src/main/java/sample/spring/chapter03/bankapp/controller*

```
package sample.spring.chapter03.bankapp.controller;

public class UserRequestControllerImpl implements UserRequestController {
    private ServiceTemplate serviceTemplate;

    public UserRequestControllerImpl(ServiceTemplate serviceTemplate) {
        this.serviceTemplate = serviceTemplate;
    }
}
```

```

    }

    @Override
    public void submitRequest(Request request) {
        // Wykonanie operacji z użyciem ServiceTemplate.
        serviceTemplate.getJmsMessageSender(); // Na przykład wysłanie komunikatu JMS.
        .....
    }
}

```

W listingu 3.8 pokazujemy, że odwołanie do egzemplarza `ServiceTemplate` (przedstawianego przez definicję komponentu `serviceTemplate`) jest przekazywane konstruktorowi `UserRequestControllerImpl` za pomocą atrybutu `ref` elementu `<constructor-arg>`.

Listing 3.8. Plik `applicationContext.xml` pokazujący przekazanie odwołania komponentu Springa za pomocą argumentu konstruktora

Projekt: `ch03-bankapp-constructor-args-by-type`

Położenie pliku: `src/main/resources/META-INF/spring`

```

<bean id="serviceTemplate" class="sample.spring.chapter03.bankapp.base.ServiceTemplate">
    .....
</bean>

<bean id="userRequestController"
      class="sample.spring.chapter03.bankapp.controller.UserRequestControllerImpl">
    <constructor-arg index="0" ref="serviceTemplate" />
</bean>

```

Skoro już wiesz, jak przekazywać proste wartości i odwołania do komponentów jako argumenty konstruktora, teraz możemy przejść do tematu dopasowywania przez kontener Springa *typu* argumentu konstruktora w celu ustalenia konstruktora komponentu, który powinien być wywołany.

Dopasowanie argumentu konstruktora na podstawie typu

Jeżeli atrybut `index` elementu `<constructor-arg>` *nie* został określony, wówczas kontener Springa ustala konstruktor do wywołania przez porównanie typów wskazanych przez elementy `<constructor-arg>` z typami podanymi w konstruktorze (lub konstruktorach) klasy komponentu.

Najpierw zobaczymy, jak kontener Springa dopasowuje argumenty konstruktora, gdy są one komponentami Springa, które *nie* są powiązane przez dziedziczenie.

Argumenty konstruktora przedstawiające oddzielne komponenty Springa

W listingu 3.9 została przedstawiona klasa `ServiceTemplate` zawierająca konstruktor, który akceptuje odwołania do komponentów `JmsMessageSender`, `EmailMessageSender` i `WebServiceInvoker`.

W listingu 3.10 została przedstawiona definicja komponentu dla klasy `ServiceTemplate` i komponenty, do których się ona odwołuje.

W listingu 3.10 elementy `<constructor-arg>` komponentu `serviceTemplate` *nie* mają zdefiniowanego atrybutu `index`. Kolejność argumentów konstruktora jest wskazana przez elementy

Listing 3.9. *Klasa ServiceTemplate**Projekt: ch03-bankapp-constructor-args-by-type**Położenie pliku: src/main/java/sample/spring/chapter03/bankapp/base*

```
package sample.spring.chapter03.bankapp.base;

public class ServiceTemplate {
    .....
    public ServiceTemplate(JmsMessageSender jmsMessageSender,
        EmailMessageSender emailMessageSender,
        WebServiceInvoker webServiceInvoker) {
        .....
    }
}
```

Listing 3.10. *Plik applicationContext.xml pokazujący definicję klasy ServiceTemplate i jej zależności**Projekt: ch03-bankapp-constructor-args-by-type**Położenie pliku: src/main/resources/META-INF/spring*

```
<bean id="serviceTemplate" class="sample.spring.chapter03.bankapp.base.ServiceTemplate">
    <constructor-arg ref="emailMessageSender" />
    <constructor-arg ref="jmsMessageSender" />
    <constructor-arg ref="webServiceInvoker" />
</bean>

<bean id="jmsMessageSender"
    class="sample.spring.chapter03.bankapp.base.JmsMessageSender" />
<bean id="emailMessageSender"
    class="sample.spring.chapter03.bankapp.base.EmailMessageSender" />
<bean id="webServiceInvoker"
    class="sample.spring.chapter03.bankapp.base.WebServiceInvoker" />
```

<constructor-arg> i przedstawia się następująco: EmailMessageSender, JmsMessageSender, WebServiceInvoker. Natomiast kolejność, w jakiej argumenty konstruktora są zdefiniowane w konstruktorze klasy ServiceTemplate, to: JmsMessageSender, EmailMessageSender, WebServiceInvoker. Jak widzisz, kolejność argumentów zdefiniowanych przez elementy <constructor-arg> jest inna od wskazanej w konstruktorze klasy ServiceTemplate.

Jeżeli wykonasz metodę main() klasy BankApp w projekcie *ch03-bankapp-constructor-args-by-type*, to zobaczysz, że kontener Springa z powodzeniem utworzył egzemplarz komponentu ServiceTemplate. Wynika to z faktu, że klasy JmsMessageSender, EmailMessageSender i WebServiceInvoker są różne z natury (czyli nie są powiązane ze względu na dziedziczenie), co znacznie ułatwia kontenerowi Springa wstrzykiwanie ich egzemplarzy w konstruktorze klasy ServiceTemplate we właściwej kolejności.

Jeżeli zaś typy argumentów konstruktora będą powiązane na skutek dziedziczenia, kontener Springa potrzebuje informacji dodatkowych, aby poradzić sobie z tego rodzaju argumentami konstruktora. Spójrz teraz, jak kontener Springa dopasowuje argumenty konstruktora, gdy wskazywane przez nie komponenty są powiązane ze sobą na skutek dziedziczenia.

Argumenty konstruktora przedstawiające powiązane ze sobą komponenty Springa

W listingu 3.11 została przedstawiona klasa komponentu `SampleBean`, której konstruktor akceptuje typy powiązane na skutek dziedziczenia.

Listing 3.11. Klasa `SampleBean`

```
public class SampleBean {  
    public SampleBean(ABean aBean, BBean bBean) { ..... }  
    .....  
}
```

Z kodu przedstawionego w listingu 3.11 wynika, że konstruktor klasy `SampleBean` akceptuje argumenty typu `ABean` i `BBean`. Wymienione typy `ABean` i `BBean` przedstawiają komponenty Springa powiązane ze sobą na skutek dziedziczenia — `BBean` jest podklasą klasy `ABean`.

W listingu 3.12 został przedstawiony plik XML kontekstu aplikacji zawierający definicje komponentów dla klas `SampleBean`, `ABean` i `BBean`.

Listing 3.12. Plik `applicationContext.xml` pokazujący definicję klas `SampleBean`, `ABean` i `BBean`

```
<bean id="aBean" class="example.ABean"/>  
<bean id="bBean" class="example.BBean"/>  
  
<bean id="sampleBean" class="example.SampleBean">  
    <constructor-arg ref="bBean"/>  
    <constructor-arg ref="aBean"/>  
</bean>
```

Ponieważ komponenty `aBean` i `bBean` są powiązane ze sobą na skutek dziedziczenia, kontener Springa stosuje argumenty konstruktora w konstruktorze `SampleBean` w kolejności, w jakiej elementy `<constructor-arg>` pojawiają się w definicji komponentu dla klasy `SampleBean`. W omawianej tutaj definicji komponentu `sampleBean` pierwszy element `<constructor-arg>` odwołuje się do komponentu `bBean`, natomiast drugi element `<constructor-arg>` odwołuje się do komponentu `aBean`. Oznacza to, że `bBean` jest przekazywany jako pierwszy, a `aBean` jako drugi argument konstruktora `SampleBean`. Ponieważ egzemplarz `ABean` (superklasa) nie może być przekazany tam, gdzie oczekiwany jest egzemplarz `BBean` (podklasa), drugi element `<constructor-arg>` w definicji komponentu `sampleBean` powoduje zgłoszenie wyjątku przez kontener Springa. Aby obsłużyć tego rodzaju sytuację, można użyć atrybutu `index` lub `type` elementu `<constructor-arg>` w celu identyfikacji argumentu konstruktora, dla którego zastosowanie ma dany element `<constructor-arg>`. Na przykład w przedstawionej w listingu 3.13 definicji komponentu `sampleBean` wykorzystano atrybut `type` do wskazania typu argumentu konstruktora, do którego ma zastosowanie dany element `<constructor-arg>`.

Listing 3.13. Atrybut `type` elementu `<constructor-arg>` identyfikuje typ argumentu konstruktora

```
<bean id="sampleBean" class="example.SampleBean">  
    <constructor-arg type="sample.spring.chapter03.bankapp.controller.BBean"  
        ref="bBean"/>  
    <constructor-arg type="sample.spring.chapter03.bankapp.controller.ABean"  
        ref="aBean"/>  
</bean>
```

Atrybut `type` elementu `<constructor-arg>` określa w pełni kwalifikowaną nazwę typu, do którego zastosowanie ma dany element `<constructor-arg>`. W listingu 3.13 pierwszy element `<constructor-arg>` ma zastosowanie do argumentu konstruktora typu `BBean`, natomiast drugi element `<constructor-arg>` odwołuje się do argumentu konstruktora typu `ABean`. Określenie atrybutu `type` pozwala na rozwianie wszelkich wątpliwości, jakie mogą się pojawić, gdy argumenty konstruktora są ze sobą powiązane na skutek dziedziczenia.



Jeżeli mamy dwa lub więcej argumentów konstruktora tego samego typu, jedynym rozwiązaniem będzie użycie atrybutu `index` w celu powiązania argumentów konstruktora z poszczególnymi elementami `<constructor-arg>`.

Jak dotąd analizowaliśmy dopasowanie typu argumentu konstruktora, gdy poszczególne argumenty są przedstawiane przez oddzielne lub powiązane ze sobą komponenty Springa. Teraz zobaczymy, jak następuje dopasowanie typów argumentów konstruktora dla typów standardowych Javy (np. `int`, `long`, `boolean`, `String`, `Date`) i niestandardowych.

Argumenty konstruktora przedstawiające typy danych Javy lub niestandardowe

Jeżeli typem argumentu konstruktora jest typ podstawowy (np. `int`, `long`, `boolean`), `String` lub niestandardowy (np. `Address`), wówczas atrybut `value` elementu `<constructor-arg>` jest używany do określenia wartości. W przypadku istnienia dwóch lub więcej argumentów konstruktora, dla których wartość w postaci ciągu tekstowego i określona przez atrybut `value` może być skonwertowana, kontener Springa nie potrafi ustalić typu argumentu konstruktora (np. czy wartość przedstawia typ `int`, `long` czy `String`). W takich sytuacjach trzeba wyraźnie wskazać typ argumentu konstruktora, używając do tego atrybutu `type`.

W listingu 3.14 została przedstawiona klasa `TransferFundsServiceImpl` definiująca konstruktor, który akceptuje argumenty typów `String`, `boolean`, `long` i `int`.

Listing 3.14. Klasa `TransferFundsServiceImpl`

Projekt: `ch03-bankapp-constructor-args-by-type`

Położenie pliku: `src/main/java/sample/spring/chapter03/bankapp/service`

```
package sample.spring.chapter03.bankapp.service;

public class TransferFundsServiceImpl implements TransferFundsService {
    public TransferFundsServiceImpl(String webServiceUrl, boolean active, long timeout,
        int numberOfRetrialAttempts) {.....}
    .....
}
```

Jak możesz zobaczyć w listingu 3.14, konstruktor klasy `TransferFundsServiceImpl` akceptuje następujące argumenty: `webServiceUrl`, `active`, `timeout` i `numberOfRetrialAttempts`. Definicja komponentu klasy `TransferFundsServiceImpl` znajduje się w listingu 3.15 i wyraźnie wskazuje, jakie wartości argumentów konstruktora mogą być przekazane konstruktorowi klasy `TransferFundsServiceImpl`.

Listing 3.15. Definicja komponentu dla klasy *TransferFundsServiceImpl*

```
<bean id="transferFundsService"
      class="sample.spring.chapter03.bankapp.service.TransferFundsServiceImpl">
  <constructor-arg value="http://someUrl.com/xyz" />
  <constructor-arg value="true" />
  <constructor-arg value="5" />
  <constructor-arg value="200" />
</bean>
```

Przyjmujemy założenie, że trzeci element `<constructor-arg>` (wartość jego atrybutu `value` wynosi 5) ma na celu dostarczenie wartości dla argumentu konstruktora `numberOfRetrialAttempts`, natomiast czwarty element `<constructor-arg>` (wartość jego atrybutu `value` wynosi 200) ma dostarczyć wartość dla argumentu konstruktora `timeout`. Kontener Springa stosuje elementy `<constructor-arg>` w konstruktorze klasy *TransferFundsServiceImpl*, zachowując kolejność, w jakiej poszczególne elementy `<constructor-arg>` występują w definicji komponentu *transferFundsService*. Oznacza to, że trzeci element `<constructor-arg>` będzie zastosowany dla argumentu `timeout`, a czwarty element `<constructor-arg>` będzie zastosowany dla argumentu `numberOfRetrialAttempts`. W celu obsługi tego rodzaju wątpliwości można wskazać *typ* argumentu konstruktora, używając atrybutu `type` elementu `<constructor-arg>`, jak przedstawia listing 3.16.

Listing 3.16. Plik *applicationContext.xml*, w którym pogrubiono atrybut `type` elementów `<constructor-arg>`

Projekt: *ch03-bankapp-constructor-args-by-type*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean id="transferFundsService"
      class="sample.spring.chapter03.bankapp.service.TransferFundsServiceImpl">
  <constructor-arg type="java.lang.String" value="http://someUrl.com/xyz" />
  <constructor-arg type="boolean" value="true" />
  <constructor-arg type="int" value="5" />
  <constructor-arg type="long" value="200" />
</bean>
```

W przedstawionej w listingu 3.16 definicji komponentu dla klasy *TransferFundsServiceImpl* atrybut `type` został użyty w celu wskazania typu argumentu konstruktora. Kontener Springa może teraz użyć dopasowania typu i odpowiednio zastosować argumenty konstruktora.



Jeżeli mamy dwa lub więcej argumentów konstruktora tego samego typu, jedynym rozwiązaniem będzie użycie atrybutu `index` w celu powiązania argumentów konstruktora z poszczególnymi elementami `<constructor-arg>`.

W tym podrozdziale zobaczyłeś, jak Spring przeprowadza dopasowanie typu, aby prawidłowo zastosować argumenty konstruktora. Teraz sprawdzimy, jak można nakazać Springowi przeprowadzenie operacji dopasowania argumentu konstruktora na podstawie nazwy argumentu.

IMPORT **Rozdział 103/ch03-bankapp-constructor-args-by-name.** (Ten projekt pokazuje aplikację *MyBank*, w której argumenty konstruktora są dopasowywane na podstawie *nazwy*. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie *BankApp* projektu).

Dopasowanie argumentu konstruktora na podstawie nazwy argumentu

Atrybut `name` elementu `<constructor-arg>` jest używany w celu podania nazwy argumentu konstruktora, do którego ma zastosowanie dany element `<constructor-arg>`. W listingu 3.17 ponownie przedstawiono klasę `TransferFundsServiceImpl`, której konstruktor akceptuje wiele argumentów.

Listing 3.17. Klasa `TransferFundsServiceImpl`

Projekt: `ch03-bankapp-constructor-args-by-name`

Położenie pliku: `src/main/java/sample/spring/chapter03/bankapp/service`

```
package sample.spring.chapter03.bankapp.service;

public class TransferFundsServiceImpl implements TransferFundsService {
    ....
    public TransferFundsServiceImpl(String webServiceUrl, boolean active, long timeout,
        int numberOfRetrialAttempts) { .... }
}
```

Na podstawie listingu 3.17 można stwierdzić, że nazwy argumentów konstruktora zdefiniowanych w klasie `TransferFundsServiceImpl` to `webServiceUrl`, `active`, `timeout` i `numberOfRetrialAttempts`.



Konstruktor klasy `TransferFundsServiceImpl` akceptuje argumenty, które są prostymi typami Javy (np. `int`, `long`, `boolean`, `String`). Jednak koncepcje wyjaśnione w tym podrozdziale mają zastosowanie także w sytuacjach, gdy argumenty konstruktora odwołują się do komponentów frameworku Spring.

W listingu 3.18 została przedstawiona definicja komponentu dla klasy `TransferFundsServiceImpl`. Atrybut `name` elementu `<constructor-arg>` został wykorzystany do podania nazwy argumentu konstruktora, dla którego ma zastosowanie dany element `<constructor-arg>`.

Listing 3.18. Plik `applicationContext.xml` przedstawiający przykład użycia atrybutu `name` elementu `<constructor-arg>`

Projekt: `ch03-bankapp-constructor-args-by-name`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="transferFundsService"
    class="sample.spring.chapter03.bankapp.service.TransferFundsServiceImpl">

    <constructor-arg name="webServiceUrl" value="http://someUrl.com/xyz" />
    <constructor-arg name="active" value="true" />
    <constructor-arg name="numberOfRetrialAttempts" value="5" />
    <constructor-arg name="timeout" value="200" />

</bean>
```

Konfiguracja przedstawiona w listingu 3.18 będzie działała tylko wtedy, gdy klasa `TransferFundsServiceImpl` zostanie skompilowana wraz z włączoną opcją debug (patrz: opcja `-g` polecenia `javac`). Po włączeniu opcji debug nazwy argumentów konstruktora są zachowywane w wygenerowanym pliku `.class`. Jeżeli przeprowadzisz kompilację bez włączonej opcji debug, nazwy argumentów

konstruktora zostaną utracone w trakcie kompilacji. Dlatego Spring nie będzie miał możliwości odszukania argumentu konstruktora odpowiadającego nazwie podanej w atrybucie `name` elementu `<constructor-arg>`.

Jeśli nie chcesz kompilować klas wraz z włączoną opcją debug, możesz wykorzystać adnotację `@ConstructorProperties` (została wprowadzona w Javie SE 6) w celu wyraźnego podania nazw argumentów konstruktora, jak w przypadku klasy `TransferFundsServiceImpl` przedstawionej w listingu 3.19.

Listing 3.19. Przykład użycia adnotacji `@ConstructorProperties`

Projekt: *ch03-bankapp-constructor-args-by-name*

Położenie pliku: *src/main/java/sample/spring/chapter03/bankapp/service*

```
package sample.spring.chapter03.bankapp.service;

import java.beans.ConstructorProperties;

public class TransferFundsServiceImpl implements TransferFundsService {

    @ConstructorProperties({"webServiceUrl", "active", "timeout",
        "numberOfRetrialAttempts"})
    public TransferFundsServiceImpl(String webServiceUrl, boolean active, long timeout,
        int numberOfRetrialAttempts) { ..... }
}
```

W listingu 3.19 adnotacja `@ConstructorProperties` wskazuje nazwy argumentów konstruktora w kolejności, w jakiej pojawiają się w konstruktorze klasy komponentu. Koniecznie *musisz* się upewnić, że używane są takie same nazwy argumentów konstruktora jak zdefiniowane w elementach `<constructor-arg>`.

Przekonamy się teraz, jaki wpływ ma adnotacja `@ConstructorProperties` na dziedziczenie definicji komponentu.

Adnotacja `@ConstructorProperties` i dziedziczenie definicji komponentu

Jeżeli konstruktor klasy odpowiadający definicji komponentu *nadrzędnego* jest oznaczony adnotacją `@ConstructorProperties`, wówczas klasa odpowiadająca definicji komponentu *potomnego* również *musi* być oznaczona adnotacją `@ConstructorProperties`.

W listingu 3.20 zostały przedstawione definicje komponentów `serviceTemplate` (nadrzędny) i `FixedDepositService` (potomny).

Listing 3.20. Plik `applicationContext.xml` przedstawiający definicje komponentów nadrzędnego i potomnego

Projekt: *ch03-bankapp-constructor-args-by-name*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean id="serviceTemplate"
    class="sample.spring.chapter03.bankapp.base.ServiceTemplate">
    <constructor-arg name="emailMessageSender" ref="emailMessageSender" />
    <constructor-arg name="jmsMessageSender" ref="jmsMessageSender" />
    <constructor-arg name="webServiceInvoker" ref="webServiceInvoker" />
</bean>
```



```
<bean id="FixedDepositService"
      class="sample.spring.chapter03.bankapp.service.FixedDepositServiceImpl"
      parent="serviceTemplate">
    <property name="fixedDepositDao" ref="FixedDepositDao" />
</bean>
```

W listingu 3.20 przedstawiona definicja komponentu `serviceTemplate` nie jest *abstrakcyjna*, czyli kontener Springa utworzy egzemplarz `serviceTemplate`. W definicji `serviceTemplate` znajdują się trzy elementy `<constructor-arg>` odpowiadające trzem argumentom zdefiniowanym w klasie `ServiceTemplate` (patrz: listing 3.21). Ponieważ w definicji komponentu `serviceTemplate` argumenty konstruktora zostały podane *według nazw*, konstruktor klasy `ServiceTemplate` ma adnotację `@ConstructorProperties`. Gwarantuje ona, że nazwy argumentów konstruktora będą dla Springa dostępne w trakcie działania aplikacji.

Listing 3.21. Klasa `ServiceTemplate`

Projekt: `ch03-bankapp-constructor-args-by-name`

Położenie pliku: `src/main/java/sample/spring/chapter03/bankapp/base`

```
package sample.spring.chapter03.bankapp.base;

import java.beans.ConstructorProperties;

public class ServiceTemplate {
    ....
    @ConstructorProperties({"jmsMessageSender", "emailMessageSender",
        "webServiceInvoker"})
    public ServiceTemplate(JmsMessageSender jmsMessageSender,
        EmailMessageSender emailMessageSender,
        WebServiceInvoker webServiceInvoker) { .... }
}
```

Ponieważ `FixedDepositService` to definicja komponentu potomnego dla `serviceTemplate`, konfiguracja `<constructor-arg>` w definicji komponentu `serviceTemplate` jest dziedziczona przez definicję komponentu `FixedDepositService`. Oznacza to, że klasa `FixedDepositServiceImpl` *musi* definiować konstruktor akceptujący ten sam zestaw argumentów, jakie są zdefiniowane w klasie `ServiceTemplate`. Ponadto *musi* mieć adnotację `@ConstructorProperties`. Jeżeli konstruktor `FixedDepositServiceImpl` nie będzie miał adnotacji `@ConstructorProperties`, kontener Springa *nie* będzie w stanie dopasować dziedziczonych elementów `<constructor-arg>` do argumentów konstruktora podanych w konstruktorze `FixedDepositServiceImpl`.

Jak wyjaśnimy niżej, adnotacji `@ConstructorProperties` nie można użyć w celu przekazania argumentów według nazwy do metody fabryki *statycznej* lub *egzemplarza*.

Adnotacja `@ConstructorProperties` i metody fabryki

Z podrozdziału 2.3 dowiedziałeś się, że elementy `<constructor-arg>` są używane także w celu przekazywania argumentów do metod fabryki, zarówno *statycznych*, jak i *egzemplarza*. Być może sądzisz, że możesz przekazać argumenty według nazwy do metody fabryki przez użycie atrybutu `name` elementu `<constructor-arg>` i dodanie adnotacji `@ConstructorProperties` do metody fabryki.

Zwróć jednak uwagę, że adnotacja `@ConstructorProperties` jest przeznaczona *tylko* dla konstruktorów. Nie można więc metod oznaczać adnotacją `@ConstructorProperties`. Dlatego też jeżeli do metody fabryki *statycznej* lub *egzemplarza* chcesz przekazać argumenty według nazwy, jedynym rozwiązaniem pozostaje kompilacja klas wraz z włączoną opcją debug.



Kompilacja klas wraz z włączoną opcją debug powoduje wygenerowanie większych plików `.class`, ale to nie ma wpływu na wydajność działania aplikacji. Jedynym skutkiem ubocznym jest wydłużenie czasu potrzebnego na wczytanie klas.

Spójrzmy teraz, jak można włączyć lub wyłączyć opcję debug w środowisku IDE Eclipse.

Włączenie lub wyłączenie opcji debug w Eclipse

Włączenie opcji debug dla projektów wymaga w Eclipse wykonania następujących kroków:

1. Wybierz opcję *Windows/Preferences*, a następnie *Java/Compiler*.
2. Zobaczysz sekcję zatytułowaną *Classfile Generation*. W tej sekcji zaznaczenie pola wyboru o nazwie *Add variable attributes to generated class files (used by the debugger)* powoduje włączenie opcji debug. Z kolei usunięcie zaznaczenia tego pola tekstowego *wyłącza* opcję debug.

Dotychczas przedstawiliśmy przede wszystkim przykłady definicji komponentów, w których właściwościami i argumentami konstruktora były odwołania do innych komponentów. Przejdziemy teraz do przykładów definicji komponentów, w których właściwościami i argumentami konstruktora są typy proste, kolekcje, `java.util.Date`, `java.util.Properties` itd.

3.4. Konfiguracja różnego typu właściwości komponentu i argumentów konstruktora

W rzeczywistości tworzonych aplikacjach właściwości i argumenty konstruktora komponentu Springa mogą mieć różne typy, począwszy od tekstowego (`String`), przez odwołania do innych standardowych komponentów (np. `java.util.Date`, `java.util.Properties`), aż po typ niestandardowy (np. `Address`). Dotychczas przedstawiliśmy przykłady dostarczania wartości dla właściwości typu `String` (za pomocą atrybutu `value` elementu `<property>`) i argumentów konstruktora typu `String` (za pomocą atrybutu `value` elementu `<constructor-arg>`). Dowiedziałeś się także, jak wstrzykiwać zależności za pomocą właściwości komponentu (używając atrybutu `ref` elementu `<property>`) i argumenty konstruktora (używając atrybutu `ref` elementu `<constructor-arg>`).

W tym podrozdziale spojrzymy na wbudowane implementacje `PropertyEditor`, które ułatwiają przekazywanie właściwości i argumentów konstruktora typu `java.util.Date`, `java.util.Currency`, typów prostych itd. Zobaczysz także, jak podawać wartości dla kolekcji (takich jak `java.util.List`, `java.util.Map`) w pliku XML kontekstu aplikacji oraz jak zarejestrować własną implementację `PropertyEditor` w Springu.

Przechodzimy teraz do przykładów definicji komponentów prezentujących użycie wbudowanych implementacji `PropertyEditor`.

IMPORT `Rozdział03/ch03-simple-types-examples`. (Ten projekt pokazuje aplikację Springa, w której właściwości komponentu i argumenty konstruktora są typu prostego, `java.util.Date`, `java.util.List`, `java.util.Map` itd. Ten projekt pokazuje także, jak zarejestrować własną implementację `PropertyEditor` w kontenerze Springa. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `SampleApp` projektu).

Wbudowane edytory właściwości

Oferowana przez JavaBeans implementacja `PropertyEditor` zapewnia logikę niezbędną do konwersji typu Javy na wartość w postaci ciągu tekstowego i na odwrót. Spring dostarcza kilku wbudowanych implementacji `PropertyEditor`, które są używane do konwersji wartości w postaci ciągu tekstowego właściwości komponentu lub argumentu konstruktora (podanego za pomocą atrybutu `value` elementów `<property>` i `<constructor-arg>`) na rzeczywisty typ Javy właściwości lub argument konstruktora.

Zanim przejdziemy do przykładów wykorzystujących wbudowane implementacje `PropertyEditor`, musisz zrozumieć wagę tych implementacji podczas ustawiania wartości właściwości komponentu lub argumentów konstruktora.

Rozważmy przedstawiony w listingu 3.22 przykład klasy `BankDetails`, którą chcemy skonfigurować jako klasę o zasięgu singleton wraz z predefiniowanymi wartościami dla jej atrybutów.

Listing 3.22. Klasa `BankDetails`

```
public class BankDetails {
    private String bankName;

    public void setBankName(String bankName) {
        this.bankName = bankName;
    }
}
```

W powyższym listingu `bankName` to atrybut klasy `BankDetails`, a jego typem jest `String`. W listingu 3.23 znajduje się definicja komponentu dla klasy `BankDetails` pokazująca, jak można przypisać atrybutowi `bankName` wartość `Mój osobisty bank`.

Listing 3.23. Definicja komponentu dla klasy `BankDetails`

```
<bean id="bankDetails" class="BankDetails">
    <property name="bankName" value="Mój osobisty bank"/>
</bean>
```

W powyższym listingu atrybut `value` elementu `<property>` określa wartość w postaci ciągu tekstowego dla właściwości `bankName`. Jak możesz zobaczyć w listingu, jeżeli właściwość komponentu jest typu `String`, wówczas do przypisania jej wartości wykorzystywany jest atrybut `value` elementu `<property>`. Podobnie, gdy argument konstruktora jest ciągiem tekstowym (typ `String`), do ustawienia jego wartości można użyć atrybutu `value` elementu `<constructor-arg>`.

Przyjmujemy założenie, że wymienione poniżej atrybuty (wraz z odpowiednimi metodami typu setter) zostały dodane do klasy `BankDetails`: atrybut `bankPrimaryBusiness` typu `byte[]`, atrybut

headOfficeAddress typu `char[]`, atrybut `privateBank` typu `char`, atrybut `primaryCurrency` typu `java.util.Currency`, atrybut `dateOfInception` typu `java.util.Date` i atrybut `branchAddresses` typu `java.util.Properties`. W listingu 3.24 przedstawiamy zmodyfikowaną wersję klasy `BankDetails`.

Listing 3.24. Klasa `BankDetails` zawierająca różne typy właściwości

Projekt: *ch03-simple-types-examples*

Położenie pliku: *src/main/java/sample/spring/chapter03/beans*

```
package sample.spring.chapter03.beans;
.....
public class BankDetails {
    private String bankName;
    private byte[] bankPrimaryBusiness;
    private char[] headOfficeAddress;
    private char privateBank;
    private Currency primaryCurrency;
    private Date dateOfInception;
    private Properties branchAddresses;
    .....
    public void setBankName(String bankName) {
        this.bankName = bankName;
    }
    // Kolejne metody typu setter.
}
```

Klasę `BankDetails` można skonfigurować jako komponent Springa przez określenie wartości w postaci ciągu tekstowego dla właściwości, a następnie pozwolić kontenerowi Springa na konwersję tych wartości na odpowiadające im typy właściwości Javy przez użycie zarejestrowanych implementacji `PropertyEditor`.

W listingu 3.25 została przedstawiona definicja klasy `BankDetails` pokazującej, że proste wartości w postaci ciągu tekstowego zostały zdefiniowane dla różnego rodzaju typów właściwości.

Listing 3.25. Plik *applicationContext.xml* przedstawiający definicję komponentu dla klasy `BankDetails`

Projekt: *ch03-simple-types-examples*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean id="bankDetails" class="sample.spring.chapter03.beans.BankDetails">
    <property name="bankName" value="Mój osobisty bank" />
    <property name="bankPrimaryBusiness" value="Bankowość detaliczna" />
    <property name="headOfficeAddress" value="Adres centrali banku" />
    <property name="privateBank" value="Y" />
    <property name="primaryCurrency" value="INR" />
    <property name="dateOfInception" value="30-01-2012"></property>
    <property name="branchAddresses">
        <value>
            x = Adres placówki X
            y = Adres placówki Y
        </value>
    </property>
</bean>
```

Z listingu 3.25 wynika, że wartości w postaci ciągu tekstowego zostały określone dla właściwości typu `java.util.Date`, `java.util.Currency`, `char[]`, `byte[]`, `char` i `java.util.Properties`. Kontener Springa używa zarejestrowanych implementacji `PropertyEditors` do konwersji wartości właściwości lub argumentu konstruktora na odpowiadający im typ Javy dla właściwości lub argumentu konstruktora. Na przykład kontener Springa skonwertuje wartość `30-01-2012` właściwości `dateOfInception` na typ `java.util.Date`, używając do tego celu `CustomDateEditor` (wbudowana implementacja `PropertyEditor` dla typu `java.util.Date`).

Jeżeli przyjrzyj się sposobowi konfiguracji właściwości `branchAddresses` (typu `java.util.Properties`) w listingu 3.25, dostrzeżesz, że w celu określenia wartości dla właściwości zamiast atrybutu `value` elementu `<property>` użyto elementu `<value>` znajdującego się wewnątrz elementu `<property>`. W przypadku właściwości składających się z pojedynczej wartości preferowane jest użycie atrybutu `value` elementu `<property>` zamiast elementu `<value>`. Jednak gdy zachodzi konieczność określenia wielu wartości dla właściwości lub jeśli wartości muszą być podane w oddzielnych wierszach (jak w przypadku właściwości `branchAddresses`), preferowanym rozwiązaniem jest wykorzystanie elementu `<value>` zamiast atrybutu `value`. W następnym podrozdziale zobaczysz, że właściwości dla wartości (lub argumentów konstruktora) typu `java.util.Properties` mogą być określane także za pomocą elementu `<props>` znajdującego się w elemencie `<property>` (lub w `<constructor-arg>`).

Spring jest dostarczany wraz z kilkoma wbudowanymi implementacjami `PropertyEditor`, które wykonują zadanie konwersji wartości podanych w pliku XML kontekstu aplikacji na typ Javy właściwości komponentu lub argumentu konstruktora. W tabeli 3.1 wymieniamy niektóre z wbudowanych w Spring implementacji `PropertyEditor`.

Tabela 3.1. Wybrane wbudowane w Spring implementacje `PropertyEditor`

Nazwa	Opis
<code>CustomBooleanEditor</code>	Konwertuje ciąg tekstowy na typ boolowski.
<code>CustomNumberEditor</code>	Konwertuje ciąg tekstowy na liczbę (<code>int</code> , <code>long</code> itd.).
<code>CharacterEditor</code>	Konwertuje ciąg tekstowy na typ <code>char</code> .
<code>ByteArrayPropertyEditor</code>	Konwertuje ciąg tekstowy na typ <code>byte[]</code> .
<code>CustomDateEditor</code>	Konwertuje ciąg tekstowy na typ <code>java.util.Date</code> .
<code>PropertiesEditor</code>	Konwertuje ciąg tekstowy na typ <code>java.util.Properties</code> .

W tabeli 3.1 wymieniamy jedynie kilka wbudowanych w Spring implementacji `PropertyEditor`. Pełną listę znajdziesz w pakiecie `org.springframework.beans.propertyeditors`. Warto w tym miejscu zwrócić uwagę, że nie wszystkie wbudowane w Spring implementacje `PropertyEditor` są domyślnie zarejestrowane w kontenerze Springa. Na przykład konieczne jest wyraźne zarejestrowanie `CustomDateEditor`, aby umożliwić kontenerowi Springa przeprowadzanie konwersji z wartości w postaci ciągu tekstowego na typ `java.util.Date`. Z dalszej części podrozdziału dowiesz się, jak w kontenerze Springa zarejestrować edytor właściwości.

Zobaczymy teraz, jak określać wartości dla właściwości komponentu (lub argumentów konstruktora) typu `java.util.List`, `java.util.Set` i `java.util.Map`.

Określanie wartości dla różnych typów kolekcji

Zdefiniowane w schemacie beans elementy `<list>`, `<map>` i `<set>` znajdujące się w elementach `<property>` i `<constructor-arg>` są używane do określania wartości właściwości i argumentów konstruktora typów odpowiednio `java.util.List`, `java.util.Map` i `java.util.Set`.



Schemat util w Springu oferuje także elementy `<list>`, `<set>` i `<map>`, które ułatwiają ustawianie właściwości i argumentów konstruktora różnych typów kolekcji. W dalszej części rozdziału dokładnie zajmiemy się elementami schematu util.

W listingu 3.26 została przedstawiona klasa `DataTypesExample`, której konstruktor akceptuje argumenty różnych typów.

Listing 3.26. Klasa `DataTypesExample`

Projekt: `ch03-simple-types-examples`

Położenie pliku: `src/main/java/sample/spring/chapter03/beans`

```
package sample.spring.chapter03.beans;

import java.beans.ConstructorProperties;
.....
public class DataTypesExample {
    private static Logger logger = Logger.getLogger(DataTypesExample.class);

    @SuppressWarnings("rawtypes")
    @ConstructorProperties({ "byteArrayType", "charType", "charArrayType",
        "classType", "currencyType", "booleanType", "dateType", "longType",
        "doubleType", "propertiesType", "listType", "mapType", "setType",
        "anotherPropertiesType" })
    public DataTypesExample(byte[] byteArrayType, char charType,
        char[] charArray, Class classType, Currency currencyType,
        boolean booleanType, Date dateType, long longType,
        double doubleType, Properties propertiesType, List<Integer> listType,
        Map mapType, Set setType, Properties anotherPropertiesType) {
        .....
        logger.info("classType " + classType.getName());
        logger.info("listType " + listType);
        logger.info("mapType " + mapType);
        logger.info("setType " + setType);
        logger.info("anotherPropertiesType " + anotherPropertiesType);
    }
}
```

W powyższym listingu widać, że konstruktor klasy `DataTypesExample` akceptuje argumenty typu `java.util.List`, `java.util.Map`, `java.util.Set`, `java.util.Properties` itd. Wartość każdego argumentu konstruktora jest wyświetlana.

W listingu 3.27 została przedstawiona definicja komponentu dla klasy `DataTypesExample`.

Listing 3.27. Plik *applicationContext.xml* przedstawiający definicję dla klasy *DataTypesExample*Projekt: *ch03-simple-types-examples*Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
    .....
    <constructor-arg name="anotherPropertiesType">
        <props>
            <prop key="book">Rozpoczęcie pracy ze Spring Framework</prop>
        </props>
    </constructor-arg>
    <constructor-arg name="listType" value-type="java.lang.Integer" >
        <list>
            <value>1</value>
            <value>2</value>
        </list>
    </constructor-arg>
    <constructor-arg name="mapType">
        <map>
            <entry>
                <key>
                    <value>klucz 1</value>
                </key>
                <value>wartość klucza 1</value>
            </entry>
        </map>
    </constructor-arg>
    <constructor-arg name="setType">
        <set>
            <value>Element 1</value>
            <value>Element 2</value>
        </set>
    </constructor-arg>
</bean>
```

Z powyższego listingu można odczytać następujące informacje:

- Wartość *anotherPropertiesType* (typu *java.util.Properties*) została określona za pomocą elementu *<props>* znajdującego się w elemencie *<constructor-arg>*. Każdy element *<prop>* zawiera pary klucz – wartość. Atrybut *key* określa nazwę klucza, natomiast element *<prop>* przechowuje wartość klucza. W celu określenia wartości dla argumentu *anotherPropertiesType* zamiast użycia elementu *<props>* możesz wykorzystać *<value>* znajdujący się w elemencie *<constructor-arg>*.
- Wartość argumentu konstruktora *listType* (typu *java.util.List*) została podana za pomocą elementu *<list>* znajdującego się w elemencie *<constructor-arg>*. Elementy *<value>* zdefiniowane w elementach *<list>* określają elementy znajdujące się na liście. Atrybut *value-type* elementu *<list>* wskazuje, że argument konstruktora akceptuje typ Javy *java.util.List*. Ponieważ argument konstruktora *listType* jest typu *List<Integer>* (patrz: listing 3.26), wartość atrybutu *value-type* została podana jako typu *java.lang.Integer*. Atrybut *value-type* jest *opcjonalny*, ale okazuje się szczególnie użyteczny w przypadku użycia *parametryzowanego* typu *List*, na przykład *List<Integer>*. Jeżeli podajesz atrybut *value-type*, kontener Springa używa zarejestrowanych edytorów właściwości do przeprowadzenia konwersji wartości na typ wskazany

przez atrybut `value-type`. Jeżeli zajdzie potrzeba, zostanie wykonana konwersja wartości na typ akceptowany przez parametryzowany typ `List`. Jeżeli nie zastosujesz atrybutu `value-type`, kontener Springa po prostu użyje edytora właściwości do przeprowadzenia konwersji wartości na typ akceptowany przez parametryzowany typ `List`.

- Wartość argumentu konstruktora `mapType` (typu `java.util.Map`) jest określana za pomocą elementu `<map>` znajdującego się w elemencie `<constructor-arg>`. Element `<entry>` zdefiniowany w elemencie `<map>` zawiera parę klucz – wartość. W elemencie `<key>` mamy nazwę klucza, natomiast w elemencie `<value>` znajduje się wartość klucza. Atrybuty `key-type` i `value-type` elementu `<map>` określają typ Javy dla kluczy i ich wartości akceptowany przez `java.util.Map`. Atrybuty `key-type` i `value-type` są opcjonalne, ale okazują się szczególnie użyteczne w przypadku użycia parametryzowanego typu `Map`, na przykład `Map<Integer, Integer>`. Kontener Springa użyje zarejestrowanych edytorów właściwości do przeprowadzenia konwersji kluczy i ich wartości na typy wskazywane przez atrybuty `key-type` i `value-type`, a także na typy akceptowane przez parametryzowany typ `Map`.
- Wartość argumentu konstruktora `setType` (typu `java.util.Set`) jest określana za pomocą elementu `<set>` znajdującego się w elemencie `<constructor-arg>`. Element `<value>` znajdujący się w `<set>` definiuje element w kolekcji `Set`. Atrybut `value-type` elementu `<set>` określa typ Javy dla elementów akceptujących `java.util.Set`. Atrybut `value-type` jest opcjonalny, ale okazuje się szczególnie użyteczny w przypadku użycia parametryzowanego typu `Set`, na przykład `Set<Integer>`. Kontener Springa użyje zarejestrowanych edytorów właściwości do przeprowadzenia konwersji wartości na typ określony przez atrybut `value-type`, a także na typy akceptowane przez parametryzowany typ `Set`.

W klasie `DataTypesExample` (patrz: listingi 3.26 i 3.27) argumenty konstruktora typu `List`, `Map` i `Set` zawierały elementy typu `String` lub `Integer`. W aplikacji kolekcja może zawierać elementy typu `Map`, `Set`, `Class`, `Properties` lub innego dowolnego typu Javy. Ponadto elementy znajdujące się w kolekcji mogą być odwołaniami do komponentów. Aby zapewnić obsługę tego rodzaju sytuacji, kontener Springa pozwala na użycie elementów takich jak `<map>`, `<set>`, `<list>`, `<props>`, `<ref>` itd. wewnątrz elementów `<list>`, `<map>` i `<set>`. Teraz przeanalizujemy kilka przykładów pokazujących, jak dodać różnego typu elementy do argumentów konstruktora i właściwości komponentu typu `Map`, `List` i `Set`.

Dodanie do kolekcji elementów typu List, Map, Set i Properties

Jeżeli właściwość komponentu lub argument konstruktora jest typu `List<List>`, można po prostu użyć zagnieżdżonego elementu `<list>`, jak pokazuje listing 3.28.

Listing 3.28. Przykład użycia zagnieżdżonych elementów `<list>`

```
<constructor-arg name="nestedList">
  <list>
    <list>
      <value>Prosta wartość typu String zagnieżdżona w elemencie list</value>
      <value>Inna prosta wartość typu String zagnieżdżona w elemencie list</value>
    </list>
  </list>
</constructor-arg>
```


W powyższym listingu element `<constructor-arg>` dostarcza wartości dla argumentu konstruktora o nazwie `nestedList` i typie `List<List>`. Zagnieżdżony element `<list>` przedstawia elementy typu `List`. Podobnie istnieje możliwość użycia elementów `<map>`, `<set>` i `<props>` wewnątrz elementu `<list>` w celu określenia wartości właściwości lub argumentów konstruktora typu odpowiednio `List<Map>`, `List<Set>` i `List<Properties>`. Podobnie jak w przypadku `<list>`, element `<set>` może zawierać elementy `<set>`, `<list>`, `<map>` lub `<props>`. Natomiast w przypadku elementu `<map>` do określenia klucza i wartości można użyć elementów `<map>`, `<set>`, `<list>` lub `<props>`.

W listingu 3.29 pokazujemy, jak można zastosować wartości dla argumentu konstruktora typu `Map<List, Set>`.

Listing 3.29. Przykład konfiguracji, w której element `<map>` zawiera klucz typu `List` i wartość typu `Set`

```
<constructor-arg name="nestedListAndSetMap">
  <map>
    <entry>
      <key>
        <list>
          <value>Element List</value>
        </list>
      </key>
      <set>
        <value>Element Set</value>
      </set>
    </entry>
  </map>
</constructor-arg>
```

W listingu 3.29 zostało pokazane, że element `nestedListAndSetMap` argumentu konstruktora jest typu `Map`, którego klucz jest typu `List`, natomiast wartość typu `Set`. Element `<key>` może zawierać w sobie elementy wymienionych typów: `<map>`, `<set>`, `<list>` i `<props>`. Z kolei wartość klucza może zostać zdefiniowana za pomocą elementu `<map>`, `<set>`, `<list>` lub `<props>`.

Dodanie do kolekcji odwołania do komponentu

Istnieje możliwość użycia elementów `<ref>` wewnątrz elementów `<list>` i `<set>` w celu umieszczenia odwołań do komponentu we właściwościach lub argumentach konstruktora typu odpowiednio `List` i `Set`.

W listingu 3.30 pokazujemy, w jaki sposób odwołania do komponentów mogą być dodawane do argumentu konstruktora typu `List`.

Listing 3.30. Przykład konfiguracji, w której element `<list>` zawiera odwołania do komponentów

```
<bean .....>
  <constructor-arg name="myList">
    <list>
      <ref bean="aBean" />
      <ref bean="bBean" />
    </list>
  </constructor-arg>
</bean>
```

```
<bean id="aBean" class="somepackage.ABean" />
<bean id="bBean" class="somepackage.BBean" />
```

W listingu 3.30 zostało pokazane, że argument konstruktora `myList` jest typu `List` i zawiera dwa elementy: odwołanie do komponentu `aBean` i odwołanie do komponentu `bBean`. Atrybut `bean` elementu `<ref>` określa nazwę komponentu, do którego się odwołuje element `<ref>`.

Podobnie jak w przypadku elementu `<list>`, istnieje możliwość użycia elementów `<ref>` wewnątrz `<set>` w celu umieszczenia w argumencie konstruktora typu `Set` lub właściwości komponentu odwołania do komponentu. W przypadku elementu `<map>` można użyć elementu `<ref>` wewnątrz `<key>` w celu zdefiniowania klucza w postaci odwołania do komponentu oraz elementu `<ref>` do zdefiniowania odwołania do obiektu jako wartości dla wspomnianego klucza. W listingu 3.31 pokazano argument konstruktora typu `Map` zawierający pojedynczą parę klucz – wartość, w której zarówno klucz, jak i wartość są odwołaniami do komponentów.

Listing 3.31. Przykład konfiguracji, w której element `<map>` zawiera odwołania komponentów podane jako klucze i wartości

```
<bean ....>
  <constructor-arg name="myMapWithBeanRef">
    <map>
      <entry>
        <key>
          <ref bean="aBean" />
        </key>
        <ref bean="bBean" />
      </entry>
    </map>
  </constructor-arg>
</bean>

<bean id="aBean" class="somepackage.ABean" />
<bean id="bBean" class="somepackage.BBean" />
```

W powyższym listingu został przedstawiony argument konstruktora `myMapWithBeanRef` typu `Map` zawierający parę klucz – wartość, w której klucz jest odwołaniem do komponentu `aBean`, natomiast wartością tego klucza jest odwołanie do komponentu `bBean`.

Dodanie do kolekcji nazw komponentów

Jeżeli nazwę komponentu (w postaci określonej przez atrybut `id` elementu `<bean>`) chcesz dodać do argumentu konstruktora lub właściwości komponentu typu `List`, `Map` lub `Set`, możesz użyć elementu `<idref>` wewnątrz elementu `<map>`, `<set>` i `<list>`. Listing 3.32 przedstawia argument konstruktora typu `Map` zawierający pojedynczą parę klucz – wartość, w której kluczem jest nazwa komponentu, natomiast wartością tego klucza jest odwołanie do komponentu.

Listing 3.32. Przykład konfiguracji, w której element `<map>` zawiera klucz w postaci nazwy komponentu i wartość w postaci odwołania do komponentu

```
<constructor-arg name="myExample">
  <map>
    <entry>
      <key>
```

```

        <idref bean="sampleBean" />
      </key>
      <ref bean="sampleBean" />
    </entry>
  </map>
</constructor-arg>

<bean id="sampleBean" class="somepackage.SampleBean" />

```

W listingu 3.32 argument konstruktora `myExample` jest typu `Map`. Kluczem jest ciąg tekstowy `sample` ➔ `Bean`, natomiast wartością jest komponent `sampleBean`. Wprawdzie można użyć elementu `<value>` w celu określenia ciągu tekstowego `sampleBean` jako klucza, ale wykorzystano element `<idref>`, ponieważ kontener Springa sprawdza istnienie komponentu `sampleBean` w trakcie wdrażania aplikacji.



Istnieje możliwość użycia elementu `<idref>` wewnątrz `<property>` lub `<constructor-arg>` w celu zdefiniowania nazwy komponentu jako wartości właściwości komponentu lub argumentu konstruktora.

Dodanie wartości null do kolekcji

Wartość `null` można dodać do kolekcji typu `Set` i `List`, używając do tego elementu `<null>`. W listingu 3.33 pokazujemy, jak dodać wartość `null` do argumentu konstruktora typu `Set`, posługując się w tym celu elementem `<null>`.

Listing 3.33. Przykład konfiguracji, w której element `<set>` zawiera element `<null>`

```

<constructor-arg name="setWithNullElement">
  <set>
    <value>Element 1</value>
    <value>Element 2</value>
    <null />
  </set>
</constructor-arg>

```

W listingu 3.33 argument konstruktora `setWithNullElement` zawiera trzy elementy: `Element 1`, `Element 2` i `null`.

Aby dodać do właściwości lub argumentu konstruktora typu `Map` klucz w postaci `null`, można użyć elementu `<null>` umieszczonego wewnątrz elementu `<key>`. Podobnie w celu dodania wartości `null` można wykorzystać element `<null>` wewnątrz elementu `<entry>`. W listingu 3.34 został pokazany argument konstruktora typu `Map` zawierający klucz i wartość w postaci `null`.

Listing 3.34. Przykład konfiguracji, w której element `<map>` zawiera klucz i wartość mające postać `null`

```

<constructor-arg name="mapType">
  <map>
    <entry>
      <key>
        <null />
      </key>
      <null />
    </entry>
  </map>
</constructor-arg>

```

```

    </entry>
  </map>
</constructor-arg>

```

W listingu 3.34 mamy przykład użycia elementu `<null>` w celu dodania do argumentu konstruktora `mapType` elementu, którego klucz i wartość to `null`.



Istnieje możliwość użycia elementu `<null>` wewnątrz `<property>` lub `<constructor-arg>` w celu zdefiniowania wartości `null` odpowiednio dla właściwości lub argumentów konstruktora.

Zobaczmy teraz, jak zdefiniować wartości dla właściwości i argumentów konstruktora będących tablicami.

Definiowanie wartości dla tablic

Jeżeli klasa komponentu definiuje właściwość w postaci tablicy, jej wartość można określić za pomocą elementu `<array>` umieszczonego w elemencie `<property>`. Podobnie wartość argumentu konstruktora w postaci tablicy można zdefiniować za pomocą elementu `<array>` umieszczonego w elemencie `<constructor-arg>`.

W listingu 3.35 pokazujemy, jak ustawić wartość właściwości komponentu typu `int[]`.

Listing 3.35. Ustawianie wartości właściwości komponentu typu `int[]`

```

<property name="numbersProperty">
  <array>
    <value>1</value>
    <value>2</value>
  </array>
</property>

```

W listingu 3.35 każdy element `<value>` znajdujący się w `<array>` przedstawia element tablicy `numbersProperty`. Edytory właściwości zarejestrowane przez kontener Springa są używane w celu przeprowadzenia konwersji ciągu tekstowego określanego przez element `<value>` na wartość typu `int`. Istnieje możliwość użycia elementu `<array>` w `<list>`, `<set>` i `<map>`. Ponadto można użyć elementy `<list>`, `<set>`, `<map>`, `<props>` i `<ref>` wewnątrz `<array>` i tym samym tworzyć tablice wartości typu odpowiednio `List`, `Set`, `Map`, `Properties` i odwołań do komponentów. Jeżeli chcesz utworzyć tablicę przechowującą inne tablice, w elemencie `<array>` umieść inne elementy `<array>`.

Dowiedziałeś się, że elementy `<list>`, `<map>` i `<set>` są używane do ustawiania właściwości i argumentów konstruktora typu odpowiednio `List`, `Map` i `Set`. Przechodzimy teraz do analizy domyślnych implementacji kolekcji tworzonych przez Spring dla każdego z wymienionych elementów.

Domyślne implementacje kolekcji dla elementów `<list>`, `<set>` i `<map>`

W tabeli 3.2 zostały wymienione domyślne implementacje kolekcji tworzone przez Spring dla elementów `<list>`, `<set>` i `<map>`.

Tabela 3.2. Domyślne implementacje kolekcji dla elementów `<list>`, `<set>` i `<map>`

Element kolekcji	Domyślna implementacja kolekcji utworzonej przez Spring
<code><list></code>	<code>java.util.ArrayList</code>
<code><set></code>	<code>java.util.LinkedHashSet</code>
<code><map></code>	<code>java.util.LinkedHashMap</code>

Na podstawie tabeli 3.2 można wyciągnąć następujące wnioski:

- Jeżeli wartość właściwości (lub argumentu konstruktora) została podana za pomocą elementu `<list>`, wówczas Spring tworzy egzemplarz `ArrayList` i przypisuje go właściwości (bądź argumentowi konstruktora).
- Jeżeli wartość właściwości (lub argumentu konstruktora) została podana za pomocą elementu `<set>`, wówczas Spring tworzy egzemplarz `LinkedHashSet` i przypisuje go właściwości (bądź argumentowi konstruktora).
- Jeżeli wartość właściwości (lub argumentu konstruktora) została podana za pomocą elementu `<map>`, wówczas Spring tworzy egzemplarz `LinkedHashMap` i przypisuje go właściwości (bądź argumentowi konstruktora).

Istnieje całkiem duże prawdopodobieństwo, że zechcesz użyć innej implementacji `List`, `Set` lub `Map` dla właściwości komponentu lub argumentu konstruktora. Na przykład zamiast `java.util.ArrayList` możesz właściwości komponentu typu `List` przypisać egzemplarz `java.util.LinkedList`. W takich przypadkach zaleca się użycie elementów `<list>`, `<map>` i `<set>` schematu `utils` oferowanego przez Spring (dokładne wyjaśnienie znajdziesz w podrozdziale 3.8). Elementy `<list>`, `<set>` i `<map>` schematu `utils` pozwalają na określenie w pełni kwalifikowanej nazwy konkretnej klasy kolekcji, która ma zostać przypisana właściwości lub argumentowi konstruktora komponentu.

Teraz przejdziemy do przeanalizowania niektórych z wbudowanych w Spring edytorów właściwości.

3.5. Wbudowane edytory właściwości

Spring oferuje kilka wbudowanych edytorów właściwości, które są użyteczne podczas definiowania właściwości komponentu lub argumentów konstruktora. Teraz pokrótce zapoznamy się z edytorami `CustomCollectionEditor`, `CustomMapEditor` i `CustomDateEditor`. Pełną listę wbudowanych edytorów właściwości znajdziesz w pakiecie `org.springframework.beans.propertyeditors`.

Edytor `CustomCollectionEditor`

Edytor właściwości `CustomCollectionEditor` jest odpowiedzialny za konwersję kolekcji źródłowej (takiej jak `java.util.LinkedList`) na inną (np. `java.util.ArrayList`). Domyślnie edytor `CustomCollectionEditor` został zarejestrowany dla typów `Set`, `SortedSet` i `List`.

Rozważ przedstawioną w listingu 3.36 klasę `CollectionTypesExample` definiującą atrybuty (i odpowiadające im metody typu `setter`) typów `Set` i `List`.

Listing 3.36. *Klasa CollectionTypesExample*Projekt: *ch03-simple-types-examples*Położenie pliku: *src/main/java/sample/spring/chapter03/beans*

```
package sample.spring.chapter03.beans;

import java.util.List;
import java.util.Set;

public class CollectionTypesExample {
    private Set setType;
    private List listType;
    ....
    // Metody setter dla atrybutów.
    public void setSetType(Set setType) {
        this.setType = setType;
    }
    ....
}
```

Klasa `CollectionTypesExample` definiuje atrybuty `setType` i `listType` typu odpowiednio `Set` i `List`. Listing 3.37 przedstawia definicję komponentu dla klasy `CollectionTypesExample`.

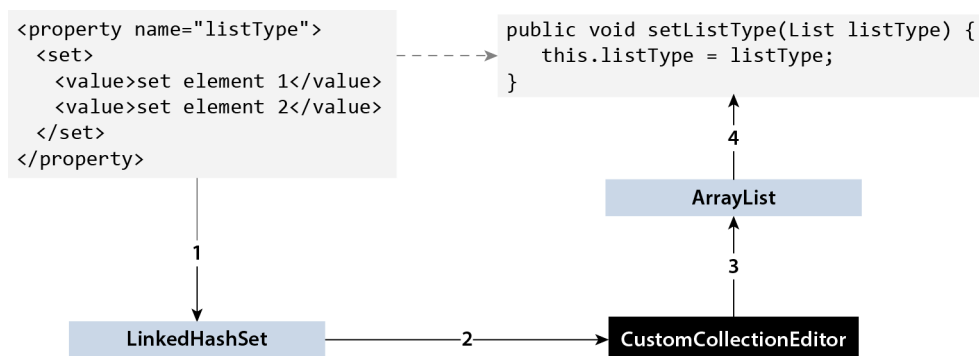
Listing 3.37. *Plik applicationContext.xml przedstawiający definicję komponentu dla klasy CollectionTypesExample*Projekt: *ch03-simple-types-examples*Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean class="sample.spring.chapter03.beans.CollectionTypesExample">
  <property name="listType">
    <set>
      <value>element set 1</value>
      <value>element set 2</value>
    </set>
  </property>
  <property name="setType">
    <list>
      <value>element list 1</value>
      <value>element list 2</value>
    </list>
  </property>
  ....
</bean>
```

Być może uważasz, że konfiguracja przedstawiona w listingu 3.37 jest nieprawidłowa, ponieważ element `<set>` został użyty do ustawienia wartości właściwości `listType` (typu `List`), natomiast element `<list>` został użyty do ustawienia wartości właściwości `setType` (typu `Set`).

Jednak powyższa konfiguracja jest całkowicie poprawna, a kontener Springa nie zgłasza *żadnych* zastrzeżeń. To wynika z faktu, że edytor `CustomCollectionEditor` konwertuje egzemplarz `ArrayList` (utworzony dla odpowiadającego mu elementu typu `<list>`) na typ `LinkedHashSet` (implementacja typu `Set`), *zanim* ustawi wartość właściwości `setType`. Ponadto edytor `CustomCollectionEditor` konwertuje egzemplarz `LinkedHashSet` (utworzony dla odpowiadającego mu elementu typu `<set>`) na typ `ArrayList` (implementacja typu `List`) *przed* ustawieniem wartości właściwości `listType`.

Na rysunku 3.4 widać, że edytor `CustomCollectionEditor` konwertuje elementy typu `LinkedHashSet` na `ArrayList` w celu ustawienia wartości właściwości `listType` klasy `CollectionTypesExample`. Na rysunku możesz zobaczyć sekwencję kroków wykonywanych przez Spring w trakcie operacji ustawiania wartości właściwości `listType`. Na początku Spring tworzy egzemplarz `LinkedHashSet` odpowiadający elementowi `<set>`. Ponieważ właściwość `listType` jest typu `List` (patrz: listing 3.36), podczas operacji ustawienia wartości dla wymienionej właściwości do gry wchodzi edytor `CustomCollectionEditor`. Jeżeli typem właściwości komponentu będzie `List`, to edytor `CustomCollectionEditor` utworzy egzemplarz `ArrayList` i wypełni go elementami `LinkedHashSet`. Na końcu wartością zmiennej `listType` będzie implementacja `ArrayList` utworzona przez `CustomCollectionEditor`.

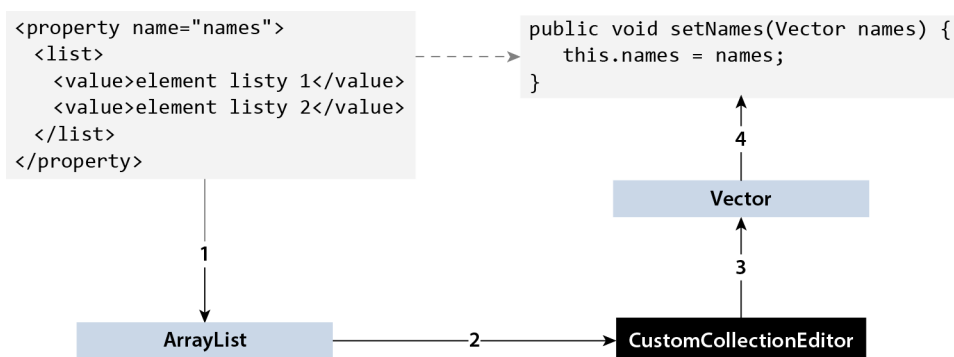


Rysunek 3.4. Edytor `CustomCollectionEditor` konwertuje elementy typu `LinkedHashSet` na `ArrayList`

Trzeba koniecznie zwrócić uwagę na jeden fakt. Jeżeli typem właściwości lub argumentu konstruktora będzie konkretna klasa kolekcji (taka jak `LinkedList`), wówczas edytor `CustomCollectionEditor` tworzy egzemplarz wspomnianej klasy i dodaje do niego elementy z kolekcji źródłowej. Na rysunku 3.5 została pokazana sytuacja, w której właściwość komponentu jest typu `java.util.Vector` (konkretna klasa kolekcji).

Z rysunku 3.5 wynika, że edytor `CustomCollectionEditor` tworzy egzemplarz `Vector` (konkretniej klasy kolekcji) i dodaje do niego elementy pochodzące z kolekcji źródłowej, tutaj jest to `ArrayList`.

Teraz przechodzimy do edytora właściwości `CustomMapEditor`.

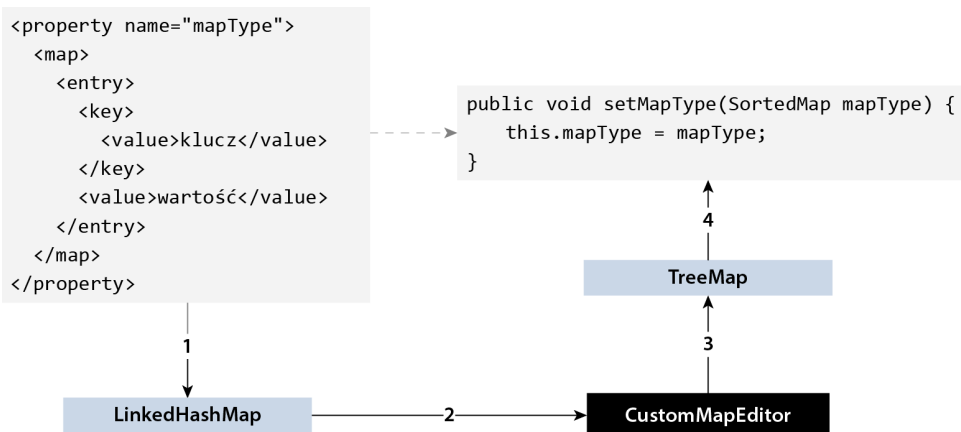


Rysunek 3.5. Edytor `CustomCollectionEditor` konwertuje element typu `ArrayList` na typ `Vector`

Edytor CustomMapEditor

Edytor właściwości CustomMapEditor zajmuje się konwersją źródłowego typu Map (np. HashMap) na docelowy typ Map (np. TreeMap). Domyślnie edytor CustomMapEditor jest zarejestrowany tylko dla typu SortedMap.

Na rysunku 3.6 pokazano sytuację, w której edytor CustomMapEditor konwertuje LinkedHashMap (źródłowy typ Map) na TreeMap (implementacja typu SortedMap).



Rysunek 3.6. Edytor CustomMapEditor konwertuje LinkedHashMap (źródłowy typ Map) na TreeMap (docelowy typ Map)

Na rysunku 3.6 pokazujemy sekwencję kroków wykonywanych przez Spring w celu ustawienia wartości właściwości `mapType`. Na początku Spring tworzy egzemplarz `LinkedHashMap` odpowiadający elementowi `<map>`. Ponieważ właściwość `mapType` jest typu `SortedMap`, podczas operacji ustawienia wartości dla wymienionej właściwości do gry wchodzi edytor `CustomMapEditor`. Edytor `CustomMapEditor` tworzy egzemplarz `TreeMap` (konkretna implementacja interfejsu `SortedSet`), dodaje pary klucz – wartość z `LinkedHashMap` do nowo utworzonego egzemplarza `TreeMap`, a następnie przypisuje go właściwości `mapType`.

Edytor CustomDateEditor

`CustomDateEditor` to edytor właściwości dla właściwości komponentu i argumentów konstruktora typu `java.util.Date`. Obsługuje on typ `java.text.DateFormat` używany do formatowania ciągu tekstowego daty i godziny na postać obiektu typu `java.util.Date` oraz do przetwarzania obiektów typu `java.util.Date` na postać ciągu tekstowego daty i godziny. W kolejnym podrozdziale zobaczysz, jak edytora `CustomDateEditor` można użyć w celu ustawienia właściwości komponentu i argumentów konstruktora typu `java.util.Date`. W projekcie *ch03-simple-types-examples* edytor `CustomDateEditor` został wykorzystany do przeprowadzenia konwersji ciągu tekstowego właściwości komponentu (patrz: atrybut `dateOfInception` klasy `BankDetails`) lub argumentu konstruktora (patrz: argument konstruktora `dateType` klasy `DataTypesExample`) na typ `java.util.Date`.

W projekcie *ch03-simple-types-examples* zostały wykorzystane także inne wbudowane edytory właściwości, między innymi `ByteArrayPropertyEditor` do konwersji ciągu tekstowego na typ `byte[]` (patrz: atrybut `bankPrimaryBusiness` klasy `BankDetails`), `CurrencyEditor` do konwersji kodu waluty na obiekt typu `java.util.Currency` (patrz: atrybut `primaryCurrency` klasy `BankDetails`), a także `CharacterArrayPropertyEditor` do konwersji ciągu tekstowego na typ `char[]` (patrz: atrybut `head` ↪ `OfficeAddress` klasy `BankDetails`) itd.

Przechodzimy teraz do tematu rejestracji edytorów właściwości w kontenerze Springa.

3.6. Rejestracja edytorów właściwości w kontenerze Springa

Klasa `BeanWrapperImpl` rejestruje w kontenerze Springa kilka wbudowanych edytorów właściwości. Na przykład edytory `CustomCollectionEditor`, `CustomMapEditor`, `CurrencyEditor`, `ByteArray` ↪ `PropertyEditor` i `CharacterArrayEditor` są domyślnie rejestrowane w kontenerze Springa, ale edytor `CustomDateEditor` już *nie*. W celu zarejestrowania w kontenerze Springa edytora właściwości należy użyć komponentu *specjalnego* o nazwie `CustomEditorConfigurer`. Wymieniona klasa `CustomEditorConfigurer` implementuje oferowany przez Spring interfejs `BeanFactoryPostProcessor` (jego szczegółowe omówienie znajdziesz w podrozdziale 5.4) oraz jest automatycznie wykrywana i wykonywana przez kontener Springa.

W projekcie *ch03-simple-types-examples* klasa `BankDetails` (patrz: listing 3.24) definiuje właściwość `dateOfInception` typu `java.util.Date`. Wartością podaną dla właściwości `dateOfInception` jest 30-01-2012 (patrz: listing 3.25). W celu przeprowadzenia konwersji ciągu tekstowego 30-01-2012 na typ `java.util.Date` konieczne jest zarejestrowanie własnego edytora właściwości dla typu `java`. ↪ `util.Date` lub też wykorzystanie wbudowanego w kontener Springa edytora `CustomDateEditor`.

Rejestracja edytora właściwości w kontenerze Springa wymaga wykonania następujących kroków:

1. Utwórz klasę implementującą interfejs `PropertyEditorRegistrar`. Ta klasa będzie odpowiedzialna za rejestrację edytorów właściwości w kontenerze Springa.
2. W pliku XML konspektu aplikacji skonfiguruj implementację `PropertyEditorRegistrar` jako komponent.
3. W pliku XML konspektu aplikacji skonfiguruj komponent *specjalny* o nazwie `CustomEditor` ↪ `Configurer` i podaj mu odwołanie do implementacji `PropertyEditorRegistrar` utworzonej w kroku 1. i skonfigurowanej w kroku 2.

Przeanalizujemy teraz, jak w projekcie *ch03-simple-types-examples* została przeprowadzona rejestracja edytora `CustomDateEditor` w kontenerze Springa.

Utworzenie implementacji `PropertyEditorRegistrar`

W listingu 3.38 została przedstawiona klasa `MyPropertyEditorRegistrar`, która implementuje interfejs `PropertyEditorRegistrar`.

Listing 3.38. Klasa *MyPropertyEditorRegistrar**Projekt: ch03-simple-types-examples**Położenie pliku: src/main/java/sample/spring/chapter03/beans*

```
package sample.spring.chapter03.beans;

import java.text.SimpleDateFormat;
import java.util.Date;

import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;
import org.springframework.beans.propertyeditors.CustomDateEditor;

public class MyPropertyEditorRegistrar implements PropertyEditorRegistrar {

    @Override
    public void registerCustomEditors(PropertyEditorRegistry registry) {
        registry.registerCustomEditor(Date.class, new CustomDateEditor(
            new SimpleDateFormat("dd-MM-yyyy"), false));
    }
}
```

W listingu 3.38 pokazano, że klasa *MyPropertyEditorRegistrar* implementuje dostarczany przez Spring interfejs *PropertyEditorRegistrar*, a także zapewnia implementację metody *registerCustomEditors()* zdefiniowanej w interfejsie *PropertyEditorRegistrar*. Egzemplarz *PropertyEditorRegistry* przekazywany metodzie *registerCustomEditors()* jest używany do rejestracji edytorów właściwości. Metoda *registerCustomEditor()* klasy *PropertyEditorRegistry* jest odpowiedzialna za rejestrację w kontenerze Springa implementacji *PropertyEditor*. W omawianym listingu metoda *registerCustomEditor()* klasy *PropertyEditorRegistry* rejestruje w kontenerze Springa edytor *CustomDateEditor*.

Konfiguracja klasy *CustomEditorConfigurer*

W listingu 3.39 pokazujemy, jak klasa *CustomEditorConfigurer* została skonfigurowana w pliku XML kontekstu aplikacji.

Listing 3.39. Plik *applicationContext.xml* przedstawiający konfigurację klasy *CustomEditorConfigurer**Projekt: ch03-simple-types-examples**Położenie pliku: src/main/resources/META-INF/spring*

```
<bean id="myPropertyEditorRegistrar"
      class="sample.spring.chapter03.beans.MyPropertyEditorRegistrar" />

<bean id="editorConfigurer"
      class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="propertyEditorRegistrars">
        <list>
            <ref bean="myPropertyEditorRegistrar" />
        </list>
    </property>
</bean>
```

W powyższym listingu definicja komponentu `myPropertyEditorRegistrar` konfiguruje klasę `MyPropertyEditorRegistrar` jako komponent Springa. Klasa `MyPropertyEditorRegistrar` implementuje interfejs `PropertyEditorRegistrar` i jest odpowiedzialna za rejestrację w kontenerze Springa dodatkowych edytorów właściwości. Klasa `CustomEditorConfigurer` ma właściwość `propertyEditorRegistrars`, która wskazuje listę implementacji `PropertyEditorRegistrar`. W omawianym listingu komponent `myPropertyEditorRegistrar` został podany jako jedna z wartości właściwości `propertyEditorRegistrars`. Komponent `CustomEditorConfigurer` jest automatycznie wykrywany i wykonywany przez kontener Springa, a skutkiem będzie rejestracja edytorów właściwości przez egzemplarz `MyPropertyEditorRegistrar`.

W kolejnym podrozdziale zobaczysz, jak wykorzystać przestrzeń nazw `p` (dla właściwości komponentu) i `c` (dla argumentów konstruktora), aby móc stosować zwięźlejsze definicje komponentów w pliku XML kontekstu aplikacji.

3.7. Zwięzłe definicje komponentów dzięki użyciu przestrzeni nazw `p` oraz `c`

Aby definicje komponentów w pliku XML kontekstu aplikacji mogły być mniej rozwlekłe, Spring oferuje przestrzenie nazw `p` oraz `c` pozwalające na określanie wartości odpowiednio dla właściwości komponentu i argumentów konstruktora. Wymienione przestrzenie nazw stanowią alternatywę dla użycia elementów odpowiednio `<property>` i `<constructor-arg>`.

W pierwszej kolejności przeanalizujemy przestrzeń nazw `p`.

IMPORT `Rozdział03/ch03-namespaces-example`. (Ten projekt pokazuje aplikację Springa, w której właściwości komponentu i argumenty konstruktora są definiowane za pomocą przestrzeni nazw odpowiednio `p` oraz `c`. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `SampleApp` projektu).

Przestrzeń nazw `p`

Aby użyć przestrzeni nazw `p` do ustawienia właściwości komponentu, musisz je najpierw zdefiniować jako atrybuty elementu `<bean>`, a następnie określić każdą właściwość jako znajdującą się w przestrzeni nazw `p`.

W przedstawionej w listingu 3.40 definicji komponentu pokazujemy, jak używać przestrzeni nazw `p` do ustawienia właściwości komponentu.

Listing 3.40. Plik `applicationContext.xml` przedstawiający przykład użycia przestrzeni nazw `p`

Projekt: `ch03-namespaces-example`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:p="http://www.springframework.org/schema/p" xsi:schemaLocation="....">

  <bean id="bankDetails" class="sample.spring.chapter03.beans.BankDetails"
    p:bankName="My Personal Bank" p:bankPrimaryBusiness="Retail banking"
```

```

    p:headOfficeAddress="Address of head office" p:privateBank="Y"
    p:primaryCurrency="INR" p:dateOfInception="30-01-2012"
    p:branchAddresses-ref="branchAddresses"/>
    .....
</beans>

```

W przedstawionym w listingu pliku XML kontekstu aplikacji przestrzeń nazw `p` została określona za pomocą atrybutu `xmlns`. Podczas określania właściwości komponentu definicja komponentu `bankDetails` używa prefiksu `p` dla wymienionej przestrzeni nazw. Jeżeli omawiany kod porównasz z przedstawionym w listingu 3.25, to zauważysz, że listing 3.40 jest *znacznie* zwięźlejszy. Wprawdzie można użyć połączenia elementów `<property>` i przestrzeni nazw `p` do określenia właściwości komponentu, ale zaleca się wybór tylko jednego stylu definiowania właściwości komponentu i jego spójne stosowanie we wszystkich definicjach komponentów.



Ponieważ przestrzeń nazw `p` jest implementowana jako część Springa, *nie* istnieje odpowiadający jej schemat. Dlatego też w listingu 3.40 nie znajdziesz żadnego odwołania do schematu dla przestrzeni nazw `p`. Jeżeli chcesz, aby środowisko IDE automatycznie uzupełniało nazwy właściwości komponentu, gdy stosowana jest przestrzeń nazw `p`, rozważ użycie produktu IntelliJ IDEA lub SpringSource Tool Suite (STS).

Jeżeli właściwość komponentu *nie* jest odwołaniem do innego komponentu, wówczas będzie określona z użyciem następującej składni:

```
p:<nazwa-właściwości>=<wartość-właściwości>
```

W powyższej składni `<nazwa-właściwości>` oznacza nazwę właściwości komponentu, natomiast `<wartość-właściwości>` jej wartość.

Jeżeli zaś właściwość komponentu jest odwołaniem do innego komponentu, wówczas będzie określona z użyciem następującej składni:

```
p:<nazwa-właściwości>-ref=<odwołanie-do-komponentu>
```

W powyższej składni `<nazwa-właściwości>` oznacza nazwę właściwości komponentu, natomiast `<odwołanie-do-komponentu>` jest identyfikatorem komponentu, do którego następuje odwołanie. Trzeba koniecznie zwrócić uwagę na fakt, że nazwa właściwości ma przyrostek `-ref`. Ponieważ właściwość `branchAddresses` komponentu `BankDetails` przedstawia odwołanie do komponentu `branchAddresses`, w listingu 3.40 właściwość `branchAddresses` została określona w postaci `p:branchAddresses-ref`.

Przechodzimy teraz do przeanalizowania sposobu, w jaki przestrzeń nazw `c` jest używana do określania wartości argumentów konstruktora.

Przestrzeń nazw `c`

Aby użyć przestrzeni nazw `c` do ustawienia wartości argumentów konstruktora, musisz je najpierw zdefiniować jako atrybuty elementu `<bean>`, a następnie określić każdy argument konstruktora jako znajdujący się w przestrzeni nazw `c`.

W listingu 3.41 została przedstawiona klasa `BankStatement`, którą za pomocą przestrzeni nazw `c` skonfigurujemy jako komponent Springa.

Listing 3.41. Klasa `BankStatement`

Projekt: `ch03-namespaces-example`

Położenie pliku: `src/main/java/sample/spring/chapter03/beans`

```
package sample.spring.chapter03.beans;

import java.beans.ConstructorProperties;

public class BankStatement {
    ....
    @ConstructorProperties({ "transactionDate", "amount", "transactionType",
        "referenceNumber" })
    public BankStatement(Date transactionDate, double amount,
        String transactionType, String referenceNumber) {
        this.transactionDate = transactionDate;
        this.amount = amount;
        ....
    }
    ....
}
```

W listingu 3.42 została przedstawiona definicja dla klasy `BankStatement` pokazująca użycie przestrzeni nazw `c` w celu ustawienia wartości dla argumentów konstruktora.

Listing 3.42. Plik `applicationContext.xml` przedstawiający przykład użycia przestrzeni nazw `c`

Projekt: `ch03-namespaces-example`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="....">
    ....
    <bean id="bankStatement" class="sample.spring.chapter03.beans.BankStatement"
        c:transactionDate = "30-01-2012"
        c:amount = "1000"
        c:transactionType = "Credit"
        c:referenceNumber = "1110202" />
    ....
</beans>
```

W przedstawionym w listingu pliku XML kontekstu aplikacji przestrzeń nazw `c` została określona za pomocą atrybutu `xmlns`. Definicja komponentu `bankStatement` używa prefiksu `c` dla wymienionej przestrzeni nazw w celu określenia argumentów konstruktora. Składnia stosowana podczas określania argumentów konstruktora z użyciem przestrzeni nazw `c` jest podobna do składni stosowanej w przestrzeni nazw `p`.

Jeżeli argument konstruktora *nie* jest odwołaniem do innego komponentu, wówczas będzie określony z użyciem następującej składni:

```
p:<nazwa-argumentu-konstruktora>=<wartość-argumentu-konstruktora>
```



Ponieważ przestrzeń nazw `c` jest implementowana jako część Springa, *nie* istnieje odpowiadający jej schemat. Dlatego też w listingu 3.42 nie znajdziesz żadnego odwołania do schematu dla przestrzeni nazw `c`. Jeżeli chcesz, aby środowisko IDE automatycznie uzupełniało nazwy właściwości komponentu, gdy stosowana jest przestrzeń nazw `p`, rozważ użycie produktu IntelliJ IDEA lub SpringSource Tool Suite (STS).

W powyższej składni `<nazwa-argumentu-konstruktor>` oznacza nazwę argumentu konstruktora, natomiast `<wartość-argumentu-konstruktor>` jego wartość.

Jeżeli zaś argument konstruktora jest odwołaniem do innego komponentu, wówczas będzie określony z użyciem następującej składni:

```
p:<nazwa-argumentu-konstruktor>-ref="<odwołanie-do-komponentu>"
```

W powyższej składni `<nazwa-argumentu-konstruktor>` oznacza nazwę argumentu konstruktora, natomiast `<odwołanie-do-komponentu>` jest identyfikatorem komponentu, do którego następuje odwołanie. Trzeba koniecznie zwrócić uwagę na fakt, że nazwa argumentu konstruktora ma przyrostek `-ref`. Na przykład jeśli argument konstruktora o nazwie `myargument` przedstawia odwołanie do komponentu o identyfikatorze `x`, wówczas argument konstruktora podajemy w następującej postaci:

```
c:myargument-ref = "x"
```

Jak wcześniej wspomnieliśmy, jeżeli klasa jest kompilowana wraz z ustawioną opcją debug, wtedy nazwy argumentów konstruktora zostaną zachowane w wygenerowanym pliku `.class`. Jeżeli klasa `BankStatement` *nie* będzie skompilowana wraz z ustawioną opcją debug, to konfiguracja przedstawiona w listingu 3.42 nie działa. W takich sytuacjach wartości dla argumentów konstruktora należy podawać z użyciem ich indeksu, jak pokazano w listingu 3.43.

Listing 3.43. Podanie wartości argumentów konstruktora za pomocą ich indeksu

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:c="http://www.springframework.org/schema/c"
  xsi:schemaLocation="....">
  ....
  <bean id="bankStatement" class="sample.spring.chapter03.beans.BankStatement"
    c:_0 = "30-01-2012"
    c:_1 = "1000"
    c:_2 = "Credit"
    c:_3 = "1110202" />
  ....
</beans>
```

W powyższym listingu przedstawiamy definicję komponentu dla klasy `BankStatement`, która w celu dostarczenia wartości używa indeksu argumentu konstruktora zamiast nazwy argumentu konstruktora. Trzeba koniecznie zwrócić uwagę na fakt, że indeks argumentu konstruktora jest poprzedzony ukośnikiem, ponieważ nazwy atrybutów w XML *nie* mogą rozpoczynać się od cyfry. Jeżeli argument konstruktora będzie odwołaniem do innego komponentu, do indeksu *trzeba* dodać przyrostek `-ref`. Na przykład jeśli argument konstruktora o indeksie `0` przedstawia odwołanie do

innego komponentu, powinien być podany jako `c:_0-ref`. Wprawdzie można użyć połączenia elementów `<constructor-arg>` i przestrzeni nazw `c` do określenia argumentów konstruktora, ale zaleca się wybór tylko jednego stylu definiowania argumentów konstruktora i jego spójne stosowanie we wszystkich definicjach komponentów.

We wcześniejszej części rozdziału zobaczyłeś, jak elementy `<list>`, `<map>` i `<set>` są używane do ustawiania właściwości lub argumentów konstruktora typu odpowiednio `List`, `Map` i `Set`. Przejdziemy teraz do sprawdzenia, jak oferowany przez Spring schemat `util` ułatwia tworzenie typów kolekcji, typów `Properties`, stałych itd. oraz ułatwia udostępnienie ich w postaci komponentów Springa.

3.8. Schemat util

Oferowany przez Spring schemat `util` ułatwia zadanie konfiguracji komponentów, ponieważ zapewnia spójny sposób przeprowadzania najczęstszych operacji konfiguracyjnych. W tabeli 3.3 wymieniono różne elementy schematu `util`.

Tabela 3.3. Różne elementy schematu `util`

Element	Opis
<code><list></code>	Tworzy typ <code>java.util.List</code> i udostępnia go w postaci komponentu.
<code><map></code>	Tworzy typ <code>java.util.Map</code> i udostępnia go w postaci komponentu.
<code><set></code>	Tworzy typ <code>java.util.Set</code> i udostępnia go w postaci komponentu.
<code><constant></code>	Właściwość <code>public static</code> obiektu jest udostępniana w postaci komponentu.
<code><property-path></code>	Właściwość komponentu jest udostępniana w postaci komponentu.
<code><properties></code>	Tworzy typ <code>java.util.Properties</code> na podstawie pliku właściwości i udostępnia go w postaci komponentu.



Wszystkie elementy schematu `util` w Springu akceptują atrybut `scope`, który określa sposób udostępnienia komponentu: o zasięgu `singleton` lub `prototype`.

Spring oferuje interfejs `FactoryBean`, który może być implementowany w celu utworzenia obiektu fabryki odpowiedzialnego za tworzenie egzemplarzy komponentu. Zamiast używać wymienionych w tabeli 3.3 elementów schematu `util` można wykorzystać dostępną standardowo implementację `FactoryBean` i otrzymać tę samą funkcjonalność. W tym podrozdziale przeanalizujemy elementy schematu `util`, a także wbudowane implementacje `FactoryBean`, których można używać zamiast elementów schematu `util`.

IMPORT `Rozdział103/ch03-util-schema-examples`. (Ten projekt pokazuje aplikację Springa, w której wykorzystano elementy schematu `util` w celu utworzenia współdzielonych egzemplarzy `List`, `Set`, `Map` itd. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `SampleApp` projektu).

Zacznijmy od przeanalizowania elementu `<list>`.

Element <list>

Element <list> oferowanego przez Spring schematu util jest używany do tworzenia obiektów typu `java.util.List`, jak przedstawia listing 3.44.

Listing 3.44. Plik `applicationContext.xml` przedstawiający użycie elementu <list> schematu util

Projekt: `ch03-util-schema-examples`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="..... http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-4.0.xsd">

    <bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
        .....
        <constructor-arg name="listType" ref="listType" />
        .....
    </bean>

    <util:list id="listType" list-class="java.util.ArrayList">
        <value>Prosty ciąg tekstowy w elemencie list</value>
        <value>Kolejny prosty ciąg tekstowy w elemencie list</value>
    </util:list>
</beans>
```

Przede wszystkim do kodu trzeba dołączyć schemat util, aby można było uzyskać dostęp do jego elementów. W powyższym listingu element <list> schematu util tworzy egzemplarz `java.util.ArrayList` i udostępnia go w postaci komponentu. Atrybut `id` określa identyfikator komponentu, w postaci którego udostępniono `java.util.ArrayList`. Z kolei atrybut `list-class` wskazuje konkretną implementację `java.util.List` przeznaczoną do utworzenia. Jeżeli nie określisz atrybutu `list-class`, wtedy domyślnie tworzony jest egzemplarz `java.util.ArrayList`. Element <value> schematu beans w Springu jest używany w celu faktycznego podania elementów listy.

Ponieważ element <list> schematu util udostępnia egzemplarz `List` w postaci komponentu, do wymienionego komponentu możesz się odwoływać z poziomu innych komponentów. Na przykład w listingu 3.44 argument konstruktora `listType` (typu `java.util.List`) komponentu `DataTypesExample` określa `listType` jako wartość atrybutu `ref` pozwalającego odwołać się do egzemplarza `List` utworzonego przez element <list> schematu util.

Jeżeli porównasz element <list> schematu util przedstawiony w listingu 3.44 z elementem <list> schematu beans (patrz: listing 3.27), zauważysz, że element <list> schematu util daje większą kontrolę nad tworzoną implementacją `List`. Na przykład jeśli chcesz utworzyć egzemplarz `Vector` zamiast `ArrayList`, wystarczy podać wartość `java.util.Vector` dla atrybutu `list-class`.

Przejdziemy teraz do elementu `ListFactoryBean`, który w schemacie util można wykorzystać zamiast elementu <list>.

ListFactoryBean

Alternatywą dla użycia elementu `<list>` schematu `util` jest oferowany przez Spring element `ListFactoryBean`, czyli fabryka przeznaczona do tworzenia egzemplarzy typu `java.util.List` i udostępniania ich w postaci komponentu.

W listingu 3.45 pokazujemy, jak można użyć elementu `ListFactoryBean` zamiast elementu `<list>` schematu `util`.

Listing 3.45. Przykład użycia elementu `ListFactoryBean`

```
<beans ....>

  <bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
    ....
    <constructor-arg name="listType" ref="listType" />
    ....
  </bean>

  <bean id="listType" class="
    org.springframework.beans.factory.config.ListFactoryBean">
    <property name="sourceList">
      <list>
        <value>Prosty ciąg tekstowy w elemencie list</value>
        <value>Kolejny prosty ciąg tekstowy w elemencie list</value>
      </list>
    </property>
  </bean>
</beans>
```

W powyższym listingu właściwość `sourceList` elementu `ListFactoryBean` określa elementy listy. Domyślnie `ListFactoryBean` tworzy egzemplarz typu `java.util.ArrayList`. Jeżeli chcesz utworzyć egzemplarz innej implementacji `List` (np. `Vector`), musisz zdefiniować właściwość `targetListClass` elementu `ListFactoryBean`. Właściwość ta podaje w pełni kwalifikowaną nazwę konkretnej implementacji klasy `java.util.List`, która powinna być utworzona przez `ListFactoryBean`.

Jeżeli porównasz kod przedstawiony w listingach 3.44 i 3.45, to zauważysz, że użycie elementu `<list>` schematu `util` jest znacznie prostsze niż użycie `ListFactoryBean` do utworzenia egzemplarza typu `List` i udostępnienia go w postaci komponentu.

Element `<map>`

Element `<map>` schematu `util` oferowanego przez Spring jest używany do tworzenia obiektu typu `java.util.Map` i udostępniania go w postaci komponentu, jak przedstawia listing 3.46.

Listing 3.46. Plik `applicationContext.xml` przedstawiający element `<map>` schematu `util`

Projekt: `ch03-util-schema-examples`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans ....
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="..... http://www.springframework.org/schema/util
  http://www.springframework.org/schema/util/spring-util-4.0.xsd">
```

```
<bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
    ....
    <constructor-arg name="mapType" ref="mapType" />
    ....
</bean>

<util:map id="mapType" map-class="java.util.TreeMap">
    <entry key="map key 1" value="map key 1's value"/>
</util:map>
    ....
</beans>
```

W powyższym listingu element `<map>` schematu `util` tworzy egzemplarz `java.util.TreeMap` i udostępnia go jako komponent. Atrybut `id` określa identyfikator, pod którym ten komponent będzie dostępny dla innych komponentów. Z kolei atrybut `map-class` określa w pełni kwalifikowaną nazwę konkretnej implementacji klasy `java.util.Map`, która powinna być tworzona przez element `<map>`. Element `<entry>` schematu `beans` w Spring określa parę klucz – wartość w utworzonym egzemplarzu `Map`.

Ponieważ element `<map>` udostępnia egzemplarz `Map` jako komponent, do tak zaoferowanego egzemplarza `Map` można się odwoływać z poziomu innych komponentów. Na przykład w omawianym listingu klasa `DataTypesExample` ma argument konstruktora `mapType` (typu `java.util.Map`) określający wartość atrybutu `ref` jako `mapType` w celu odwołania się do egzemplarza `TreeMap` utworzonego przez element `<map>`.



Wcześniej w tym rozdziale zobaczyłeś, że elementy `<key>` i `<value>` znajdujące się w elemencie `<entry>` są używane w celu zdefiniowania par klucz – wartość przechowywanych przez egzemplarz `Map`. Listing 3.46 pokazuje, że pary klucz – wartość zawarte w egzemplarzu `Map` można określić także za pomocą atrybutów `key` i `value` elementu `<entry>`.

Jeżeli porównasz element `<map>` schematu `util` przedstawiony w listingu 3.46 z elementem `<map>` schematu `beans` (patrz: listing 3.27), zauważysz, że element `<map>` schematu `util` daje większą kontrolę nad tworzoną implementacją `Map`. Na przykład jeśli chcesz utworzyć egzemplarz `LinkedHashMap` zamiast `TreeMap`, wystarczy podać wartość `java.util.LinkedHashMap` dla atrybutu `map-class`. W przypadku braku określonego atrybutu `map-class` kontener Springa domyślnie utworzy egzemplarz `java.util.LinkedHashMap`.

Przejdziemy teraz do elementu `MapFactoryBean`, który w schemacie `util` można wykorzystać zamiast elementu `<map>`.

MapFactoryBean

Alternatywą dla użycia elementu `<map>` schematu `util` jest oferowany przez Spring element `MapFactoryBean`, czyli fabryka przeznaczona do tworzenia egzemplarzy typu `java.util.Map` i udostępniania ich w postaci komponentu.

W listingu 3.47 pokazujemy przykład użycia elementu `MapFactoryBean`.

Listing 3.47. *Przykład użycia elementu MapFactoryBean*

```

<beans .....>

    <bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
        .....
        <constructor-arg name="mapType" ref="mapType" />
        .....
    </bean>

    <bean id="mapType" class="org.springframework.beans.factory.config.MapFactoryBean">
        <property name="sourceMap">
            <map>
                <entry key="klucz 1" value="wartość klucza 1"/>
            </map>
        </property>
    </bean>
    .....
</beans>

```

W powyższym listingu właściwość `sourceMap` elementu `MapFactoryBean` określa pary klucz – wartość znajdujące się w egzemplarzu `Map` utworzonym przez `MapFactoryBean`. Domyślnie `MapFactoryBean` tworzy egzemplarz typu `java.util.LinkedHashMap`. Rodzaj utworzonego przez `MapFactoryBean` egzemplarza `Map` można wskazać za pomocą właściwości `targetMapClass`. Właściwość ta podaje w pełni kwalifikowaną nazwę konkretnej implementacji klasy `java.util.Map`. Na przykład jeśli podasz `java.util.HashMap` jako wartość właściwości `targetMapClass`, wówczas `MapFactoryBean` utworzy egzemplarz `java.util.HashMap`.

Jeżeli porównasz kod przedstawiony w listingach 3.46 i 3.47, to zauważysz, że użycie elementu `<map>` schematu `util` jest znacznie prostsze niż użycie `MapFactoryBean` do utworzenia egzemplarza typu `Map` i udostępnienia go w postaci komponentu.

Element <set>

Element `<set>` schematu `util` oferowanego przez Spring jest używany do tworzenia obiektu typu `java.util.Set` i udostępniania go w postaci komponentu, jak przedstawia listing 3.48.

Listing 3.48. *Plik `applicationContext.xml` przedstawiający element <set> schematu util*

Projekt: `ch03-util-schema-examples`

Położenie pliku: `src/main/resources/META-INF/spring`

```

<beans .....
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="..... http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-4.0.xsd">

    <bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
        .....
        <constructor-arg name="setType" ref="setType" />
    </bean>
    <util:set id="setType" set-class="java.util.HashSet">
        <value>Element 1</value>
    </util:set>

```

```

        <value>Element 2</value>
    </util:set>
    .....
</beans>

```

W powyższym listingu element `<set>` schematu `util` tworzy egzemplarz `HashSet` i udostępnia go jako komponent o identyfikatorze `setType`. Atrybut `id` określa identyfikator, pod którym ten komponent będzie dostępny dla innych komponentów. Z kolei atrybut `set-class` określa w pełni kwalifikowaną nazwę konkretnej implementacji klasy `java.util.Set`, która powinna być tworzona przez element `<set>`. Element `<value>` schematu `beans` w Springu określa element w utworzonym egzemplarzu `Set`.

Do egzemplarza `Set` utworzonego przez element `<set>` można się odwoływać z poziomu innych komponentów. Na przykład w omawianym listingu klasa `DataTypesExample` ma argument konstruktora `setType` (typu `java.util.Set`) odwołującego się do egzemplarza `HashSet` utworzonego przez element `<set>`.

Przejdziemy teraz do elementu `SetFactoryBean`, który w schemacie `util` można wykorzystać zamiast elementu `<set>`.

SetFactoryBean

Oferowany przez Spring element `SetFactoryBean` to fabryka przeznaczona do tworzenia egzemplarzy typu `java.util.Set`.

W listingu 3.49 pokazujemy, jak można użyć elementu `SetFactoryBean` zamiast elementu `<set>` schematu `util`.

Listing 3.49. Przykład użycia elementu `SetFactoryBean`

```

<beans .....>

    <bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
        .....
        <constructor-arg name="setType" ref="setType" />
        .....
    </bean>

    <bean id="setType" class="org.springframework.beans.factory.config.SetFactoryBean">
        <property name="sourceSet">
            <set>
                <value>Element 1</value>
                <value>Element 2</value>
            </set>
        </property>
    </bean>
    .....
</beans>

```

W powyższym listingu właściwość `sourceSet` elementu `SetFactoryBean` określa elementy znajdujące się w egzemplarzu `Set` utworzonym przez `SetFactoryBean`. Oferowana przez `SetFactoryBean` właściwość `targetSetClass` pozwala na podanie w pełni kwalifikowanej nazwy klasy implementującej

interfejs `java.util.Set`. W przypadku określenia wymienionej właściwości `SetFactoryBean` utworzy egzemplarz klasy wskazanej przez właściwość `targetSetClass` i udostępni go w postaci komponentu Springa. Na przykład jeśli podasz `java.util.HashSet` jako wartość właściwości `targetSetClass`, wówczas `SetFactoryBean` utworzy egzemplarz `java.util.HashSet`. Jeżeli właściwość `targetSetClass` nie jest zdefiniowana, `SetFactoryBean` utworzy egzemplarz `java.util.LinkedHashSet`.

W omawianym listingu pokazujemy, że użycie elementów `<set>` schematu `util` skutkuje uzyskaniem znacznie zwęższej konfiguracji niż w przypadku wykorzystania `SetFactoryBean` do tworzenia egzemplarzy `Set`.

Element `<properties>`

Oferowany przez schemat `util` element `<properties>` jest użyteczny, gdy chcesz utworzyć egzemplarz `java.util.Properties` na podstawie pliku właściwości i udostępnić obiekt `java.util.Properties` jako komponent.

W listingu 3.50 przedstawiamy przykład użycia elementu `<properties>`.

Listing 3.50. Plik `applicationContext.xml` przedstawiający element `<properties>` schematu `util`

Projekt: `ch03-util-schema-examples`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans .....
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="....http://www.springframework.org/schema/util
  http://www.springframework.org/schema/util/spring-util-4.0.xsd">

  <bean id="bankDetails" class="sample.spring.chapter03.beans.BankDetails">
    .....
    <property name="branchAddresses" ref="branchAddresses" />
  </bean>
  .....
  <util:properties id="branchAddresses"
    location="classpath:META-INF/addresses.properties" />
</beans>
```

W powyższym listingu element `<properties>` tworzy egzemplarz `java.util.Properties` zawierający właściwości wczytane z pliku `addresses.properties` (wskazany przez atrybut `location`) i udostępniający egzemplarz `java.util.Properties` jako komponent o identyfikatorze `branchAddresses` (wskazany przez atrybut `id`). W omawianym listingu można także zobaczyć, że właściwość `branchAddresses` (typu `java.util.Properties`) komponentu `BankDetails` odwołuje się do komponentu `branchAddresses` utworzonego za pomocą elementu `<properties>` schematu `util`.

Alternatywą dla użycia elementu `<properties>` jest wykorzystanie `PropertiesFactoryBean`.

PropertiesFactoryBean

Oferowany przez Spring element `PropertiesFactoryBean` to fabryka przeznaczona do tworzenia egzemplarzy typu `java.util.Properties`.

W listingu 3.51 pokazujemy, jak można użyć elementu `PropertiesFactoryBean` zamiast elementu `<Properties>` schematu `util`.

Listing 3.51. Przykład użycia elementu *PropertiesFactoryBean*

```
<beans .....>

    <bean id="bankDetails" class="sample.spring.chapter03.beans.BankDetails">
        .....
        <property name="branchAddresses" ref="branchAddresses" />
    </bean>

    <bean id="branchAddresses"
        class="org.springframework.beans.factory.config.PropertiesFactoryBean">
        <property name="location" value="classpath:META-INF/addresses.properties"/>
    </bean>
    .....
</beans>
```

W listingu 3.51 definicja komponentu dla `PropertiesFactoryBean` tworzy egzemplarz `java.util.Properties` na podstawie właściwości wczytanych z pliku `addresses.properties` (wskazany przez właściwość `location`) i udostępnia egzemplarz `java.util.Properties` jako komponent o identyfikatorze `branchAddresses`.

Element `<constant>`

Oferowany przez schemat `util` element `<constant>` jest używany w celu udostępnienia właściwości `public static` obiektu jako komponentu Springa.

W listingu 3.52 przedstawiamy przykład użycia elementu `<constant>`.

Listing 3.52. Plik *applicationContext.xml* przedstawiający element `<constant>` schematu `util`

Projekt: *ch03-util-schema-examples*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<beans ..... xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="..... http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-4.0.xsd">

    <bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
        .....
        <constructor-arg name="booleanType" ref="booleanTrue" />
        .....
    </bean>

    <util:constant id="booleanTrue" static-field="java.lang.Boolean.TRUE" />
    .....
</beans>
```

Element `<constant>` udostępnia jako komponent Springa wartość określoną za pomocą atrybutu `static-field`. W listingu 3.52 element `<constant>` udostępnia komponent o wartości `java.lang.Boolean.TRUE` i identyfikatorze `booleanTrue`. Dowolną właściwość `public static` możesz podać jako wartość atrybutu `static-field`, a następnie odwoływać się do niej z poziomu innych kom-

ponentów w kontenerze Springa. Na przykład w omawianym listingu odwołanie do komponentu `booleanType` następuje z poziomu argumentu konstruktora `booleanType` typu `boolean` klasy `DataTypesExample`.

Znacznie mniej zwięzłym sposobem udostępniania właściwości `public static` jako komponentów Springa jest użycie elementu `FieldRetrievingFactoryBean`.

FieldRetrievingFactoryBean

Oferowany przez Spring element `FieldRetrievingFactoryBean` to fabryka przeznaczona do pobierania wartości właściwości `public static` określonej przez właściwość `staticField` klasy `FieldRetrievingFactoryBean`. Wartość pobrana przez `FieldRetrievingFactoryBean` będzie udostępniona w postaci komponentu. `FieldRetrievingFactoryBean` można wykorzystać także do pobrania wartości właściwości innej niż statyczna.

W listingu 3.53 przedstawiamy przykład użycia elementu `FieldRetrievingFactoryBean`.

Listing 3.53. Przykład użycia elementu `FieldRetrievingFactoryBean`

```
<beans .....>

  <bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
    .....
    <constructor-arg name="booleanType" ref="booleanTrue" />
    .....
  </bean>

  <bean id="booleanTrue"
        class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
    <property name="staticField" value="java.lang.Boolean.TRUE"/>
  </bean>
  .....
</beans>
```

W listingu 3.53 element `FieldRetrievingFactoryBean` pobiera wartość właściwości `java.lang.Boolean.TRUE` i udostępnia ją w postaci komponentu. Do komponentu udostępnionego przez `FieldRetrievingFactoryBean` odwołujemy się z poziomu argumentu konstruktora `booleanType` typu `boolean` klasy `DataTypesExample`.

Element `<property-path>`

Oferowany przez schemat `util` element `<property-path>` jest używany w celu udostępnienia właściwości komponentu jako komponentu Springa.

W listingu 3.54 przedstawiamy przykład użycia elementu `<property-path>`.

Listing 3.54. Plik `applicationContext.xml` przedstawiający element `<property-path>` schematu `util`

Projekt: `ch03-util-schema-examples`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans .....>
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="..... http://www.springframework.org/schema/util
  http://www.springframework.org/schema/util/spring-util-4.0.xsd">
```

```

<bean id="bankDetails" class="sample.spring.chapter03.beans.BankDetails">
    .....
    <property name="dateOfInception" ref="dateType" />
    .....
</bean>

<util:property-path id="dateType" path="dataTypes.dateType" />

<bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
    .....
    <property name="dateType" value="30-01-2012" />
    .....
</bean>
</beans>

```

W listingu 3.54 zdefiniowana przez `DataTypesExample` właściwość `dateType` (typu `java.util.Date`) ma przypisaną wartość `30-01-2012`. Element `<property-path>` pobiera tę właściwość, a następnie udostępnia ją jako komponent o identyfikatorze `dateType`. Atrybut `path` elementy `<property-path>` ma przedstawioną poniżej składnię:

```
<nazwa-komponentu>.<nazwa-właściwości>
```

W tej składni `<nazwa-komponentu>` oznacza nazwę lub identyfikator komponentu, natomiast `<nazwa-właściwości>` wskazuje nazwę właściwości.

Ponieważ element `<property-path>` udostępnia komponent, do tak udostępnionego komponentu można się odwołać z poziomu innego komponentu w kontenerze Springa. Na przykład w omawianym listingu do komponentu `dateType` odwołujemy się we właściwości `dateOfInception` komponentu `BankDetails`.

Zamiast używać elementu `<property-path>` można wykorzystać element `PropertyPathFactoryBean` do udostępnienia wartości właściwości komponentu jako komponentu.

PropertyPathFactoryBean

Element `PropertyPathFactoryBean` to fabryka używana w celu tworzenia egzemplarzy przedstawiających wartość właściwości komponentu.

Przykład użycia elementu `PropertyPathFactoryBean` przedstawiamy w listingu 3.55.

Listing 3.55. Przykład użycia elementu `PropertyPathFactoryBean`

```

<beans .....
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="..... http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-4.0.xsd">

    <bean id="bankDetails" class="sample.spring.chapter03.beans.BankDetails">
        .....
        <property name="dateOfInception" ref="dateType" />
        .....
    </bean>

```



```
<bean id="dataType"
      class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
  <property name="targetBeanName" value="dataTypes"/>
  <property name="propertyPath" value="dateType"/>
</bean>

<bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample">
  .....
  <property name="dateType" value="30-01-2012" />
  .....
</bean>
</beans>
```

W listingu 3.55 element `PropertyPathFactoryBean` został wykorzystany w celu utworzenia egzemplarza komponentu przedstawiającego właściwość `dateType` komponentu `dataTypes`. Udostępniany przez `PropertyPathFactoryBean` atrybut `targetBeanName` pozwala na określenie identyfikatora lub nazwy komponentu zawierającego właściwość. Z kolei atrybut `propertyPath` określa nazwę właściwości, której wartość jest udostępniana jako komponent. Do egzemplarza komponentu utworzonego przez `PropertyPathFactoryBean` można uzyskać dostęp z poziomu innych komponentów znajdujących się w kontenerze Springa. W omawianym listingu do komponentu `dataType` utworzonego przez `PropertyPathFactoryBean` odwołujemy się we właściwości `dateOfInception` (typu `java.util.Date`) komponentu `BankDetails`.

Skoro dokładnie przeanalizowaliśmy elementy schematu `util`, możemy teraz przejść do omówienia interfejsu `FactoryBean` w Springu.

3.9. Interfejs `FactoryBean`

W Springu interfejs `FactoryBean` jest implementowany przez klasy działające w charakterze fabryki tworzącej egzemplarze komponentu. Z poprzedniego podrozdziału dowiedziałeś się, że klasy implementujące interfejs `FactoryBean` są konfigurowane w pliku XML kontekstu aplikacji, podobnie jak wszelkie inne komponenty. Interfejs `FactoryBean` jest szczególnie użyteczny, gdy zachodzi potrzeba przeprowadzenia skomplikowanych operacji sprawdzających, na podstawie których podejmowana jest decyzja o typie tworzonego komponentu. Ponadto interfejs będzie użyteczny do wykonania skomplikowanej logiki inicjalizującej komponent.

Przechodzimy teraz do aplikacji, w której interfejs `FactoryBean` wykorzystamy do wyboru typu komponentu, a następnie do jego utworzenia.

Aplikacja `MyBank` — przechowywanie zdarzeń w bazie danych

W aplikacji `MyBank` ważne zdarzenia, takie jak transakcje przeprowadzane kartami kredytowymi i debetowymi oraz zakładanie i likwidowanie lokat, są zapisywane w bazie danych. Aplikacja `MyBank` może zapisywać te zdarzenia bezpośrednio w bazie danych lub też pośrednio przez wysłanie zdarzenia do pośrednika bądź do usługi sieciowej. W tabeli 3.4 zostały wymienione klasy definiowane przez aplikację `MyBank`, które są przeznaczone do bezpośredniego lub pośredniego zapisu zdarzeń.

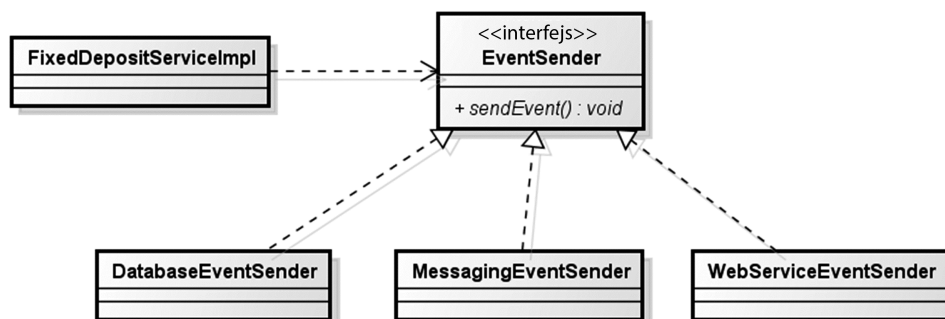
Tabela 3.4. Klasy aplikacji MyBank przeznaczone do zapisu zdarzeń

Klasa	Opis
DatabaseEventSender	Klasa zawierająca funkcjonalność zachowywania zdarzeń w bazie danych.
MessagingEventSender	Klasa zawierająca funkcjonalność wysyłania zdarzeń do pośrednika.
WebServiceEventSender	Klasa zawierająca funkcjonalność wysyłania zdarzeń do zdalnej usługi sieciowej.

Decyzja o bezpośrednim zapisie zdarzeń w bazie danych albo o wysłaniu ich do pośrednika lub do usługi sieciowej jest podejmowana na podstawie konfiguracji. Na przykład jeżeli aplikacja stwierdzi istnienie pliku o nazwie *database.properties*, wówczas odczytuje z niego informacje konfiguracyjne (takie jak adres URL bazy danych, nazwę użytkownika i hasło) i tworzy egzemplarz *DatabaseEventSender*. Podobnie w przypadku istnienia pliku *messaging.properties* aplikacja MyBank tworzy egzemplarz *MessagingEventSender*. Natomiast po znalezieniu pliku *webservice.properties* tworzony jest egzemplarz *WebServiceEventSender*.

Inicjacja egzemplarzy *DatabaseEventSender*, *MessagingEventSender* i *WebServiceEventSender* może wymagać wykonania skomplikowanej logiki inicjalizacyjnej. Na przykład może wystąpić konieczność utworzenia (lub pobrania z JNDI) egzemplarzy *javax.jms.ConnectionFactory* i *javax.jms.Destination*, a następnie ustawienia ich w egzemplarzu *MessagingEventSender*, aby umożliwić *MessagingEventSender* wysyłanie komunikatów JMS do pośrednika.

Diagram na rysunku 3.7 pokazuje, że klasa *FixedDepositServiceImpl* aplikacji MyBank używa egzemplarza *DatabaseEventSender*, *MessagingEventSender* lub *WebServiceEventSender* w celu bezpośredniego bądź pośredniego zapisu w bazie danych zdarzeń powiązanych z lokatami.

Rysunek 3.7. Klasa *FixedDepositServiceImpl* używa implementacji interfejsu *EventSender*

W diagramie pokazanym na rysunku 3.7 metoda *sendEvent()* interfejsu *EventSender* definiuje logikę odpowiedzialną za bezpośrednie lub pośrednie zapisanie zdarzeń w bazie danych. Klasy *DatabaseEventSender*, *MessagingEventSender* i *WebServiceEventSender* implementują interfejs *EventSender* i dostarczają odpowiednią implementację metody *sendEvent()*.

Przekonajmy się teraz, jak interfejs *FactoryBean* ułatwia wybór odpowiedniej implementacji interfejsu *EventSender*, a następnie ją inicjalizuje.

IMPORT **Rozdział03/ch03-bankapp-factorybean.** (Ten projekt pokazuje aplikację MyBank, w której użyto implementacji interfejsu `FactoryBean` do utworzenia obiektów typu `EventSender`. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Aplikacja MyBank — przykład użycia interfejsu `FactoryBean`

W aplikacji MyBank wybór odpowiedniej implementacji `EventSender` i jej inicjalizacja jest dość skomplikowanym zadaniem, a tym samym stanowi idealny scenariusz do użycia implementacji interfejsu `FactoryBean`. Interfejs `FactoryBean` definiuje następujące metody, które muszą być zaimplementowane:

- `getObjectType()` — wartością zwrótną tej metody jest *typ* obiektu zarządzanego przez implementację `FactoryBean`. W przypadku naszej aplikacji MyBank implementacja `FactoryBean` tworzy i zwraca obiekty typu `EventSender`.
- `getObject()` — wartością zwrótną tej metody jest obiekt zarządzany przez implementację `FactoryBean`. W przypadku naszej aplikacji MyBank metoda `getObject()` zwraca egzemplarz `DatabaseEventSender`, `MessagingEventSender` lub `WebServiceEventSender`.
- `isSingleton()` — wartością zwrótną tej metody będzie `true`, jeżeli implementacja `FactoryBean` jest fabryką dla obiektów typu `singleton`. Gdy metoda `isSingleton()` zwraca `true`, wówczas obiekt zwrócony przez metodę `getObject()` jest buforowany przez kontener Springa i ten sam egzemplarz będzie zwracany w kolejnych żądaniach. Natomiast jeśli implementacja `FactoryBean` jest fabryką dla obiektów o zakresie prototypu, wtedy wartością zwrótną metody `isSingleton()` jest `false`. Oznacza to, że metoda `getObject()` będzie tworzyć nowy egzemplarz dla każdego żądania. W przypadku naszej aplikacji MyBank implementacja `FactoryBean` zwraca egzemplarz klasy `DatabaseEventSender`, `MessagingEventSender` lub `WebServiceEventSender`. Po utworzeniu ten sam egzemplarz jest używany w trakcie całego cyklu życiowego aplikacji MyBank, a więc wartością zwrótną metody `isSingleton()` jest `true`.

W listingu 3.56 przedstawiamy przykład użycia `EventSenderFactoryBean`, czyli implementacji `FactoryBean` tworzącej i zwracającej obiekty typu `EventSender`.

Listing 3.56. Klasa `EventSenderFactoryBean`

Projekt: *ch03-bankapp-factorybean*

Położenie pliku: *src/main/java/sample/spring/chapter03/bankapp/event*

```
package sample.spring.chapter03.bankapp.event;

import org.springframework.beans.factory.FactoryBean;
import org.springframework.beans.factory.FactoryBeanNotInitializedException;
import org.springframework.core.io.ClassPathResource;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

public class EventSenderFactoryBean implements FactoryBean<EventSender> {
    private String databasePropertiesFile;
    private String webServicePropertiesFile;
    private String messagingPropertiesFile;
    private boolean isSingleton = true;

    public EventSender getObject() throws Exception {
        EventSender eventSender = null;
        try {
            eventSender = new DatabaseEventSender(
                new ClassPathResource(databasePropertiesFile).getInputStream(),
                new ClassPathResource(webServicePropertiesFile).getInputStream(),
                new ClassPathResource(messagingPropertiesFile).getInputStream()
            );
        } catch (IOException e) {
            throw new FactoryBeanNotInitializedException(e);
        }
        return eventSender;
    }

    public Class<EventSender> getObjectType() {
        return EventSender.class;
    }

    public boolean isSingleton() {
        return isSingleton;
    }
}
```

```

        Properties properties = new Properties();

        ClassPathResource databaseProperties = null;
        if(databasePropertiesFile != null) {
            databaseProperties = new ClassPathResource(databasePropertiesFile);
        }
        ....
        if (databaseProperties != null && databaseProperties.exists()) {
            InputStream inStream = databaseProperties.getInputStream();
            properties.load(inStream);
            eventSender = new DatabaseEventSender(properties);
        }
        else if (webServiceProperties != null && webServiceProperties.exists()) {....}
        else if (messagingProperties != null && messagingProperties.exists()) {.....}

        return eventSender;
    }

    public Class<?> getObjectType() {
        return EventSender.class;
    }

    public boolean isSingleton() {
        return true;
    }
}

```

W powyższym listingu widać, że klasa `EventSenderFactoryBean` implementuje interfejs `FactoryBean`. Parametr `EventSender` w `FactoryBean<EventSender>` wskazuje, że wartością zwrótną metody `getObject()` klasy `FactoryBean` jest obiektu typu `EventSender`. Elementy `databasePropertiesFile`, `webServicePropertiesFile` i `messagingPropertiesFile` są właściwościami klasy `EventSender` ↪ `FactoryBean` i przedstawiają położenie plików *database.properties*, *webservice.properties* i *messaging.properties* w ścieżce dostępu zdefiniowanej przez zmienną środowiskową `CLASSPATH`.

Metoda `getObject()` używa oferowanej przez Spring klasy `ClassPathResource` do przeprowadzenia weryfikacji, czy wymienione pliki właściwości istnieją we wspomnianej lokalizacji. Jeżeli pliki właściwości istnieją, zdefiniowane w tych plikach właściwości zostaną wczytane i przekazane konstruktorowi klasy implementacji `EventSender`. Na przykład jeżeli kod w omawianym listingu znajdzie plik *database.properties* (przedstawiany przez właściwość `databasePropertiesFile`), wówczas właściwości z tego pliku zostaną wczytane, a następnie przekazane jako argument konstruktora `DatabaseEventSender`. Wartością zwrótną metody `getObjectType()` będzie typ `EventSender`, ponieważ metoda `getObject()` klasy `EventSenderFactoryBean` zwróci obiekty typu `EventSender`. Z kolei wartością zwrótną metody `isSingleton()` będzie wartość `true`, co oznacza, że obiekt zwracany przez metodę `getObject()` będzie buforowany przez Spring i ten sam egzemplarz zostanie zwrócony w trakcie każdego wywołania metody `getObject()` klasy `EventSenderFactoryBean`.

Skoro poznałeś sposób implementacji klasy `EventSenderFactoryBean` w naszej aplikacji `MyBank`, prawdopodobnie domyślasz się, jak są implementowane wbudowane w Spring implementacje interfejsu `FactoryBean`, na przykład `ListFactoryBean` (do tworzenia egzemplarzy typu `List`), `Map` ↪ `FactoryBean` (do tworzenia egzemplarzy typu `Map`), `SetFactoryBean` (do tworzenia egzemplarzy typu `Set`) itd.

W listingu 3.57 przedstawiamy sposób konfiguracji klasy `EventSenderFactoryBean` w pliku XML kontekstu aplikacji.

Listing 3.57. Plik `applicationContext.xml` przedstawiający konfigurację klasy `EventSenderFactoryBean`

Projekt: `ch03-bankapp-factorybean`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans .....>

  <bean id="service"
    class="sample.spring.chapter03.bankapp.service.FixedDepositServiceImpl">
    ....
    <property name="eventSender" ref="eventSenderFactory" />
  </bean>
  ....
  <bean id="eventSenderFactory"
    class="sample.spring.chapter03.bankapp.event.EventSenderFactoryBean">
    <property name="databasePropertiesFile"
      value="META-INF/config/database.properties"/>
  </bean>
</beans>
```

W powyższym listingu pokazujemy, że klasa `EventSenderFactoryBean` została skonfigurowana jak każdy inny komponent Springa. Pomimo skonfigurowania implementacji `FactoryBean` jak innego dowolnego komponentu Springa jest on traktowany *inaczej* przez kontener Springa. Najważniejsza różnica polega na tym, że komponent jest zależny od implementacji `FactoryBean`. Kontener Springa wywołuje metodę `getObject()` interfejsu `FactoryBean` i zwrócony przez tę metodę obiekt wstrzykuje do komponentu zależnego.



Zwróć uwagę, że jeśli wartością zwrótną metody `isSingleton()` będzie `true`, to metoda `getObject()` interfejsu `FactoryBean` jest przez kontener Springa wywoływana tylko jeden raz.

W omawianym listingu definicja komponentu dla klasy `FixedDepositServiceImpl` pokazuje zależność od `EventSenderFactoryBean`, czyli implementacji `FactoryBean`. Dlatego kontener Springa wywołuje metodę `getObject()` klasy `EventSenderFactoryBean` i otrzymany obiekt typu `EventSender` wstrzykuje do egzemplarza `FixedDepositServiceImpl`.

W listingu 3.58 została przedstawiona klasa `FixedDepositServiceImpl`, która wymaga egzemplarza `EventSender` utworzonego przez `EventSenderFactoryBean`.

Listing 3.58. Klasa `FixedDepositServiceImpl`

Projekt: `ch03-bankapp-factorybean`

Położenie pliku: `src/main/java/sample/spring/chapter03/bankapp/service`

```
package sample.spring.chapter03.bankapp.service;

import sample.spring.chapter03.bankapp.event.EventSender;

public class FixedDepositServiceImpl implements FixedDepositService {
  ....
  private EventSender eventSender;
```

```

public void setEventSender(EventSender eventSender) {
    this.eventSender = eventSender;
}
.....
public void createFixedDeposit(FixedDepositDetails fixedDepositDetails) {
    .....
    eventSender.sendEvent(event);
}
}

```

W powyższym listingu widać, że klasa `FixedDepositServiceImpl` zależy od egzemplarza `EventSender`, ale *nie* od egzemplarza `EventSenderFactoryBean`. Kontener Springa pobiera egzemplarz `EventSender` przez wywołanie metody `getObject()` klasy `EventSenderFactoryBean`, a następnie wstrzykuje otrzymany egzemplarz `EventSender` do egzemplarza `FixedDepositServiceImpl`.

Zobaczmy teraz, jak uzyskać dostęp do samego egzemplarza `FactoryBean`, a *nie* komponentu tworzonego i zwracanego za pomocą metody `getObject()`.

Uzyskanie dostępu do egzemplarza `FactoryBean`

Jeżeli z kontenera Springa chcesz pobrać sam egzemplarz `FactoryBean`, wówczas nazwę (lub identyfikator) komponentu fabryki musisz poprzedzić znakiem ampersand (&).

Przyjmujemy założenie, że klasa `FixedDepositServiceImpl` wymaga uzyskania dostępu do `EventSenderFactoryBean`, jak przedstawia listing 3.59.

Listing 3.59. Klasa `FixedDepositServiceImpl`, która jest zależna od egzemplarza `EventSenderFactoryBean`

```

package sample.spring.chapter03.bankapp.service;

import sample.spring.chapter03.bankapp.event.EventSenderFactoryBean;
import sample.spring.chapter03.bankapp.event.EventSender;

public class FixedDepositServiceImpl implements FixedDepositService {
    .....
    private EventSenderFactoryBean eventSenderFactoryBean;

    public void setEventSenderFactoryBean (EventSenderFactoryBean
        eventSenderFactoryBean) {
        this.eventSenderFactoryBean = eventSenderFactoryBean;
    }
    .....
    public void createFixedDeposit(FixedDepositDetails fixedDepositDetails) {
        .....
        EventSender eventSender = eventSenderFactoryBean.getObject();
        eventSender.sendEvent(event);
    }
}

```

W powyższym listingu klasa `FixedDepositServiceImpl` zależy od egzemplarza `EventSenderFactoryBean` i używa jego metody `getObject()` w celu pobrania egzemplarza `EventSender`.

W listingu 3.57 zobaczyłeś, że podczas definiowania komponentu `EventSenderFactoryBean` jako zależności komponentu `FixedDepositServiceImpl` kontener Springa wywołuje metodę `getObject()`

klasy `EventSenderFactoryBean` i wstrzykuje do komponentu `FixedDepositServiceImpl` otrzymany obiekt typu `EventSender`. Aby kontenerowi Springa nakazać wstrzyknięcie egzemplarza `EventSender` ↪ `FactoryBean`, nazwę komponentu zdefiniowaną w atrybucie `ref` poprzedź znakiem `&`, jak pokazuje listing 3.60.

Listing 3.60. Wstrzyknięcie egzemplarza `EventSenderFactoryBean` do komponentu `FixedDepositServiceImpl`

```
<beans .....>
  <bean id="service"
    class="sample.spring.chapter03.bankapp.service.FixedDepositServiceImpl">
    .....
    <property name="eventSenderFactoryBean" ref="&eventSenderFactory" />
  </bean>
  .....
  <bean id="eventSenderFactory"
    class="sample.spring.chapter03.bankapp.event.EventSenderFactoryBean">
    <property name="databasePropertiesFile"
      value="META-INF/config/database.properties"/>
  </bean>
</beans>
```

W powyższym listingu wymieniony poniżej element `<property>` określa, że komponent `FixedDepositServiceImpl` jest zależny od `EventSenderFactoryBean`:

```
<property name="eventSenderFactoryBean" ref="&eventSenderFactory" />
```

Zwróć uwagę na wartość atrybutu `ref` wynoszącą `&eventSenderFactory`. Prefiks `&` nakazuje kontenerowi Springa wstrzyknięcie do komponentu `FixedDepositServiceImpl` egzemplarza `EventSenderFactoryBean`.

Użycie znaku `&` jest wymagane również wtedy, gdy chcesz otrzymać egzemplarz `FactoryBean` za pomocą metody `getBean()` klasy `ApplicationContext`. W listingu 3.61 została pokazana klasa `BankApp` naszej aplikacji `MyBank`. Klasa ta pobiera obiekt `EventSender` utworzony przez `EventSender` ↪ `FactoryBean` oraz sam egzemplarz `EventSenderFactoryBean`.

Listing 3.61. Klasa `BankApp`

Projekt: `ch03-bankapp-factorybean`

Położenie pliku: `src/main/java/sample/spring/chapter03/bankapp`

```
package sample.spring.chapter03.bankapp;
.....
public class BankApp {
  private static Logger logger = Logger.getLogger(BankApp.class);

  public static void main(String args[]) {
    ApplicationContext context = new ClassPathXmlApplicationContext(
      .....
      logger.info("Wywołanie getBean(\"eventFactory\") zwraca : " +
        context.getBean("eventSenderFactory"));
      logger.info("Wywołanie getBean(\"&eventFactory\") zwraca : " +
        context.getBean("&eventSenderFactory"));
    }
  }
}
```

Jeżeli wykonasz metodę `main()` klasy `BankApp` przedstawionej w listingu 3.61, przekonasz się, że wywołanie `getBean("eventSenderFactory")` zwraca egzemplarz klasy `DatabaseEventSender`, natomiast wywołanie `getBean("&eventSenderFactory")` zwraca egzemplarz klasy `EventSenderFactoryBean`.

3.10. Podsumowanie

W tym rozdziale zobaczyłeś, jak można użyć dziedziczenia definicji komponentu w celu tworzenia mniej rozwlekłych i łatwiejszych w zarządzaniu definicji komponentów. Większa część rozdziału została poświęcona na pokazanie sposobów konfiguracji różnego typu właściwości komponentu i argumentów konstruktora za pomocą wbudowanych implementacji interfejsu `FactoryBean` oferowanego przez Spring schematu `util` oraz przestrzeni nazw `p` i `c`. Ponadto przeanalizowaliśmy niektóre z wbudowanych w Spring implementacji `PropertyEditor`. Dowiedziałeś się również, jak w kontenerze Springa zarejestrować dodatkowe edytory właściwości. W kolejnym rozdziale zajmiemy się dokładnym omówieniem funkcji wstrzykiwania zależności w Springu.

Rozdział 4. Wstrzykiwanie zależności

4.1. Wprowadzenie

W poprzednim rozdziale analizowaliśmy konfigurację podzespołów za pomocą oferowanego przez Spring schematu `util`, a także przestrzeni nazw `p` oraz `c`, implementacji `FactoryBean` itd. W tym rozdziale skoncentrujemy się na innych scenariuszach wstrzykiwania zależności, które najczęściej występują podczas tworzenia rzeczywistych aplikacji. Dowiesz się, jak Spring radzi sobie w tych sytuacjach.

Na początku rozdziału zajmiemy się *komponentami wewnętrznymi*, czyli alternatywą dla użycia atrybutu `ref` oraz elementów `<property>` i `<constructor-arg>`. Następnie przejdziemy do atrybutu `depends-on` elementu `<bean>`. W drugiej części rozdziału przeanalizujemy kwestie, jakie mogą się pojawić, gdy komponenty o zasięgu `singleton` lub prototypu współpracują ze sobą, aby dostarczyć odpowiednią funkcjonalność aplikacji. Rozdział zakończy się dokładnym omówieniem funkcji *automatycznego wiązania zależności* oferowanej przez Spring.

IMPORT **Rozdział 04/ch04-bankapp-dependencies.** (Ten projekt pokazuje użycie wewnętrznych komponentów oraz atrybutu `depends-on` elementu `<bean>`. Ponadto przedstawia implikacje podczas definiowania zależności komponentów o zasięgu `singleton` od komponentów o zasięgu prototypu i na odwrót. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

4.2. Komponenty wewnętrzne

Jeżeli zależność komponentu *nie* jest współdzielona przez wiele komponentów, warto rozważyć jej zdefiniowanie jako *komponent wewnętrzny*. Komponent wewnętrzny jest definiowany wewnątrz elementu `<property>` lub `<constructor-arg>` za pomocą elementu `<bean>` schematu `util` dostarczanego przez Spring. Zwróć uwagę, że komponent wewnętrzny jest dostępny jedynie dla komponentu nadrzędnego, ale już *nie* dla innych komponentów zarejestrowanych w kontenerze Springa.

W listingu 4.1 przedstawiamy ogólny sposób wyrażania zależności komponentu.

Listing 4.1. Zdefiniowanie zależności komponentu za pomocą atrybutu `ref` elementu `<property>`

```
<bean id="service"
      class="sample.spring.chapter04.bankapp.service.FixedDepositServiceImpl">
  <property name="fixedDepositDao" ref="dao" />
</bean>

<bean id="dao" class="sample.spring.chapter04.bankapp.dao.FixedDepositDaoImpl" />
```

W powyższym listingu pokazano, że komponent `service` jest zależny od komponentu `dao`. Jeżeli komponent `service` będzie jedynym komponentem zależnym od `dao`, wówczas `dao` można zdefiniować jako *komponent wewnętrzny* komponentu `service`, jak przedstawia listing 4.2.

Listing 4.2. Plik *applicationContext.xml* przedstawiający przykład komponentu wewnętrznegoProjekt: *ch04-bankapp-dependencies*Położenie pliku: *src/main/resources/META-INF/spring*

```

<bean id="service"
      class="sample.spring.chapter04.bankapp.service.FixedDepositServiceImpl">
  <property name="fixedDepositDao">
    <bean class="sample.spring.chapter04.bankapp.dao.FixedDepositDaoImpl" />
  </property>
</bean>

```

W powyższym listingu definicja komponentu dla klasy `FixedDepositDaoImpl` znajduje się wewnątrz elementu `<property>` komponentu `service`. Jeżeli omawiany listing porównasz z listingiem 4.1, to zauważysz, że element `<property>` nie zawiera atrybutu `ref`, a element `<bean>` odpowiadający klasie `FixedDepositDaoImpl` nie zawiera atrybutu `id`.

Element `<bean>` odpowiadający definicji komponentu wewnętrznego nie zawiera atrybutu `id`, ponieważ komponent wewnętrzny *nie* jest rejestrowany w kontenerze Springa. Jeżeli mimo to podasz atrybut `id` dla komponentu wewnętrznego, zostanie on zignorowany przez kontener Springa. Komponent wewnętrzny *zawsze* ma zasięg prototypu. Dlatego jeśli element `<bean>` odpowiadający definicji komponentu wewnętrznego zawiera atrybut `scope`, atrybut ten zostanie zignorowany przez kontener Springa. Warto w tym miejscu zwrócić uwagę na fakt, że komponent wewnętrzny z natury jest *anonimowy*, więc pozostaje niedostępny dla innych komponentów (z wyjątkiem jego komponentu nadrzędnego) w kontenerze Springa.



Podobnie jak w przypadku definicji zwykłego komponentu, wewnątrz elementu `<bean>` definicji komponentu wewnętrznego można używać elementów `<property>`, `<constructor-arg>` itd.

Z poprzedniego rozdziału dowiedziałeś się, że elementy oferowanego przez Spring schematu `util` są używane do tworzenia komponentów przedstawiających `List`, `Set`, `Map` itd. Zobaczyłeś również, że do komponentów tworzonych przez elementy schematu `util` można odwoływać się z poziomu innych komponentów. Koncepcja komponentu wewnętrznego pozwala na użycie elementów schematu `util` wewnątrz elementów `<property>` i `<constructor-arg>`, jak pokazuje listing 4.3.

Listing 4.3. Element `<list>` schematu `util` definiuje komponent wewnętrzny

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="..... http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-4.0.xsd">

  <bean id="someBean" class="com.sample.SomeBean">
    .....
    <constructor-arg name="listType">
      <util:list list-class="java.util.ArrayList">
        <value>Prosty ciąg tekstowy w elemencie list</value>
        <value>Kolejny prosty ciąg tekstowy w elemencie list</value>
      </util:list>
    </constructor-arg>
  </bean>

```

```
.....  
</bean>  
</beans>
```

W powyższym listingu argument konstruktora `listType` jest typu `java.util.List`. Wartość przekazywana argumentowi konstruktora `listType` jest definiowana za pomocą elementu `<list>` schematu `util`. Zwróć uwagę na brak zdefiniowanego atrybutu `id` elementu `<list>`, ponieważ kontener Springa ignoruje atrybuty `id` w komponentach wewnętrznych.

Przechodzimy teraz do przeanalizowania atrybutu `depends-on` elementu `<bean>`.

4.3. Wyrażna kontrola kolejności inicjalizacji procesu za pomocą atrybutu `depends-on`

Z podrozdziału 1.4 dowiedziałeś się, że komponenty są tworzone w kolejności, w jakiej je zdefiniowano w pliku XML kontekstu aplikacji. Kolejność tworzenia komponentów jest ustalana także na podstawie zależności między komponentami. Na przykład jeżeli komponent A akceptuje argument konstruktora w postaci egzemplarza komponentu B, wówczas kontener Springa *najpierw* utworzy komponent B, a dopiero później A, niezależnie od kolejności ich zdefiniowania w pliku XML konspektu aplikacji. Takie zachowanie kontenera Springa gwarantuje, że zależności komponentu (w omawianym przykładzie zależnością jest komponent B) będą w pełni skonfigurowane, zanim nastąpi ich wstrzyknięcie (w omawianym przykładzie komponentem zależnym jest A).

W pewnych sytuacjach zależności komponentów nie są wyraźnie określone za pomocą elementów `<property>` i `<constructor-arg>`. Jeżeli zależności komponentu *nie* są wyraźnie zdefiniowane, można użyć atrybutu `depends-on` elementu `<bean>` w celu wyraźnego określenia zależności komponentu. Kontener Springa gwarantuje, że zależności zdefiniowane za pomocą atrybutu `depends-on` będą zainicjowane *przed* komponentem wskazanym przez atrybut `depends-on`.

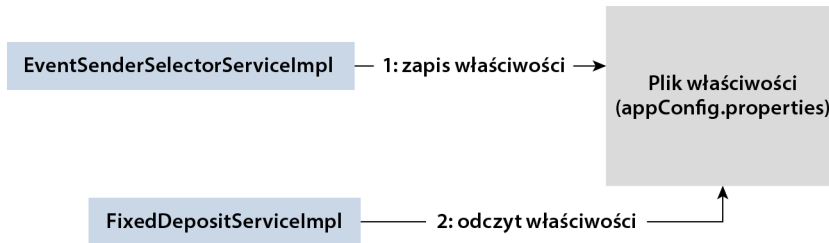
Przeanalizujemy teraz przykładowy scenariusz, w którym atrybut `depends-on` jest używany do kontrolowania kolejności inicjalizacji komponentów.

Aplikacja MyBank — zasugerowane zależności między komponentami

W naszej aplikacji MyBank z poprzedniego rozdziału implementacja `FactoryBean` tworzyła obiekt `EventSender`, który był używany przez egzemplarz `FixedDepositServiceImpl` w celu bezpośredniego lub pośredniego umieszczania zdarzeń w bazie danych (więcej informacji szczegółowych na ten temat znajdziesz w podrozdziale 3.9). Przyjmujemy założenie, że do utworzenia egzemplarza `EventSender` zamiast użyć implementacji `FactoryBean` zastosujemy podejście przedstawione na rysunku 4.1.

Na podstawie diagramu pokazanego na rysunku 4.1 można wyciągnąć następujące wnioski:

- Klasa `EventSenderSelectorServiceImpl` jest używana do wyboru implementacji `EventSender` (`DatabaseEventSender`, `WebServiceEventSender` lub `MessagingEventSender`), która będzie używana przez klasę `FixedDepositServiceImpl`.
- Klasa `EventSenderSelectorServiceImpl` przechowuje w pliku `appConfig.properties` w pełni kwalifikowaną nazwę implementacji `EventSender`.



Rysunek 4.1. Klasa `EventSenderSelectorServiceImpl` zapisuje nazwę implementacji `EventSender` w pliku `appConfig.properties`, który następnie jest odczytywany przez egzemplarz `FixedDepositServiceImpl`

- Klasa `FixedDepositServiceImpl` odczytuje z pliku `appConfig.properties` w pełni kwalifikowaną nazwę implementacji `EventSender`, tworzy obiekt `EventSender` i używa go do umieszczania w bazie danych zdarzeń dotyczących lokat.

Powyższe podejście sugeruje, że egzemplarz `FixedDepositServiceImpl` nie będzie działał prawidłowo, gdy klasie `EventSenderSelectorServiceImpl` nie uda się zapisać w pliku `appConfig.properties` w pełni kwalifikowanej nazwy implementacji `EventSender`. Dlatego też klasa `FixedDepositServiceImpl` jest w niejawnym sposób zależna od klasy `EventSenderSelectorServiceImpl`.

Przechodzimy teraz do przeanalizowania implikacji niejawnej zależności egzemplarza `FixedDepositServiceImpl` od egzemplarza `EventSenderSelectorServiceImpl`.

Problem niejawnej zależności

Rozważ przedstawiony w listingu 4.4 plik XML kontekstu aplikacji zawierający definicje komponentów dla klas `FixedDepositServiceImpl` i `EventSenderSelectorServiceImpl`.

Listing 4.4. Plik `applicationContext.xml` przedstawiający przykład niejawnej zależności

Projekt: `ch04-bankapp-dependencies`

Położenie pliku: `src/main/resources/META-INF/spring`

```

<beans .....>
  <bean id="service"
    class="sample.spring.chapter04.bankapp.service.FixedDepositServiceImpl">
    .....
    <constructor-arg index="0" value="META-INF/config/appConfig.properties" />
  </bean>

  <bean id="eventSenderSelectorService"
    class="sample.spring.chapter04.bankapp.service.EventSenderSelectorServiceImpl">
    <constructor-arg index="0" value="META-INF/config/appConfig.properties" />
  </bean>
</beans>
  
```

W przedstawionym powyżej pliku XML kontekstu aplikacji widać, że konstruktory klas `FixedDepositServiceImpl` i `EventSenderSelectorServiceImpl` akceptują położenie pliku `appConfig.properties`. Egzemplarz `EventSenderSelectorServiceImpl` używa pliku `appConfig.properties` w celu podania egzemplarzowi `FixedDepositServiceImpl` w pełni kwalifikowanej nazwy implementacji `EventSender`. Ponieważ między komponentami `service` i `eventSenderSelectorService` *nie* istnieje

wyraźna zależność, kontener Springa tworzy ich egzemplarze z zachowaniem kolejności podanej w pliku XML kontekstu aplikacji. W pliku tym komponent service jest zdefiniowany *przed* event ↪ SenderSelectorService, a więc egzemplarz FixedDepositServiceImpl zostanie utworzony *przed* egzemplarzem EventSenderSelectorServiceImpl. Wkrótce zobaczysz, że jeżeli egzemplarz FixedDepositServiceImpl zostanie utworzony *przed* egzemplarzem EventSenderSelectorServiceImpl, to FixedDepositServiceImpl *nie* ma możliwości odczytania w pełni kwalifikowanej nazwy klasy implementacji EventSender zapisanej w pliku *appConfig.properties*.

Przechodzimy teraz do dokładnej analizy klas EventSenderSelectorServiceImpl i FixedDepositServiceImpl oraz pliku *appConfig.properties*.

Klasa EventSenderSelectorServiceImpl — zapis właściwości

W listingu 4.5 została przedstawiona klasa EventSenderSelectorServiceImpl.

Listing 4.5. Klasa EventSenderSelectorServiceImpl

Projekt: ch04-bankapp-dependencies

Położenie pliku: src/main/java/sample/spring/chapter04/bankapp/service

```
package sample.spring.chapter04.bankapp.service;

import org.springframework.core.io.ClassPathResource;
import sample.spring.chapter04.bankapp.Constants;

public class EventSenderSelectorServiceImpl {

    public EventSenderSelectorServiceImpl(String configFile) throws Exception {
        ClassPathResource resource = new ClassPathResource(configFile);
        OutputStream os = new FileOutputStream(resource.getFile());

        Properties properties = new Properties();
        properties
            .setProperty(Constants.EVENT_SENDER_CLASS_PROPERTY,
                "sample.spring.chapter04.bankapp.event.DatabaseEventSender");
        properties.store(os, null);
        ....
    }
}
```

W powyższym listingu widać, że informacje o lokalizacji pliku *appConfig.properties* są przekazywane jako argument konstruktora do klasy EventSenderSelectorServiceImpl. Konstruktor klasy EventSenderSelectorServiceImpl w pliku *appConfig.properties* zapisuje właściwość o nazwie eventSender ↪ Class (która jest wartością stałej EVENT_SENDER_CLASS_PROPERTY zdefiniowanej w klasie Constants). Właściwość eventSenderClass określa w pełni kwalifikowaną nazwę implementacji EventSender przeznaczoną do użycia przez egzemplarz FixedDepositServiceImpl w celu bezpośredniego lub pośredniego umieszczania zdarzeń w bazie danych. Aby zachować prostotę przykładu, konstruktor klasy EventSenderSelectorServiceImpl ustawia w pełni kwalifikowaną nazwę klasy DatabaseEventSender jako wartość właściwości eventSenderClass.

Plik appConfig.properties

Poniższy wiersz kodu zostanie przez klasę `EventSenderSelectorServiceImpl` dodany do pliku `appConfig.properties`:

```
eventSenderClass=sample.spring.chapter04.bankapp.event.DatabaseEventSender
```

Klasa `FixedDepositServiceImpl` — odczyt właściwości

Właściwość `eventSenderClass` zapisana przez egzemplarz `EventSenderSelectorServiceImpl` jest odczytywana w klasie `FixedDepositServiceImpl`, jak przedstawia listing 4.6.

Listing 4.6. Klasa `FixedDepositServiceImpl`

Projekt: `ch04-bankapp-dependencies`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp/service`

```
package sample.spring.chapter04.bankapp.service;

import org.springframework.core.io.ClassPathResource;
import sample.spring.chapter04.bankapp.Constants;

public class FixedDepositServiceImpl implements FixedDepositService {
    private FixedDepositDao fixedDepositDao;
    private EventSender eventSender;

    public FixedDepositServiceImpl(String configFile) throws Exception {
        ClassPathResource configProperties = new ClassPathResource(configFile);

        if (configProperties.exists()) {
            InputStream inStream = configProperties.getInputStream();
            Properties properties = new Properties();
            properties.load(inStream);

            String eventSenderClassString =
                properties.getProperty(Constants.EVENT_SENDER_CLASS_PROPERTY);

            if (eventSenderClassString != null) {
                Class<?> eventSenderClass = Class.forName(eventSenderClassString);
                eventSender = (EventSender) eventSenderClass.newInstance();
                logger.info("Utworzono klasę EventSender");
            } else {
                logger.info("Plik appConfig.properties nie zawiera informacji " +
                    "o klasie EventSender.");
            }
        }
    }

    public void createFixedDeposit(FixedDepositDetails fixedDepositDetails)
        throws Exception {
        ....
        eventSender.sendEvent(event);
    }
}
```

Powyższy listing przedstawia następującą sekwencję zadań, które są wykonywane przez konstruktor klasy `FixedDepositServiceImpl`:

- Wczytanie właściwości z pliku *appConfig.properties*. Argument konstruktora *configFile* określa położenie pliku *appConfig.properties*.
- Pobranie właściwości o nazwie *eventSenderClass* (przedstawianej przez stałą *EVENT_SENDER_CLASS_PROPERTY* zdefiniowaną w klasie *Constants*) z danych, które zostały wczytane z pliku *appConfig.properties*. Wartością właściwości *eventSenderClass* jest w pełni kwalifikowana nazwa klasy implementacji *EventSender*, która będzie używana przez *FixedDepositServiceImpl*. Wartość właściwości *eventSenderClass* jest przechowywana w zmiennej lokalnej o nazwie *eventSenderClassString*.
- Utworzenie egzemplarza implementacji *EventSender*, której w pełni kwalifikowana nazwa jest przechowywana w zmiennej *eventSenderClassString*. Nowo utworzony egzemplarz jest przechowywany przez zmienną egzemplarza o nazwie *eventSender*. Zmienna *eventSender* będzie później używana przez metodę *createFixedDeposit()* klasy *FixedDepositServiceImpl* (patrz: listing) w celu bezpośredniego lub pośredniego umieszczania zdarzeń w bazie danych.

Zauważ, że jeśli właściwość o nazwie *eventSenderClass* *nie* zostanie znaleziona w pliku *appConfig.properties*, wówczas zmienna *eventSenderClassString* *nie* będzie ustawiona. W takim przypadku konstruktor klasy *FixedDepositServiceImpl* wyświetli w konsoli komunikat o treści Plik *appConfig.properties* nie zawiera informacji o klasie *EventSender*.

W listingu 4.4 widzieliśmy definicje komponentów dla klas *EventSenderSelectorServiceImpl* i *FixedDepositServiceImpl*. Na podstawie tych definicji wyciągnęliśmy wniosek, że egzemplarz *FixedDepositServiceImpl* jest tworzony przed egzemplarzem *EventSenderSelectorServiceImpl*, ponieważ kontener Springa inicjalizuje komponenty w kolejności, w jakiej zostały zdefiniowane w pliku XML kontekstu aplikacji. Natomiast w listingu 4.5 widzieliśmy, że utworzenie egzemplarza *EventSenderSelectorServiceImpl* skutkuje zapisem właściwości *eventSenderClass* w pliku *appConfig.properties*. Dlatego jeżeli egzemplarz *FixedDepositServiceImpl* zostanie utworzony *przed* egzemplarzem *EventSenderSelectorServiceImpl*, wówczas egzemplarz *FixedDepositServiceImpl* *nie* znajdzie żadnej właściwości *eventSenderClass* w pliku *appConfig.properties*. Mamy więc wyraźny dowód, że klasa *FixedDepositServiceImpl* jest *niejawnie* zależna od klasy *EventSenderSelectorServiceImpl*, a tym samym egzemplarz *EventSenderSelectorServiceImpl* *musi* być utworzony przed egzemplarzem *FixedDepositServiceImpl*.

Jak rozwiązać problem niejawnej zależności?

Problem niejawnej zależności można rozwiązać na dwa sposoby:

- Zmiana kolejności, w jakiej definicje komponentów dla klas *EventSenderSelectorServiceImpl* i *FixedDepositServiceImpl* będą podane w pliku XML kontekstu aplikacji. Jeżeli definicja komponentu dla klasy *EventSenderSelectorServiceImpl* będzie znajdowała się *przed* definicją komponentu dla klasy *FixedDepositServiceImpl*, to egzemplarz *EventSenderSelectorServiceImpl* zostanie utworzony *przed* egzemplarzem *FixedDepositServiceImpl*.
- Użycie atrybutu *depends-on* elementu *<bean>* w celu *wyrażnego* wskazania, że komponent *service* (odpowiadający klasie *FixedDepositServiceImpl*) jest zależny od komponentu *eventSender* (odpowiadającego klasie *EventSenderSelectorServiceImpl*).

W listingu 4.7 pokazujemy sposób użycia atrybutu *depends-on* elementu *<bean>*.

Listing 4.7. Przykład użycia atrybutu *depends-on* elementu *<bean>*

```
<beans .....>
  <bean id="service"
        class="sample.spring.chapter04.bankapp.service.FixedDepositServiceImpl"
        depends-on="eventSenderSelectorService">
    .....
  </bean>

  <bean id="eventSenderSelectorService"
        class="sample.spring.chapter04.bankapp.service.EventSenderSelectorServiceImpl">
    .....
  </bean>
</beans>
```

W powyższym listingu komponent *service* używa atrybutu *depends-on* w celu wyraźnego wskazania zależności od komponentu *eventSenderSelectorService*. Atrybut *depends-on* podaje identyfikatory lub nazwy komponentów, od których zależy dany komponent. Ponieważ komponent *service* definiuje swoją zależność od komponentu *eventSenderSelectorService*, kontener Springa tworzy egzemplarz komponentu *eventSenderSelectorService* (odpowiadającego klasie *EventSenderSelectorServiceImpl*) *przed* egzemplarzem komponentu *service* (odpowiadającego klasie *FixedDepositServiceImpl*).



Jeżeli wywołasz metodę *main()* klasy *BankApp* w projekcie *ch04-bankapp-dependencies*, zobaczysz, że egzemplarz *FixedDepositServiceImpl* jest tworzony *przed* egzemplarzem *EventSenderSelectorServiceImpl*. Dlatego w konsoli zostanie wyświetlony komunikat o treści Plik *appConfig.properties* nie zawiera informacji o klasie *EventSender*.

Wiele niejawnych zależności

Jeżeli komponent ma wiele niejawnych zależności, identyfikatory lub nazwy ich wszystkich można podać jako wartość atrybutu *depends-on*, jak przedstawia listing 4.8.

Listing 4.8. Przykład użycia atrybutu *depends-on* wraz z wieloma niejawnymi zależnościami

```
<beans .....>
  <bean id="abean" ..... depends-on="bBean, cBean">
    .....
  </bean>
  .....
</beans>
```

W powyższym listingu pokazano możliwość podania wielu identyfikatorów lub nazw komponentów jako wartości atrybutu *depends-on*.

Atrybut *depends-on* i dziedziczenie definicji komponentu

Trzeba koniecznie zwrócić uwagę na fakt, że atrybut *depends-on* *nie* jest dziedziczny przez definicje komponentów potomnych. W listingu 4.9 została pokazana definicja *abstrakcyjnego* komponentu nadrzędnego *serviceTemplate*, która używa atrybutu *depends-on* w celu wskazania *baseService* jako zależności.

Listing 4.9. Atrybut *depends-on* i dziedziczenie definicji komponentu

```
<bean id="serviceTemplate" class=".....ServiceTemplate" depends-on="baseService"
    abstract="true"/>

<bean id="someService" class=".....SomeServiceImpl" parent="serviceTemplate"/>

<bean id="someOtherService" class=".....SomeOtherServiceImpl"
    parent="serviceTemplate"/>

<bean id="baseService" class=".....BaseServiceImpl" />
```

W powyższym listingu definicje komponentów potomnych `someService` i `someOtherService` nie dziedziczą atrybutu `depends-on` po definicji komponentu nadrzędnego `serviceTemplate`. Ponieważ kontener Springa tworzy komponenty w kolejności ich zdefiniowania w pliku XML kontekstu aplikacji, więc komponent `baseService` zostaje utworzony *po* komponentach `someService` i `someOtherService`.

Zobaczmy teraz, w jaki sposób kontener Springa zajmuje się zarządzaniem komponentami o zasięgu singleton i prototypu.

4.4. Zależności w postaci komponentów o zasięgu singleton i prototypu

Komponent o zasięgu singleton (i jego zależności o takim samym zasięgu) jest tworzony w chwili powstawania egzemplarza `ApplicationContext`. Natomiast komponent o zasięgu prototypu (i jego zależności o takim samym zasięgu) jest tworzony w trakcie każdego wywołania metody `getBean()` klasy `ApplicationContext` mającego na celu pobranie komponentu o zasięgu prototypu.

Jeżeli komponent o zasięgu singleton jest zależny od komponentu o zasięgu prototypu lub na odwrót, wówczas sprawy nieco się komplikują. Na przykład jeśli komponent o zasięgu singleton zależy od komponentu o zasięgu prototypu, możesz zadawać sobie pytanie, czy kontener Springa utworzy komponent o zasięgu prototypu (zależność) *przed* komponentem o zasięgu singleton (komponent zależny), czy może kontener Springa utworzy i wstrzyknie komponent o zasięgu prototypu tylko wtedy, gdy nastąpi wywołanie metody `getBean()` klasy `ApplicationContext` mające na celu pobranie egzemplarza komponentu o zasięgu singleton. Odpowiedzi na to pytanie kryją się w sposobie zarządzania przez kontener Springa zależnościami, co omówimy w dalszej części rozdziału.

Zależności komponentu o zasięgu singleton

Listing 4.10 przedstawia istniejący w aplikacji `MyBank` komponent `customerRequestService` o zasięgu singleton i jego zależności.

Listing 4.10. Plik `applicationContext.xml` przedstawiający zależności komponentu `customerRequestService`

Projekt: `ch04-bankapp-dependencies`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="customerRequestService"
    class="sample.spring.chapter04.bankapp.service.CustomerRequestServiceImpl">
    <constructor-arg name="customerRequestDetails" ref="customerRequestDetails" />
```

```

    <constructor-arg name="customerRequestDao" ref="customerRequestDao" />
</bean>

<bean id="customerRequestDetails"
      class="sample.spring.chapter04.bankapp.domain.CustomerRequestDetails"
      scope="prototype" />

<bean id="customerRequestDao"
      class="sample.spring.chapter04.bankapp.dao.CustomerRequestDaoImpl" />

```

W powyższym listingu widać, że komponent `customerRequestService` (o zasięgu singleton) jest zależny od komponentów `customerRequestDetails` (o zasięgu prototypu) i `customerRequestDao` (o zasięgu singleton). Obiekt `CustomerRequestService` (przedstawiany przez komponent `customerRequestService`) definiuje usługę wywoływaną, gdy klient banku generuje nowe żądanie, na przykład książeczki czekowej. Szczegóły dotyczące żądania wygenerowanego przez klienta obiekt `CustomerRequestService` umieszcza w obiekcie `CustomerRequestDetails` (przedstawiany przez komponent `customerRequestDetails`) oraz za pomocą obiektu `CustomerRequestDao` (przedstawiany przez komponent `customerRequestDao`) zapisuje w magazynie danych.

W listingu 4.11 została przedstawiona metoda `main()` klasy `BankApp`, która wczytuje definicje komponentu pokazane w listingu 4.10.

Listing 4.11. Klasa `BankApp`

Projekt: `ch04-bankapp-dependencies`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp`

```

package sample.spring.chapter04.bankapp;

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

public class BankApp {
    private static Logger logger = Logger.getLogger(BankApp.class);

    public static void main(String args[]) throws Exception {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");
        ....
        logger.info("Rozpoczęcie pracy z CustomerRequestService");
        CustomerRequestService customerRequestService_1
            = context.getBean(CustomerRequestService.class);
        ....
        CustomerRequestService customerRequestService_2
            = context.getBean(CustomerRequestService.class);
        ....
        logger.info("Zakończenie pracy z CustomerRequestService");
    }
}

```

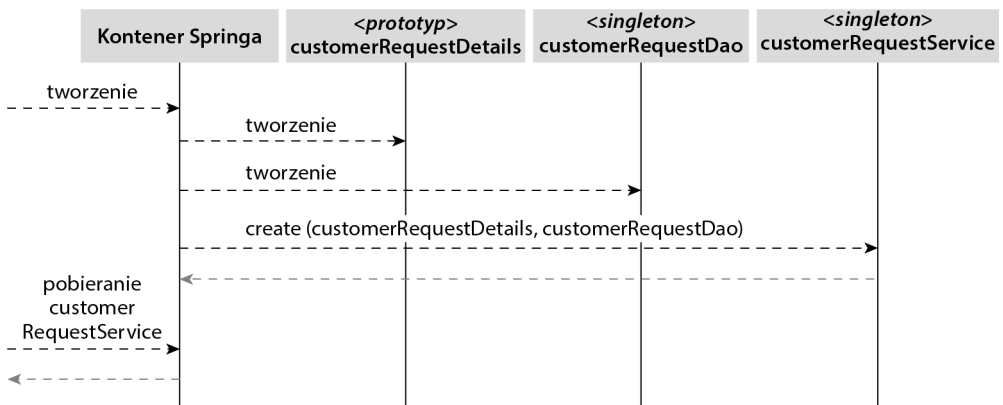
W powyższym listingu widać, że po utworzeniu egzemplarza `ApplicationContext` następuje dwukrotne wywołanie metody `getBean()` klasy `ApplicationContext` w celu pobrania odwołania do komponentu `customerRequestService`.

Jeżeli wykonasz metodę `main()` klasy `BankApp`, otrzymasz następujące dane wyjściowe:

```
Utworzono egzemplarz CustomerRequestDetails
Utworzono egzemplarz CustomerRequestDaoImpl
Utworzono egzemplarz CustomerRequestServiceImpl
.....
Rozpoczęcie pracy z CustomerRequestService
Zakończenie pracy z CustomerRequestService
```

Widoczne w powyższych danych wyjściowych komunikaty `Utworzono...` są wyświetlane przez konstruktory wskazanych klas. Na podstawie powyższych danych wyjściowych możemy stwierdzić, że zależności `customerRequestDetails` (o zasięgu prototypu) i `customerRequestDao` (o zasięgu singleton) komponentu `customerRequestService` (o zasięgu singleton) są tworzone i wstrzykiwane do egzemplarza `customerRequestService` podczas tworzenia kontenera Springa. Ponieważ żaden komunikat `Utworzono...` nie pojawia się w konsoli między komunikatami `Rozpoczęcie...` i `Zakończenie...`, kontener Springa nie tworzy żadnych egzemplarzy podczas wywoływania metody `getBean()` klasy `ApplicationContext` w celu pobrania `customerRequestService`.

Diagram na rysunku 4.2 pokazuje sekwencję zdarzeń zachodzących podczas wykonywania metody `main()` klasy `BankApp` (patrz: listing 4.11). Jak możesz zobaczyć, podczas tworzenia kontenera Springa następuje utworzenie najpierw komponentów `customerRequestDetails` (o zasięgu prototypu) i `customerRequestDao` (o zasięgu singleton), a dopiero potem `customerRequestService` (o zasięgu singleton). Oparty na konstruktorze mechanizm wstrzykiwania zależności został użyty w celu wstrzyknięcia komponentów `customerRequestDetails` i `customerRequestDao` do `customerRequestService`. Ponieważ komponent o zasięgu singleton jest przez kontener Springa tworzony *tylko jeden raz*, kontener Springa ma *tylko jedną* szansę na wstrzyknięcie zależności komponentu `customerRequestService`. Dlatego też kontener Springa *tylko raz* wstrzykuje egzemplarz `customerRequestDetails` do egzemplarza `customerRequestService`. Implikacją takiego zachowania jest to, że komponent `customerRequestService` w ciągu całego swojego cyklu życiowego zawiera odwołanie do *tego samego* komponentu `customerRequestDetails`.



Rysunek 4.2. Sekwencja zdarzeń zachodzących podczas tworzenia kontenera Springa i pobierania z niego komponentu `customerRequestService`

Trzeba koniecznie zwrócić uwagę na następujący fakt: nawet jeśli oparty na metodzie typu setter mechanizm wstrzykiwania zależności zostanie użyty w celu wstrzyknięcia `customerRequestDetails` (o zasięgu prototypu) do komponentu `customerRequestService`, to w trakcie cyklu życiowego komponentu `customerRequestService` kontener Springa *tylko raz* wywoła metodę typu setter. Oznacza to, że niezależnie od rodzaju użytego mechanizmu wstrzykiwania zależności (opartego na metodzie typu setter lub konstruktorze) komponent o zasięgu singleton jest w trakcie cyklu życiowego tworzony i konfigurowany tylko jednokrotnie.

Teraz po utworzeniu kontenera Springa każde żądanie komponentu `customerRequestService` (o zasięgu singleton) będzie zwracało ten sam buforowany egzemplarz komponentu `customerRequestService`. Z tego powodu po wykonaniu metody `main()` klasy `BankApp` (patrz: listing 4.11) w konsoli między komunikatami *Rozpoczęcie...* i *Zakończenie...* nie pojawia się żaden komunikat *Utworzono...*

Ponieważ komponent `customerRequestService` (o zasięgu singleton) zawsze przechowuje odwołanie do tego *samego* komponentu `customerRequestDetails` (o zasięgu prototypu), to może mieć wpływ na sposób działania aplikacji `MyBank`. Na przykład jeśli wielu klientów jednocześnie wykonuje żądania do egzemplarza `CustomerRequestServiceImpl`, wszystkie te żądania będą modyfikować ten sam obiekt `CustomerRequestDetails` przechowywany przez `CustomerRequestService`. W idealnej sytuacji klasa `CustomerRequestServiceImpl` powinna tworzyć nowy egzemplarz obiektu `CustomerRequestDetails` dla każdego żądania. W podrozdziale 4.5 poznasz modyfikacje, jakie trzeba wprowadzić w klasie komponentu o zasięgu singleton, aby w trakcie każdego wywołania metody następowało pobieranie nowego egzemplarza komponentu o zasięgu prototypu.

Przekonajmy się teraz, jak kontener Springa zarządza zależnościami komponentu o zasięgu prototypu.

Zależności komponentu o zasięgu prototypu

W aplikacji `MyBank` klient przeprowadza rejestrację, wykonując sekwencję kroków. Przede wszystkim musi podać informacje o sobie i o posiadanym koncie. Jeżeli aplikacja `MyBank` znajdzie dopasowany rekord, wówczas klient zostanie zapytany o szczegóły dotyczące karty debetowej. Klasa `CustomerRegistrationServiceImpl` aplikacji `MyBank` zawiera niezbędną do rejestrowania klientów logikę biznesową. Ponieważ rejestrując się w aplikacji klienci wykonują pewną sekwencję kroków, obiekt `CustomerRegistrationServiceImpl` zachowuje stan między wywołaniami metod.

W listingu 4.12 przedstawiono komponent `customerRegistrationService` (o zasięgu prototypu) i jego zależności. Ten komponent przedstawia klasę `CustomerRegistrationServiceImpl`.

Listing 4.12. *Plik `applicationContext.xml` przedstawiający komponent `customerRegistrationService` i jego zależności*

Projekt: `ch04-bankapp-dependencies`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="customerRegistrationService"
      class="sample.spring.chapter04.bankapp.service.CustomerRegistrationServiceImpl"
      scope="prototype">
  <constructor-arg name="customerRegistrationDetails"
    ref="customerRegistrationDetails" />
</bean>
```

```
<constructor-arg name="customerRegistrationDao" ref="customerRegistrationDao" />
</bean>
<bean id="customerRegistrationDetails"
      class="sample.spring.chapter04.bankapp.domain.CustomerRegistrationDetails"
      scope="prototype" />

<bean id="customerRegistrationDao"
      class="sample.spring.chapter04.bankapp.dao.CustomerRegistrationDaoImpl" />
```

Powyższy listing pokazuje, że komponent `customerRegistrationService` (o zasięgu prototypu) zależy od komponentów `customerRegistrationDetails` (o zasięgu prototypu) i `customerRegistrationDao` (o zasięgu singleton).

Egzemplarz `CustomerRegistrationServiceImpl` zawiera informacje o postępie procesu rejestracji i przechowuje w obiekcie `CustomerRegistrationDetails` (przedstawiany przez komponent `customerRegistrationDetails`) dane dostarczone przez klienta w trakcie rejestracji. Ponieważ obiekty `CustomerRegistrationServiceImpl` i `CustomerRegistrationDetails` z natury przechowują informacje o stanie, `customerRegistrationService` i `customerRegistrationDetails` zostały zdefiniowane jako komponenty o zasięgu prototypu.

W listingu 4.13 przedstawiamy metodę `main()` klasy `BankApp`, która wczytuje komponenty związane z rejestracją (patrz: listing 4.12) i przeprowadza rejestrację dla dwóch klientów.

Listing 4.13. Klasa `BankApp`

Projekt: `ch04-bankapp-dependencies`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp`

```
package sample.spring.chapter04.bankapp;

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

public class BankApp {
    private static Logger logger = Logger.getLogger(BankApp.class);

    public static void main(String args[]) throws Exception {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");
        ....
        logger.info("Rozpoczęcie pracy z CustomerRegistrationService");

        CustomerRegistrationService customerRegistrationService_1 = context
            .getBean(CustomerRegistrationService.class);
        customerRegistrationService_1.setAccountNumber("account_1");
        customerRegistrationService_1.setAddress("address_1");
        customerRegistrationService_1.setDebitCardNumber("debitCardNumber_1");
        customerRegistrationService_1.register();
        logger.info("Zarejestrowano klienta o identyfikatorze account_1");

        CustomerRegistrationService customerRegistrationService_2 = context
            .getBean(CustomerRegistrationService.class);
        ....
        logger.info("Zarejestrowano klienta o identyfikatorze account_2");
```

```

        logger.info("Zakończenie pracy z CustomerRegistrationService");
    }
}

```

W powyższym listingu widać, że metoda `main()` klasy `BankApp` dwukrotnie wywołuje metodę `getBean()` klasy `ApplicationContext` w celu pobrania odwołania do komponentu `customerRegistrationService`. Po otrzymaniu egzemplarza komponentu `customerRegistrationService` następuje wywołanie w nim metod `setAccountNumber()`, `setAddress()`, `setDebitCardNumber()` i `register()`. Jeżeli wykonasz metodę `main()` klasy `BankApp`, wówczas w konsoli otrzymasz następujące dane wyjściowe:

```

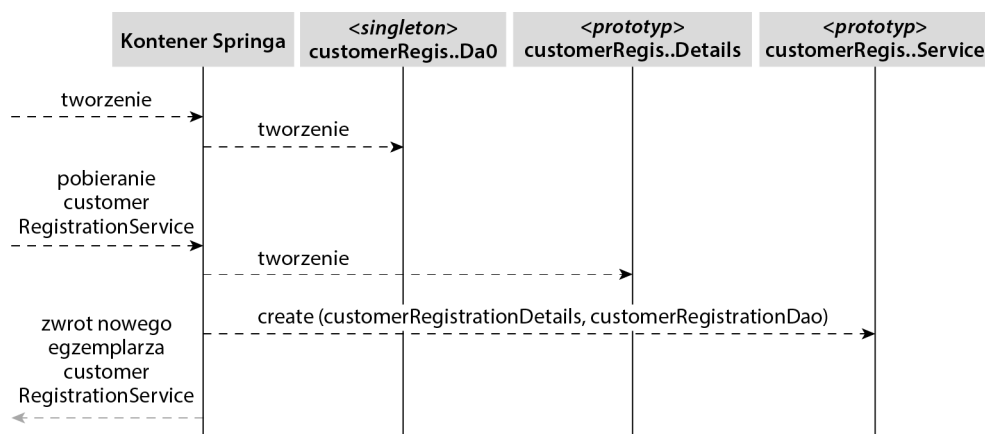
Utworzono egzemplarz CustomerRegistrationDaoImpl
.....
Rozpoczęcie pracy z CustomerRegistrationService
Utworzono egzemplarz CustomerRegistrationDetails
Utworzono egzemplarz CustomerRegistrationServiceImpl
Zarejestrowano klienta o identyfikatorze account_1
Utworzono egzemplarz CustomerRegistrationDetails
Utworzono egzemplarz CustomerRegistrationServiceImpl
Zarejestrowano klienta o identyfikatorze account_2
Zakończenie pracy z CustomerRegistrationService

```

Widoczne w powyższych danych wyjściowych komunikaty `Utworzono...` są wyświetlane przez konstruktory wskazanych klas. Na podstawie powyższych danych wyjściowych możemy stwierdzić, że komponent o zasięgu singleton i nazwie `customerRegistrationDao` (przedstawiający klasę `CustomerRegistrationDaoImpl`) jest tworzony tylko w chwili powstawania egzemplarza `ApplicationContext`. Komunikaty `Utworzono...` znajdujące się między komunikatami `Rozpoczęcie...` i `Zakończenie...` wskazują, że w trakcie każdego wywołania metody `getBean()` klasy `ApplicationContext` w celu pobrania komponentu `customerRegistrationService` kontener Springa tworzy nowy egzemplarz komponentu `customerRegistrationService` i jego zależność o zasięgu prototypu (komponent `customerRegistrationDetails`).

Diagram na rysunku 4.3 pokazuje sekwencję zdarzeń zachodzących podczas wykonywania metody `main()` klasy `BankApp` (patrz: listing 4.13). Jak możesz zobaczyć, komponent `customerRegistrationDao` o zasięgu singleton jest tworzony *tylko raz* podczas operacji tworzenia egzemplarza `ApplicationContext`. Gdy z kontenera Springa żądany jest komponent `customerRegistrationService` o zasięgu prototypu, wówczas kontener Springa najpierw tworzy egzemplarz komponentu `customerRegistrationDetails` (będący zależnością o zasięgu prototypu komponentu `customerRegistrationService`), a dopiero później komponent `customerRegistrationService`. W ten sposób jeżeli komponent `X` o zasięgu prototypu jest zależny od komponentu `Y` również o zasięgu prototypu, kontener Springa utworzy nowe egzemplarza `X` i `Y` w trakcie każdego żądania komponentu `X` z kontenera Springa.

We wcześniejszej części rozdziału zobaczyłeś, że jeśli komponent o zasięgu singleton jest zależny od komponentu o zasięgu prototypu, wówczas w trakcie jego całego cyklu życiowego komponent o zasięgu singleton będzie powiązany z tym *samym* egzemplarzem komponentu o zasięgu prototypu. Teraz poznamy różne sposoby, na jakie komponent o zasięgu singleton może pobierać z kontenera Springa nowy egzemplarz komponentu o zasięgu prototypu.



Rysunek 4.3. Sekwencja zdarzeń zachodzących podczas tworzenia kontenera Springa i pobierania z niego komponentu `customerRegistrationService`

4.5. Pobieranie w komponencie singleton nowego egzemplarza komponentu o zasięgu prototypu

W poprzednim podrozdziale zobaczyłeś, że zależności o zasięgu prototypu komponentu o zasięgu singleton są wstrzykiwane w chwili tworzenia komponentu o zasięgu singleton (patrz: rysunek 4.2). Kontener Springa tylko *jeden raz* tworzy egzemplarz komponentu o zasięgu singleton, a więc ten komponent w trakcie całego cyklu życiowego przechowuje odwołanie do *tego samego* komponentu o zasięgu prototypu. Metody komponentu o zasięgu singleton mogą pobierać z kontenera Springa nowy egzemplarz zależności o zasięgu prototypu, korzystając z dowolnego z poniższych podejść:

- Implementacja interfejsu `ApplicationContextAware` przez klasę komponentu o zasięgu singleton.
- Użycie elementu `<lookup-method>` schematu beans oferowanego przez Spring.
- Użycie elementu `<replaced-method>` schematu beans oferowanego przez Spring.



Istnieje możliwość wykorzystania słowa kluczowego `new` w celu utworzenia egzemplarza klasy komponentu o zasięgu prototypu w metodzie komponentu o zasięgu singleton, a następnie użycia nowego egzemplarza. Ponieważ odpowiedzialność za utworzenie egzemplarza komponentu spoczywa na kontenerze Springa, *nie* powinniśmy próbować bezpośrednio tworzyć egzemplarza komponentu za pomocą słowa kluczowego `new`.

IMPORT **Rozdział104/ch04-bankapp-context-aware.** (Ten projekt pokazuje scenariusz, w którym komponent o zasięgu singleton implementuje interfejs `ApplicationContextAware` w celu pobierania z kontenera Springa egzemplarzy komponentów o zasięgu prototypu. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Na początek zapoznamy się z interfejsem `ApplicationContextAware`.

Interfejs ApplicationContextAware

Oferowany przez Spring interfejs `ApplicationContextAware` jest implementowany przez komponenty wymagające uzyskania dostępu do egzemplarza `ApplicationContext`, w którym działają. Interfejs `ApplicationContextAware` definiuje pojedynczą metodę o nazwie `setApplicationContext()`, która implementującym go komponentom zapewnia dostęp do egzemplarza obiektu `ApplicationContext`.

Interfejs `ApplicationContextAware` jest *interfejsem cyklu życiowego*, co oznacza, że kontener Springa komponenty implementujące interfejs `ApplicationContextAware` wywołuje w odpowiednich momentach ich cyklu życiowego. Na przykład metoda `setApplicationContext()` klasy `ApplicationContextAware` jest przez kontener Springa wywoływana *po* utworzeniu egzemplarza komponentu, ale jeszcze *przed* jego pełnym zainicjowaniem. Egzemplarz komponentu zostanie uznany za zainicjowany w pełni dopiero po wywołaniu jego *metody inicjacyjnej* (patrz: podrozdział 5.2) przez kontener Springa. Trzeba koniecznie zwrócić uwagę na fakt, że po przeprowadzeniu pełnej inicjalizacji egzemplarza komponentu zostanie on przez kontener Springa wstrzyknięty do egzemplarza komponentów zależnych. W rozdziale 5. dokładnie zajmiemy się tematem cyklu życiowego interfejsów w Springu.

Komponent implementujący interfejs `ApplicationContextAware` może uzyskać dostęp do innych komponentów zarejestrowanych przez egzemplarz `ApplicationContext` dzięki wywołaniu metody `getBean()` klasy `ApplicationContext`. Oznacza to, że jeśli klasa komponentu o zasięgu singleton implementuje interfejs `ApplicationContextAware`, to z kontenera Springa może pobrać nowy egzemplarz komponentu o zasięgu prototypu przez wywołanie metody `getBean()` klasy `ApplicationContext`. Ponieważ komponent o zasięgu singleton jawnie pobiera z kontenera Springa zależność w postaci komponentu o zasięgu prototypu, używając wywołania metody `getBean()` klasy `ApplicationContext`, nie ma konieczność definiowania komponentu zależności w pliku XML kontekstu aplikacji.

W listingu 4.14 została przedstawiona klasa `CustomerRequestServiceImpl` wymagająca nowego egzemplarza obiektu `CustomerRequestDetails` w trakcie każdego wywołania metody `submitRequest()` klasy `CustomerRequestServiceImpl`.

Listing 4.14. Klasa `CustomerRequestServiceImpl`

Projekt: `ch04-bankapp-context-aware`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp/service`

```
package sample.spring.chapter04.bankapp.service;

import sample.spring.chapter04.bankapp.dao.CustomerRequestDao;
import sample.spring.chapter04.bankapp.domain.CustomerRequestDetails;

public class CustomerRequestServiceImpl implements CustomerRequestService {
    private CustomerRequestDetails customerRequestDetails;
    private CustomerRequestDao customerRequestDao;

    @ConstructorProperties({ "customerRequestDetails", "customerRequestDao" })
    public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails,
        CustomerRequestDao customerRequestDao) {
        this.customerRequestDetails = customerRequestDetails;
        this.customerRequestDao = customerRequestDao;
    }
}
```



```

    }

    public void submitRequest(String requestType, String requestDescription) {
        // Wypełnienie obiektu CustomerRequestDetails i jego zapisanie.
        customerRequestDetails.setType(requestType);
        customerRequestDetails.setDescription(requestDescription);
        customerRequestDao.submitRequest(customerRequestDetails);
    }
}

```

W powyższym listingu obiekty `CustomerRequestDetails` i `CustomerRequestDao` są przekazywane jako argumenty konstruktorowi klasy `CustomerRequestServiceImpl`. Metoda `submitRequest()` wypełnia egzemplarz `CustomerRequestDetails` i zapisuje go w bazie danych, wywołując metodę `submitRequest()` klasy `CustomerRequestDao`. Jeżeli wielu klientów jednocześnie próbuje wykonać żądanie, wówczas metoda `submitRequest()` będzie modyfikować ten sam egzemplarz obiektu `CustomerRequestDetails`, a skutkiem będzie nieoczekiwane zachowanie aplikacji `MyBank`. Aby rozwiązać ten problem, w trakcie każdego wywołania metoda `submitRequest()` musi pobierać nowy egzemplarz obiektu `CustomerRequestDetails` z kontenera Springa.

W listingu 4.15 została przedstawiona klasa `CustomerRequestServiceContextAwareImpl` (to jest zmodyfikowana wersja klasy `CustomerRequestServiceImpl` pokazanej w listingu 4.14), która implementuje interfejs `ApplicationContextAware`.

Listing 4.15. Klasa `CustomerRequestServiceContextAwareImpl` implementująca oferowany przez Spring interfejs `ApplicationContextAware`

Projekt: `ch04-bankapp-context-aware`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp/service`

```

package sample.spring.chapter04.bankapp.service;

import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class CustomerRequestServiceContextAwareImpl implements
    CustomerRequestService, ApplicationContextAware {

    private CustomerRequestDao customerRequestDao;
    private ApplicationContext applicationContext;

    @ConstructorProperties({ "customerRequestDao" })
    public CustomerRequestServiceContextAwareImpl(CustomerRequestDao customerRequestDao)
    {
        this.customerRequestDao = customerRequestDao;
    }

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

    public void submitRequest(String requestType, String requestDescription) {
        CustomerRequestDetails customerRequestDetails = applicationContext
            .getBean(CustomerRequestDetails.class);
        customerRequestDetails.setType(requestType);
    }
}

```

```

        customerRequestDetails.setDescription(requestDescription);
        customerRequestDao.submitRequest(customerRequestDetails);
    }
}

```

W powyższym listingu metoda `setApplicationContext()` dostarcza klasę `CustomerRequestService` ↪ `ContextAwareImpl` wraz z egzemplarzem obiektu `ApplicationContext`. Egzemplarz `ApplicationContext` ↪ `Context` jest następnie używany przez metodę `submitRequest()` w celu pobrania z kontenera Springa egzemplarza obiektu `CustomerRequestDetails`.

Ponieważ klasa `CustomerRequestServiceContextAwareImpl` jawnie pobiera z kontenera Springa obiekt `CustomerRequestDetails`, *nie* ma konieczności użycia oferowanego przez Spring mechanizmu wstrzykiwania zależności, aby wstrzyknąć egzemplarz `CustomerRequestDetails` do egzemplarza `CustomerRequestServiceContextAwareImpl`. Z tego powodu konstruktor klasy `CustomerRequestService` ↪ `ContextAwareImpl` (patrz: listing 4.15) nie wymaga określenia argumentu w postaci obiektu `CustomerRequestDetails`. Jeżeli teraz przejdiesz do projektu *ch04-bankapp-context-aware* i wykonasz metodę `main()` klasy `BankApp`, zobaczysz, że w trakcie każdego wywołania metody `submitRequest()` następuje pobranie z kontenera Springa nowego egzemplarza obiektu `CustomerRequestDetails`.

W kontekście aplikacji `MyBank` przekonaliśmy się, że interfejs `ApplicationContextAware` okazuje się użyteczny, jeżeli komponent wymaga uzyskania dostępu do innych komponentów. Wadą implementacji interfejsu `ApplicationContextAware` jest powstanie powiązania między klasą komponentu i Springiem. Istnieje możliwość uniknięcia powstania powiązania między klasami komponentów i Springiem przy jednoczesnym zachowaniu dostępu do innych komponentów z poziomu kontenera Springa. Ta alternatywa polega na użyciu technik *wstrzykiwania metod* oferowanych przez elementy `<lookup-method>` i `<replaced-method>` schematu beans w Springu.

Najpierw przeanalizujemy element `<lookup-method>`.

IMPORT **Rozdział 104/ch04-bankapp-lookup-method.** (Ten projekt pokazuje aplikację `MyBank` używającą elementu `<lookup-method>` schematu beans oferowanego przez Spring. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Element `<lookup-method>`

Jeżeli klasa komponentu definiuje *metodę wyszukiwania komponentu*, której typ wartości zwrótej przedstawia komponent, wówczas element `<lookup-method>` nakazuje kontenerowi Springa dostarczenie implementacji wspomnianej metody. Dostarczana przez kontener Springa implementacja metody jest odpowiedzialna za pobieranie egzemplarza komponentu z kontenera Springa i jego zwrot.

Atrybut bean elementu `<lookup-method>` wskazuje nazwę wyszukiwanego komponentu, który zostanie zwrócony przez implementację metody. Natomiast atrybut `name` określa nazwę metody, której implementacja będzie dostarczona przez kontener Springa. Trzeba koniecznie zapamiętać, że metoda wyszukiwania zdefiniowana przez klasę komponentu może być abstrakcyjna lub konkretna.



Użycie elementu `<lookup-method>` w celu nakazania kontenerowi Springa dostarczenia implementacji dla metody wyszukiwania komponentu jest określane mianem *techniki wstrzykiwania metody*, ponieważ implementacja metody wyszukiwania jest przez element `<lookup-method>` wstrzykiwana do klasy komponentu.

W listingu 4.16 została przedstawiona metoda *abstrakcyjna* `getCustomerRequestDetails()` klasy `CustomerRequestServiceImpl`. Wartością zwrótną tej metody jest egzemplarz `CustomerRequestDetails`.

Listing 4.16. Klasa `CustomerRequestServiceImpl` definiująca metodę wyszukiwania komponentu

Projekt: `ch04-bankapp-lookup-method`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp/service`

```
package sample.spring.chapter04.bankapp.service;

public abstract class CustomerRequestServiceImpl implements CustomerRequestService {
    private CustomerRequestDao customerRequestDao;

    @ConstructorProperties({ "customerRequestDao" })
    public CustomerRequestServiceImpl(CustomerRequestDao customerRequestDao) {
        this.customerRequestDao = customerRequestDao;
    }

    public abstract CustomerRequestDetails getCustomerRequestDetails();

    @Override
    public void submitRequest(String requestType, String requestDescription) {
        // Wypełnienie obiektu CustomerRequestDetails i jego zapisanie.
        CustomerRequestDetails customerRequestDetails = getCustomerRequestDetails();
        .....
    }
}
```

W powyższym listingu widać, że klasa `CustomerRequestServiceImpl` została zdefiniowana jako abstrakcyjna, ponieważ zawiera abstrakcyjną metodę wyszukiwania komponentu `getCustomerRequestDetails()`. Można zdefiniować `getCustomerRequestDetails()` nie jako abstrakcyjną, lecz jako konkretną metodę. Natomiast metoda `submitRequest()` wywołuje metodę `getCustomerRequestDetails()` w celu uzyskania dostępu do egzemplarza `CustomerRequestDetails`.

W listingu 4.17 zostały przedstawione definicje komponentów dla klas `CustomerRequestServiceImpl` i `CustomerRequestDetails`.

Listing 4.17. Plik `applicationContext.xml` pokazujący sposób użycia elementu `<lookup-method>`

Projekt: `ch04-bankapp-lookup-method`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="customerRequestService">
    class="sample.spring.chapter04.bankapp.service.CustomerRequestServiceImpl">
    <constructor-arg name="customerRequestDao" ref="customerRequestDao" />
    <lookup-method bean="customerRequestDetails" name="getCustomerRequestDetails"/>
</bean>
```

```
<bean id="customerRequestDetails"  
      class="sample.spring.chapter04.bankapp.domain.CustomerRequestDetails"  
      scope="prototype" />
```

W powyższym listingu widać, że definicja dla klasy `CustomerRequestServiceImpl` zawiera element `<lookup-method>`. Wartością atrybutu `name` elementu `<lookup-method>` jest `getCustomerRequestDetails()`. Dlatego kontener Springa dostarczy implementację dla metody wyszukiwania `getCustomerRequestDetails()` klasy `CustomerRequestServiceImpl`, jak pokazano w listingu 4.16. Wartością atrybutu `bean` elementu `<lookup-method>` jest `customerRequestDetails`, co oznacza, że implementacja metody `getCustomerRequestDetails()` pobierze z kontenera Springa komponent o identyfikatorze (nazwie) `customerRequestDetails` i zwróci go wywołującej metodzie. Ponieważ komponent `customerRequestDetails` przedstawia obiekt `CustomerRequestDetails` (patrz: definicja komponentu `customerRequestDetails` w listingu 4.17), wartością zwrótną implementacji metody `getCustomerRequestDetails()` będzie obiekt `CustomerRequestDetails`.

W listingu 4.16 metoda `submitRequest()` klasy `CustomerRequestService` wywołuje metodę wyszukiwania `getCustomerRequestDetails()` w celu pobrania egzemplarza `CustomerRequestDetails`. Ponieważ klasa `CustomerRequestDetails` jest w pliku XML kontekstu aplikacji przedstawiona jako komponent o zasięgu prototypu (patrz: listing 4.17), każde wywołanie metody `submitRequest()` spowoduje pobranie z kontenera Springa nowego egzemplarza obiektu `CustomerRequestDetails`.

Aby upewnić się o dostarczaniu przez element `<lookup-method>` odpowiedniej implementacji metody wyszukiwania `getCustomerRequestDetails()` klasy `CustomerRequestService`, w metodzie `main()` klasy `BankApp` z kontenera Springa pobieramy egzemplarz `CustomerRequestService` i wielokrotnie wywołujemy jego metodę `submitRequest()`. Jeżeli skutkiem każdego wywołania metody `submitRequest()` będzie pobranie z kontenera Springa nowego egzemplarza obiektu `CustomerRequestDetails`, oznacza to, że element `<lookup-method>` dostarcza prawidłowej implementacji metody `getCustomerRequestDetails()` klasy `CustomerRequestService`.

W listingu 4.18 została przedstawiona metoda `main()` klasy `BankApp`, która wywołuje metodę `submitRequest()` klasy `CustomerRequestService`.

Listing 4.18. Klasa *BankApp*

Projekt: *ch04-bankapp-lookup-method*

Położenie pliku: *src/main/java/sample/spring/chapter04/bankapp*

```
package sample.spring.chapter04.bankapp;  
.....  
public class BankApp {  
    private static Logger logger = Logger.getLogger(BankApp.class);  
  
    public static void main(String args[]) throws Exception {  
        ApplicationContext context = new ClassPathXmlApplicationContext(  
            "classpath:META-INF/spring/applicationContext.xml");  
        .....  
        logger.info("Rozpoczęcie pracy z CustomerRequestService");  
        CustomerRequestService customerRequestService_1 = context  
            .getBean(CustomerRequestService.class);  
        customerRequestService_1.submitRequest("checkBookRequest",
```

```
"Request to send a 50-leaf check book");
customerRequestService_1.submitRequest("checkBookRequest",
    "Request to send a 100-leaf check book");
.....
logger.info("Zakończenie pracy z CustomerRequestService");
}
}
```

Jeżeli wykonasz metodę `main()` klasy `BankApp`, wówczas w konsoli otrzymasz następujące dane wyjściowe:

```
Rozpoczęcie pracy z CustomerRequestService
Utworzono egzemplarz CustomerRequestDetails
Utworzono egzemplarz CustomerRequestDetails
.....
Zakończenie pracy z CustomerRequestService
```

Widoczne w powyższych danych wyjściowych komunikaty `Utworzono...` są wyświetlane przez konstruktory odpowiednich klas komponentów. Na podstawie pokazanych danych można stwierdzić, że każde wywołanie metody `submitRequest()` klasy `CustomerRequestService` powoduje pobranie z kontenera Springa nowego egzemplarza `CustomerRequestService`.

Ponieważ implementacja metody wyszukiwania komponentu jest dostarczana przez kontener Springa, względem sygnatury tego rodzaju metody istnieją pewne ograniczenia. Na przykład metoda wyszukiwania *musi* być zdefiniowana jako publiczna (`public`) lub chroniona (`protected`), a ponadto *nie może* akceptować żadnych argumentów. W trakcie działania aplikacji kontener Springa tworzy podklasy klasy komponentu zawierającej metodę wyszukiwania w celu dostarczenia implementacji metody wyszukiwania, więc klasa komponentu i znajdująca się w niej metoda wyszukiwania *nie mogą* być zdefiniowane jako `final`.



W trakcie działania aplikacji kontener Springa musi mieć możliwość tworzenia podklas klasy komponentu zawierającej metodę wyszukiwania, aby dostarczyć implementację wspomnianej metody wyszukiwania, dlatego kontener Springa używa biblioteki CGLIB (<https://github.com/cglib/cglib>) do utworzenia klas potomnych. Począwszy od Springa w wersji 3.2, klasy CGLIB są umieszczane w pliku JAR o nazwie `spring-core` i dlatego biblioteki nie trzeba wyraźnie definiować jako zależności projektu.

Element `<lookup-method>` zapewnia obsługę techniki wstrzykiwania metody polegającej na tym, że klasa komponentu definiuje metodę wyszukiwania, której implementacja jest dostarczana przez kontener Springa. W celu przeprowadzenia operacji wstrzyknięcia metody możesz rozważyć użycie elementu `<replaced-method>` schematu beans oferowanego przez Spring zamiast elementu `<lookup-method>`.

IMPORT

Rozdział04/ch04-bankapp-replaced-method. (Ten projekt pokazuje aplikację `MyBank` używającą elementu `<replaced-method>` schematu beans oferowanego przez Spring. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Element <replaced-method>

Element <replaced-method> pozwala na zastąpienie dowolnej metody w klasie komponentu zupełnie inną implementacją. Listing 4.19 przedstawia klasę `CustomerRequestServiceImpl`, której będziemy używać jako przykładu podczas demonstracji sposobu wykorzystania elementu <replaced-method>.

Listing 4.19. Klasa `CustomerRequestServiceImpl`

Projekt: `ch04-bankapp-replaced-method`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp/service`

```
package sample.spring.chapter04.bankapp.service;
.....
public class CustomerRequestServiceImpl implements CustomerRequestService {
    private CustomerRequestDao customerRequestDao;
    .....
    public Object getMyBean(String beanName) {
        return null;
    }

    @Override
    public void submitRequest(String requestType, String requestDescription) {
        // Wypełnienie obiektu CustomerRequestDetails i jego zapisanie.
        CustomerRequestDetails customerRequestDetails =
            (CustomerRequestDetails) getMyBean("customerRequestDetails");
        customerRequestDetails.setType(requestType);
        customerRequestDetails.setDescription(requestDescription);
        customerRequestDao.submitRequest(customerRequestDetails);
    }
}
```

Powyższy listing pokazuje, że klasa `CustomerRequestServiceImpl` definiuje metodę `getMyBean()`. Metoda ta akceptuje argument w postaci nazwy komponentu i zamiast zwrócić egzemplarz komponentu odpowiadający podanej nazwie wartością zwrótną metody `getMyBean()` jest `null`. Metoda `submitRequest()` przekazuje ciąg tekstowy `customerRequestDetails` jako argument metodzie `getMyBean()`. W kodzie metody `submitRequest()` przyjęto założenie, że metoda `getMyBean()` zwraca egzemplarz komponentu `customerRequestDetails`. Używając elementu <replaced-method>, można nadpisać metodę `getMyBean()` wersją, która zwraca egzemplarz komponentu odpowiadający podanemu w argumencie `name`.

Element <replaced-method> wymaga informacji o metodzie *nadpisanej* (w omawianym przykładzie to będzie metoda `getMyBean()` klasy `CustomerRequestServiceImpl`) i metodzie *nadpisującej*. Metoda nadpisująca jest dostarczana przez klasę implementującą interfejs `MethodReplacer` oferowany przez Spring. Listing 4.20 przedstawia klasę `MyMethodReplacer` implementującą interfejs `MethodReplacer`.

Listing 4.20. Klasa `MyMethodReplacer`

Projekt: `ch04-bankapp-replaced-method`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp/service`

```
package sample.spring.chapter04.bankapp.service;

import org.springframework.beans.factory.support.MethodReplacer;
import org.springframework.context.ApplicationContextAware;
```

```
public class MyMethodReplacer implements MethodReplacer, ApplicationContextAware {
    private ApplicationContext applicationContext;

    @Override
    public Object reimplement(Object obj, Method method, Object[] args) throws Throwable
    {
        return applicationContext.getBean((String) args[0]);
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
        throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

Interfejs `MethodReplacer` w Springu definiuje metodę `reimplement()`, której implementacja jest dostarczana przez klasę `MyMethodReplacer`. Metoda `reimplement()` przedstawia metodę *nadpisującą*. Klasa `MyMethodReplacer` implementuje także interfejs `ApplicationContextAware`, aby metoda `reimplement()` mogła uzyskać dostęp do egzemplarza `ApplicationContext`. Metoda `reimplement()` używa metody `getBean()` klasy `ApplicationContext` w celu pobierania komponentów z kontenera Springa.

Metoda `reimplement()` akceptuje następujące argumenty:

- `Object obj` — wskazuje obiekt, którego metoda jest nadpisywana. W omawianym przykładzie to będzie obiekt `CustomerRequestServiceImpl`.
- `Method method` — wskazuje metodę klasy komponentu nadpisywaną przez `reimplement()`. W omawianym przykładzie to będzie metoda `getMyBean()` klasy `CustomerRequestServiceImpl`.
- `Object[] args` — wskazuje argumenty przekazywane nadpisywanej metodzie. W omawianym przykładzie `args` oznacza argumenty przekazywane metodzie `getMyBean()` klasy `CustomerRequestServiceImpl`. W listingu 4.20 fragment `args[0]` w metodzie `reimplement()` odwołuje się do argumentu w postaci nazwy komponentu przekazanej metodzie `getMyBean()` klasy `CustomerRequestServiceImpl`.

Jeżeli spojrzysz na przedstawioną w listingu 4.20 metodę `reimplement()` klasy `MyMethodReplacer`, to zauważysz, że używa argumentu `args` w celu pobrania nazwy komponentu przekazanej metodzie `getMyBean()` klasy `CustomerRequestServiceImpl`. Następnie wywołuje metodę `getBean()` klasy `ApplicationContext` w celu pobrania odpowiedniego egzemplarza komponentu. Ponieważ metoda `reimplement()` klasy `MyMethodReplacer` nadpisuje metodę `getMyBean()` klasy `CustomerRequestServiceImpl`, wywołanie `getMyBean()` w trakcie działania aplikacji spowoduje zwrot egzemplarza komponentu, którego nazwa została przekazana metodzie `getMyBean()`.

Element `<replaced-method>` informuje kontener Springa, że metoda `reimplement()` klasy `MyMethodReplacer` nadpisuje metodę `getMyBean()` klasy `CustomerRequestServiceImpl`. Przykład użycia elementu `<replaced-method>` przedstawia listing 4.21.

Listing 4.21. Plik *applicationContext.xml* przedstawiający użycie elementu `<replaced-method>`Projekt: *ch04-bankapp-replaced-method*Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean id="customerRequestService"
      class="sample.spring.chapter04.bankapp.service.CustomerRequestServiceImpl">
  <constructor-arg name="customerRequestDao" ref="customerRequestDao" />
  <replaced-method name="getMyBean" replacer="methodReplacer" />
</bean>

<bean id="methodReplacer"
      class="sample.spring.chapter04.bankapp.service.MyMethodReplacer" />
```

W powyższym listingu zostały przedstawione definicje komponentów dla klas *MyMethodReplacer* i *CustomerRequestServiceImpl*. Atrybut `name` elementu `<replaced-method>` określa nazwę nadpisywanej metody, natomiast atrybut `replacer` zawiera odwołanie do komponentu implementującego interfejs *MethodReplacer*. Metoda podana przez atrybut `name` jest nadpisywana przez metodę `reimplement()` komponentu wskazanego przez atrybut `replacer`.

Podobnie jak w przypadku elementu `<lookup-method>`, metoda `main()` klasy *BankApp* projektu *ch04-bankapp-replaced-method* sprawdza, czy element `<replaced-method>` nadpisuje metodę `getMyBean()` klasy *CustomerRequestService* metodą `reimplement()` klasy *MyMethodReplacer*. Klasa *BankApp* projektu *ch04-bankapp-replaced-method* ma taką samą postać jak dla przedstawionego w listingu 4.18 projektu *ch04-bankapp-lookup-method*. Jeżeli wykonasz metodę `main()` klasy *BankApp*, przekonasz się, że element `<replaced-method>` nadpisuje metodę `getMyBean()` klasy *CustomerRequestServiceImpl* metodą `reimplement()` klasy *MyMethodReplacer*. Dlatego też z kontenera Springa nowy egzemplarz *CustomerRequestDetails* będzie pobierany w trakcie każdego wywołania metody `submitRequest()` klasy *CustomerRequestServiceImpl* (patrz: listing 4.19).

Warto pamiętać o możliwości użycia elementu `<replaced-method>` w celu zastąpienia metody abstrakcyjnej lub konkretnej komponentu klasy zupełnie inną implementacją metody. Na przykład `getMyBean()` można zdefiniować jako metodę abstrakcyjną, a następnie użyć elementu `<replaced-method>` w dokładnie taki sam sposób jak tutaj omówiliśmy.



W trakcie działania aplikacji kontener Springa musi mieć możliwość tworzenia podklas, aby zastąpić metodę zupełnie inną implementacją, dlatego kontener Springa używa biblioteki CGLIB (<https://github.com/cglib/cglib>) do utworzenia klas potomnych. Począwszy od Springa w wersji 3.2, klasy CGLIB są umieszczane w pliku JAR o nazwie *spring-core* i dlatego tej biblioteki nie trzeba wyraźnie definiować jako zależności projektu.

Zobaczmy teraz, jak element `<replaced-method>` unikalnie identyfikuje metodę komponentu przeznaczoną do nadpisania.

Unikalna identyfikacja metody komponentu

Może się zdarzyć sytuacja, że nazwa nie pozwoli na prawidłową identyfikację metody komponentu, której implementacja ma być zastąpiona za pomocą elementu `<replaced-method>`. Na przykład w listingu 4.22 przedstawiona została klasa komponentu zawierająca przeciążone metody `perform()`.

Listing 4.22. *Przeciążone metody w klasie komponentu*

```
public class MyBean {  
    public void perform(String task1, String task2) { ..... }  
    public void perform(String task) { ..... }  
    public void perform(my.Task task) { ..... }  
}
```

W powyższym listingu klasa `MyBean` zawiera wiele metod o nazwie `perform()`. W celu unikalnej identyfikacji metody komponentu przeznaczonej do zastąpienia element `<replaced-method>` używa elementu `<arg-type>`, aby wskazać typy argumentów metody. Na przykład w listingu 4.23 widać, jak element `<replaced-method>` wskazuje metodę `perform(String, String)` przeznaczoną do zastąpienia w klasie `MyBean`.

Listing 4.23. *Przykład użycia elementu `<replaced-method>` zawierającego elementy `<arg-type>`*

```
<bean id="mybean" class="MyBean">  
    <replaced-method name="perform" replacer=".....">  
        <arg-type>java.lang.String</arg-type>  
        <arg-type>java.lang.String</arg-type>  
    </replaced-method>  
</bean>
```

Zamiast używać w pełni kwalifikowanej nazwy jako wartości elementu `<arg-type>` można podać podciąg tekstowy wspomnianej nazwy. Na przykład zamiast `java.lang.String` można użyć `Str` lub `String` jako wartości elementu `<arg-type>` w listingu 4.23.

Przechodzimy teraz do oferowanej przez Spring funkcji automatycznego wiązania zależności, dzięki której można uniknąć konieczności podawania zależności w pliku XML kontekstu aplikacji.

4.6. Automatyczne wiązanie zależności

Zależności komponentów w Springu można zdefiniować jawnie za pomocą elementów `<property>` i `<constructor-arg>` lub też pozwolić mu na ich automatyczne rozwiązanie. Proces, w trakcie którego zależności są automatycznie rozwiązywane przez Spring, nosi nazwę *automatycznego wiązania*.

IMPORT **Rozdział104/ch04-bankapp-autowiring.** (Ten projekt pokazuje aplikację `MyBank`, która wykorzystuje do wstrzykiwania zależności oferowaną przez Spring funkcję automatycznego wiązania. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Atrybut `autowire` elementu `<bean>` określa sposób automatycznego rozwiązywania zależności przez Spring. Atrybut ten może przyjmować jedną z następujących wartości: `default`, `byName`, `byType`, `constructor` i `no`. Teraz dokładnie przeanalizujemy poszczególne wartości atrybutu `autowire`.



Trzeba pamiętać, że atrybut `autowire` elementu `<bean>` nie jest dziedziczony przez definicje komponentów potomnych.

Wartość byType

Jeżeli wartością atrybutu `autowire` będzie `byType`, wówczas Spring przeprowadza automatyczne wiązanie właściwości komponentu na podstawie ich typu. Na przykład jeśli komponent A definiuje właściwość typu `X`, Spring odszuka typ `X` w egzemplarzu `ApplicationContext` i wstrzyknie go do komponentu A. Spójrzmy teraz na przykład użycia automatycznego wiązania `byType` w aplikacji `MyBank`.

W listingu 4.24 została przedstawiona klasa `CustomerRegistrationServiceImpl` aplikacji `MyBank`.

Listing 4.24. Klasa `CustomerRegistrationServiceImpl`

Projekt: `ch04-bankapp-autowiring`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp/service`

```
package sample.spring.chapter04.bankapp.service;

public class CustomerRegistrationServiceImpl implements CustomerRegistrationService {

    private CustomerRegistrationDetails customerRegistrationDetails;
    private CustomerRegistrationDao customerRegistrationDao;
    ....
    public void setCustomerRegistrationDetails(
        CustomerRegistrationDetails customerRegistrationDetails) {
        this.customerRegistrationDetails = customerRegistrationDetails;
    }
    public void setCustomerRegistrationDao(
        CustomerRegistrationDao customerRegistrationDao) {
        this.customerRegistrationDao = customerRegistrationDao;
    }
    .....
}
```

W powyższym listingu widać, że klasa `CustomerRegistrationServiceImpl` definiuje właściwości o nazwach `customerRegistrationDetails` (typu `CustomerRegistrationDetails`) i `customerRegistrationDao` (typu `CustomerRegistrationDao`). Oznacza to, że obiekty `CustomerRegistrationDetails` i `CustomerRegistrationDao` są zależnościami obiektu `CustomerRegistrationServiceImpl`.

W listingu 4.25 zostały przedstawione definicje komponentów dla klas `CustomerRegistrationServiceImpl`, `CustomerRegistrationDetails` i `CustomerRegistrationDaoImpl` (implementacja interfejsu `CustomerRegistrationDao`).

Listing 4.25. Plik `applicationContext.xml` przedstawiający przykład automatycznego wiązania według typu

Projekt: `ch04-bankapp-autowiring`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="customerRegistrationService"
    class="sample.spring.chapter04.bankapp.service.CustomerRegistrationServiceImpl"
    scope="prototype" autowire="byType" />

<bean id="customerRegistrationDetails"
    class="sample.spring.chapter04.bankapp.domain.CustomerRegistrationDetails"
    scope="prototype" />
```

```
<bean id="customerRegistrationDao"
      class="sample.spring.chapter04.bankapp.dao.CustomerRegistrationDaoImpl" />
```

W powyższym listingu definicja komponentu `customerRegistrationService` nie zawiera elementów `<property>` przeznaczonych do ustawienia właściwości `customerRegistrationDetails` i `customerRegistrationDao` (patrz: listing 4.24). Zamiast tego element `<bean>` zawiera atrybut `autowire` o wartości `byType` nakazujący Springowi przeprowadzenie automatyczne rozwiązania zależności komponentu `customerRegistrationService` na podstawie ich typów. Spring wyszukuje w egzemplarzu `ApplicationContext` typy `CustomerRequestDetails` i `CustomerRegistrationDao`, a następnie wstrzykuje je do komponentu `customerRegistrationService`. Ponieważ komponenty `customerRegistrationDetails` i `customerRegistrationDao` przedstawiają komponenty typów `CustomerRegistrationDetails` i `CustomerRegistrationDao`, kontener Springa wstrzykuje egzemplarze `customerRegistrationDetails` i `customerRegistrationDao` do komponentu `customerRegistrationService`.

Może się zdarzyć, że Spring nie znajdzie w egzemplarzu `ApplicationContext` żadnych zarejestrowanych typów odpowiadających typowi właściwości. W takim przypadku nie dochodzi do zgłoszenia wyjątku, a właściwość komponentu po prostu *nie* zostaje ustawiona. Na przykład jeżeli komponent definiuje właściwość `x` typu `Y`, a egzemplarz `ApplicationContext` nie ma zarejestrowanego komponentu typu `Y`, wówczas właściwość `x` nie zostanie ustawiona. Gdy w egzemplarzu `ApplicationContext` Spring znajdzie wiele komponentów dopasowujących typ właściwości, nastąpi zgłoszenie wyjątku. W takich przypadkach zamiast używać funkcji automatycznego wiązania należy skorzystać z elementów `<property>` w celu wyraźnego zdefiniowania zależności komponentu lub ustawienia komponentu jako *podstawowego kandydata* do automatycznego wiązania, co odbywa się przez przypisanie wartości `true` atrybutowi `primary` elementu `<bean>`.

Wartość constructor

Jeżeli wartością atrybutu `autowire` będzie `constructor`, Spring przeprowadzi automatyczne wiązanie argumentów konstruktora klasy komponentu na podstawie ich typu. Na przykład jeśli konstruktor komponentu `A` akceptuje argumenty typu `X` i `Y`, Spring odszuka typy `X` i `Y` w egzemplarzu `ApplicationContext` i wstrzyknie je jako argumenty konstruktora komponentu `A`. Spójrzmy teraz na przykład użycia automatycznego wiązania `constructor` w aplikacji `MyBank`.

W listingu 4.26 została przedstawiona klasa `CustomerRequestServiceImpl` aplikacji `MyBank`.

Listing 4.26. Klasa `CustomerRequestServiceImpl`

Projekt: `ch04-bankapp-autowiring`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp/service`

```
package sample.spring.chapter04.bankapp.service;

public class CustomerRequestServiceImpl implements CustomerRequestService {
    private CustomerRequestDetails customerRequestDetails;
    private CustomerRequestDao customerRequestDao;

    @ConstructorProperties({ "customerRequestDetails", "customerRequestDao" })
    public CustomerRequestServiceImpl(
        CustomerRequestDetails customerRequestDetails,
        CustomerRequestDao customerRequestDao) {
```

```

        this.customerRequestDetails = customerRequestDetails;
        this.customerRequestDao = customerRequestDao;
    }
    .....
}

```

Klasa `CustomerRequestServiceImpl` definiuje konstruktor akceptujący argumenty typów `CustomerRequestDetails` i `CustomerRequestDao`.

W listingu 4.27 zostały przedstawione definicje komponentów dla klas `CustomerRequestServiceImpl`, `CustomerRequestDetails` i `CustomerRequestDaoImpl` (implementacja interfejsu `CustomerRequestDao`).

Listing 4.27. Plik `applicationContext.xml` przedstawiający przykład automatycznego wiązania według typu argumentu konstruktora

Projekt: `ch04-bankapp-autowiring`

Położenie pliku: `src/main/resources/META-INF/spring`

```

<bean id="customerRequestService"
      class="sample.spring.chapter04.bankapp.service.CustomerRequestServiceImpl"
      autowire="constructor">
</bean>

<bean id="customerRequestDetails"
      class="sample.spring.chapter04.bankapp.domain.CustomerRequestDetails"
      scope="prototype" />

<bean id="customerRequestDao"
      class="sample.spring.chapter04.bankapp.dao.CustomerRequestDaoImpl" />

```

W powyższym listingu definicja komponentu `customerRequestService` przypisuje atrybutowi `autowire` wartość `constructor`. Spring wyszukuje więc w egzemplarzu `ApplicationContext` komponenty typów `CustomerRequestDetails` i `CustomerRequestDao`, a następnie przekazuje je jako argumenty do konstruktora klasy `CustomerRequestServiceImpl`. Ponieważ komponenty `customerRequestDetails` i `customerRequestDao` są typów `CustomerRequestDetails` i `CustomerRequestDao`, Spring automatycznie wstrzykuje egzemplarze tych komponentów do komponentu `customerRequestService`.

Jeżeli w egzemplarzu `ApplicationContext` Spring nie znajdzie żadnych komponentów, których typ odpowiada typowi argumentu konstruktora, wówczas wspomniany argument konstruktora po prostu *nie* zostanie ustawiony. Gdy w egzemplarzu `ApplicationContext` Spring znajdzie wiele komponentów dopasowujących typ argumentu konstruktora, wtedy nastąpi zgłoszenie wyjątku. W takich przypadkach należy użyć elementów `<constructor-arg>` w celu wyraźnego zdefiniowania zależności komponentu lub ustawienia komponentu jako *podstawowego kandydata* do automatycznego wiązania, co odbywa się przez przypisanie wartości `true` atrybutowi `primary` elementu `<bean>`.

Wartość `byName`

Jeżeli wartością atrybutu `autowire` będzie `byName`, Spring przeprowadzi automatyczne wiązanie właściwości komponentu na podstawie ich nazw. Na przykład jeśli komponent A definiuje właściwość o nazwie `x`, Spring odszuka w egzemplarzu `ApplicationContext` komponent o nazwie

x i wstrzyknie go do komponentu A. Spójrzmy teraz na przykład użycia automatycznego wiązania `byName` w aplikacji `MyBank`.

W listingu 4.28 została przedstawiona klasa `FixedDepositServiceImpl` aplikacji `MyBank`.

Listing 4.28. Klasa `FixedDepositServiceImpl`

Projekt: `ch04-bankapp-autowiring`

Położenie pliku: `src/main/java/sample/spring/chapter04/bankapp/service`

```
package sample.spring.chapter04.bankapp.service;

import sample.spring.chapter04.bankapp.dao.FixedDepositDao;
import sample.spring.chapter04.bankapp.domain.FixedDepositDetails;

public class FixedDepositServiceImpl implements FixedDepositService {
    private FixedDepositDao myFixedDepositDao;

    public void setMyFixedDepositDao(FixedDepositDao myFixedDepositDao) {
        this.myFixedDepositDao = myFixedDepositDao;
    }
    .....
}
```

W powyższym listingu widać, że klasa `FixedDepositServiceImpl` definiuje właściwość o nazwie `myFixedDepositDao` i typie `FixedDepositDao`.

W listingu 4.29 zostały przedstawione definicje komponentów dla klas `FixedDepositServiceImpl` i `FixedDepositDaoImpl` (implementacja interfejsu `FixedDepositDao`).

Listing 4.29. Plik `applicationContext.xml` przedstawiający przykład automatycznego wiązania według nazwy

Projekt: `ch04-bankapp-autowiring`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="FixedDepositService"
      class="sample.spring.chapter04.bankapp.service.FixedDepositServiceImpl"
      autowire="byName" />

<bean id="myFixedDepositDao"
      class="sample.spring.chapter04.bankapp.dao.FixedDepositDaoImpl" />
```

W powyższym listingu definicja komponentu `FixedDepositService` przypisuje atrybutowi `autowire` wartość `byName`. Oznacza to, że właściwości komponentu `FixedDepositService` będą przez Spring automatycznie rozwiązane na podstawie ich nazw. Listing 4.28 pokazuje, że klasa `FixedDepositServiceImpl` definiuje właściwość o nazwie `myFixedDepositDao`. Dlatego też w egzemplarzu `ApplicationContext` Spring wyszukuje komponent o nazwie `myFixedDepositDao`, a następnie wstrzykuje go do komponentu `FixedDepositService`. W omawianym przykładzie definicja komponentu `myFixedDepositDao` przedstawia klasę `FixedDepositDaoImpl`, co oznacza, że egzemplarz `FixedDepositDaoImpl` jest wstrzykiwany do `FixedDepositService` dla właściwości o nazwie `myFixedDepositDao`.

Wartości default i no

Jeżeli atrybutowi `autowire` zostanie przypisana wartość `default` lub `no`, wówczas funkcja automatycznego wiązania jest *wyłączana* dla danego komponentu. Ponieważ Spring domyślnie traktuje komponenty jakby miały przypisaną wartość `no` dla atrybutu `autowire`, przypisanie temu atrybutowi wartości `default` oznacza brak automatycznego wiązania w danym komponencie. Możesz wyraźnie wskazać, że komponentowi *nie wolno* używać oferowanej przez Spring funkcji automatycznego wiązania. W tym celu atrybutowi `autowire` przypisz wartość `no`.



Istnieje możliwość zmiany domyślnego zachowania komponentów w zakresie automatycznego wiązania, co wymaga ustawienia atrybutu `default-autowire` elementu `<beans>`. Na przykład jeżeli atrybutowi `default-autowire` zostanie przypisana wartość `byType`, praktycznie oznacza to przypisanie wartości `byType` atrybutowi `autowire` we wszystkich elementach `<bean>` w pliku XML kontekstu aplikacji. Warto zwrócić uwagę, że jeśli atrybut `autowire` elementu `<bean>` będzie miał inną wartość niż podana w atrybucie `default-autowire` elementu `<beans>`, wówczas w komponencie zostanie zastosowana wartość atrybutu `autowire` elementu `<bean>`.

W listingu 4.30 została przedstawiona definicja komponentu dla klasy `CustomerRegistrationServiceImpl` w aplikacji `MyBank`. Klasa ta ma atrybut `autowire` o wartości `no`.

Listing 4.30. Plik `applicationContext.xml` przedstawiający brak automatycznego wiązania

Projekt: `ch04-bankapp-autowiring`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="customerRegistrationService"
      class="sample.spring.chapter04.bankapp.service.CustomerRegistrationServiceImpl"
      scope="prototype" autowire="no" />
```

W listingu 4.24 widać, że klasa `CustomerRegistrationServiceImpl` definiuje właściwości `customerRegistrationDetails` (typu `CustomerRegistrationDetails`) i `customerRegistrationDao` (typu `CustomerRegistrationDao`). Ponieważ przypisano wartość `no` atrybutowi `autowire` komponentu `customerRegistrationService_`, dla wymienionego komponentu funkcja automatycznego wiązania została wyłączona. Oznacza to, że właściwości `customerRegistrationDetails` i `customerRegistrationDao` komponentu `customerRegistrationService_` *nie* będą ustawiane przez Spring.

Dotychczas w tym podrozdziale przedstawiliśmy różne sposoby automatycznego rozwiązywania zależności komponentu przez Spring. Zobaczmy teraz, jak uczynić komponent *niedostępnym* dla funkcji automatycznego wiązania za pomocą atrybutu `autowire-candidate` elementu `<bean>`.

Uniedostępnienie komponentu dla funkcji automatycznego wiązania

Domyślne zachowanie kontenera Springa polega na udostępnianiu komponentów dla funkcji automatycznego wiązania. Istnieje jednak możliwość uniedostępnienia danego komponentu innym komponentom do celów automatycznego wiązania, co wymaga przypisania wartości `false` atrybutowi `autowire-candidate`.

W aplikacji MyBank klasa `AccountStatementServiceImpl` definiuje właściwość typu `AccountStatementDao`. Listing 4.31 przedstawia klasę `AccountStatementServiceImpl`.

Listing 4.31. Klasa `AccountStatementServiceImpl`

Projekt: *ch04-bankapp-autowiring*

Położenie pliku: *src/main/java/sample/spring/chapter04/bankapp/service*

```
package sample.spring.chapter04.bankapp.service;

import sample.spring.chapter04.bankapp.dao.AccountStatementDao;
import sample.spring.chapter04.bankapp.domain.AccountStatement;
public class AccountStatementServiceImpl implements AccountStatementService {
    private AccountStatementDao accountStatementDao;

    public void setAccountStatementDao(AccountStatementDao accountStatementDao) {
        this.accountStatementDao = accountStatementDao;
    }
    ....
}
```

Z kolei listing 4.32 przedstawia definicje komponentów dla klas `AccountStatementServiceImpl` i `AccountStatementDaoImpl` (implementacja interfejsu `AccountStatementDao`).

Listing 4.32. Plik `applicationContext.xml` przedstawiający przykład użycia atrybutu `autowire-candidate`

Projekt: *ch04-bankapp-autowiring*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean id="accountStatementService"
      class="sample.spring.chapter04.bankapp.service.AccountStatementServiceImpl"
      autowire="byType" />

<bean id="accountStatementDao"
      class="sample.spring.chapter04.bankapp.dao.AccountStatementDaoImpl"
      autowire-candidate="false" />
```

W powyższym listingu definicja komponentu `accountStatementService` przypisuje atrybutowi `autowire` wartość `byType`. Oznacza to, że właściwość typu `AccountStatementDao` komponentu `accountStatementService` będzie automatycznie wiązana według typu. Ponieważ komponent `accountStatementDao` jest typu `AccountStatementDao`, możesz sądzić, że Spring wstrzyknie egzemplarz `accountStatementDao` do komponentu `accountStatementService`. Spring jednak nie będzie brać komponentu `accountStatementDao` pod uwagę w trakcie automatycznego wiązania, gdyż w definicji komponentu `accountStatementDao` atrybutowi `autowire-candidate` przypisano wartość `false`.



Należy pamiętać, że komponent niedostępny dla innych komponentów w celach automatycznego wiązania sam może skorzystać z tej funkcji Springa, aby jego zależności zostały automatycznie rozwiązane.

Jak wcześniej wspomnieliśmy, domyślne zachowanie kontenera Springa polega na udostępnianiu komponentów w celu ich automatycznego wiązania. Aby tylko wybrany zestaw komponentów mógł być używany do automatycznego wiązania, należy ustawić atrybut `default-autowire-candidates`

elementu `<beans>`. Atrybut ten określa *wzorzec* nazwy komponentu i jedynie komponenty o nazwach dopasowanych do podanego wzorca będą dostępne w celu automatycznego wiązania. Listing 4.33 przedstawia przykład użycia atrybutu `default-autowire-candidates`.

Listing 4.33. Przykład użycia atrybutu `default-autowire-candidates`

```
<beans default-autowire-candidates="*Dao" >
    ....
    <bean id="customerRequestDetails"
        class="sample.spring.chapter04.bankapp.domain.CustomerRequestDetails"
        scope="prototype" autowire-candidate="true"/>

    <bean id="customerRequestDao"
        class="sample.spring.chapter04.bankapp.dao.CustomerRequestDaoImpl" />

    <bean id="customerRegistrationDao"
        class="sample.spring.chapter04.bankapp.dao.CustomerRegistrationDaoImpl" />
    ....
</beans>
```

W powyższym listingu wartością atrybutu `default-autowire-candidates` jest `*Dao`. Dlatego komponenty o nazwach kończących się na `Dao` (np. komponenty `customerRequestDao` i `customerRegistrationDao`) będą dostępne dla automatycznego wiązania. Jeżeli nazwa komponentu nie zostanie dopasowana do wzorca określonego przez atrybut `default-autowire-candidates` (np. komponent `customerRequestDetails`), nadal można je udostępnić funkcji automatycznego wiązania przez przypisanie wartości `true` atrybutowi `autowire-candidate` elementu `<bean>` odpowiadającego komponentowi.

Teraz przechodzimy do ograniczeń użycia automatycznego wiązania w aplikacjach.

Ograniczenia automatycznego wiązania

Jak zauważyłeś w tym podrozdziale, funkcja automatycznego wiązania pozwala na uniknięcie wysiłku związanego z koniecznością wyraźnego definiowania zależności za pomocą elementów `<property>` i `<constructor-arg>`. Funkcja ta ma jednak także swoje wady:

- Automatycznego wiązania nie można użyć do ustawienia właściwości lub argumentów konstruktora będących typami prostymi w Javie (np. `int`, `long`, `boolean`, `String`, `Date`). Natomiast automatyczne wiązanie można zastosować dla tablic i kolekcji, o ile wartością atrybutu `autowire` jest `byType` lub `constructor`.
- Ponieważ zależności komponentów są automatycznie rozwiązywane przez Spring, skutkiem będzie ukrycie ogólnej struktury aplikacji. Jeżeli do określenia zależności używasz elementów `<property>` i `<constructor-arg>`, dokładnie dokumentujesz ogólną strukturę aplikacji. Poznanie sposobu działania i konserwacja aplikacji są łatwiejsze, gdy zależności komponentów pozostają wyraźnie zdefiniowane. Dlatego też nie zaleca się stosowania funkcji automatycznego wiązania w ogromnych aplikacjach.

4.7. Podsumowanie

Z tego rozdziału dowiedziałeś się, jak Spring przeprowadza różne rodzaje wstrzykiwania zależności. Przeanalizowaliśmy sposoby użycia interfejsu `ApplicationContextAware`, elementów `<replaced-method>` i `<lookup-method>` znajdujących się w elemencie `<bean>` w celu programowego pobierania egzemplarza komponentu z egzemplarza `ApplicationContext`. Ponadto zobaczyłeś, jak oferowana przez Spring funkcja automatycznego wiązania pozwala na uniknięcie konieczności wyraźnego definiowania zależności komponentu w pliku XML kontekstu aplikacji. Z następnego rozdziału dowiesz się, jak dostosować do własnych potrzeb komponenty i ich definicje.

Rozdział 5. Dostosowanie komponentów i ich definicji

5.1. Wprowadzenie

Dotychczas przedstawiliśmy przykłady, w których kontener Springa tworzył egzemplarz komponentu na podstawie jego definicji określonej w pliku XML kontekstu aplikacji. W tym rozdziale pójdziemy o krok dalej i zajmiemy się następującymi zagadnieniami:

- Jak w komponencie wykorzystać własną logikę inicjalizacji i usuwania komponentu?
- Jak można używać nowo utworzonego egzemplarza komponentu poprzez implementację interfejsu `BeanPostProcessor`?
- W jaki sposób modyfikować definicje komponentów poprzez implementację interfejsu `BeanFactoryPostProcessor`?

5.2. Dostosowanie logiki inicjalizacji i usuwania komponentu

Z poprzednich rozdziałów dowiedziałeś się, że kontener Springa jest odpowiedzialny za tworzenie egzemplarza komponentu oraz wstrzykiwanie jego zależności. Po utworzeniu egzemplarza komponentu za pomocą wywołania konstruktora klasy komponentu kontener Springa przystępuje do przygotowania właściwości komponentu, wywołując jego metody typu setter. Jeżeli chcesz wykonać własną logikę inicjalizacji (np. otworzyć plik, nawiązać połączenie z bazą danych itd.) *po* ustawieniu właściwości komponentu, ale jeszcze *przed* jego pełnym zainicjowaniem przez kontener Springa, możesz w tym celu podać nazwę metody inicjującej jako wartość atrybutu `init-method` elementu `<bean>`. Podobnie jeżeli chcesz wykonać własną logikę czyszczącą, *zanim* zostanie usunięty kontener Springa zawierający egzemplarz komponentu, wówczas nazwę metody czyszczącej podaj jako wartość atrybutu `destroy-method` elementu `<bean>`.

IMPORT **Rozdział105/ch05-bankapp-customization.** (Ten projekt pokazuje aplikację `MyBank` używającą atrybutów `init-method` i `destroy-method` elementu `<bean>` do wskazania metod zawierających logikę niestandardowej inicjalizacji i usunięcia egzemplarza. Aby przetestować wykonanie metody zawierającej niestandardową logikę inicjalizacji, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu. Natomiast w celu przetestowania metody zawierającej niestandardową logikę usuwania egzemplarza należy wykonać metodę `main()` klasy `BankAppWithHook` projektu).

W listingu 5.1 została przedstawiona klasa `FixedDepositDaoImpl` definiująca metodę inicjującą o nazwie `initializeDbConnection()`, która jest odpowiedzialna za nawiązanie połączenia z bazą danych. Klasa ponadto zawiera metodę czyszczącą o nazwie `releaseDbConnection()`, która powoduje zamknięcie połączenia z bazą danych.

Listing 5.1. Klasa *FixedDepositDaoImpl* przedstawiająca przykład własnej logiki inicjalizacji i usunięcia komponentu

Projekt: *ch05-bankapp-customization*

Położenie pliku: *src/main/java/sample/spring/chapter05/bankapp/dao*

```
package sample.spring.chapter05.bankapp.dao;

public class FixedDepositDaoImpl implements FixedDepositDao {
    private static Logger logger = Logger.getLogger(FixedDepositDaoImpl.class);
    private DatabaseConnection connection;

    public FixedDepositDaoImpl() {
        logger.info("Wywołanie konstruktora klasy FixedDepositDaoImpl.");
    }

    public void initializeDbConnection() {
        logger.info("Wywołanie metody initializeDbConnection() klasy FixedDepositDaoImpl.");
        connection = DatabaseConnection.getInstance();
    }

    public boolean createFixedDeposit(FixedDepositDetails fixedDepositDetails) {
        logger.info("Wywołanie metody createFixedDeposit() klasy FixedDepositDaoImpl.");
        // Zapis informacji o lokatach i zwrot wartości true.
        return true;
    }

    public void releaseDbConnection() {
        logger.info("Wywołanie metody releaseDbConnection() klasy FixedDepositDaoImpl.");
        connection.releaseConnection();
    }
}
```

W powyższym listingu obiekt *DatabaseConnection* jest używany podczas pracy z bazą danych aplikacji *MyBank*. Klasa *FixedDepositDaoImpl* ma metodę *initializeDbConnection()* odpowiedzialną za inicjalizację obiektu *DatabaseConnection*, który następnie jest używany przez metodę *createFixedDeposit()* do zapisu w bazie danych aplikacji *MyBank* informacji szczegółowych o lokatach.

W listingu 5.2 widać, że klasa *FixedDepositServiceImpl* aplikacji *MyBank* do założenia nowych lokat używa egzemplarza *FixedDepositDaoImpl*.

Listing 5.2. Klasa *FixedDepositServiceImpl*

Projekt: *ch05-bankapp-customization*

Położenie pliku: *src/main/java/sample/spring/chapter05/bankapp/service*

```
package sample.spring.chapter05.bankapp.service;

public class FixedDepositServiceImpl implements FixedDepositService {
    private static Logger logger = Logger.getLogger(FixedDepositServiceImpl.class);
    private FixedDepositDao myFixedDepositDao;

    public void setMyFixedDepositDao(FixedDepositDao myFixedDepositDao) {
        logger.info("Wywołanie metody setMyFixedDepositDao() klasy
```

```

        FixedDepositServiceImpl.");
        this.myFixedDepositDao = myFixedDepositDao;
    }

    @Override
    public void createFixedDeposit(FixedDepositDetails fixedDepositDetails) throws
        Exception {
        // Utworzenie lokaty.
        myFixedDepositDao.createFixedDeposit(fixedDepositDetails);
    }
}

```

W powyższym listingu widać, że egzemplarz `FixedDepositDaoImpl` jest zależnością `FixedDepositServiceImpl` i jest przekazywany jako argument metodzie typu setter o nazwie `setMyFixedDepositDao()`. Ponadto po wywołaniu metody `createFixedDeposit()` klasy `FixedDepositServiceImpl` skutkiem będzie wywołanie metody `createFixedDeposit()` klasy `FixedDepositDaoImpl`.

W listingu 5.3 zostały przedstawione definicje komponentów dla klas `FixedDepositDaoImpl` i `FixedDepositServiceImpl`.

Listing 5.3. Plik `applicationContext.xml` przedstawiający przykład użycia atrybutów `init-method` i `destroy-method`

Projekt: `ch05-bankapp-customization`

Położenie pliku: `src/main/resources/META-INF/spring`

```

<beans .....>
  <bean id="FixedDepositService"
        class="sample.spring.chapter05.bankapp.service.FixedDepositServiceImpl">
    <property name="myFixedDepositDao" ref="myFixedDepositDao" />
  </bean>

  <bean id="myFixedDepositDao"
        class="sample.spring.chapter05.bankapp.dao.FixedDepositDaoImpl"
        init-method="initializeDbConnection" destroy-method="releaseDbConnection" />
</beans>

```

W powyższym listingu widać, że element `<bean>` odpowiadający klasie `FixedDepositDaoImpl` określa `initializeDbConnection` i `releaseDbConnection` jako wartości atrybutów odpowiednio `init-method` i `destroy-method`.



Trzeba w tym miejscu koniecznie wspomnieć, że metody zawierające logikę inicjalizacji i czyszczenia wskazywane przez atrybuty odpowiednio `init-method` i `destroy-method` elementu `<bean>` *nie mogą* akceptować żadnych argumentów, ale mogą być zdefiniowane w taki sposób, aby zgłaszać wyjątki.

W listingu 5.4 została przedstawiona klasa `BankApp`, której metoda `main()` pobiera egzemplarz `FixedDepositServiceImpl` z egzemplarza `ApplicationContext` i wywołuje metodę `createFixedDeposit()` klasy `FixedDepositServiceImpl`.

Listing 5.4. *Klasa BankApp*Projekt: *ch05-bankapp-customization*Położenie pliku: *src/main/java/sample/spring/chapter05/bankapp*

```
package sample.spring.chapter05.bankapp;

public class BankApp {
    public static void main(String args[]) throws Exception {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");

        FixedDepositService fixedDepositService =
            context.getBean(FixedDepositService.class);
        FixedDepositService.createFixedDeposit(new FixedDepositDetails(1, 1000,
            12, "nazwa-uzytkownika@nazwa-domeny.pl"));
    }
}
```

Jeżeli wykonasz teraz metodę `main()` klasy `BankApp`, w konsoli zostaną wygenerowane następujące dane wyjściowe:

```
Wywołanie konstruktora klasy FixedDepositDaoImpl.
Wywołanie metody initializeDbConnection() klasy FixedDepositDaoImpl.
Wywołanie metody setMyFixedDepositDao() klasy FixedDepositServiceImpl.
Wywołanie metody createFixedDeposit() klasy FixedDepositDaoImpl.
```

Powyższe dane wyjściowe pokazują, że kontener Springa najpierw tworzy egzemplarz `FixedDepositDaoImpl`, a następnie wywołuje metodę `initializeDbConnection()`. Po wykonaniu metody `initializeDbConnection()` egzemplarz `FixedDepositDaoImpl` zostaje wstrzyknięty do egzemplarza `FixedDepositServiceImpl`. To wyraźnie pokazuje, że kontener Springa wstrzykuje zależność (egzemplarz `FixedDepositDaoImpl`) do komponentu zależnego (`FixedDepositServiceImpl`) dopiero po wywołaniu przez kontener Springa metody inicjalizacji w komponencie zależności.

Prawdopodobnie zauważyłeś, że powyższe dane wyjściowe na skutek wywołania metody `main()` klasy `BankApp` nie zawierają komunikatu Wywołanie metody `releaseDbConnection()` klasy `FixedDepositDaoImpl` (patrz: listing 5.1). Oznacza to, że metoda `releaseDbConnection()` klasy `FixedDepositDaoImpl` nie została wywołana przez kontener Springa po zakończeniu wykonywania metody `main()` klasy `BankApp`. Podczas tworzenia rzeczywistej aplikacji tego rodzaju sytuacja oznacza, że nawiązane przez egzemplarz `FixedDepositDaoImpl` połączenie z bazą danych nie zostało zamknięte. Teraz zobaczysz, w jaki sposób Spring może elegancko usuwać egzemplarze komponentów o zasięgu singleton za pomocą wywołania metody czyszczącej określonej przez atrybut `destroy-method` elementu `<bean>`.

Wywołanie przez Spring metody czyszczącej wskazanej przez atrybut `destroy-method`

Sieciowa wersja implementacji `ApplicationContext` ma postać oferowanego przez Spring obiektu `WebApplicationContext`. Implementacja `WebApplicationContext` zawiera logikę niezbędną do wywołania metody czyszczącej (wskazanej przez atrybut `destroy-method`) komponentu o zasięgu singleton. Metoda czyszcząca będzie wywołana *przed* zakończeniem działania aplikacji.



Podjęcie omówione w tym punkcie powoduje, że Spring elegancko usuwa egzemplarze komponentów o zasięgu singleton przez wywołanie odpowiednich metod czyszczących charakterystycznych dla samodzielnych aplikacji.

W listingu 5.5 została przedstawiona klasa `BankAppWithHook` (zmodyfikowana wersja klasy `BankApp` pokazanej w listingu 5.4), której metoda `main()` gwarantuje, że metody czyszczące (określone za pomocą atrybutu `destroy-method` elementu `<bean>`) we wszystkich komponentach o zasięgu singleton zarejestrowanych w kontenerze Springa zostaną wywołane po zakończeniu działania metody `main()`.

Listing 5.5. Klasa `BankAppWithHook` przedstawiająca rejestrację zaczepu zamknięcia w JVM

Projekt: `ch05-bankapp-customization`

Położenie pliku: `src/main/java/sample/spring/chapter05/bankapp`

```
package sample.spring.chapter05.bankapp;

public class BankAppWithHook {
    public static void main(String args[]) throws Exception {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");

        context.registerShutdownHook();

        FixedDepositService fixedDepositService =
            context.getBean(FixedDepositService.class);
        FixedDepositService.createFixedDeposit(new FixedDepositDetails(1, 1000,
            12, "someemail@somedomain.com"));
    }
}
```

Oferowana przez Spring klasa `AbstractApplicationContext` implementuje interfejs `Application` ➔ `Context` i definiuje metodę `registerShutdownHook()` rejestrującą zaczep zamknięcia wirtualnej maszyny Javy (ang. *Java Virtual Machine*). Zaczep zamknięcia jest odpowiedzialny za usunięcie egzemplarza `ApplicationContext` w chwili zamykania JVM. W powyższym listingu zwróć uwagę na fakt, że egzemplarz `ClassPathXmlApplicationContext` jest przypisany typowi `AbstractApplicationContext`, natomiast metoda `registerShutdownHook()` klasy `AbstractApplicationContext` ➔ `Context` jest wywoływana w celu rejestracji zaczepu zamknięcia w JVM. Gdy metoda `main()` klasy `BankAppWithHook` istnieje, zaczep zamknięcia powoduje usunięcie wszystkich buforowanych egzemplarzy komponentu o zasięgu singleton oraz usunięcie egzemplarza `ApplicationContext`.

Jeżeli wykonasz teraz metodę `main()` klasy `BankAppWithHook` projektu `ch05-bankapp-customization`, wówczas w konsoli zostaną wygenerowane następujące dane wyjściowe:

```
Wywołanie konstruktora klasy FixedDepositDaoImpl.
Wywołanie metody initializeDbConnection() klasy FixedDepositDaoImpl.
Wywołanie metody setMyFixedDepositDao() klasy FixedDepositServiceImpl.
Wywołanie metody releaseDbConnection() klasy FixedDepositDaoImpl.
```

Wyświetlony w konsoli komunikat Wywołanie metody `releaseDbConnection()` klasy `FixedDepositDaoImpl` stanowi potwierdzenie wywołania metody `releaseDbConnection()` klasy `FixedDepositDaoImpl` (patrz: listing 5.1). Jak widzisz, skutkiem zarejestrowania zaczepu zamknięcia w JVM jest wywołanie metody czyszczącej w komponencie `myFixedDepositDao` o zasięgu singleton (jest to komponent odpowiadający klasie `FixedDepositDaoImpl`).

Zobaczmy teraz, jaki wpływ ma zaczep zamknięcia na komponent o zasięgu prototypu.

Metoda czyszcząca i komponent o zasięgu prototypu

W przypadku komponentu o zasięgu prototypu atrybut `destroy-method` jest ignorowany przez kontener Springa, ponieważ kontener oczekuje obiektu pobierającego z egzemplarza `ApplicationContext` egzemplarz komponentu o zasięgu prototypu i odpowiedzialnego za wywołanie metody czyszczącej.



Cykle życiowe komponentów o zasięgu singleton i prototypu są takie same. Jedynym wyjątkiem jest fakt, że kontener Springa *nie* wywołuje metody czyszczącej (określonej za pomocą atrybutu `destroy-method`) w egzemplarzu komponentu o zasięgu prototypu.

Zobaczmy teraz, jak można wskazać domyślne metody inicjalizacji i usuwania obiektu dla wszystkich komponentów zdefiniowanych w pliku XML kontekstu aplikacji.

Określenie domyślnych metod inicjalizacji i usuwania obiektu dla wszystkich komponentów

Atrybuty `default-init-method` i `default-destroy-method` elementu `<beans>` można wykorzystać w celu określenia domyślnych metod inicjalizacji i usuwania obiektu dla komponentów, jak pokazuje listing 5.6.

Listing 5.6. Przykład użycia atrybutów `default-init-method` i `default-destroy-method`

```
<beans ..... default-init-method="initialize" default-destroy-method="release">
  <bean id="A" class="....." init-method="initializeService" />
  <bean id="B" class="....." />
</beans>
```

Jeżeli wiele komponentów definiuje metody inicjacyjne i usuwające o takiej samej nazwie, wówczas sensowne wydaje się użycie atrybutów `default-init-method` i `default-destroy-method`. Przypisując wartość atrybutom `init-method` i `destroy-method`, element `<bean>` może nadpisywać wartości tych atrybutów zdefiniowane w elemencie `<beans>`. Na przykład w listingu 5.6 komponent A przypisuje atrybutowi `init-method` wartość `initializeService`, co oznacza, że metoda `initializeService()` — a nie metoda inicjalizacji określona przez atrybut `default-init-method` elementu `<beans>` — będzie użyta w charakterze metody inicjującej komponent A.

Zamiast używać atrybutów `init-method` i `destroy-method` elementu `<bean>` do wskazania własnych metod inicjalizacji i usuwania obiektu można zastosować interfejsy cyklu życiowego `InitializingBean` i `DisposableBean`.

Interfejsy cyklu życiowego `InitializingBean` i `DisposableBean`

Komponent taki jak `ApplicationContextAware` (patrz: podrozdział 4.5) implementujący interfejsy cyklu życiowego `InitializingBean` i `DisposableBean` otrzymuje wywołania zwrotne z kontenera Springa. Wywołania zwrotne dają egzemplarzowi komponentu możliwość przeprowadzenia pewnych akcji lub też dostarczają komponentowi informacje. Na przykład jeżeli komponent implementuje interfejs `ApplicationContextAware`, wówczas kontener wywołuje metodę `setApplicationContext()` egzemplarza komponentu w celu dostarczenia mu odwołania do egzemplarza `ApplicationContext`, w którym znajduje się dany komponent.

Interfejs `InitializingBean` definiuje metodę `afterPropertiesSet()` wywoływaną przez kontener Springa po ustawieniu właściwości. W metodzie `afterPropertiesSet()` są przeprowadzane zadania procesu inicjalizacji, takie jak nawiązanie połączenia z bazą danych, otwarcie pliku w celu jego odczytu itd. Interfejs `DisposableBean` definiuje metodę `destroy()` wywoływaną przez kontener Springa w chwili usuwania egzemplarza komponentu.



Podobnie jak w przypadku interfejsu cyklu życiowego `ApplicationContextAware` komponenty powinny implementować także interfejsy `InitializingBean` i `DisposableBean`, ponieważ wiążą one kod aplikacji z Springiem.

Przechodzimy teraz do adnotacji `@PostConstruct` i `@PreDestroy` wprowadzonych przez specyfikację JSR 250 i przeznaczonych do określenia metod inicjalizacji i usuwania obiektu.

Adnotacje `@PostConstruct` i `@PreDestroy` wprowadzone w specyfikacji JSR 250

Specyfikacja JSR 250 (najczęściej używane adnotacje na platformie Javy) definiuje standard adnotacji używanych przez różne technologie Javy. Adnotacje `@PostConstruct` i `@PreDestroy` służą do identyfikacji metod inicjalizacji i usunięcia obiektu. Klasa komponentu w Springu może określić metodę jako metodę inicjalizacji przez oznaczenie jej adnotacją `@PostConstruct`. Podobnie metoda może być określona jako metoda usunięcia przez oznaczenie jej adnotacją `@PreDestroy`.



Więcej informacji na temat specyfikacji JSR 205 znajdziesz na stronie <https://jcp.org/en/jsr/detail?id=250>.

IMPORT

Rozdział 05/ch05-bankapp-jsr250. (Ten projekt pokazuje aplikację `MyBank` używającą adnotacji `@PostConstruct` i `@PreDestroy` do określenia metod odpowiednio inicjalizacji i usuwania obiektu. Aby przetestować wykonanie metody zawierającej niestandardową logikę inicjalizacji, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu. Natomiast w celu przetestowania metody zawierającej niestandardową logikę usuwania egzemplarza należy wykonać metodę `main()` klasy `BankAppWithHook` projektu).

W listingu 5.7 została przedstawiona klasa `FixedDepositDaoImpl` projektu `ch05-bankapp-jsr250`, która używa adnotacji `@PostConstruct` i `@PreDestroy`.

Listing 5.7. Klasa *FixedDepositDaoImpl* przedstawiająca użycie adnotacji *@PostConstruct* i *@PreDestroy*Projekt: *ch05-bankapp-jsr250*Położenie pliku: *src/main/java/sample/spring/chapter05/bankapp/dao*

```

package sample.spring.chapter05.bankapp.dao;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class FixedDepositDaoImpl implements FixedDepositDao {
    private DatabaseConnection connection;
    .....
    @PostConstruct
    public void initializeDbConnection() {
        logger.info("Wywołanie metody initializeDbConnection() klasy
            FixedDepositDaoImpl.");
        connection = DatabaseConnection.getInstance();
    }
    .....
    @PreDestroy
    public void releaseDbConnection() {
        logger.info("Wywołanie metody releaseDbConnection() klasy
            FixedDepositDaoImpl.");
        connection.releaseConnection();
    }
}

```

W powyższym listingu klasa *FixedDepositDaoImpl* używa adnotacji *@PostConstruct* i *@PreDestroy* w celu identyfikacji metod inicjalizacji i usuwania obiektu. Warto pamiętać, że adnotacje *@PostConstruct* i *@PreDestroy* nie są charakterystyczne jedynie dla Springa.



Java SE 6 dostarcza adnotacje zdefiniowane przez specyfikację JSR 250. Jeżeli używasz Javy SE 6 lub nowszej, wówczas w ścieżce określonej przez zmienną środowiskową *CLASSPATH* nie musisz podawać pliku JAR dla JSR 250. Natomiast w przypadku użycia Javy SE 5 lub starszej trzeba w wymienionej ścieżce podać plik JAR dla JSR 250 oraz powiązanych z nim innych plików JAR.

Użycie adnotacji *@PostConstruct* i *@PreDestroy* w aplikacji wymaga przeprowadzenia w pliku XML kontekstu aplikacji konfiguracji oferowanej przez Spring klasy *CommonAnnotationBeanPostProcessor*. Przykład takiej konfiguracji przedstawia listing 5.8.

Listing 5.8. Plik *applicationContext.xml* przedstawiający przykład konfiguracji klasy*CommonAnnotationBeanPostProcessor*Projekt: *ch05-bankapp-jsr250*Położenie pliku: *src/main/resources/META-INF/spring*

```

<beans .....>
    <bean id="FixedDepositService"
        class="sample.spring.chapter05.bankapp.service.FixedDepositServiceImpl">
        <property name="myFixedDepositDao" ref="myFixedDepositDao" />
    </bean>

```

```
<bean id="myFixedDepositDao"
      class="sample.spring.chapter05.bankapp.dao.FixedDepositDaoImpl" />

<bean class="org.springframework.context.annotation.
      CommonAnnotationBeanPostProcessor"/>
</beans>
```

Klasa `CommonAnnotationBeanPostProcessor` implementuje oferowany przez Spring interfejs `BeanPostProcessor` (omówiony w kolejnym podrozdziale) i jest odpowiedzialna za przetwarzanie adnotacji JSR 250.

Jeżeli wykonasz metodę `main()` klas `BankApp` i `BankAppWithHook`, zauważysz, że oznaczone przez adnotacje `@PostConstruct` i `@PreDestroy` metody klasy `FixedDepositDaoImpl` są wykonywane w trakcie odpowiednio tworzenia i usuwania egzemplarzy `FixedDepositDaoImpl`.

Teraz przechodzimy do oferowanego przez Spring interfejsu `BeanPostProcessor` pozwalającego na pracę z nowo utworzonymi egzemplarzami komponentu *przed* ich zainicjowaniem lub *po* przez kontener Springa.

5.3. Użycie interfejsu `BeanPostProcessor` do pracy z nowo utworzonym egzemplarzem komponentu

Interfejs `BeanPostProcessor` jest używany do pracy z nowo utworzonymi egzemplarzami komponentu *przed* wywołaniem ich metod inicjacyjnych lub *po* (patrz: podrozdział 5.2) przez kontener Springa. Za pomocą interfejsu `BeanPostProcessor` można wykonać własną logikę *przed* wywołaniem lub *po* wywołaniu przez kontener Springa metody inicjacyjnej komponentu.



Komponent implementujący oferowany przez Spring interfejs `BeanPostProcessor` jest specjalnego typu. Kontener Springa automatycznie wykrywa i wykonuje komponent `BeanPostProcessor`.

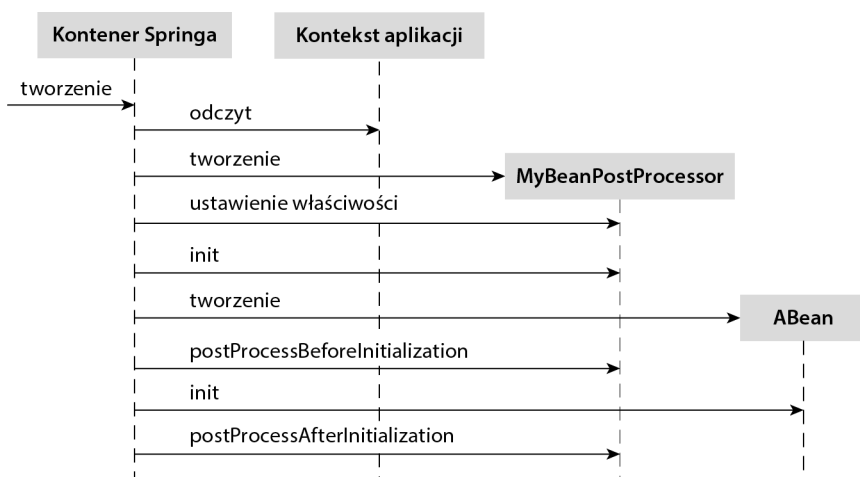
Interfejs `BeanPostProcessor` definiuje następujące metody:

- `Object postProcessBeforeInitialization(Object bean, String beanName)` —wywoływana *przed* wykonaniem metody inicjacyjnej egzemplarza komponentu.
- `Object postProcessAfterInitialization(Object bean, String beanName)` —wywoływana *po* wykonaniu metody inicjacyjnej egzemplarza komponentu.

Metody interfejsu `BeanPostProcessor` akceptują nowo utworzony egzemplarz komponentu i jego nazwę jako argumenty. Wartościami zwrótnymi metod są te same lub zmodyfikowane egzemplarze komponentu. Na przykład jeżeli w pliku XML kontekstu aplikacji skonfigurowano klasę `FixedDepositDaoImpl` jako komponent o wartości `id` wynoszącej `myFixedDepositDao` (patrz: listing 5.8), wówczas metody interfejsu `BeanPostProcessor` będą otrzymywały argumenty w postaci egzemplarza klasy `FixedDepositDaoImpl` i ciąg tekstowy `myFixedDepositDao`. Wartością zwrótną metody interfejsu `BeanPostProcessor` może być oryginalny lub zmodyfikowany egzemplarz komponentu, ewentualnie metoda zwróci obiekt opakowujący oryginalny egzemplarz komponentu.

Istnieje możliwość skonfigurowania w pliku XML kontekstu aplikacji implementacji `BeanPostProcessor` podobnie jak każdego innego komponentu Springa. Kontener Springa automatycznie wykrywa komponenty implementujące interfejs `BeanPostProcessor`, a następnie tworzy ich egzemplarze *przed* utworzeniem jakiegokolwiek innego egzemplarza komponentu zdefiniowanego w pliku XML kontekstu aplikacji. Po utworzeniu komponentów `BeanPostProcessor` kontener Springa wywołuje metody `postProcessBeforeInitialization()` i `postProcessAfterInitialization()` dla poszczególnych egzemplarzy komponentów utworzonych przez Spring.

Przyjmujemy założenie, że w pliku XML kontekstu aplikacji zdefiniowano komponent o zasięgu `singleton` i nazwie `ABean` oraz komponent implementujący interfejs `BeanPostProcessor` o nazwie `MyBeanPostProcessor`. Na rysunku 5.1 znajduje się diagram pokazujący kolejność, w jakiej metody komponentu `MyBeanPostProcessor` są wywoływane przez kontener Springa.



Rysunek 5.1. Kontener Springa wywołuje metody komponentu `MyBeanPostProcessor` przed inicjalizacją metod inicjalizacji komponentu `ABean` i po

Wywołanie metody `init()` w diagramie pokazanym na rysunku 5.1 przedstawia wywołanie metody inicjacyjnej komponentu. Diagram pokazuje, że egzemplarz `MyBeanPostProcessor` jest tworzony jeszcze przed komponentem `ABean`. Implementacja `BeanPostProcessor` jest konfigurowana tak samo jak każdy inny komponent, więc jeśli `MyBeanPostProcessor` definiuje metodę inicjacyjną, wówczas kontener wywołuje metodę inicjacyjną egzemplarza `MyBeanPostProcessor`. Po utworzeniu egzemplarza `ABean` następuje wywołanie przez kontener Springa metod typu setter egzemplarza `ABean` w celu spełnienia jego zależności oraz dostarczenia komponentowi niezbędnych informacji konfiguracyjnych. Po ustawieniu właściwości, ale jeszcze *przed* wywołaniem metody inicjacyjnej `ABean`, kontener Springa wywołuje metodę `postProcessBeforeInitialization()` obiektu `MyBeanPostProcessor`. Natomiast po wykonaniu metody inicjacyjnej `ABean` kontener Springa wywołuje metodę `postProcessAfterInitialization()` obiektu `MyBeanPostProcessor`.

Dopiero po wywołaniu metody `postProcessAfterInitialization()` egzemplarz komponentu jest przez kontener Springa uznawany za w pełni zainicjowany. Na przykład jeżeli komponent `BBean`

jest zależny od ABean, wówczas kontener Springa wstrzyknie egzemplarz ABean do BBean tylko po wykonaniu metody `postProcessAfterInitialization()` klasy `MyBeanPostProcessor` zarówno w egzemplarzu ABean, jak i BBean.

Warto zwrócić uwagę na to, że jeżeli w definicji komponentu `BeanPostProcessor` określono jego opóźnione tworzenie (patrz: atrybut `lazy-init` elementu `<bean>` lub atrybut `default-lazy-init` elementu `<beans>`, które omówiliśmy w podrozdziale 2.5), to kontener Springa ignoruje konfigurację opóźnionej inicjalizacji i tworzy egzemplarz komponentu `BeanPostProcessor` *przed* utworzeniem egzemplarzy komponentów o zasięgu singleton zdefiniowanych w pliku XML kontekstu aplikacji. Zwróć także uwagę, że komponenty implementujące interfejs `BeanFactoryPostProcessor` (omówiony w podrozdziale 5.4) są tworzone *przed* komponentami implementującymi interfejs `BeanPostProcessor`.

Przechodzimy teraz do przykładów, w których można użyć interfejsu `BeanPostProcessor`.

IMPORT **Rozdział 105/ch05-bankapp-beanpostprocessor.** (Ten projekt pokazuje aplikację `MyBank` używającą implementacji `BeanPostProcessor` do weryfikacji egzemplarzy komponentu oraz rozwiązywania jego zależności. Aby upewnić się o prawidłowym działaniu implementacji `BeanPostProcessor`, należy wykonać metodę `main()` w klasie `BankApp` projektu).

Przykład użycia interfejsu `BeanPostProcessor` — weryfikacja egzemplarzy komponentu

W aplikacji Springa może występować konieczność sprawdzenia, czy egzemplarz komponentu został prawidłowo skonfigurowany. Taka weryfikacja zwykle musi zostać przeprowadzona jeszcze przed wstrzyknięciem komponentu do komponentów zależnych lub przed uzyskaniem dostępu do niego z poziomu innych obiektów aplikacji. Zobaczmy więc, jak można użyć implementacji `BeanPostProcessor` w celu umożliwienia każdemu egzemplarzowi komponentu przeprowadzenia weryfikacji konfiguracji, zanim dany egzemplarz stanie się dostępny dla komponentów zależnych lub innych obiektów aplikacji.

W listingu 5.9 został przedstawiony interfejs `InstanceValidator`. Musi on być implementowany przez komponenty, które mają być zweryfikowane za pomocą implementacji `BeanPostProcessor`.

Listing 5.9. Interfejs `InstanceValidator`

Projekt: `ch05-bankapp-beanpostprocessor`

Położenie pliku: `src/main/java/sample/spring/chapter05/bankapp/common`

```
package sample.spring.chapter05.bankapp.common;

public interface InstanceValidator {
    void validateInstance();
}
```

Interfejs `InstanceValidator` definiuje metodę `validateInstance()` odpowiedzialną za sprawdzenie, czy dany egzemplarz komponentu został prawidłowo zainicjowany. Wkrótce zobaczysz, że metoda `validateInstance()` jest wywoływana przez implementację `BeanPostProcessor`.

W listingu 5.10 została przedstawiona klasa `FixedDepositDaoImpl` implementująca interfejs `InstanceValidator`.

Listing 5.10. Klasa `FixedDepositDaoImpl`

Projekt: `ch05-bankapp-beanpostprocessor`

Położenie pliku: `src/main/java/sample/spring/chapter05/bankapp/dao`

```
package sample.spring.chapter05.bankapp.dao;

import org.apache.log4j.Logger;
import sample.spring.chapter05.bankapp.common.InstanceValidator;
public class FixedDepositDaoImpl implements FixedDepositDao, InstanceValidator {
    private static Logger logger = Logger.getLogger(FixedDepositDaoImpl.class);
    private DatabaseConnection connection;

    public FixedDepositDaoImpl() {
        logger.info("Wywołanie konstruktora klasy FixedDepositDaoImpl.");
    }

    public void initializeDbConnection() {
        logger.info("Wywołanie metody initializeDbConnection() klasy FixedDepositDaoImpl.");
        connection = DatabaseConnection.getInstance();
    }

    @Override
    public void validateInstance() {
        logger.info("Weryfikacja egzemplarza FixedDepositDaoImpl.");
        if(connection == null) {
            logger.error("Nie udało się pobrać egzemplarza DatabaseConnection.");
        }
    }
}
```

W powyższym listingu `initializeDbConnection()` to metoda inicjacyjna, która pobiera egzemplarz `DatabaseConnection` przez wywołanie metody *statycznej* `getInstance()` klasy `DatabaseConnection`. Wartością atrybutu `connection` jest `null`, gdy egzemplarzowi `FixedDepositDaoImpl` nie udało się pobrać egzemplarza `DatabaseConnection`. Jeżeli atrybut `connection` przyjmuje wartość `null`, metoda `validateInstance()` generuje komunikat błędu informujący, że egzemplarz `FixedDepositDaoImpl` nie został prawidłowo zainicjowany. Ponieważ metoda inicjacyjna `initializeDbConnection()` jest odpowiedzialna za przypisanie wartości atrybutu `connection`, metoda `validateInstance()` *musi* być wywołana *po* `initializeDbConnection()`. W rzeczywistej aplikacji jeśli egzemplarz komponentu nie został prawidłowo skonfigurowany, wówczas metoda `validateInstance()` może podjąć pewne działania naprawcze lub zgłosić wyjątek w trakcie działania aplikacji, aby uniemożliwić jej uruchomienie. W celu zachowania prostoty omawianego przykładu, jeżeli egzemplarz komponentu nie został prawidłowo skonfigurowany, to metoda `validateInstance()` jedynie wyświetla odpowiedni komunikat.

W listingu 5.11 została przedstawiona klasa `InstanceValidationBeanPostProcessor` implementująca oferowany przez Spring interfejs `BeanPostProcessor` i odpowiedzialna za wywołanie metody `validateInstance()` nowo utworzonego komponentu.

Listing 5.11. Klasa *InstanceValidationBeanPostProcessor*Projekt: *ch05-bankapp-beanpostprocessor*Położenie pliku: *src/main/java/sample/spring/chapter05/bankapp/postprocessor*

```
package sample.spring.chapter05.bankapp.postprocessor;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.core.Ordered;

public class InstanceValidationBeanPostProcessor implements BeanPostProcessor, Ordered {
    private static Logger logger =
        Logger.getLogger(InstanceValidationBeanPostProcessor.class);
    private int order;

    public InstanceValidationBeanPostProcessor() {
        logger.info("Utworzono egzemplarz InstanceValidationBeanPostProcessor.");
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        logger.info("Wywołanie metody postProcessBeforeInitialization.");
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        logger.info("Wywołanie metody postProcessAfterInitialization.");
        if (bean instanceof InstanceValidator) {
            ((InstanceValidator) bean).validateInstance();
        }
        return bean;
    }

    public void setOrder(int order) {
        this.order = order;
    }

    @Override
    public int getOrder() {
        return order;
    }
}
```

W powyższym listingu widać, że klasa *InstanceValidationBeanPostProcessor* implementuje oferowane przez Spring interfejsy *BeanPostProcessor* i *Ordered*. Wartością zwrótną metody *postProcessBeforeInitialization()* jest po prostu egzemplarz komponentu przekazany metodzie. Jeżeli w metodzie *postProcessAfterInitialization()* okaże się, że egzemplarz komponentu jest typu *InstanceValidator*, wówczas nastąpi wywołanie jego metody *validateInstance()*. Dlatego jeżeli komponent implementuje interfejs *InstanceValidator*, klasa *InstanceValidationBeanPostProcessor* wywoła metodę *validateInstance()* egzemplarza komponentu *po* wywołaniu przez kontener Springa metody inicjacyjnej egzemplarza.

Interfejs `Ordered` definiuje metodę `getOrder()`, której wartością zwrótną jest liczba całkowita. Liczba zwrócona przez metodę `getOrder()` określa priorytet implementacji `BeanPostProcessor` względem innych implementacji `BeanPostProcessor` skonfigurowanych w pliku XML kontekstu aplikacji. Implementacja `BeanPostProcessor` o *większej* wartości `order` jest uznawana za mającą niższy priorytet i wykonywana *po* implementacjach `BeanPostProcessor` o mniejszej wartości `order`. Ponieważ chcemy, aby wartość liczbową zwracaną przez metodę `getOrder()` była skonfigurowana jako właściwość komponentu, metoda `setOrder()` i zmienna egzemplarza o nazwie `order` zostały zdefiniowane w klasie `InstanceValidationBeanPostProcessor`.

W listingu 5.12 została przedstawiona definicja komponentu dla klasy `InstanceValidationBeanPostProcessor`.

Listing 5.12. Definicja komponentu `InstanceValidationBeanPostProcessor`

Projekt: `ch05-bankapp-beanpostprocessor`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean class=".....bankapp.postprocessor.InstanceValidationBeanPostProcessor">
  <property name="order" value="1" />
</bean>
```

W powyższej definicji komponentu atrybut `id` elementu `<bean>` *nie* został określony, ponieważ zwykle nie chcemy zależności klasy `InstanceValidationBeanPostProcessor` od jakiegokolwiek innego komponentu. W elemencie `<property>` właściwość `order` została przypisana wartość 1.

Spójrzmy teraz na implementację `BeanPostProcessor` używaną w celu rozwiązywania zależności.

Przykład `BeanPostProcessor` — rozwiązywanie zależności komponentu

W rozdziale 4. zobaczyłeś, że jeśli komponent implementuje oferowany przez Spring interfejs `ApplicationContextAware`, może programowo pobierać egzemplarze komponentów za pomocą metody `getBean()` klasy `ApplicationContext`. Implementacja interfejsu `ApplicationContextAware` wiąże kod aplikacji ze Springiem i dlatego implementacja interfejsu `ApplicationContextAware` nie jest zalecana. W tym punkcie przeanalizujemy implementację `BeanPostProcessor` dostarczającą komponenty wraz z obiektem opakowującym egzemplarz `ApplicationContext`. Skutkiem jest kod aplikacji, który nie jest bezpośrednio zależny od interfejsów `ApplicationContextAware` i `ApplicationContext` oferowanych przez Spring.

W listingu 5.13 przedstawiamy użyty w aplikacji `MyBank` interfejs `DependencyResolver`. Ten interfejs jest implementowany przez komponenty, które w sposób programowy pobierają zależności z `ApplicationContext`.

Interfejs `DependencyResolver` definiuje metodę `resolveDependency()`, która akceptuje obiekt `MyApplicationContext` będący opakowaniem dla obiektu `ApplicationContext`. Wkrótce zobaczysz, że metoda `resolveDependency()` jest wywoływana przez implementację `BeanPostProcessor`.

W listingu 5.14 została przedstawiona klasa `FixedDepositServiceImpl` implementująca interfejs `DependencyResolver`.

Listing 5.13. *Interfejs DependencyResolver**Projekt: ch05-bankapp-beanpostprocessor**Położenie pliku: src/main/java/sample/spring/chapter05/bankapp/common*

```
package sample.spring.chapter05.bankapp.common;

public interface DependencyResolver {
    void resolveDependency(MyApplicationContext myApplicationContext);
}
```

Listing 5.14. *Klasa FixedDepositServiceImpl**Projekt: ch05-bankapp-beanpostprocessor**Położenie pliku: src/main/java/sample/spring/chapter05/bankapp/service*

```
package sample.spring.chapter05.bankapp.service;

import sample.spring.chapter05.bankapp.common.DependencyResolver;
import sample.spring.chapter05.bankapp.common.MyApplicationContext;

public class FixedDepositServiceImpl implements FixedDepositService, DependencyResolver {
    private FixedDepositDao fixedDepositDao;
    .....
    @Override
    public void resolveDependency(MyApplicationContext myApplicationContext) {
        FixedDepositDao = myApplicationContext.getBean(FixedDepositDao.class);
    }
}
```

Klasa `FixedDepositServiceImpl` definiuje atrybut `FixedDepositDao` typu `FixedDepositDao`. Metoda `resolveDependency()` jest odpowiedzialna za pobranie egzemplarza obiektu `FixedDepositDao` z `MyApplicationContext` (opakowanie dla obiektu `ApplicationContext`) i przypisanie go atrybutowi `FixedDepositDao`.

W listingu 5.15 została przedstawiona klasa `DependencyResolutionBeanPostProcessor` wywołująca metodę `resolveDependency()` w komponentach implementujących interfejs `DependencyResolver`.

Listing 5.15. *Klasa DependencyResolutionBeanPostProcessor**Projekt: ch05-bankapp-beanpostprocessor**Położenie pliku: src/main/java/sample/spring/chapter05/bankapp/postprocessor*

```
package sample.spring.chapter05.bankapp.postprocessor;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.core.Ordered;
import sample.spring.chapter05.bankapp.common.MyApplicationContext;

public class DependencyResolutionBeanPostProcessor implements BeanPostProcessor,
    Ordered {
    private MyApplicationContext myApplicationContext;
    private int order;

    public void setMyApplicationContext(MyApplicationContext myApplicationContext) {
        this.myApplicationContext = myApplicationContext;
    }
}
```

```

    public void setOrder(int order) {
        this.order = order;
    }

    @Override
    public int getOrder() {
        return order;
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        if (bean instanceof DependencyResolver) {
            ((DependencyResolver) bean).resolveDependency(myApplicationContext);
        }
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        return bean;
    }
}

```

Klasa `DependencyResolutionBeanPostProcessor` implementuje oferowane przez Spring interfejsy `BeanPostProcessor` i `Ordered`. Atrybut `myApplicationContext` (typu `MyApplicationContext`) przedstawia zależność od `DependencyResolutionBeanPostProcessor`. Metoda `postProcessBeforeInitialization()` wywołuje metodę `resolveDependency()` w egzemplarzu komponentu implementującym interfejs `DependencyResolver`, przekazując obiekt `MyApplicationContext` jako argument. Wartością zwrótną metody `postProcessAfterInitialization()` jest po prostu egzemplarz komponentu przekazany metodzie.

W listingu 5.16 została przedstawiona klasa `MyApplicationContext` działająca w charakterze opakowania dla oferowanego przez Spring obiektu `ApplicationContext`.

Listing 5.16. Klasa *MyApplicationContext*

Projekt: *ch05-bankapp-beanpostprocessor*

Położenie pliku: *src/main/java/sample/spring/chapter05/bankapp/common*

```

package sample.spring.chapter05.bankapp.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class MyApplicationContext implements ApplicationContextAware {
    private ApplicationContext applicationContext;

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

```

```

public <T> T getBean(Class<T> klass) {
    return applicationContext.getBean(klass);
}
}

```

Klasa `MyApplicationContext` implementuje udostępniany przez Spring interfejs `ApplicationContext` → `ContextAware` w celu pobrania odwołania do obiektu `ApplicationContext`, w którym znajduje się komponent. Klasa `MyApplicationContext` definiuje metodę `getBean()` zwracającą z egzemplarza `ApplicationContext` egzemplarz komponentu o danej nazwie.

W listingu 5.17 zostały przedstawione definicje komponentów dla klas `DependencyResolutionBeanPostProcessor` → `PostProcessor` i `MyApplicationContext`.

Listing 5.17. Plik `applicationContext.xml` zawierający definicje komponentów dla klas `DependencyResolutionBeanPostProcessor` i `MyApplicationContext`

Projekt: `ch05-bankapp-beanpostprocessor`

Położenie pliku: `src/main/resources/META-INF/spring`

```

<bean class="....postprocessor.DependencyResolutionBeanPostProcessor">
    <property name="myApplicationContext" ref="myApplicationContext" />
    <property name="order" value="0" />
</bean>

<bean id="myApplicationContext" class="....bankapp.common.MyApplicationContext" />

```

Definicja komponentu dla klasy `DependencyResolutionBeanPostProcessor` pokazuje, że jego właściwość `order` ma przypisaną wartość 0. W listingu 5.12 widziałeś, że w egzemplarzu `Instance` → `ValidationBeanPostProcessor` właściwość `order` miała przypisaną wartość 1. Ponieważ niższa wartość `order` oznacza wyższy priorytet, w egzemplarzu komponentu kontener Springa zastosuje najpierw `DependencyResolutionBeanPostProcessor`, a dopiero później `InstanceValidationBean` → `PostProcessor`.

W listingu 5.18 została przedstawiona metoda `main()` klasy `BankApp`, która sprawdza funkcjonalność `DependencyResolutionBeanPostProcessor` i `InstanceValidationBeanPostProcessor`.

Listing 5.18. Klasa `BankApp`

Projekt: `bankapp-beanpostprocessor`

Położenie pliku: `src/main/java/sample/spring/chapter05/bankapp`

```

package sample.spring.chapter05.bankapp;

public class BankApp {
    public static void main(String args[]) throws Exception {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");
        context.registerShutdownHook();

        FixedDepositService fixedDepositService =
            context.getBean(FixedDepositService.class);
        FixedDepositService.createFixedDeposit(new FixedDepositDetails(1, 1000, 12,
            "someemail@somedomain.com"));
    }
}

```

```

    }
    .....
}

```

Metoda `main()` klasy `BankApp` zwraca egzemplarz `FixedDepositService` pobrany z obiektu `ApplicationContext`, a następnie wykonuje metodę `createFixedDeposit()` klasy `FixedDepositService`. Podczas wykonywania metody `main()` klasy `BankApp` zauważysz, że kontener Springa tworzy egzemplarze komponentów `DependencyResolutionBeanPostProcessor` i `InstanceValidationBeanPostProcessor` *przed* utworzeniem jakichkolwiek innych komponentów zdefiniowanych w pliku XML kontekstu aplikacji. Komponent `DependencyResolutionBeanPostProcessor` (jego właściwość `order` ma wartość 0) jest zastosowany w nowo utworzonym egzemplarzu *przed* zastosowaniem komponentu `InstanceValidationBeanPostProcessor` (jego właściwość `order` ma wartość 1).

Zwróć uwagę, że kontener Springa nie stosuje implementacji `BeanPostProcessor` w innych implementacjach `BeanPostProcessor`. Na przykład w aplikacji `MyBank` metody `postProcessBeforeInitialization()` i `postProcessAfterInitialization()` klasy `DependencyResolutionBeanPostProcessor` nie są wywoływane przez kontener Springa podczas tworzenia egzemplarza `InstanceValidationBeanPostProcessor`.

Zobaczmy teraz, jak przedstawia się zachowanie implementacji `BeanPostProcessor` dla komponentu implementującego interfejs `FactoryBean`.

Zachowanie `BeanPostProcessor` w przypadku komponentów implementujących `FactoryBeans`

Z podrozdziału 3.9 dowiedziałeś się, że komponent implementujący oferowany przez Spring interfejs `FactoryBean` przedstawia fabrykę służącą do tworzenia egzemplarzy komponentów. Być może zadajesz sobie pytanie, czy implementacja `BeanPostProcessor` ma zastosowanie w implementacjach `FactoryBean` lub egzemplarzach utworzonych przez implementacje `FactoryBean`. W dalszej części rozdziału zobaczysz, że metody `postProcessBeforeInitialization()` i `postProcessAfterInitialization()` klasy `BeanPostProcessor` są wywoływane dla egzemplarza `FactoryBean` utworzonego przez kontener Spring. *Tylko* metoda `postProcessAfterInitialization()` jest wywoływana dla egzemplarzy komponentów utworzonych przez `FactoryBean`.

W listingu 5.19 została przedstawiona klasa `EventSenderFactoryBean` (implementacja `FactoryBean`) w aplikacji `MyBank`, która tworzy egzemplarze komponentu `EventSender`.

Listing 5.19. Klasa `EventSenderFactoryBean`

Projekt: `ch05-bankapp-beanpostprocessor`

Położenie pliku: `src/main/java/sample/spring/chapter05/bankapp/factory`

```

package sample.spring.chapter05.bankapp.factory;

import org.springframework.beans.factory.FactoryBean;
import org.springframework.beans.factory.InitializingBean;

public class EventSenderFactoryBean implements FactoryBean<EventSender>,
    InitializingBean {
    .....
    @Override

```

```

public EventSender getObject() throws Exception {
    logger.info("Wywołanie metody getObject klasy EventSenderFactoryBean.");
    return new EventSender();
}

@Override
public Class<?> getObjectType() {
    return EventSender.class;
}

@Override
public boolean isSingleton() {
    return false;
}

@Override
public void afterPropertiesSet() throws Exception {
    logger.info("Wywołanie metody afterPropertiesSet klasy EventSenderFactoryBean");
}
}

```

Klasa `EventSenderFactoryBean` implementuje oferowane przez Spring interfejsy `InitializingBean` i `FactoryBean`. Metoda `getObject()` zwraca egzemplarz obiektu `EventSender`. Ponieważ wartością zwrótną metody `isSingleton()` jest `false`, metoda `getObject()` klasy `EventSenderFactoryBean` jest wywoływana za każdym razem, gdy `EventSenderFactoryBean` otrzymuje żądanie obiektu `EventSender`.

W listingu 5.20 została przedstawiona metoda `main()` klasy `BankApp` w projekcie *ch05-bankapp-beanpostprocessor*, która pobiera egzemplarze `EventSender` z obiektu `EventSenderFactoryBean`.

Listing 5.20. Klasa `BankApp`

Projekt: *ch05-bankapp-beanpostprocessor*

Położenie pliku: *src/main/java/sample/spring/chapter05/bankapp*

```

package sample.spring.chapter05.bankapp;

public class BankApp {
    public static void main(String args[]) throws Exception {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");
        context.registerShutdownHook();
        ....
        context.getBean("eventSenderFactory");
        context.getBean("eventSenderFactory");
    }
}

```

W powyższym listingu metoda `getBean()` klasy `ApplicationContext` jest wywoływana dwukrotnie w celu pobrania dwóch różnych egzemplarzy `EventSender` z `EventSenderFactoryBean`. Jeżeli wykonasz metodę `main()` klasy `BankApp`, wówczas w konsoli zobaczysz wygenerowane następujące komunikaty:

Utworzenie `EventSenderFactoryBean`

Wywołanie metody `postProcessBeforeInitialization` klasy `DependencyResolutionBeanPostProcessor`
 dla.....`EventSenderFactoryBean`

```
Wywołanie metody postProcessBeforeInitialization klasy InstanceValidationBeanPostProcessor
↳ dla ....EventSenderFactoryBean
Wywołanie metody afterPropertiesSet klasy EventSenderFactoryBean
Wywołanie metody postProcessAfterInitialization klasy DependencyResolutionBeanPostProcessor
↳ dla ....EventSenderFactoryBean
Wywołanie metody postProcessAfterInitialization klasy InstanceValidationBeanPostProcessor
↳ dla ....EventSenderFactoryBean
```

W powyższych danych wyjściowych widać, że metody `postProcessBeforeInitialization()` i `postProcessAfterInitialization()` klasy `BeanPostProcessor` są wywoływane dla utworzonego przez kontener Springa egzemplarza `EventSenderFactoryBean`.

Wykonanie metody `main()` klasy `BankApp` powoduje wyświetlenie w konsoli następujących danych wyjściowych:

```
Wywołanie metody getObject klasy EventSenderFactoryBean
Wywołanie metody postProcessAfterInitialization klasy DependencyResolutionBeanPostProcessor
↳ dla ....EventSender
Wywołanie metody getObject klasy EventSenderFactoryBean
Wywołanie metody postProcessAfterInitialization klasy DependencyResolutionBeanPostProcessor
↳ dla ....EventSender
```

W powyższych danych wyjściowych wyraźnie widać, że *tylko* metoda `postProcessAfterInitialization()` klasy `BeanPostProcessor` jest wywoływana dla egzemplarza `EventSender` utworzonego przez `EventSenderFactoryBean`. Jeżeli chcesz, to w metodzie `postProcessAfterInitialization()` możesz wprowadzić modyfikacje dla egzemplarza `EventSender`.

Przechodzimy teraz do analizy wbudowanej w Spring implementacji `RequiredAnnotationBeanPostProcessor`, którą można wykorzystać do zagwarantowania, aby *wymagane* właściwości komponentu zostały skonfigurowane w pliku XML kontekstu aplikacji.

RequiredAnnotationBeanPostProcessor

Jeżeli metoda typu setter dla właściwości komponentu jest oznaczona adnotacją `@Required`, to oferowana przez Spring implementacja `RequiredAnnotationBeanPostProcessor` (odmiana implementacji `BeanPostProcessor`) sprawdza, czy dana właściwość komponentu jest skonfigurowana w pliku XML kontekstu aplikacji.



Należy zwrócić uwagę, że implementacja `RequiredAnnotationBeanPostProcessor` nie jest automatycznie rejestrowana w kontenerze Springa. Dlatego też trzeba ją samodzielnie zarejestrować przez zdefiniowanie w pliku XML kontekstu aplikacji.

W listingu 5.21 przedstawiamy przykład użycia adnotacji `@Required`.

Listing 5.21. Przykład użycia adnotacji `@Required`

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

public class FixedDepositServiceImpl implements FixedDepositService {
    private FixedDepositDao fixedDepositDao;
```

```
@Required
public void setFixedDepositDao(FixedDepositDao fixedDepositDao) {
    this.fixedDepositDao = fixedDepositDao;
}
.....
}
```

W powyższym listingu metoda typu setter o nazwie `setFixedDepositDao()` dla właściwości `FixedDepositDao` została oznaczona adnotacją `@Required`. Jeżeli w pliku XML kontekstu aplikacji masz zdefiniowaną implementację `RequiredAnnotationBeanPostProcessor`, wówczas `RequiredAnnotationBeanPostProcessor` sprawdzi, czy istnieje element `<property>` (lub czy użyto przestrzeni nazw `p`) do ustawienia wartości właściwości `FixedDepositDao`. Brak skonfigurowanej właściwości `FixedDepositDao` w definicji komponentu dla klasy `FixedDepositServiceImpl` znajdującej się w pliku XML kontekstu aplikacji będzie skutkowało zgłoszeniem wyjątku. To pokazuje, że implementację `RequiredAnnotationBeanPostProcessor` można wykorzystać do zagwarantowania, aby wszystkie egzemplarze komponentu w aplikacji zostały prawidłowo skonfigurowane w pliku XML kontekstu aplikacji.

Implementacja `RequiredAnnotationBeanPostProcessor` gwarantuje jedynie, że właściwość komponentu będzie skonfigurowana w jego definicji. Natomiast *nie* daje żadnej gwarancji, że właściwość będzie miała przypisaną poprawną wartość. Na przykład zamiast prawidłowej wartości można właściwości przypisać wartość `null`. Dlatego też komponenty mogą nadal wymagać implementacji metod inicjujących, które będą sprawdzały, czy właściwościom zostały przypisane prawidłowe wartości.

Przechodzimy teraz do oferowanego przez Spring interfejsu `DestructionAwareBeanPostProcessor`, który jest subinterfejsem `BeanPostProcessor`.

DestructionAwareBeanPostProcessor

Dotychczas zobaczyłeś, że implementacja `BeanPostProcessor` jest używana do pracy z nowo utworzonymi egzemplarzami komponentów. W pewnych sytuacjach może zachodzić potrzeba pracy z egzemplarzem komponentu przed jego usunięciem. Rozwiązaniem będzie skonfigurowanie w pliku XML kontekstu aplikacji komponentu implementującego oferowany przez Spring interfejs `DestructionAwareBeanPostProcessor`. `DestructionAwareBeanPostProcessor` to subinterfejs `BeanPostProcessor`, który definiuje następującą metodę:

```
void postProcessBeforeDestruction(Object bean, String beanName)
```

Jako argumenty metoda `postProcessBeforeDestruction()` akceptuje egzemplarz komponentu, który ma zostać usunięty przez kontener Springa. Metoda `postProcessBeforeDestruction()` jest przez kontener Springa wywoływana dla każdego egzemplarza komponentu o zasięgu singleton *przed* usunięciem danego egzemplarza. Najczęstszy powód użycia metody `postProcessBeforeDestruction()` to dostarczenie własnych metod usunięcia egzemplarza komponentu. Trzeba koniecznie zwrócić uwagę na fakt, że metoda `postProcessBeforeDestruction()` *nie* będzie wywoływana dla komponentów o zasięgu prototypu.

Przechodzimy teraz do oferowanego przez Spring interfejsu `BeanFactoryPostProcessor`, który pozwala na wprowadzanie modyfikacji w definicjach komponentów.

5.4. Modyfikacja definicji komponentu za pomocą `BeanFactoryPostProcessor`

Oferowany przez Spring interfejs `BeanFactoryPostProcessor` jest implementowany przez klasy, które mają wprowadzać modyfikacje w definicjach komponentów. Implementacja `BeanFactoryPostProcessor` jest wykonywana *po* wczytaniu definicji komponentów przez kontener Springa, ale jeszcze przed utworzeniem jakiegokolwiek egzemplarza. Utworzenie implementacji `BeanFactoryPostProcessor` następuje *przed* innymi komponentami zdefiniowanymi w pliku XML kontekstu aplikacji. W ten sposób implementacja `BeanFactoryPostProcessor` zyskuje możliwość wprowadzenia modyfikacji w definicjach innych komponentów. Konfigurację implementacji `BeanFactoryPostProcessor` przeprowadza się w pliku XML kontekstu aplikacji, podobnie jak w przypadku każdego innego komponentu Springa.



Jeżeli zamiast definicji komponentu chcesz modyfikować egzemplarze komponentów lub z nimi pracować, użyj implementacji `BeanPostProcessor` (patrz: podrozdział 5.3), a nie `BeanFactoryPostProcessor`.

Interfejs `BeanFactoryPostProcessor` definiuje pojedynczą metodę o nazwie `postProcessBeanFactory()`. Metoda ta akceptuje argument typu `ConfigurableListableBeanFactory`, który można wykorzystać do pobrania i modyfikacji definicji komponentów wczytanych przez kontener Springa. Istnieje możliwość utworzenia egzemplarza komponentu z poziomu metody `postProcessBeanFactory()`, co odbywa się przez wywołanie metody `getBean()` klasy `ConfigurableListableBeanFactory`. Jednak *nie* zaleca się tworzenia w metodzie `postProcessBeanFactory()`. Trzeba koniecznie pamiętać, że omówione w podrozdziale 5.3 implementacje `BeanPostProcessors` *nie* są wykonywane dla egzemplarzy komponentów utworzonych z poziomu metody `postProcessBeanFactory()`.

Należy także pamiętać, że klasa `ConfigurableListableBeanFactory` zapewnia dostęp do kontenera Springa, podobnie jak obiekt `ApplicationContext`. Ponadto `ConfigurableListableBeanFactory` pozwala na konfigurację kontenera Springa, iterację przez komponenty oraz modyfikowanie definicji komponentów. Na przykład za pomocą obiektu `ConfigurableListableBeanFactory` można zarejestrować edytor właściwości `PropertyEditorRegistrar` (patrz: podrozdział 3.6), zarejestrować `BeanPostProcessors` itd. W dalszej części rozdziału zobaczysz, jak obiekt `ConfigurableListableBeanFactory` jest używany do wprowadzania modyfikacji w definicjach komponentów.

Przechodzimy teraz do przeanalizowania sposobów użycia implementacji `BeanFactoryPostProcessor` do modyfikowania definicji komponentów.

IMPORT **Rozdział 105/ch05-bankapp-beanfactorypostprocessor.** (Ten projekt pokazuje aplikację `MyBank`, w której użyto implementacji `BeanFactoryPostProcessor` w celu wyłączenia w aplikacji funkcji automatycznego wiązania. Jeżeli okaże się, że komponent o zasięgu singleton jest zależny

od komponentu o zasięgu prototypu, wówczas aplikacja wyświetli komunikat błędu. Aby upewnić się o prawidłowym działaniu implementacji `BeanFactoryPostProcessor`, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Przykład użycia `BeanFactoryPostProcessor`

W poprzednim rozdziale zobaczyłeś, że automatyczne wiązanie powoduje ukrycie ogólnej struktury aplikacji (patrz: podrozdział 4.6). Wspomnieliśmy także, że aby określić zależność komponentu o zasięgu singleton od komponentu o zasięgu prototypu, należy zamiast elementu `<property>` użyć elementu `<lookup-method>` lub `<replaced-method>` (więcej informacji na ten temat znajdziesz w podrozdziałach 4.4 i 4.5) do programowego pobrania zależności. Teraz przechodzimy do implementacji `BeanFactoryPostProcessor`, która powoduje *uniedostępnienie* komponentów dla funkcji automatycznego wiązania (patrz: omówienie w podrozdziale 4.6 atrybutu `autowire-candidate` elementu `<bean>`) i wyświetlenie komunikatu błędu, jeśli komponent o zasięgu singleton okaże się zależny od komponentu o zasięgu prototypu. W celu zachowania prostoty przyjęto założenie, że komponent o zasięgu singleton używa elementu `<property>` do określenia zależności od komponentu o zasięgu prototypu.



Komponent implementujący oferowany przez Spring interfejs `BeanFactoryPostProcessor` jest *specjalnym* typem komponentu. Kontener Springa automatycznie wykrywa i wykonuje komponenty `BeanFactoryPostProcessor`.

W listingu 5.22 została przedstawiona istniejąca w aplikacji `MyBank` klasa `ApplicationConfigurer`, która implementuje interfejs `BeanFactoryPostProcessor`.

Listing 5.22. Klasa `ApplicationConfigurer`, czyli przykład implementacji `BeanFactoryPostProcessor`

Projekt: `ch05-bankapp-beanfactorypostprocessor`

Położenie pliku: `src/main/java/sample/spring/chapter05/bankapp/postprocessor`

```
package sample.spring.chapter05.bankapp.postprocessor;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;

public class ApplicationConfigurer implements BeanFactoryPostProcessor {

    public ApplicationConfigurer() {
        logger.info("Utworzono egzemplarz ApplicationConfigurer.");
    }

    @Override
    public void postProcessBeanFactory(
        ConfigurableListableBeanFactory beanFactory) throws BeansException {
        String[] beanDefinitionNames = beanFactory.getBeanDefinitionNames();

        // Pobranie definicji wszystkich komponentów.
        for (int i = 0; i < beanDefinitionNames.length; i++) {
            String beanName = beanDefinitionNames[i];
```

```

        BeanDefinition beanDefinition = beanFactory.getBeanDefinition(beanName);
        beanDefinition.setAutowiredCandidate(false);

        //Pobranie zależności komponentu.
        if (beanDefinition.isSingleton()) {
            if (hasPrototypeDependency(beanFactory, beanDefinition)) {
                logger.error("Komponent o zasięgu singleton " + beanName
                    + " jest zależny od komponentu o zasięgu prototypu.");
            }
        }
    }
}
.....
}

```

W metodzie `postProcessBeanFactory` wykonywana jest następująca sekwencja zadań:

1. Przede wszystkim metoda `postProcessBeanFactory()` wywołuje metodę `getBeanDefinitionNames()` klasy `ConfigurableListableBeanFactory` w celu pobrania nazw wszystkich definicji komponentów wczytanych przez kontener Springa. Zwróć uwagę, że nazwa definicji komponentu jest wartością atrybutu `id` elementu `<bean>`.
2. Po pobraniu wszystkich definicji komponentów metoda `postProcessBeanFactory()` wywołuje metodę `getBeanDefinition()` klasy `ConfigurableListableBeanFactory` w celu pobrania obiektów `BeanDefinition` odpowiadających poszczególnym definicjom komponentów. Jako argument metoda `getBeanDefinition()` pobiera nazwę definicji komponentu ustaloną w kroku 1.
3. Obiekt `BeanDefinition` przedstawia definicję komponentu i może być użyty do modyfikacji jego konfiguracji. Dla każdej definicji wczytanej przez kontener Springa metoda `postProcessBeanFactory()` wywołuje metodę `setAutowiredCandidate()` klasy `BeanDefinition` w celu uniedostępnienia *wszystkich* komponentów dla funkcji automatycznego wiązania.
4. Wartością zwrótną metody `isSingleton()` klasy `BeanDefinition` jest `true`, jeżeli definicja komponentu dotyczy komponentu o zasięgu singleton. W takim przypadku metoda `postProcessBeanFactory()` wywołuje metodę `hasPrototypeDependency()` w celu sprawdzenia, czy dany komponent singleton jest zależny od jakiegokolwiek komponentu o zasięgu prototypu. Jeżeli wspomniana zależność zostanie znaleziona, metoda `postProcessBeanFactory()` wyświetli komunikat błędu.

W listingu 5.23 została przedstawiona implementacja metody `hasPrototypeDependency()` klasy `ApplicationConfigurer` zwracającej wartość `true`, jeśli komponent o zasięgu singleton jest zależny od komponentu o zasięgu prototypu.

Listing 5.23. Metoda `hasPrototypeDependency()` klasy `ApplicationConfigurer`

Projekt: `ch05-bankapp-beanfactorypostprocessor`

Położenie pliku: `src/main/java/sample/spring/chapter05/bankapp/postprocessor`

```

import org.springframework.beans.MutablePropertyValues;
import org.springframework.beans.PropertyValue;
import org.springframework.beans.factory.config.RuntimeBeanReference;

public class ApplicationConfigurer implements BeanFactoryPostProcessor {
    .....
}

```

```

private boolean hasPrototypeDependency(ConfigurableListableBeanFactory beanFactory,
    BeanDefinition beanDefinition) {
    boolean isPrototype = false;
    MutablePropertyValues mutablePropertyValues =
        beanDefinition.getPropertyValues();
    PropertyValue[] propertyValues = mutablePropertyValues.getPropertyValues();

    for (int j = 0; j < propertyValues.length; j++) {
        if (propertyValues[j].getValue() instanceof RuntimeBeanReference) {
            String dependencyBeanName = ((RuntimeBeanReference) propertyValues[j]
                .getValue()).getBeanName();
            BeanDefinition dependencyBeanDef = beanFactory
                .getBeanDefinition(dependencyBeanName);
            if (dependencyBeanDef.isPrototype()) {
                isPrototype = true;
                break;
            }
        }
    }
    return isPrototype;
}

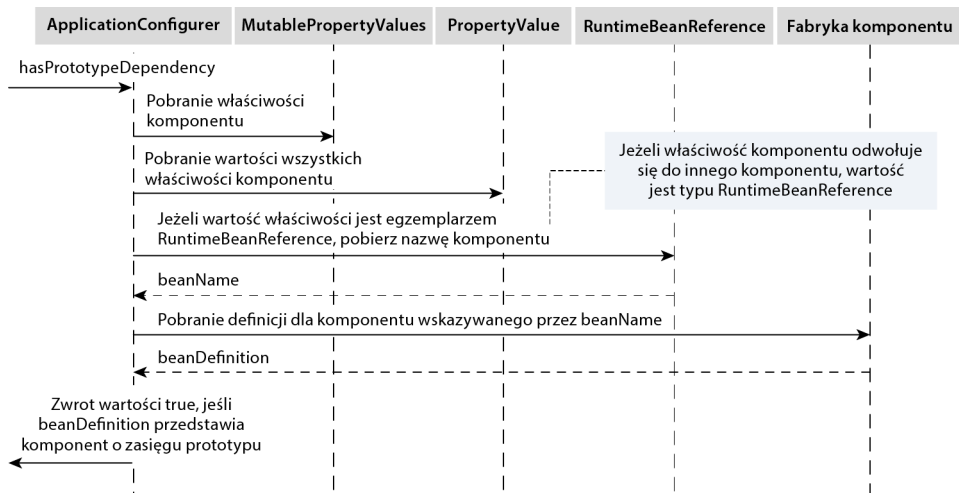
```

Metoda `hasPrototypeDependency()` sprawdza, czy komponent przedstawiany przez argument `BeanDefinition` jest zależny od komponentu o zasięgu prototypu. Argument `ConfigurableListableBeanFactory` zapewnia dostęp do definicji komponentów wczytanych przez kontener Springa. Wymieniona poniżej sekwencja zdarzeń jest wykonywana przez metodę `hasPrototypeDependency()` w celu ustalenia, czy komponent przedstawiany przez `BeanDefinition` ma zależność w postaci komponentu o zasięgu prototypu:

1. Przede wszystkim metoda `hasPrototypeDependency()` wywołuje metodę `getPropertyValues()` klasy `BeanDefinition` w celu pobrania właściwości komponentu zdefiniowanych przez elementy `<property>`. Wartością zwrótną metody `getPropertyValues()` klasy `BeanDefinition` jest obiekt typu `MutablePropertyValues`, za pomocą którego można zmodyfikować właściwości komponentu. Na przykład dodatkowe właściwości do definicji komponentu można dodać za pomocą metod `addPropertyValue()` i `addPropertyValues()` klasy `MutablePropertyValues`.
2. Ponieważ chcemy przeprowadzić iterację przez wszystkie właściwości komponentu i sprawdzić, czy którakolwiek z nich odwołuje się do komponentu o zasięgu prototypu, metoda `getPropertyValues()` klasy `MutablePropertyValues` jest wywoływana w celu pobrania tablicy obiektów `PropertyValue`. Obiekt `PropertyValue` przechowuje informacje o właściwości komponentu.
3. Jeżeli właściwość komponentu odwołuje się do komponentu Springa, wywołanie metody `getValue()` klasy `PropertyValue` zwraca egzemplarz obiektu `RuntimeBeanReference` przechowującego nazwę komponentu, którego dotyczy odwołanie. Skoro jesteśmy zainteresowani właściwościami komponentu odwołującymi się do komponentów Springa, wartość zwrótna metody `getValue()` klasy `PropertyValue` jest sprawdzana pod kątem tego, czy przedstawia egzemplarz typu `RuntimeBeanReference`. Jeżeli tak, wtedy obiekt zwrócony przez metodę `getValue()` będzie rzutowany na typ `RuntimeBeanReference`, a nazwa komponentu, którego dotyczy odwołanie, zostanie ustalona za pomocą wywołania metody `getBeanName()` klasy `RuntimeBeanReference`.

4. Mając ustaloną nazwę komponentu wskazywanego przez właściwość, obiekt `BeanDefinition` dla wspomnianego komponentu jest pobierany za pomocą wywołania metody `getBeanDefinition()` klasy `ConfigurableListableBeanFactory`. Oczywiście można sprawdzić, czy wskazywany komponent ma zasięg prototypu. W tym celu należy wywołać metodę `isPrototype()` klasy `BeanDefinition`.

Diagram na rysunku 5.2 przedstawia podsumowanie dotyczące sposobu działania metody `hasPrototypeDependency()`.



Rysunek 5.2. Metoda `hasPrototypeDependency()` przeprowadza iterację przez zależności definicji komponentu i zwraca wartość `true` w przypadku znalezienia zależności w postaci komponentu o zasięgu prototypu

W diagramie na rysunku 5.2 obiekt `ConfigurableListableBeanFactory` został przedstawiony jako *fabryka komponentu*.

W listingu 5.24 został przedstawiony plik XML kontekstu aplikacji w projekcie `ch05-bankapp-beanfactorypostprocessor` zawierający definicje komponentów dla klas `ApplicationConfigurer` (implementacja `BeanFactoryPostProcessor`), `InstanceValidationBeanPostProcessor` (implementacja `BeanPostProcessor`) oraz definicje obiektów charakterystycznych dla danej aplikacji.

Listing 5.24. Plik `applicationContext.xml` przedstawiający definicję komponentu implementacji `BeanFactoryPostProcessor`

Projekt: `ch05-bankapp-beanfactorypostprocessor`
 Położenie pliku: `src/main/resources/META-INF/spring`

```

<beans .....>
.....
  <bean id="FixedDepositDao"
    class="sample.spring.chapter05.bankapp.dao.FixedDepositDaoImpl"..... >
    <property name="fixedDepositDetails" ref="FixedDepositDetails" />
  </bean>

  <bean id="FixedDepositDetails"

```

```

        class="sample.spring.chapter05.bankapp.domain.FixedDepositDetails"
        scope="prototype" />

<bean class="....postprocessor.InstanceValidationBeanPostProcessor">
  <property name="order" value="1" />
</bean>

<bean
  class="sample.spring.chapter05.bankapp.postprocessor.ApplicationConfigurer" />
</beans>

```

W przedstawionej powyżej definicji komponent o zasięgu singleton (`FixedDepositDao`) jest zależny od komponentu o zasięgu prototypu (`FixedDepositDetails`).

Po wykonaniu metody `main()` klasy `BankApp` projektu *ch05-bankapp-beanfactorypostprocessor* w konsoli zostaną wygenerowane następujące dane wyjściowe:

```

Utworzono egzemplarz ApplicationConfigurer.
Komponent o zasięgu singleton FixedDepositDao jest zależny od komponentu o zasięgu prototypu.
Utworzono egzemplarz InstanceValidationBeanPostProcessor.

```

W powyższych danych wyjściowych wyraźnie widać, że kontener Springa tworzy egzemplarz `ApplicationConfigurer` (implementacja `BeanFactoryPostProcessor`) i wykonuje jego metodę `postProcessBeanFactory()` jeszcze *przed* utworzeniem egzemplarza `InstanceValidationBeanPostProcessor` (implementacja `BeanPostProcessor`). Trzeba koniecznie zwrócić uwagę na fakt, że komponenty implementujące interfejs `BeanFactoryPostProcessor` są przetwarzane *przed* komponentami implementującymi interfejs `BeanPostProcessor`. Dlatego też *nie* można wykorzystać implementacji `BeanPostProcessor` do wprowadzania modyfikacji w egzemplarzu `BeanFactoryPostProcessor`. Implementacja `BeanFactoryPostProcessor` daje możliwość modyfikacji definicji komponentów wczytanych przez kontener Springa, natomiast implementacja `BeanFactoryPostProcessor` pozwala na wprowadzanie modyfikacji w nowo utworzonych egzemplarzach komponentów.

Spójrzmy teraz na pewne podobieństwa zachodzące między implementacjami `BeanPostProcessors` i `BeanFactoryPostProcessors`:

- W pliku XML kontekstu aplikacji można skonfigurować wiele implementacji `BeanFactoryPostProcessors`. W celu kontrolowania kolejności wykonywania przez kontener Springa poszczególnych implementacji `BeanFactoryPostProcessors` należy wykorzystać oferowany przez Spring interfejs `Ordered` (więcej informacji na jego temat znajdziesz w podrozdziale 5.3).
- Nawet po określeniu, że implementacja `BeanFactoryPostProcessors` ma być inicjalizowana z opóźnieniem przez kontener Springa, egzemplarze `BeanFactoryPostProcessors` są tworzone w chwili tworzenia egzemplarza kontenera Springa.

W rozdziale 3. omówiliśmy edytor `CustomEditorConfigurer`, czyli oferowaną przez Spring implementację `BeanFactoryPostProcessors`, która pozwala na rejestrację własnych edytorów właściwości. Przechodzimy teraz do omówienia kolejnych implementacji `BeanFactoryPostProcessors` standardowo dostarczanych przez Spring.

Komponent PropertySourcesPlaceholderConfigurer

Dotychczas przedstawiliśmy przykłady definicji komponentów, w których atrybut `value` elementu `<property>` lub `<constructor-arg>` był używany do określenia rzeczywistej wartości dla właściwości komponentu lub argumentu konstruktora. `PropertySourcesPlaceholderConfigurer` (implementacja `BeanFactoryPostProcessor`) pozwala na podanie w pliku właściwości ciągu tekstowego wartości dla właściwości komponentu lub argumentu konstruktora. W definicji komponentu trzeba jedynie podać *miejsce zarezerwowane właściwości* (w postaci `${<nazwa_właściwości_w_pliku_właściwości>}`) jako wartość atrybutu `value` elementu `<property>` lub `<constructor-arg>`. Kiedy definicje komponentów są wczytywane przez kontener Springa, `PropertySourcesPlaceholderConfigurer` pobiera faktyczne wartości z pliku właściwości, a następnie umieszcza je w miejscach zarezerwowanych znajdujących się w definicji komponentu.

IMPORT **Rozdział 105/ch05-propertySourcesPlaceholderConfigurer-example.** (Ten projekt pokazuje aplikację Springa używającą `PropertySourcesPlaceholderConfigurer` w celu ustawienia właściwości na podstawie wartości podanych w pliku właściwości. Aby upewnić się o prawidłowym działaniu `PropertySourcesPlaceholderConfigurer`, wystarczy wywołać metodę `main()` w klasie `SampleApp` projektu).

W listingu 5.25 zostały przedstawione definicje komponentów dla klas `DataSource` i `WebServiceConfiguration`, które używają miejsc zarezerwowanych dla właściwości.

Listing 5.25. *Plik `applicationContext.xml` przedstawiający definicje komponentów używające miejsc zarezerwowanych*

Projekt: `ch05-propertySourcesPlaceholderConfigurer-example`
Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="datasource" class="sample.spring.chapter05.domain.DataSource">
  <property name="url" value="${database.url}" />
  <property name="username" value="${database.username}" />
  <property name="password" value="${database.password}" />
  <property name="driverClass" value="${database.driverClass}" />
</bean>

<bean id="webServiceConfiguration"
  class="sample.spring.chapter05.domain.WebServiceConfiguration">
  <property name="webServiceUrl" value="${webservice.url}" />
</bean>
```

W powyższym listingu widać, że wartości `value` poszczególnych elementów `<property>` określają miejsca zarezerwowane dla właściwości. Kiedy definicja komponentu jest wczytywana przez kontener Springa, wówczas `PropertySourcesPlaceholderConfigurer` zastępuje miejsca zarezerwowane wartościami pobranymi z pliku właściwości. Na przykład jeśli plik właściwości zawiera zdefiniowaną właściwość o nazwie `database.username`, wtedy jej wartość zastąpi miejsce zarezerwowane `${database.username}` w komponencie `dataSource`.

Definicja komponentu dla `PropertySourcesPlaceholderConfigurer` określa, że pliki właściwości mają być przeszukiwane w celu dostarczenia wartości dla miejsc zarezerwowanych, jak pokazuje listing 5.26.

Listing 5.26. Plik *applicationContext.xml* przedstawiający definicję komponentu*PropertySourcesPlaceholderConfigurer*Projekt: *ch05-propertySourcesPlaceholderConfigurer-example*Położenie pliku: *src/main/resources/META-INF/spring*

```

<bean class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
  <property name="locations">
    <list>
      <value>classpath:database.properties</value>
      <value>classpath:webservice.properties</value>
    </list>
  </property>
  <property name="ignoreUnresolvablePlaceholders" value="false" />
</bean>

```

Właściwość *locations* komponentu *PropertySourcesPlaceholderConfigurer* wskazuje, że pliki właściwości będą przeszukiwane w celu pobrania wartości dla miejsc zarezerwowanych. W powyższym listingu komponent *PropertySourcesPlaceholderConfigurer* przeszukuje pliki *database.properties* i *webservice.properties* pod kątem wartości dla miejsc zarezerwowanych. Właściwość *ignoreUnresolvablePlaceholders* określa zachowanie komponentu *PropertySourcesPlaceholderConfigurer* w przypadku niezalezienia w żadnym z wymienionych plików wartości dla miejsca zarezerwowanego. Możliwe jest po prostu zignorowanie braku wartości lub zgłoszenie wyjątku. Wartość *false* oznacza, że komponent *PropertySourcesPlaceholderConfigurer* zgłosi wyjątek, jeżeli w pliku właściwości *database.properties* lub *webservice.properties* nie zostanie znaleziona wartość dla miejsca zarezerwowanego.

W listingu 5.27 zostały przedstawione właściwości zdefiniowane w plikach *database.properties* i *webservice.properties*.

Listing 5.27. Właściwości zdefiniowane w plikach *database.properties* i *webservice.properties*Projekt: *ch05-propertySourcesPlaceholderConfigurer-example*Położenie pliku: *src/main/resources/META-INF*

```

----- Plik database.properties -----
database.url=some_url
database.username=some_username
database.password=some_password
database.driverClass=some_driverClass

----- Plik webservice.properties -----
webservice.url=some_url

```

Jeżeli właściwości zdefiniowane w plikach *database.properties* i *webservice.properties* porównasz z miejscami zarezerwowanymi w definicjach komponentów *datasource* i *WebServiceConfiguration* (patrz: listing 5.25), zauważysz, że dla każdego z tych miejsc w jednym z plików została zdefiniowana właściwość.

Metoda *main()* klasy *SampleApp* projektu *ch05-propertySourcesPlaceholderConfigurer-example* pobiera komponenty *WebServiceConfiguration* i *DataSource* z egzemplarza *ApplicationContext*, a następnie

wyświetla w konsoli ich właściwości. Jeżeli wykonasz metodę `main()` klasy `SampleApp`, w konsoli otrzymasz następujące dane wyjściowe:

```
DataSource [url=some_url, username=some_username, password=some_password,
↳driverClass=some_driverClass]
WebServiceConfiguration [webServiceUrl=some_url]
```

Na podstawie powyższych danych wyjściowych możemy wyciągnąć następujące wnioski:

- Właściwość `url` komponentu `DataSource` ma przypisaną wartość `some_url`, właściwości `username` przypisano wartość `some_username`, właściwości `password` wartość `some_password`, natomiast właściwości `driverClass` wartość `some_driverClass`.
- Właściwość `webServiceUrl` komponentu `WebServiceConfiguration` ma przypisaną wartość `some_url`.

W przypadku usunięcia jakiegokolwiek właściwości z pliku `database.properties` lub `webservice.properties` wykonanie metody `main()` klasy `SampleApp` spowoduje zgłoszenie wyjątku.

Przystępujemy teraz do omówienia właściwości `localOverride` komponentu `PropertySourcesPlaceholderConfigurer`.

Właściwość `localOverride`

Jeżeli właściwości lokalne (zdefiniowane za pomocą elementu `<props>`) mają nadpisywać odczytane z pliku właściwości, wtedy można przypisać wartość `true` właściwości `localOverride` komponentu `PropertySourcesPlaceholderConfigurer`.

IMPORT **Rozdział05/ch05-localOverride-example.** (Ten projekt pokazuje aplikację Springa używającą właściwości `localOverride` komponentu `PropertySourcesPlaceholderConfigurer`. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `SampleApp` projektu).

W listingu 5.28 zostały przedstawione definicje komponentów dla klas `DataSource` i `WebServiceConfiguration`.

Listing 5.28. Plik `applicationContext.xml` przedstawiający definicje komponentów używających miejsc zarezerwowanych dla właściwości

Projekt: `ch05-localOverride-example`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="datasource" class="sample.spring.chapter05.domain.DataSource">
  <property name="url" value="${database.url}" />
  <property name="username" value="${database.username}" />
  <property name="password" value="${database.password}" />
  <property name="driverClass" value="${database.driverClass}" />
</bean>

<bean id="webServiceConfiguration"
  class="sample.spring.chapter05.domain.WebServiceConfiguration">
  <property name="webServiceUrl" value="${webservice.url}" />
</bean>
```


Definicje komponentów dla klas `DataSource` i `WebServiceConfiguration` są teraz takie same jak w listingu 5.25.

W listingu 5.29 zostały przedstawione właściwości zdefiniowane w plikach *database.properties* i *webservice.properties*.

Listing 5.29. Właściwości zdefiniowane w plikach *database.properties* i *webservice.properties*

Projekt: *ch05-localOverride-example*

Położenie pliku: *src/main/resources/META-INF*

```
----- Plik database.properties -----
database.url=some_url
database.username=some_username

----- Plik webservice.properties -----
webservice.url=some_url
```

Jeżeli właściwości zdefiniowane w plikach *database.properties* i *webservice.properties* porównasz z miejscami zarezerwowanymi w definicjach komponentów `datasource` i `webServiceConfiguration` (patrz: listing 5.28), wówczas w pliku *database.properties* zauważysz brak zdefiniowanych właściwości dla miejsc zarezerwowanych `${database.password}` i `${database.driverClass}`.

W listingu 5.30 została przedstawiona definicja komponentu dla klasy `PropertySourcesPlaceholderConfigurer`.

Listing 5.30. Plik *applicationContext.xml* przedstawiający definicję komponentu dla klasy *PropertySourcesPlaceholderConfigurer*

Projekt: *ch05-localOverride-example*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
  <property name="locations">
    <list>
      <value>classpath:database.properties</value>
      <value>classpath:webservice.properties</value>
    </list>
  </property>
  <property name="properties">
    <props>
      <prop key="database.password">locally-set-password</prop>
      <prop key="database.driverClass">locally-set-driverClass</prop>
      <prop key="webservice.url">locally-set-webServiceUrl</prop>
    </props>
  </property>
  <property name="ignoreUnresolvablePlaceholders" value="false" />
  <property name="localOverride" value="true" />
</bean>
```

Właściwość `properties` komponentu `PropertySourcesPlaceholderConfigurer` definiuje właściwości *lokalne*. Dlatego też `database.password`, `database.driverClass` i `webservice.url` są właściwościami

lokalnymi. Właściwość `localOverride` określa, czy właściwości lokalne mają pierwszeństwo przed właściwościami odczytanymi z zewnętrznego pliku właściwości. Ponieważ wartość właściwości `localOverride` wynosi `true`, właściwości lokalne mają pierwszeństwo.

Metoda `main()` klasy `SampleApp` projektu *ch05-localOverride-example* pobiera komponenty `WebServiceConfiguration` i `DataSource` z egzemplarza `ApplicationContext`, a następnie wyświetla w konsoli ich właściwości. Jeżeli wykonasz metodę `main()` klasy `SampleApp`, w konsoli otrzymasz następujące dane wyjściowe:

```
DataSource [url=some_url, username=some_username, password=locally-set-password,
↳driverClass=locally-set-driverClass]
WebServiceConfiguration [webServiceUrl=locally-set-webServiceUrl]
```

Powyższe dane wyjściowe pokazują, że wartościami właściwości `password` i `driverClass` komponentu `DataSource` są odpowiednio `locally-set-password` i `locally-set-driverClass`. Dlatego też wartości dla właściwości `password` i `driverClass` komponentu `DataSource` pochodzą z właściwości lokalnie zdefiniowanych przez komponent `PropertySourcesPlaceholderConfigurer` (patrz: listing 5.30). To pokazuje, że jeśli komponent `PropertySourcesPlaceholderConfigurer` nie znajduje w zewnętrznych plikach właściwości wartości dla danego miejsca zarezerwowanego, wówczas przystępuje do jej szukania we właściwościach lokalnych zdefiniowanych przez komponent `PropertySourcesPlaceholderConfigurer`. Powyższe dane wyjściowe pokazują także, że wartość właściwości `webServiceUrl` komponentu `WebServiceConfiguration` jest pobierana z właściwości lokalnych zdefiniowanych przez komponent `PropertySourcesPlaceholderConfigurer` (patrz: listing 5.30). Wartością właściwości `localOverride` komponentu `PropertySourcesPlaceholderConfigurer` jest `true`, a więc lokalnie zdefiniowana właściwość `webService.url` ma pierwszeństwo przed właściwością o tej samej nazwie odczytaną z pliku *webService.properties*.



Podczas definiowania właściwości lokalnych alternatywą dla właściwości `properties` komponentu `PropertySourcesPlaceholderConfigurer` jest użycie elementu `<properties>` schematu `util` oferowanego przez Spring (patrz: podrozdział 3.8) lub implementacji `PropertiesFactoryBean`.

Zamiast bezpośrednio konfigurować komponent `PropertySourcesPlaceholderConfigurer` w pliku XML kontekstu aplikacji można skorzystać z elementu `<property-placeholder>` schematu `context` oferowanego przez Spring. Element `<property-placeholder>` przeprowadza konfigurację egzemplarza `PropertySourcesPlaceholderConfigurer`. Przechodzimy więc do dokładniejszej analizy elementu `<property-placeholder>`.

IMPORT **Rozdział 105/ch05-property-placeholder-element-example.** (Ten projekt pokazuje aplikację Springa, w której użyto elementu `<property-placeholder>`. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `SampleApp` projektu).

Element `<property-placeholder>`

W listingu 5.31 przedstawiamy sposób użycia elementu `<property-placeholder>` do przeprowadzenia konfiguracji egzemplarza `PropertySourcesPlaceholderConfigurer`. Sama konfiguracja jest dokładnie taka sama jak w listingu 5.30.

Listing 5.3.1. Plik `applicationContext.xml` przedstawiający użycie elementu `<property-placeholder>`Projekt: `ch05-property-placeholder-element-example`Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:util="http://www.springframework.org/schema/util" ....>
  ....
  <context:property-placeholder ignore-unresolvable="false"
    location="classpath:database.properties, classpath:webService.properties"
    local-override="true" order="1" properties-ref="localProps" />
  <util:properties id="localProps">
    <prop key="database.password">locally-set-password</prop>
    <prop key="database.driverClass">locally-set-driverClass</prop>
    <prop key="webService.url">locally-set-webServiceUrl</prop>
  </util:properties>
</beans>
```

W powyższym listingu dodane zostało odwołanie do oferowanego przez Spring schematu `context`, aby można było uzyskać dostęp do elementów. W listingu widać, że użycie elementów `<property-placeholder>` oznacza bardziej związłą konfigurację komponentu `PropertySourcesPlaceholderConfigurer`. Atrybuty `ignore-unresolvable`, `location` i `local-override` odpowiadają właściwościom `ignoreUnresolvablePlaceholders`, `locations` i `localOverride` komponentu `PropertySourcesPlaceholderConfigurer`. Ponieważ klasa `PropertySourcesPlaceholderConfigurer` implementuje oferowany przez Spring interfejs `Ordered`, wartość atrybutu `order` jest używana w celu ustawienia właściwości `order` egzemplarza `PropertySourcesPlaceholderConfigurer`. Atrybut `properties-ref` odwołuje się do obiektu `java.util.Properties` przedstawiającego *właściwości lokalne*. W powyższym listingu element `<properties>` schematu `util` oferowanego przez Spring (patrz: podrozdział 3.8) tworzy egzemplarz obiektu `java.util.Properties`, do którego odwołujemy się w atrybucie `properties-ref` elementu `<property-placeholder>`.

Zobaczmy teraz, jak komponent Springa `PropertyOverrideConfigurer` (implementacja `BeanFactoryPostProcessor`) pozwala na zdefiniowanie w zewnętrznych plikach właściwości wartości dla właściwości komponentu.

Komponent `PropertyOverrideConfigurer`

Komponent `PropertyOverrideConfigurer` jest podobny do `PropertySourcesPlaceholderConfigurer` pod tym względem, że pozwala na dostarczenie wartości właściwości w zewnętrznym pliku właściwości. Podczas użycia `PropertyOverrideConfigurer` wartość właściwości w zewnętrznym pliku jest podawana w następującym formacie:

```
<nazwa-komponentu>.<nazwa-właściwości-komponentu>=<wartość>
```

W powyższej składni `<nazwa-komponentu>` oznacza nazwę komponentu, `<nazwa-właściwości-komponentu>` oznacza nazwę właściwości w komponencie, natomiast `<wartość>` to po prostu wartość, która ma zostać przypisana właściwości komponentu.

Oto różnice występujące między klasami `PropertyOverrideConfigurer` i `PropertySourcesPlaceholderConfigurer`:

- Klasy `PropertyOverrideConfigurer` można używać jedynie dla pochodzących z zewnątrz wartości właściwości komponentu. Oznacza to brak możliwości jej użycia dla zewnętrznych wartości argumentów konstruktora.
- Klasa `PropertySourcesPlaceholderConfigurer` *nie* pozwala na podanie wartości domyślnych dla właściwości. Natomiast klasa `PropertyOverrideConfigurer` umożliwia określenie wartości domyślnych dla właściwości komponentu.

Spójrzmy na przykład użycia klasy `PropertyOverrideConfigurer`.

IMPORT **Rozdział 105/ch05-propertyOverrideConfigurer-example.** (Ten projekt pokazuje aplikację Springa, w której użyto klasy `PropertyOverrideConfigurer`. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `SampleApp` projektu).

Przykład użycia klasy `PropertyOverrideConfigurer`

W listingu 5.32 przedstawiamy definicje komponentów dla klas `DataSource` i `WebServiceConfiguration`, których właściwości zostały ustawione za pomocą `PropertyOverrideConfigurer`.

Listing 5.32. Plik `applicationContext.xml` przedstawiający definicje komponentów dla klas `DataSource` i `WebServiceConfiguration`

Projekt: `ch05-propertyOverrideConfigurer-example`
 Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="datasource" class="sample.spring.chapter05.domain.DataSource">
  <property name="url" value="wartość testowa dla url" />
  <property name="username" value="wartość testowa dla username" />
  <property name="password" value="wartość testowa dla password" />
  <property name="driverClass" value="wartość testowa dla driverClass" />
</bean>

<bean id="webServiceConfiguration"
  class="sample.spring.chapter05.domain.WebServiceConfiguration">
  <property name="webServiceUrl" value="tę wartość trzeba zastąpić inną" />
</bean>
```

W powyższym listingu atrybut `value` elementu `<bean>` określa wartość domyślną dla właściwości komponentu.

W listingu 5.33 została przedstawiona definicja komponentu dla klasy `PropertyOverrideConfigurer`, która wartości domyślne właściwości komponentu (określone w listingu 5.32) zastępuje wartościami odczytanymi z plików `database.properties` i `webservice.properties`.

Listing 5.33. Plik `applicationContext.xml` przedstawiający konfigurację komponentu dla klasy `PropertyOverrideConfigurer`

Projekt: `ch05-propertyOverrideConfigurer-example`
 Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">
  <property name="locations">
    <list>
      <value>classpath:database.properties</value>
```

```

        <value>classpath:webservice.properties</value>
    </list>
</property>
</bean>

```

W powyższym listingu właściwość `locations` klasy `PropertyOverrideConfigurer` wskazuje pliki właściwości zawierające wartości dla właściwości.



Zamiast bezpośrednio konfigurować `PropertyOverrideConfigurer` możesz użyć elementu `<property-override>` schematu context oferowanego przez Spring i za jego pomocą przeprowadzić konfigurację `PropertyOverrideConfigurer`.

W listingu 5.34 zostały przedstawione pliki `database.properties` i `webservice.properties` zawierające wartości dla właściwości komponentu.

Listing 5.34. Właściwości zdefiniowane w plikach `database.properties` i `webservice.properties`

Projekt: `ch05-propertyOverrideConfigurer-example`

Położenie pliku: `src/main/resources/META-INF`

```

----- Plik database.properties -----
datasource.url=some_url
datasource.username=some_username
datasource.password=some_password

----- Plik webservice.properties -----
webServiceConfiguration.webServiceUrl=some_url

```

Wpisy w plikach `database.properties` i `webservice.properties` pokazują, że nazwy właściwości stosują format `<nazwa-komponentu>.<nazwa-właściwości>`. Kiedy właściwości są wczytywane przez kontener Springa, komponent `PropertyOverrideConfigurer` zastępuje wartość domyślną właściwości komponentu wartością odczytaną z pliku `database.properties` lub `webservice.properties`. Na przykład właściwość `url` komponentu `datasource` ma zdefiniowaną w pliku `database.properties` wartość `datasource.url`. Podobnie właściwość `webServiceUrl` komponentu `webServiceConfiguration` ma zdefiniowaną w pliku `webservice.properties` wartość `webServiceConfiguration.webServiceUrl`.

Jeżeli w zewnętrznym pliku właściwości nie zostanie znaleziona wartość dla właściwości, wówczas właściwości komponentu zachowują swoją wartość domyślną. W listingu 5.32 widać, że właściwość `driverClass` komponentu `datasource` ma określoną wartość domyślną wartość testowa dla `driverClass`. W listingu 5.34 widać, że w plikach `database.properties` i `webservice.properties` nie ma zdefiniowanej właściwości o nazwie `datasource.driverClass`, a tym samym właściwość `driverClass` zachowuje swoją wartość domyślną.

Metoda `main()` klasy `SampleApp` projektu `ch05-propertyOverrideConfigurer-example` pobiera komponenty `WebServiceConfiguration` i `DataSource` z egzemplarza `ApplicationContext`, a następnie wyświetla w konsoli ich właściwości. Jeżeli wykonasz metodę `main()` klasy `SampleApp`, w konsoli otrzymasz następujące dane wyjściowe:

```
DataSource [url=some_url, username=some_username, password=some_password, driverClass=wartość  
↳testowa dla driverClass]  
WebServiceConfiguration [webServiceUrl=some_url]
```

W powyższych danych wyjściowych widać, że wartości domyślne wszystkich komponentów poza `driverClass` zostały zastąpione wartościami, które zdefiniowano w zewnętrznych plikach właściwości.

Ponieważ komponenty `PropertyOverrideConfigurer` i `PropertySourcesPlaceholderConfigurer` dziedziczą po klasie Springa o nazwie `PropertyResourceConfigurer`, można dostrzec, że obie klasy dziedziczą wiele wspólnych opcji konfiguracyjnych. Na przykład można określić wartość właściwości `localOverride` klasy `PropertyOverrideConfigurer` i tym samym wskazać, czy właściwości lokalne mają pierwszeństwo przed wczytanymi z zewnętrznych plików właściwości. Właściwości `properties` klasy `PropertyOverrideConfigurer` można użyć do zdefiniowania właściwości lokalnych itd.

5.5. Podsumowanie

W tym rozdziale zobaczyłeś, jak do egzemplarza komponentu dodać własną logikę odpowiedzialną za jego inicjalizację i usunięcie. Dowiedziałeś się, jak można modyfikować nowo utworzone egzemplarze komponentów, używając do tego implementacji `BeanPostProcessor`, a także jak modyfikować definicje komponentów za pomocą implementacji `BeanFactoryPostProcessor`. Wewnętrznie Spring wykorzystuje implementacje `BeanPostProcessors` i `BeanFactoryPostProcessor` w celu dostarczenia wielu funkcji frameworku. W kolejnym rozdziale poznasz oferowaną przez Spring obsługę programowania opartego na adnotacjach.

Rozdział 6. Programowanie oparte na adnotacjach

6.1. Wprowadzenie

W poprzednich rozdziałach widziałeś, że znajdujące się w pliku XML kontekstu aplikacji definicje komponentów są używane przez kontener Springa w charakterze matryc do tworzenia egzemplarzy komponentów. Definicja komponentu zawiera informacje o jego zależnościach, metodach inicjalizacyjnej i usuwającej egzemplarz, strategii tworzenia komponentu, zasięgu komponentu itd. W tym rozdziale przejdziemy do omawiania adnotacji, za pomocą których informacje te można podać w samej klasie, a tym samym uniknąć konieczności wyraźnej konfiguracji komponentu w pliku XML kontekstu aplikacji. Poruszymy także temat SpEL (ang. *Spring Expression Language*) oraz weryfikacji obiektów za pomocą oferowanego przez Spring interfejsu `Validator` lub za pomocą adnotacji zdefiniowanych w specyfikacji JSR 303. Na końcu rozdziału pokrótce zajmiemy się programowym definiowaniem komponentów Springa przy użyciu adnotacji `@Configuration` i `@Bean`.

Najpierw przeanalizujemy oferowaną przez Spring adnotację `@Component`, która wskazuje, że dana klasa przedstawia komponent Springa.

6.2. Identyfikacja komponentów Springa za pomocą adnotacji `@Component`

Oferowana przez Spring adnotacja `@Component` jest adnotacją na poziomie typu i wskazuje, że dana klasa przedstawia komponent Springa. Zaleca się użycie bardziej specjalizowanych form adnotacji `@Component` w przypadku adnotacji kontrolerów, usług i obiektów dostępu do danych (tzw. DAO) w aplikacji. Na przykład dla kontrolerów należy stosować adnotację `@Controller`, dla usług — `@Service`, natomiast dla obiektów DAO — `@Repository`.

IMPORT **Rozdział06/ch06-bankapp-annotations.** (Ten projekt pokazuje aplikację MyBank, w której adnotacji użyto do rejestracji komponentów w kontenerze Springa oraz do automatycznego wiązania zależności. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

W listingu 6.1 została przedstawiona klasa `FixedDepositServiceImpl` aplikacji MyBank używająca adnotacji `@Service`.

Listing 6.1. Klasa `FixedDepositServiceImpl` wykorzystująca adnotację `@Service`

Projekt: `ch06-bankapp-annotations`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp/service`

```
package sample.spring.chapter06.bankapp.service;

import org.springframework.stereotype.Service;

@Service(value="FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService { ..... }
```

Ponieważ klasa `FixedDepositServiceImpl` została opatrzona adnotacją `@Service`, przedstawia komponent Springa. Adnotacja `@Service` akceptuje atrybut `value` określający nazwę, pod jaką komponent jest zarejestrowany w kontenerze Springa. Na przykład klasa `FixedDepositServiceImpl` została zarejestrowana w kontenerze Springa jako komponent o nazwie `FixedDepositService`. Atrybut `value` pełni taką samą rolę jak atrybut `id` w elemencie `<bean>`.

Podobnie jak adnotacja `@Service`, także `@Component`, `@Repository` i `@Controller` określają nazwy komponentu za pomocą atrybutu `value`. Istnieje możliwość podania nazwy komponentu Springa bez konieczności wyraźnego definiowania atrybutu `value`. Oznacza to, że `@Service(value="FixedDepositService")` ma taki sam efekt jak `@Service("FixedDepositService")`. Jeżeli nie zostanie podana nazwa dla komponentu, wówczas Spring przyjmie założenie, że jest ona taka sama jak nazwa klasy komponentu. Jedyna różnica polega na tym, że nazwa komponentu rozpoczyna się *małą* literą. Powinieneś określać własną nazwę dla komponentu, ponieważ okazuje się to szczególnie użyteczne podczas automatycznego wiązania zależności „według nazw”.

Po włączeniu oferowanej przez Spring funkcji skanowania ścieżki klas klasy oznaczone adnotacjami `@Component`, `@Controller`, `@Service` i `@Repository` będą automatycznie rejestrowane w kontenerze Springa. Włączenie funkcji skanowania ścieżki klas odbywa się za pomocą elementu `<component-scan>` schematu `context`.

W listingu 6.2 przedstawiamy przykład użycia elementu `<component-scan>`.

Listing 6.2. Plik `applicationContext.xml` pokazujący przykład użycia elementu `<component-scan>`

Projekt: `ch06-bankapp-annotations`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="....http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.0.xsd">

  <context:component-scan base-package="sample.spring"/>
</beans>
```

W powyższym listingu znajduje się odwołanie do schematu `context`, aby można było uzyskać dostęp do elementów. Atrybut `base-package` elementu `<component-scan>` zawiera rozdzieloną przecinkami listę pakietów, które powinny być przeszukiwane pod kątem komponentów Springa. Ponieważ wartością atrybutu `base-package` jest `sample.spring`, komponenty Springa będą szukane w pakiecie `sample.spring` oraz wszystkich znajdujących się w nim pakietach. Jak pokazaliśmy w listingu 6.1, klasa `FixedDepositServiceImpl` jest oznaczona adnotacją `@Service` i znajduje się w pakiecie `sample.spring.chapter06.bankapp.service`. Dlatego element `<component-scan>` w omawianym listingu automatycznie rejestruje w kontenerze Springa klasę `FixedDepositServiceImpl` jako komponent. To będzie odpowiednikiem umieszczenia w pliku XML kontekstu aplikacji przedstawionej w listingu 6.3 definicji komponentu dla klasy `FixedDepositServiceImpl`.

Jeżeli chcesz filtrować klasy komponentów, które powinny być uwzględnione podczas automatycznej rejestracji w kontenerze Springa, wówczas użyj atrybutu `resource-pattern` elementu `<component-scan>`. Wartość domyślna atrybutu `resource-pattern` wynosi `**/*.class`, co oznacza,

Listing 6.3. Definicja komponentu dla klasy *FixedDepositServiceImpl*

```
<bean id="FixedDepositService"
      class="sample.spring.chapter06.bankapp.service.FixedDepositServiceImpl" />
```

że wszystkie klasy komponentów w pakiecie wskazanym przez atrybut `base-package` będą uwzględniane podczas automatycznej rejestracji. Elementy `<include-filter>` i `<exclude-filter>` znajdujące się w elemencie `<component-scan>` zapewniają znacznie zwęższy sposób wskazania klas komponentów uwzględnianych podczas automatycznej rejestracji oraz klas, które powinny zostać zignorowane. W listingu 6.4 mamy przykład użycia elementów `<include-filter>` i `<exclude-filter>`.

Listing 6.4. Przykład użycia elementów `<include-filter>` i `<exclude-filter>`

```
<beans ....>
  <context:component-scan base-package="sample.example">
    <context:include-filter type="annotation"
                          expression="example.annotation.MyAnnotation"/>
    <context:exclude-filter type="regex" expression=".*Details"/>
  </context:component-scan>
</beans>
```

Elementy `<exclude-filter>` i `<include-filter>` definiują atrybut `type` określający strategię stosowaną podczas filtrowania klas komponentów oraz atrybut `expression` określający wyrażenie filtru. W przedstawionym powyżej listingu element `<include-filter>` wskazuje, że klasy komponentów oznaczone adnotacją `MyAnnotation` będą automatycznie rejestrowane w kontenerze Springa. Natomiast element `<exclude-filter>` określa, że klasy komponentów o nazwach kończących się na `Details` będą ignorowane przez element `<component-scan>`.

W tabeli 6.1 zostały wymienione wartości akceptowane przez atrybut `type` elementów `<include-filter>` i `<exclude-filter>`.



W tym podrozdziale skoncentrujemy się na przykładzie użycia adnotacji `@Service`. Adnotacje `@Component`, `@Controller` i `@Repository` są określane w taki sam sposób jak `@Service`. W klasach `CustomerRegistrationDetails` i `CustomerRequestDetails` projektu *ch06-bankapp-annotations* znajdziesz przykład użycia adnotacji `@Component`. Natomiast w klasach DAO znajdujących się w projekcie *ch06-bankapp-annotations* znajdziesz przykład użycia adnotacji `@Repository`.

Ponieważ komponenty Springa *nie* są definiowane w pliku XML kontekstu aplikacji, brakuje możliwości użycia elementu `<property>` lub `<constructor-arg>` w celu wskazania ich zależności. Dlatego aby wskazać zależności, komponenty Springa wykorzystują takie adnotacje jak `@Autowired`, `@Inject` itd.

Przechodzimy teraz do omówienia oferowanej przez Spring adnotacji `@Autowired`.

Tabela 6.1. Wartości akceptowane przez atrybut `type` elementów `<include-filter>` i `<exclude-filter>`

Wartość atrybutu <code>type</code>	Opis
<code>annotation</code>	Jeżeli wartością atrybutu <code>type</code> jest <code>annotation</code> , wówczas atrybut <code>expression</code> określa w pełni kwalifikowaną nazwę klasy, jaką <i>musi</i> mieć klasa oznaczona tą adnotacją. Na przykład jeśli wartością atrybutu <code>expression</code> jest <code>example.↳annotation.MyAnnotation</code> , to klasa komponentu opatrzona adnotacją <code>MyAnnotation</code> zostanie uwzględniona (w przypadku elementu <code><include-filter></code>) lub nie zostanie uwzględniona (w przypadku elementu <code><exclude-filter></code>).
<code>assignable</code>	Jeżeli wartością atrybutu <code>type</code> jest <code>assignable</code> , wówczas atrybut <code>expression</code> określa w pełni kwalifikowaną nazwę klasy lub interfejsu, do którego musi być przypisana klasa komponentu.
<code>aspectj</code>	Jeżeli wartością atrybutu <code>type</code> jest <code>aspectj</code> , wówczas atrybut <code>expression</code> określa wyrażenie AspectJ, które jest używane do filtrowania klas komponentów.
<code>regex</code>	Jeżeli wartością atrybutu <code>type</code> jest <code>regex</code> , wówczas atrybut <code>expression</code> określa wyrażenie regularne, które jest używane do filtrowania klas komponentów według ich nazw.
<code>custom</code>	Jeżeli wartością atrybutu <code>type</code> jest <code>custom</code> , wówczas atrybut <code>expression</code> określa implementację <code>org.springframework.core.type.TypeFilter</code> , która jest używana do filtrowania klas komponentów.

6.3. Adnotacja `@Autowired`, czyli automatyczne wiązanie zależności według ich typu

Adnotacja `@Autowired` jest używana do automatycznego wiązania zależności według ich typu. Oferuje ona dokładnie taką samą funkcjonalność jak omówiona w rozdziale 4. funkcja automatycznego wiązania. Jednak użycie adnotacji `@Autowired` to znacznie zwęższe i elastyczniejsze podejście podczas automatycznego wiązania zależności. Ponadto adnotacja `@Autowired` może być używana na poziomie *konstruktora*, *metody* lub *właściwości*.

W listingu 6.5 została przedstawiona klasa `AccountStatementServiceImpl` używająca adnotacji `@Autowired` na poziomie właściwości.

Listing 6.5. Klasa `AccountStatementServiceImpl` przedstawiająca użycie adnotacji `@Autowired` na poziomie właściwości

Projekt: `ch06-bankapp-annotations`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp/service`

```
package sample.spring.chapter06.bankapp.service;

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

@Service(value="accountStatementService")
public class AccountStatementServiceImpl implements AccountStatementService {

    @Autowired
    private AccountStatementDao accountStatementDao;
```

```
@Override
public AccountStatement getAccountStatement(Date from, Date to) {
    return accountStatementDao.getAccountStatement(from, to);
}
}
```

W powyższym listingu właściwość `accountStatementDao` (typu `AccountStatementDao`) została oznaczona adnotacją `@Autowired`. Podczas tworzenia egzemplarza `AccountStatementServiceImpl` implementacja `AutowiredAnnotationBeanPostProcessor` (czyli implementacja `BeanPostProcessor`) będzie odpowiedzialna za automatyczne wiązanie właściwości `accountStatementDao`. Implementacja `AutowiredAnnotationBeanPostProcessor` pobiera z kontenera Springa odwołanie do komponentu typu `AccountStatementDao` i przypisuje go właściwości `accountStatementDao`. Trzeba koniecznie zwrócić uwagę na fakt, że właściwość oznaczona adnotacją `@Autowired` *nie* musi być publiczna ani nie musi mieć odpowiadającej jej publicznej metody typu setter.



Oferowana przez Spring implementacja `AutowiredAnnotationBeanPostProcessor` przeprowadza automatyczne wiązanie właściwości, metod i konstruktorów oznaczonych przez adnotację `@Autowired` dostarczaną przez Spring lub adnotację `@Inject` dostarczaną przez specyfikację JSR 330 (jej dokładne omówienie znajdziesz w podrozdziale 6.5).

W listingu 6.6 została przedstawiona klasa `CustomerRegistrationServiceImpl` używająca adnotacji `@Autowired` na poziomie metody.

Listing 6.6. Klasa `CustomerRegistrationServiceImpl` przedstawiająca użycie adnotacji `@Autowired` na poziomie metody

Projekt: `ch06-bankapp-annotations`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp/service`

```
package sample.spring.chapter06.bankapp.service;

@Service("customerRegistrationService")
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class CustomerRegistrationServiceImpl implements CustomerRegistrationService {

    private CustomerRegistrationDetails customerRegistrationDetails;
    ....
    @Autowired
    public void obtainCustomerRegistrationDetails(
        CustomerRegistrationDetails customerRegistrationDetails) {
        this.customerRegistrationDetails = customerRegistrationDetails;
    }
    ....
    @Override
    public void setAccountNumber(String accountNumber) {
        customerRegistrationDetails.setAccountNumber(accountNumber);
    }
    ....
}
```

W powyższym listingu metoda `obtainCustomerRegistrationDetails()` została oznaczona adnotacją `@Autowired`. Po oznaczeniu metody adnotacją `@Autowired` argumenty danej metody będą automatycznie wiązane. Ponieważ metoda `obtainCustomerRegistrationDetails()` jest opatrzona adnotacją `@Autowired`, jej argument `CustomerRegistrationDetails` będzie automatycznie wiązany według *typu*. Trzeba w tym miejscu koniecznie wspomnieć, że metoda oznaczona adnotacją `@Autowired` *nie* musi być publiczna.



Metoda oznaczona adnotacją `@Autowired` jest wywoływana po utworzeniu egzemplarza komponentu, natomiast właściwości oznaczone adnotacją `@Autowired` są wstrzykiwane wraz z dopasowanymi egzemplarzami komponentu.

W listingu 6.7 została przedstawiona klasa `CustomerRequestServiceImpl` definiująca konstruktor oznaczony adnotacją `@Autowired`.

Listing 6.7. Klasa `CustomerRequestServiceImpl` przedstawiająca przykład użycia adnotacji `@Autowired` na poziomie konstruktora

Projekt: `ch06-bankapp-annotations`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp/service`

```
package sample.spring.chapter06.bankapp.service;

@Service(value="customerRequestService")
public class CustomerRequestServiceImpl implements CustomerRequestService {
    private CustomerRequestDetails customerRequestDetails;
    private CustomerRequestDao customerRequestDao;

    @Autowired
    public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails,
                                     CustomerRequestDao customerRequestDao) {
        this.customerRequestDetails = customerRequestDetails;
        this.customerRequestDao = customerRequestDao;
    }
    .....
}
```

W powyższym listingu konstruktor klasy `CustomerRequestServiceImpl` jest oznaczony adnotacją `@Autowired`. Jeżeli konstruktor jest oznaczony adnotacją `@Autowired`, argumenty konstruktora będą automatycznie wiązane. Ponieważ konstruktor klasy `CustomerRequestServiceImpl` jest oznaczony adnotacją `@Autowired`, jego argumenty `CustomerRequestDetails` i `CustomerRequestDao` będą automatycznie wiązane według *typu*. Trzeba w tym miejscu wspomnieć, że konstruktor oznaczony adnotacją `@Autowired` *nie* musi być publiczny.

Podczas użycia adnotacji `@Autowired`, jeżeli *nie* zostanie znaleziony komponent dopasowany do wymaganego typu, nastąpi zgłoszenie wyjątku. Rozważmy na przykład listing 6.7. Jeżeli w kontenerze Springa nie znajdzie się zarejestrowany komponent typu `CustomerRequestDetails` lub `CustomerRequestDao`, to podczas tworzenia egzemplarza `CustomerRequestServiceImpl` nastąpi zgłoszenie wyjątku. Atrybut `required` adnotacji `@Autowired` określa, czy automatyczne wiązanie zależności jest obowiązkowe, czy jednak opcjonalne. Jeżeli wartością atrybutu `required` adnotacji

@Autowired będzie false, to automatyczne wiązanie zależności jest uznawane za *opcjonalne*. Oznacza to, że po przypisaniu wartości false atrybutowi required nie nastąpi zgłoszenie wyjątku, jeśli w kontenerze Springa nie będzie dopasowany komponent wymaganego typu. Wartością domyślną atrybutu required jest true, więc zależności muszą być spełnione przez kontener Springa.

Jeżeli klasa komponentu definiuje konstruktor oznaczony adnotacją @Autowired, której atrybut required ma wartość true, wówczas *nie* może mieć innego konstruktora z adnotacją @Autowired. Na przykład przeanalizuj kod w listingu 6.8 przedstawiający klasę zawierającą dwa konstruktory wraz z adnotacjami @Autowired.

Listing 6.8. Klasa komponentu zawierająca dwa konstruktory wraz z adnotacjami @Autowired

```
@Service(value="customerRequestService")
public class CustomerRequestServiceImpl implements CustomerRequestService {
    ....
    @Autowired(required=false)
    public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails)
    { .... }

    @Autowired
    public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails,
        CustomerRequestDao customerRequestDao) { .... }
}
```

Ponieważ w przypadku jednego konstruktora wymagane jest automatyczne wiązanie zależności (atrybut required adnotacji @Autowired ma wartość true), natomiast opcjonalne dla drugiego konstruktora w klasie (atrybut required adnotacji @Autowired ma wartość false), wynikiem jest zgłoszenie wyjątku przez Spring.

Klasa komponentu może definiować wiele konstruktorów oznaczonych adnotacją @Autowired, której atrybut required ma wartość false. W takim przypadku jeden z konstruktorów będzie wywołany przez Spring w celu utworzenia egzemplarza klasy komponentu. Listing 6.9 przedstawia klasę komponentu definiującą dwa konstruktory wraz z adnotacjami @Autowired(required=false) oraz konstruktor domyślny.

Listing 6.9. Klasa komponentu definiująca wiele konstruktorów oznaczonych adnotacjami @Autowired, których atrybut required ma przypisaną wartość false

```
@Service(value="customerRequestService")
public class CustomerRequestServiceImpl implements CustomerRequestService {
    public CustomerRequestServiceImpl() {
        ....
    }
    @Autowired(required=false)
    public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails) {
        ....
    }
    @Autowired(required=false)
    public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails,
        CustomerRequestDao customerRequestDao) {
        ....
    }
}
```

W powyższym listingu oba konstruktory oznaczone adnotacjami `@Autowired` są kandydatami do automatycznego wiązania przez Spring w celu utworzenia egzemplarza klasy `CustomerRequestServiceImpl`. Wybrany zostanie konstruktor z największą liczbą spełnionych zależności. W przypadku klasy `CustomerRequestServiceImpl`, gdy kontener Springa zawiera zarejestrowane komponenty typu `CustomerRequestDetails` i `CustomerRequestDao`, wówczas Spring wywoła konstruktor `CustomerRequestServiceImpl(CustomerRequestDetails, CustomerRequestDao)`. W przypadku zarejestrowanego w kontenerze komponentu typu `CustomerRequestDetails` i jednocześnie braku zarejestrowanego komponentu typu `CustomerRequestDao` wywołany zostanie konstruktor `CustomerRequestServiceImpl(CustomerRequestDetails)`. Natomiast jeśli żadne zależności nie będą spełnione, nastąpi wywołanie domyślnego konstruktora klasy `CustomerRequestServiceImpl`.

Przekonajmy się teraz, jak użyć oferowanej przez Spring adnotacji `@Qualifier` wraz z adnotacją `@Autowired` w celu automatycznego wiązania zależności *według nazwy*.

6.4. Adnotacja `@Qualifier` pozwalająca na automatyczne wiązanie zależności według ich nazwy

Adnotacji `@Autowired` można użyć wraz z oferowaną przez Spring adnotacją `@Qualifier` i tym samym zapewnić możliwość automatycznego wiązania zależności *według nazwy*. Adnotację `@Qualifier` można stosować na poziomie właściwości, parametru metody i konstruktora.

W listingu 6.10 została przedstawiona klasa `FixedDepositServiceImpl`, która używa adnotacji `@Qualifier`.

Listing 6.10. Klasa `FixedDepositServiceImpl` przedstawiająca użycie adnotacji `@Qualifier`

Projekt: `ch06-bankapp-annotations`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp/service`

```
package sample.spring.chapter06.bankapp.service;

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

@Service(value="FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {

    @Autowired
    @Qualifier(value="myFixedDepositDao")
    private FixedDepositDao myFixedDepositDao;
    ....
}
```

W powyższym listingu właściwość `myFixedDepositDao` została oznaczona adnotacjami `@Autowired` i `@Qualifier`. Atrybut `value` adnotacji `@Qualifier` określa nazwę komponentu przypisanego właściwości `myFixedDepositDao`.

Spring najpierw wyszukuje kandydatów do automatycznego wiązania według typu właściwości, konstruktory i metody oznaczone adnotacją `@Autowired`. Następnie nazwa komponentu oznaczona adnotacją `@Qualifier` jest używana do znalezienia *unikalnego* komponentu na liście kandydatów

do automatycznego wiązania. Na przykład w listingu 6.10 Spring w pierwszej kolejności wyszukuje komponenty typu `FixedDepositDao` dla właściwości `myFixedDepositDao`, a następnie szuka komponentu o nazwie `myFixedDepositDao` na liście kandydatów przeznaczonych do automatycznego wiązania. Po znalezieniu komponentu o nazwie `myFixedDepositDao` Spring przypisuje go właściwości `myFixedDepositDao`.



Obie formy zapisu — `@Qualifier(value="myFixedDepositDao")` i `@Qualifier("myFixedDepositDao")` — mają dokładnie ten sam skutek. Nie trzeba używać atrybutu `value` w celu podania nazwy komponentu przeznaczonego do automatycznego wiązania.

W listingu 6.11 przedstawiamy przykład użycia adnotacji `@Qualifier` na poziomach parametru metody i argumentu konstruktora.

Listing 6.11. Przykład użycia adnotacji `@Qualifier` na poziomach parametru metody i argumentu konstruktora

```
public class Sample {  
  
    @Autowired  
    public Sample(@Qualifier("aBean") ABean bean) { .... }  
  
    @Autowired  
    public void doSomething(@Qualifier("bBean") BBean bean, CBean cBean) { ..... }  
}
```

W powyższym listingu adnotacja `@Qualifier` została użyta dla argumentu konstruktora i argumentu metody. Podczas tworzenia egzemplarza klasy `Sample` Spring wyszukuje komponent typu `ABean` o nazwie `aBean`, a następnie przekazuje go jako argument konstruktora klasy `Sample`. Podczas wywoływania metody `doSomething()` klasy `Sample` Spring wyszukuje komponent typu `BBean` (o nazwie `bBean`) oraz inny komponent typu `CBean`, a następnie przekazuje oba jako argumenty metody `doSomething()`. Trzeba w tym miejscu wspomnieć, że zależność dla `BBean` jest automatycznie wiązana *według nazwy*, natomiast zależność `CBean` jest automatycznie wiązana *według typu*.

Przechodzimy teraz do adnotacji `@Inject` i `@Named` specyfikacji JSR 330. Adnotacji tych można używać zamiast `@Autowired` i `@Qualifier`.

6.5. Adnotacje `@Inject` i `@Named` zdefiniowane przez specyfikację JSR 330

Specyfikacja JSR 330 (wstrzykiwanie zależności w Javie) standaryzuje adnotacje dotyczące wstrzykiwania zależności na platformie Javy. Definiuje ona adnotacje `@Inject` i `@Named` podobne do oferowanych przez Spring adnotacji odpowiednio `@Autowired` i `@Qualifier`. Spring zapewnia obsługę adnotacji `@Inject` i `@Named`.

IMPORT **Rozdział 106/ch06-bankapp-jsr330.** (Ten projekt pokazuje aplikację `MyBank`, w której użyto zdefiniowanych przez specyfikację JSR 330 adnotacji `@Inject` i `@Named` w celu przeprowadzania automatycznego wiązania zależności. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

W listingu 6.12 została przedstawiona klasa `FixedDepositServiceImpl` używająca adnotacji `@Inject` i `@Named` specyfikacji JSR 330.

Listing 6.12. Klasa `FixedDepositServiceImpl`

Projekt: `ch06-bankapp-jsr330`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp/service`

```
package sample.spring.chapter06.bankapp.service;

import javax.inject.Inject;
import javax.inject.Named;

@Named(value="FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {

    @Inject
    @Named(value="myFixedDepositDao")
    private FixedDepositDao myFixedDepositDao;
    ....
}
```

Jeżeli klasę `FixedDepositServiceImpl` przedstawioną w powyższym listingu porównasz z klasą `FixedDepositServiceImpl` w listingu 6.10, zauważysz, że definiowana przez specyfikację JSR 330 adnotacja `@Named` została użyta zamiast `@Service` i `@Qualifier`, natomiast definiowana przez specyfikację JSR 330 adnotacja `@Inject` została użyta w miejsce `@Autowired`.

Adnotacje `@Autowired` i `@Inject` charakteryzują się taką samą semantyką, są używane do automatycznego wiązania zależności *według typu*. Podobnie jak w przypadku adnotacji `@Autowired`, także `@Inject` może być używana na poziomie metody, konstruktora lub właściwości. W pierwszej kolejności preferowane jest wstrzykiwanie zależności konstruktorów, następnie właściwości i na koniec metod. Wcześniej widziałeś, że atrybut `required` adnotacji `@Autowired` określa, czy operacja automatycznego wiązania właściwości jest obowiązkowa czy opcjonalna. W adnotacji `@Inject` nie znajdziemy żadnego odpowiednika atrybutu `required` adnotacji `@Autowired`.

Jeżeli adnotacja `@Named` zostanie użyta na poziomie typu, działa podobnie jak oferowana przez Spring adnotacja `@Component`. Natomiast jeśli zostanie użyta na poziomie parametru metody lub argumentu konstruktora, działa podobnie jak oferowana przez Spring adnotacja `@Qualifier`. Jeżeli klasa będzie oznaczona adnotacją `@Named`, element `<component-scan>` schematu context traktuje ją jak klasę komponentu oznaczoną adnotacją `@Component`.

Aby użyć adnotacji `@Named` i `@Inject`, do projektu trzeba dodać plik JAR wraz ze specyfikacją JSR 330. W projekcie `ch06-bankapp-jsr330` plik ten został wskazany za pomocą elementu `<dependency>` w pliku `pom.xml`:

```
<dependency>
  <groupId>javax.inject</groupId>
  <artifactId>javax.inject</artifactId>
  <version>1</version>
</dependency>
```


W rozdziale 5. miałeś okazję zobaczyć zdefiniowane przez specyfikację JSR 250 adnotacje `@PostConstruct` i `@PreDestroy`, które są używane w celu identyfikacji metod inicjalizacji i usuwania egzemplarza komponentu. Przekonajmy się teraz, jak zdefiniowana przez specyfikację JSR 250 adnotacja `@Resource` może być użyta do automatycznego wiązania zależności *według nazwy*.

6.6. Zdefiniowana przez JSR 250 adnotacja `@Resource`

W przypadku właściwości i metod Spring obsługuje automatyczne wiązanie zależności *według nazwy* za pomocą zdefiniowanej przez specyfikację JSR 250 adnotacji `@Resource`. Atrybut `name` tej adnotacji określa nazwę komponentu, który ma być automatycznie wiązany. Trzeba koniecznie pamiętać, że adnotacji `@Resource` nie można użyć w celu automatycznego wiązania argumentów konstruktora.

W listingu 6.13 pokazujemy, jak klasę `FixedDepositServiceImpl` z listingu 6.12 można przystosować do użycia adnotacji `@Resource`.

Listing 6.13. Przykład użycia adnotacji `@Resource` na poziomie właściwości

```
import javax.annotation.Resource;

@Named(value="FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {

    @Resource(name="myFixedDepositDao")
    private FixedDepositDao myFixedDepositDao;
    ....
}
```

W powyższym listingu adnotacja `@Resource` została użyta w celu automatycznego wiązania właściwości `myFixedDepositDao`. Ponieważ wartością atrybutu `name` jest `myFixedDepositDao`, Spring szuka w kontenerze komponentu o nazwie `myFixedDepositDao`, a następnie przypisuje go właściwości `myFixedDepositDao`.

Do automatycznego wiązania zależności *według nazwy* należy zamiast adnotacji `@Autowired` i `@Qualifier` używać adnotacji `@Resource`. Jak wcześniej wspomnieliśmy, w przypadku użycia adnotacji `@Autowired` i `@Qualifier` do przeprowadzania automatycznego wiązania *według nazwy* Spring w pierwszej kolejności wyszukuje na podstawie typu właściwości (lub typu argumentu metody bądź konstruktora) komponenty przeznaczone do automatycznego wiązania, a następnie zawęża wybór do unikalnego komponentu, opierając się na jego nazwie określonej przez adnotację `@Qualifier`. Jednak w przypadku użycia adnotacji `@Resource` Spring wykorzystuje podaną przez adnotację `@Resource` nazwę do wyszukania unikalnego komponentu. Oznacza to, że podczas użycia adnotacji `@Resource` typ właściwości (lub argument metody typu setter) przeznaczonej do automatycznego wiązania *nie* będzie uwzględniany przez Spring.



Ponieważ adnotacje `@Autowired`, `@Inject` i `@Resource` są przetwarzane przez `BeanPostProcessors`, nie powinieneś ich używać w klasach komponentów implementujących interfejs `BeanFactoryPostProcessor` lub `BeanPostProcessor`.

Przechodzimy teraz do analizy adnotacji @Scope, @Lazy, @DependsOn i @Primary.

6.7. Adnotacje @Scope, @Lazy, @DependsOn i @Primary

Zasięg (singleton lub prototypu) dla komponentu Springa można określić za pomocą adnotacji @Scope. Domyślnie komponenty Springa mają zasięg singleton. Jeżeli chcesz, aby komponent miał zasięg prototypu, powinieneś wyrazić to wskazać za pomocą adnotacji @Scope. Adnotacja @Scope odgrywa taką samą rolę jak atrybut scope elementu <bean> (więcej informacji na temat atrybutu scope znajdziesz w podrozdziale 2.5).

W listingu 6.14 została przedstawiona klasa CustomerRequestDetails, która używa adnotacji @Scope.

Listing 6.14. Przykład użycia adnotacji @Scope

Projekt: ch06-bankapp-jsr330

Położenie pliku: src/main/java/sample/spring/chapter06/bankapp/domain

```
package sample.spring.chapter06.bankapp.domain;

import javax.inject.Named;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;

@Named(value="customerRequestDetails")
@Scope(value=ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class CustomerRequestDetails { .... }
```

Adnotacja @Scope akceptuje atrybut value określający zasięg komponentu. Atrybutowi value można przypisać wartość prototype lub singleton i tym samym wskazać, że komponent ma zasięg odpowiednio prototypu lub singleton. Ewentualnie atrybutowi value można przypisać zdefiniowaną przez klasę ConfigurableBeanFactory stałą SCOPE_SINGLETON (wartość wynosi singleton) lub SCOPE_PROTOTYPE (wartość wynosi prototype).

Domyślnie komponenty Springa o zasięgu singleton są *wcześniej* inicjalizowane, czyli wraz z kontenerem Springa. Jeżeli komponent Springa ma być inicjalizowany z opóźnieniem, klasę komponentu o zasięgu singleton należy oznaczyć adnotacją @Lazy.



Adnotacja @Lazy służy do takich samych celów jak atrybut lazy-init elementu <bean>. Więcej informacji na temat atrybutu lazy-init znajdziesz w podrozdziale 2.5.

W listingu 6.15 przedstawiamy przykład użycia adnotacji @Lazy.

Listing 6.15. Przykład użycia adnotacji @Lazy

```
@Lazy(value=true)
@Component
public class Sample { .... }
```

Atrybut `value` adnotacji `@Lazy` określa sposób inicjalizacji komponentu: wczesny lub opóźniony. Jeżeli wartością tego atrybutu jest `true`, komponent będzie inicjalizowany z opóźnieniem.

Zależności komponentu można wyraźnie wskazać za pomocą adnotacji `@DependsOn`. Listing 6.16 przedstawia przykład użycia adnotacji `@DependsOn`.

Listing 6.16. Przykład użycia adnotacji `@DependsOn`

```
@DependsOn(value = {"beanA", "beanB"})
@Component
public class Sample { ..... }
```

W powyższym listingu adnotacja `@DependsOn` w klasie `Sample` informuje kontener Springa o konieczności utworzenia komponentów `beanA` i `beanB` przed utworzeniem egzemplarza klasy `Sample`.

➔ Adnotacja `@DependsOn` służy do takich samych celów jak atrybut `depends-on` elementu `<bean>`. Więcej informacji na temat atrybutu `depends-on` znajdziesz w podrozdziale 4.3.

W przypadku istnienia dla zależności wielu kandydatów do automatycznego wiązania adnotacja `@Primary` wskazuje komponent jako podstawowego kandydata do automatycznego wiązania. Listing 6.17 przedstawia przykład użycia adnotacji `@Primary`.

Listing 6.17. Przykład użycia adnotacji `@Primary`

```
@Primary
@Component
public class Sample { ..... }
```

➔ Adnotacja `@Primary` służy do takich samych celów jak atrybut `primary` elementu `<bean>`. Więcej informacji na temat atrybutu `primary` znajdziesz w podrozdziale 4.6.

Przechodzimy teraz do omówienia oferowanej przez Spring adnotacji `@Value`, która ułatwia proces konfiguracji klas komponentów.

6.8. Ułatwienie konfiguracji komponentu za pomocą adnotacji `@Value`

W poprzednich rozdziałach przedstawiliśmy przykłady, w których niezbędne informacje konfiguracyjne komponentu były dostarczane za pomocą atrybutu `value` elementów `<property>` i `<constructor-arg>`. Ponieważ komponenty Springa nie są definiowane w pliku XML kontekstu aplikacji, oferowana przez Spring adnotacja `@Value` jest wykorzystywana do takich samych celów jak atrybut `value` elementów `<property>` i `<constructor-arg>`. Trzeba zwrócić uwagę na fakt, że adnotacji `@Value` można używać na poziomach właściwości, metody, parametru metody i argumentu konstruktora.

IMPORT **Rozdział06/ch06-value-annotation.** (Ten projekt pokazuje aplikację, w której oferowana przez Spring adnotacja `@Value` jest używana do konfiguracji komponentów Springa. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `SampleApp` projektu).

W listingu 6.18 przedstawiamy przykład użycia adnotacji `@Value` na poziomie właściwości.

Listing 6.18. Klasa *Sample* przedstawiająca przykład użycia adnotacji `@Value`

Projekt: *ch06-value-annotation*

Położenie pliku: *src/main/java/sample/spring/chapter06/beans*

```
package sample.spring.chapter06.beans;

import org.springframework.beans.factory.annotation.Value;

@Component(value="sample")
public class Sample {
    @Value("Dowolna waluta")
    private String currency;
    ....
}
```

W powyższym listingu właściwość `currency` jest oznaczona adnotacją `@Value`. Atrybut `value` adnotacji `@Value` określa wartość domyślną właściwości. Użycie atrybutu `value` jest opcjonalne, a więc zapisy `@Value(value="Dowolna waluta")` i `@Value("Dowolna waluta")` oznaczają dokładnie to samo.

Istnieje również możliwość użycia wyrażenia SpEL jako wartości adnotacji `@Value`. SpEL to *język wyrażień*, którego można używać do wykonywania zapytań i do operowania obiektami w trakcie działania aplikacji. W listingu 6.19 przedstawiamy przykład adnotacji `@Value` wykorzystującej wyrażenie SpEL.

Listing 6.19. Klasa *Sample* przedstawiająca przykład użycia adnotacji `@Value` wykorzystującej wyrażenie SpEL

Projekt: *ch06-value-annotation*

Położenie pliku: *src/main/java/sample/spring/chapter06/beans*

```
package sample.spring.chapter06.beans;

import org.springframework.beans.factory.annotation.Value;
@Component(value="sample")
public class Sample {
    @Value("#{configuration.environment}")
    private String environment;
    ....
    @Value("#{configuration.getCountry()}")
    private String country;

    @Value("#{configuration.state}")
    private String state;
    ....
}
```

W powyższym listingu widać, że adnotacja `@Value` ma wartość wyrażoną w postaci składni `#{<wyrażenie-spel>}`. Wyrażenie SpEL podawane przed adnotacją `@Value` będzie przetworzone przez `BeanPostProcessor`. Wyrażenia SpEL mogą korzystać z formatu `<nazwaKomponentu>.<właściwość lub metoda>` w celu pobrania wartości. Na przykład `#{configuration.environment}` oznacza pobranie wartości właściwości `environment` komponentu o nazwie `configuration`, natomiast `#{configuration.getCountry()}` oznacza wywołanie metody `getCountry()` komponentu `configuration`.

W listingu 6.20 została przedstawiona klasa Javy komponentu `configuration`, do którego odwołujemy się przez wyrażenie SpEL pokazane w listingu 6.19.

Listing 6.20. Klasa komponentu `configuration`

Projekt: `ch06-value-annotation`

Położenie pliku: `src/main/java/sample/spring/chapter06/beans`

```
package sample.spring.chapter06.beans;

import org.springframework.stereotype.Component;

@Component("configuration")
public class Configuration {
    public static String environment = "DEV";

    public String getCountry() {
        return "Dowolny kraj";
    }

    public String getState() {
        return "Dowolne województwo";
    }

    public String[] splitName(String name) {
        return name.split(" ");
    }

    public String getCity() {
        return "Dowolne miasto";
    }
}
```

W powyższym listingu klasa `Configuration` przedstawia komponent Springa definiujący właściwości i metody. Jeżeli porównasz kod przedstawiony w listingach 6.19 i 6.20, zauważysz, że wyrażenie `#{configuration.environment}` odwołuje się do zmiennej statycznej `environment` zdefiniowanej w klasie `Configuration`, wyrażenie `#{configuration.getCountry()}` odwołuje się do metody `getCountry()` klasy `Configuration`, natomiast wyrażenie `#{configuration.state}` odwołuje się do metody `getState()` klasy `Configuration`.

Metoda `main()` klasy `SampleApp` w projekcie `ch06-value-annotation` pobiera egzemplarz komponentu `Sample` z egzemplarza `ApplicationContext`, a następnie wyświetla wartości różnych atrybutów egzemplarza komponentu `Sample`. Jeżeli wykonasz metodę `main()` klasy `SampleApp`, otrzymasz następujące dane wyjściowe:

```
Sample [environment=DEV, currency=Dowolna waluta, country=Dowolny kraj, state=Dowolne
↳województwo, splitName=[FirstName, LastName], city=Dowolne miasto]
```

Na podstawie powyższych danych wyjściowych można wyciągnąć następujące wnioski:

- Wyrażenie `#{configuration.environment}` powoduje, że właściwość `environment` klasy `Sample` będzie miała przypisaną wartość `DEV`, czyli wartość publicznej właściwości statycznej `environment` klasy `Configuration`.

- Wyrażenie `{configuration.getCountry()}` powoduje, że właściwość `country` klasy `Sample` będzie miała wartość `Dowolny kraj`, która jest wartością zwracaną przez wywołanie metody `getCountry()` klasy `Configuration`.
- Wyrażenie `{configuration.state}` powoduje, że właściwość `state` klasy `Sample` będzie miała wartość `Dowolne województwo`, która jest wartością zwracaną przez wywołanie metody `getState()` klasy `Configuration`.

W powyższym przykładzie pokazaliśmy możliwość użycia wyrażeń SpEL do pobrania informacji konfiguracyjnych z innych komponentów.



SpEL to potężny język wyrażeń, który oferuje znacznie więcej możliwości, niż przedstawiliśmy w tej książce. Warto zajrzeć do dokumentacji Springa, aby dowiedzieć się więcej na temat SpEL.

W listingu 6.21 przedstawiamy przykład użycia adnotacji `@Value` na poziomie metody i parametru metody.

Listing 6.21. Klasa `Sample` pokazująca użycie adnotacji `@Value` na poziomie metody i parametru metody

Projekt: `ch06-value-annotation`

Położenie pliku: `src/main/java/sample/spring/chapter06/beans`

```
package sample.spring.chapter06.beans;

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

@Component(value="sample")
public class Sample {
    .....
    private String[] splitName;
    private String city;

    @Autowired
    public void splitName(@Value("#{configuration.splitName(FirstName LastName')}")
        String[] splitName) {
        this.splitName = splitName;
    }

    @Autowired
    @Value("#{configuration.getCity()}")
    public void city(String city) {
        this.city = city;
    }
    .....
}
```

W powyższym listing widać, że metody oznaczone adnotacją `@Autowired` wykorzystują adnotację `@Value` na poziomie metody i parametru metody. Warto zauważyć, że adnotacja `@Value` może być stosowana na poziomie metody i parametru metody *tylko wtedy*, gdy metoda jest oznaczona adnotacją `@Autowired`, `@Resource` lub `@Inject`. Wyrażenie SpEL w postaci `{configuration.splitName('First`

↳ `Name LastName'))` powoduje wywołanie metody `splitName()` klasy `Configuration` wraz z argumentem `FirstName LastName`. To pokazuje, że wyrażenia SpEL można używać do wywoływania metod akceptujących argumenty.



Adnotacja `@Value` jest przetwarzana przez `BeanPostProcessor`. Dlatego nie powinieneś jej używać w klasach komponentów implementujących interfejs `BeanFactoryPostProcessor` lub `BeanPostProcessor`.

Użycie wyrażen SpEL *nie* jest ograniczone jedynie do adnotacji `@Value`. Wyrażenia te można stosować także w definicjach komponentów znajdujących się w pliku XML kontekstu aplikacji.

IMPORT **Rozdział106/ch06-spel-example.** (Ten projekt pokazuje aplikację używającą wyrażen SpEL w definicjach komponentów. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `SampleApp` projektu).

W listingu 6.22 przedstawiamy przykład użycia wyrażen SpEL w definicjach komponentów.

Listing 6.22. Plik `applicationContext.xml` przedstawiający przykład użycia wyrażen SpEL w definicjach komponentów

Projekt: `ch06-spel-example`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans ..... >
  <bean id="sample" class="sample.spring.chapter06.beans.Sample">
    <property name="environment" value="#{configuration.environment}" />
    <property name="currency" value="Dowolna waluta" />
    <property name="country" value="#{configuration.getCountry()}" />
    <property name="state" value="#{configuration.state}" />
  </bean>

  <bean id="configuration" class="sample.spring.chapter06.beans.Configuration" />
</beans>
```

W powyższym listingu widać, że definicja komponentu dla klasy `Sample` wykorzystuje wyrażenia SpEL (odwołujące się do komponentu `Configuration`) w celu ustawienia wartości domyślnych właściwości `environment`, `currency`, `country` i `state`.

Teraz zobaczysz, jak można przeprowadzić weryfikację obiektów w aplikacji Springa, używając do tego oferowanego przez Spring interfejsu `Validator`.

6.9. Weryfikacja obiektów za pomocą interfejsu `Validator`

Interfejs `Validator` stanowi część oferowanego przez Spring API weryfikacji i pozwala na przeprowadzenie weryfikacji obiektów. Można go wykorzystać do weryfikacji obiektów na dowolnej warstwie aplikacji, na przykład na warstwie sieciowej lub trwałego przechowywania danych.

IMPORT **Rozdział106/ch06-validator-interface.** (Ten projekt pokazuje aplikację `MyBank` używającą interfejsu `Validator` do weryfikacji obiektu `FixedDepositDetails`. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).



Alternatywą dla użycia interfejsu `Validator` jest wykorzystanie zdefiniowanych w specyfikacji JSR 303 adnotacji w celu określenia ograniczeń mających zastosowanie w danym obiekcie. Adnotacje JSR 303 omówimy w dalszej części rozdziału.

Obiekt `FixedDepositDetails` w aplikacji `MyBank` przedstawia informacje szczegółowe o lokacie. Listing 6.23 przedstawia klasę `FixedDepositDetails`.

Listing 6.23. Klasa `FixedDepositDetails`

Projekt: `ch06-validator-interface`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp/domain`

```
package sample.spring.chapter06.bankapp.domain;

public class FixedDepositDetails {
    private long id;
    private float depositAmount;
    private int tenure;
    private String email;

    public FixedDepositDetails(long id, float depositAmount, int tenure,
                               String email) {
        this.id = id;
        this.depositAmount = depositAmount;
        this.tenure = tenure;
        this.email = email;
    }
    .....
    // Metody getter i setter dla zmiennych egzemplarza.
    public float getDepositAmount() {
        return depositAmount;
    }
    .....
}
```

W powyższym listingu widać, że klasa `FixedDepositDetails` definiuje zmienne egzemplarza `id`, `depositAmount`, `tenure` i `email`. Przyjmujemy następujące założenie: przed zapisaniem w systemie informacji szczegółowych o lokacie chcemy się upewnić, że jej wysokość (określona przez zmienną egzemplarza `depositAmount`) ma wartość większą niż 0.

W celu przeprowadzenia weryfikacji właściwości `depositAmount` obiektu `FixedDepositDetails` konieczne jest utworzenie implementacji oferowanego przez Spring interfejsu `Validator`. W listingu 6.24 przedstawiamy rozwiązanie przeznaczone do weryfikacji obiektów typu `FixedDepositDetails`.

Listing 6.24. Klasa `FixedDepositValidator` przedstawiająca implementację oferowanego przez Spring interfejsu `Validator`

Projekt: `ch06-validator-interface`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp/validator`

```
package sample.spring.chapter06.bankapp.validator;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;
```



```

public class FixedDepositValidator implements Validator {
    @Override
    public boolean supports(Class<?> clazz) {
        return FixedDepositDetails.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        FixedDepositDetails fixedDepositDetails = (FixedDepositDetails) target;
        if (fixedDepositDetails.getDepositAmount() == 0) {
            errors.reject("zeroDepositAmount");
        }
    }
}

```

Interfejs `Validator` definiuje metody `supports()` i `validate()`. Metoda `supports()` sprawdza, czy podany egzemplarz obiektu (przedstawiany przez atrybut `clazz`) może być zweryfikowany. Jeżeli wartością zwrótną metody `supports()` jest `true`, wówczas metoda `validate()` będzie użyta do przeprowadzenia weryfikacji obiektu. W przedstawionym powyżej listingu metoda `supports()` klasy `FixedDepositValidator` sprawdza, czy podany egzemplarz obiektu jest typu `FixedDepositDetails`. Jeżeli metoda `supports()` zwróci wartość `true`, to metoda `validate()` obiektu `FixedDepositValidator` przeprowadzi operację weryfikacji obiektu. Metoda `validate()` akceptuje egzemplarz obiektu przeznaczony do weryfikacji oraz egzemplarz `Errors`. Metoda `reject()` egzemplarza `Errors` jest przeznaczona do przechowywania informacji o błędach, które wystąpiły w trakcie weryfikacji. Później można przeanalizować egzemplarz `Error`, aby dowiedzieć się więcej o błędach wykrytych podczas weryfikacji.

W listingu 6.25 widać, że metoda `createFixedDeposit()` klasy `FixedDepositServiceImpl` używa klasy `FixedDepositValidator` (patrz: listing 6.24) do przeprowadzenia weryfikacji obiektów `FixedDepositDetails`.

Listing 6.25. Klasa `FixedDepositServiceImpl` przeprowadzająca weryfikację obiektu `FixedDepositDetails`

Projekt: `ch06-validator-interface`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp/service`

```

package sample.spring.chapter06.bankapp.service;

import org.springframework.validation.BeanPropertyBindingResult;
import sample.spring.chapter06.bankapp.validator.FixedDepositValidator;
@Service(value="FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {

    @Autowired
    @Qualifier(value="myFixedDepositDao")
    private FixedDepositDao myFixedDepositDao;

    @Override
    public void createFixedDeposit(FixedDepositDetails fixedDepositDetails) throws
        Exception {
        BeanPropertyBindingResult bindingResult =
            new BeanPropertyBindingResult(fixedDepositDetails, "Errors");
        FixedDepositValidator validator = new FixedDepositValidator();

```

```

        validator.validate(fixedDepositDetails, bindingResult);

        if(bindingResult.getErrorCount() > 0) {
            logger.error("Podczas weryfikacji egzemplarza FixedDepositDetails wystąpiły
                błędy.");
        } else {
            myFixedDepositDao.createFixedDeposit(fixedDepositDetails);
            logger.info("Założono lokatę.");
        }
    }
}

```

Metoda `createFixedDeposit()` klasy `FixedDepositServiceImpl` przeprowadza weryfikację obiektu `FixedDepositDetails` (przedstawianego przez argument `fixedDepositDetails`), zanim zostanie przez `FixedDepositDao` zapisany w magazynie danych. Przedstawiona w powyższym listingu metoda `createFixedDeposit()` wykonuje kilka zadań:

- Utworzenie egzemplarza `FixedDepositValidator` i `BeanPropertyBindingResult`, czyli standardowo dostarczanej przez Spring domyślnej implementacji interfejsu `Errors`.
- Wywołanie metody `validate()` klasy `FixedDepositValidator` i przekazanie jej obiektu `FixedDepositDetails` oraz egzemplarza `BeanPropertyBindingResult`.
- Wywołanie metody `getErrorCount()` klasy `BeanPropertyBindingResult` w celu sprawdzenia, czy wystąpiły błędy podczas weryfikacji. W przypadku braku błędów weryfikacji następuje wywołanie metody `createFixedDeposit()` klasy `FixedDepositDao` w celu zapisania w magazynie danych informacji szczegółowych o lokacie.

W listingu 6.26 widać, że metoda `main()` klasy `BankApp` wywołuje metodę `createFixedDeposit()` klasy `FixedDepositServiceImpl` (patrz: listing 6.25) w celu sprawdzenia, czy weryfikacja została prawidłowo przeprowadzona przez metodę `validate()` klasy `FixedDepositValidator`.

Listing 6.26. Klasa *BankApp*

Projekt: *ch06-validator-interface*

Położenie pliku: *src/main/java/sample/spring/chapter06/bankapp*

```

package sample.spring.chapter06.bankapp;

public class BankApp {
    public static void main(String args[]) throws Exception {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");

        FixedDepositService fixedDepositService =
            context.getBean(FixedDepositService.class);

        FixedDepositService.createFixedDeposit(new FixedDepositDetails(1, 0,
            12, "nazwa-uzytkownika@nazwa-domeny.pl"));
        FixedDepositService.createFixedDeposit(new FixedDepositDetails(1, 1000,
            12, "nazwa-uzytkownika@nazwa-domeny.pl"));
    }
}

```

Najpierw metodzie `createFixedDeposit()` klasy `FixedDepositService` przekazywany jest obiekt `FixedDepositDetails`, w którym właściwości `depositAmount` przypisano wartość 0. W kolejnym wywołaniu przekazywany jest obiekt `FixedDepositDetails`, w którym właściwości `depositAmount` przypisano wartość 1000.

Po wykonaniu metody `main()` klasy `BankApp` w konsoli otrzymamy następujące dane wyjściowe:

Podczas weryfikacji egzemplarza `FixedDepositDetails` wystąpiły błędy.
Założono lokatę.

Komunikat o treści Podczas weryfikacji egzemplarza `FixedDepositDetails` wystąpiły błędy pokazuje, że `FixedDepositValidator` zgłasza błąd w przypadku weryfikacji egzemplarza `FixedDepositDetails`, w którym właściwości `depositAmount` przypisano wartość 0. Z kolei komunikat Założono lokatę pokazuje brak błędów podczas weryfikacji egzemplarza `FixedDepositDetails`, w którym właściwości `depositAmount` przypisano wartość 1000.



Oferowany przez Spring interfejs `Validator` jest zwykle używany w aplikacjach sieciowych opartych na Spring MVC podczas dołączania do odpowiedniego obiektu informacji wprowadzonych przez użytkownika w formularzu HTML.

Teraz zobaczymy, jak określić ograniczenia dla właściwości komponentu za pomocą adnotacji zdefiniowanych w specyfikacji JSR 303 i pozwolić Springowi na przeprowadzenie weryfikacji.

6.10. Określanie ograniczeń za pomocą adnotacji JSR 303

Specyfikacja JSR 303 (API weryfikacji komponentu) pozwala na użycie adnotacji w celu określenia ograniczeń dla komponentów JavaBeans. Podczas użycia JSR 303 wraz ze Springiem właściwości komponentu są oznaczane adnotacjami zdefiniowanymi w specyfikacji JSR 303, a Spring zajmuje się weryfikacją komponentu i dostarczeniem wyniku tej operacji.

IMPORT `Rozdział106/ch06-jsr303-validation`. (Ten projekt pokazuje aplikację `MyBank`, w której używane są adnotacje JSR 303. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

W listingu 6.27 została przedstawiona klasa `FixedDepositDetails`, która używa adnotacji JSR 303.

Listing 6.27. Klasa `FixedDepositDetails` przedstawiająca przykład użycia adnotacji JSR 303

Projekt: `ch06-jsr303-validation`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp/domain`

```
package sample.spring.chapter06.bankapp.domain;

import javax.validation.constraints.*;
import org.hibernate.validator.constraints.NotBlank;

public class FixedDepositDetails {
    @NotNull
    private long id;
```

```
@Min(1000)
@Max(500000)
private float depositAmount;

@Min(6)
private int tenure;

@NotBlank
@Size(min=5, max=100)
private String email;

public FixedDepositDetails(long id, float depositAmount, int tenure, String email) {
    this.id = id;
    this.depositAmount = depositAmount;
    this.tenure = tenure;
    this.email = email;
}
.....
}
```

`@NotNull`, `@Min`, `@Max`, `@NotBlank` i `@Size` to tylko niektóre z adnotacji zdefiniowanych przez JSR 303. W powyższym listingu widać, że dzięki użyciu adnotacji JSR 303 klasa `FixedDepositDetails` wyraźnie określa ograniczenia nakładane na jej właściwości. Z drugiej strony w przypadku użycia oferowanego przez Spring API Validation do weryfikacji obiektu informacje o ograniczeniach będą znajdowały się w implementacji interfejsu `Validator` (patrz: listing 6.24).

W tabeli 6.2 wymieniliśmy ograniczenia nakładane przez adnotacje JSR 303 w obiekcie `FixedDepositDetails` przedstawionym w listingu 6.27.



Aby użyć adnotacji JSR 303, w projekcie *ch06-jsr303-validation* została zdefiniowana zależność od pliku JAR zawierającego API dla JSR 303 (*validation-api-1.0.0.GA*) oraz od frameworku Hibernate Validator (*hibernate-validator-4.3.0.Final*). Framework ten dostarcza odpowiednią implementację dla JSR 303.

Jeżeli spojrzysz na polecenia `import` w listingu 6.27, zauważysz, że adnotacja `@NotBlank` jest definiowana przez framework Hibernate Validator, a nie specyfikację JSR 303. Hibernate Validator dostarcza dodatkowe adnotacje, których można używać wraz z JSR 303.

Po zdefiniowaniu ograniczeń JSR 303 dla klasy `FixedDepositDetails` możemy zobaczyć, jak za pomocą Springa przeprowadzić weryfikację obiektu `FixedDepositDetails`.

Obsługa specyfikacji JSR 303 w Springu

Spring obsługuje weryfikację obiektów, w których za pomocą JSR 303 zdefiniowano ograniczenia. Klasa `LocalValidatorFactoryBean` jest odpowiedzialna za wykrycie w ścieżce klas aplikacji istnienia dostawcy JSR 303 (podobnie jak framework Hibernate Validator) i jego inicjalizację. Trzeba w tym miejscu koniecznie dodać, że klasa `LocalValidatorFactoryBean` implementuje dostarczane przez JSR 303 interfejsy `Validator` i `ValidatorFactory`, a także oferowany przez Spring interfejs `Validator`.

Tabela 6.2. Ograniczenia nakładane na właściwości obiektu *FixedDepositDetails*

Adnotacja JSR 303	Opis ograniczenia
@NotNull	Właściwość oznaczona tą adnotacją nie może mieć wartości null. Na przykład właściwość <code>id</code> klasy <code>FixedDepositDetails</code> nie może mieć wartości null.
@Min	Właściwość oznaczona tą adnotacją musi mieć wartość większą lub równą podanej wartości minimalnej. Na przykład adnotacja <code>@Min(1000)</code> dla właściwości <code>depositAmount</code> obiektu <code>FixedDepositDetails</code> oznacza, że wartość właściwości <code>depositAmount</code> musi być większa lub równa 1000.
@Max	Właściwość oznaczona tą adnotacją musi mieć wartość mniejszą lub równą podanej wartości. Na przykład adnotacja <code>@Max(500000)</code> dla właściwości <code>depositAmount</code> obiektu <code>FixedDepositDetails</code> oznacza, że wartość właściwości <code>depositAmount</code> musi być mniejsza lub równa 500000.
@NotBlank	Właściwość oznaczona tą adnotacją nie może mieć wartości null lub być pusta. Na przykład właściwość <code>email</code> klasy <code>FixedDepositDetails</code> nie może być pusta ani nie może mieć przypisanej wartości null.
@Size	Wielkość wartości właściwości oznaczonej tą adnotacją musi zawierać się w przedziale definiowanym przez atrybuty <code>min</code> i <code>max</code> . Na przykład adnotacja <code>@Size(min=5, max=100)</code> dla właściwości <code>email</code> obiektu <code>FixedDepositDetails</code> oznacza, że wielkość wartości właściwości <code>email</code> musi być większa lub równa 5, ale jednocześnie mniejsza niż 100.

W listingu 6.28 przedstawiamy konfigurację komponentu `LocalValidatorFactoryBean` w pliku XML kontekstu aplikacji.

Listing 6.28. Plik *applicationContext.xml* przedstawiający konfigurację komponentu *LocalValidatorFactoryBean*

Projekt: *ch06-jsr303-validation*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean id="validator"
      class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" />
```

Jak widzisz, komponent `LocalValidatorFactoryBean` został skonfigurowany jak każdy inny komponent Springa. Po przeprowadzeniu konfiguracji komponentu `LocalValidatorFactoryBean` zobaczymy, w jaki sposób można go użyć do przeprowadzania weryfikacji.

W listingu 6.29 została przedstawiona klasa `FixedDepositServiceImpl` wymagająca, aby obiekt `FixedDepositDetails` został zweryfikowany, zanim informacje szczegółowe o lokacie zostaną zapisane w magazynie danych.

Listing 6.29. Klasa *FixedDepositServiceImpl* przeprowadzająca weryfikację obiektu *FixedDepositDetails*

Projekt: *ch06-jsr303-validation*

Położenie pliku: *src/main/java/sample/spring/chapter06/bankapp/service*

```
package sample.spring.chapter06.bankapp.service;

import org.springframework.validation.BeanPropertyBindingResult;
```

```

import org.springframework.validation.Validator;
.....
@Service(value="FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {

    @Autowired
    private Validator validator;

    @Autowired
    @Qualifier(value="myFixedDepositDao")
    private FixedDepositDao myFixedDepositDao;

    @Override
    public void createFixedDeposit(FixedDepositDetails fixedDepositDetails) throws
        Exception {
        BeanPropertyBindingResult bindingResult =
            new BeanPropertyBindingResult(fixedDepositDetails, "Errors");
        validator.validate(fixedDepositDetails, bindingResult);

        if(bindingResult.getErrorCount() > 0) {
            logger.error("Podczas weryfikacji egzemplarza FixedDepositDetails wystąpiły
                błędy.");
        } else {
            myFixedDepositDao.createFixedDeposit(fixedDepositDetails);
            logger.info("Założono lokatę.");
        }
    }
}

```

W powyższym listingu widać, że implementacja oferowanego przez Spring interfejsu `Validator` jest wskazywana przez właściwość `validator`. Ponieważ klasa `LocalValidatorFactoryBean` implementuje interfejs `Validator`, egzemplarz `LocalValidatorFactoryBean` został przypisany właściwości `validator`. Metoda `createFixedDeposit()` klasy `FixedDepositServiceImpl` wywołuje metodę `validate()` interfejsu `Validator` w celu przeprowadzenia weryfikacji obiektu `FixedDepositDetails`.

W listingu 6.29 warto zwrócić uwagę na jeden interesujący szczegół: specyfikacja JSR 303 *nie* jest używana do przeprowadzenia weryfikacji obiektu `FixedDepositDetails`. Zamiast niej wykorzystujemy API Validation oferowane przez Spring. Takie rozwiązanie jest możliwe, ponieważ klasa `LocalValidatorFactoryBean` implementuje metodę `validate()` interfejsu `Validator` w celu użycia specyfikacji JSR 303 do przeprowadzania weryfikacji obiektów, tym samym ukrywając przed programistami szczegóły dotyczące API JSR 303.

Ponieważ klasa `LocalValidatorFactoryBean` implementuje zdefiniowane przez JSR 303 interfejsy `Validator` i `ValidatorFactory`, zyskujesz możliwość użycia API JSR 303 do przeprowadzenia weryfikacji obiektu `FixedDepositDetails`. W listingu 6.30 przedstawiamy alternatywną implementację klasy `FixedDepositServiceImpl` wykorzystującą do przeprowadzania weryfikacji oferowany przez JSR 303 interfejs `Validator`.

W przedstawionym powyżej listingu odwołanie do implementacji JSR 303 znajduje się we właściwości `validator`. Ponieważ klasa `LocalValidatorFactoryBean` implementuje zdefiniowany w specyfikacji JSR 303 interfejs `Validator`, egzemplarz `LocalValidatorFactoryBean` zostaje przypisany

Listing 6.30. Klasa *FixedDepositServiceImplJsr303* przeprowadzająca weryfikację obiektu *FixedDepositDetails*Projekt: *ch06-jsr303-validation*Położenie pliku: *src/main/java/sample/spring/chapter06/bankapp/service*

```
package sample.spring.chapter06.bankapp.service;

import javax.validation.ConstraintViolation;
import javax.validation.Validator;

@Service(value = "FixedDepositServiceJsr303")
public class FixedDepositServiceJsr303Impl implements FixedDepositService {
    .....
    @Autowired
    private Validator validator;

    @Autowired
    @Qualifier(value = "myFixedDepositDao")
    private FixedDepositDao myFixedDepositDao;

    @Override
    public void createFixedDeposit(FixedDepositDetails fixedDepositDetails) throws
        Exception {
        Set<ConstraintViolation<FixedDepositDetails>> violations =
            validator.validate(fixedDepositDetails);

        Iterator<ConstraintViolation<FixedDepositDetails>> itr = violations.iterator();

        if (itr.hasNext()) {
            logger.error("Podczas weryfikacji egzemplarza FixedDepositDetails wystąpiły
                błędy.");
        } else {
            myFixedDepositDao.createFixedDeposit(fixedDepositDetails);
            logger.info("Założono lokatę.");
        }
    }
}
```

właściwości `validator`. Metoda `createFixedDeposit()` przeprowadza weryfikację obiektu `FixedDepositDetails` za pomocą wywołania metody `validate()` interfejsu `Validator`. Wartością zwrótną metody `validate()` jest obiekt `java.util.Set` zawierający informacje o zgłoszonych przez dostawcę JSR 303 złamaniach nałożonych ograniczeń. Obiekt `java.util.Set` zwrócony przez metodę `validate()` można sprawdzić, aby dowiedzieć się, czy nastąpiło złamanie jakichkolwiek ograniczeń. Na przykład w powyższym listingu metoda `createFixedDeposit()` wywołuje metodę `createFixedDeposit()` klasy `FixedDepositDao` tylko wtedy, gdy obiekt `java.util.Set` nie zawiera żadnych informacji o złamaniu nałożonych ograniczeń.

Z tego podrozdziału dowiedziałeś się, jak wykorzystać oferowaną przez Spring obsługę JSR 303 do przeprowadzania weryfikacji obiektów. Przeanalizowaliśmy jedynie ograniczenia, takie jak `@NotNull`, `@Size` itd., standardowo dostarczane przez JSR 303. Warto wiedzieć, że JSR 303 pozwala również na tworzenie własnych ograniczeń i użycie ich w aplikacji. Na przykład można utworzyć własne ograniczenie o nazwie `@MyConstraint` oraz odpowiedni weryfikator w celu wymuszenia na obiekcie przestrzegania nałożonych ograniczeń.

Przechodzimy teraz do adnotacji, za pomocą których można programowo konfigurować komponenty Springa.

6.11. Programowa konfiguracja komponentów Springa za pomocą adnotacji @Configuration i @Bean

Adnotacji @Configuration i @Bean można używać do przeprowadzenia programowej konfiguracji komponentów Springa. Jeżeli klasa zostanie oznaczona adnotacją @Configuration, wskazuje to, że klasa zawiera metody oznaczone adnotacją @Bean, które zwracają egzemplarze komponentu przeznaczone do zarejestrowania w kontenerze Springa.



W celu użycia klas oznaczonych adnotacją @Configuration do definiowania komponentów konieczna jest biblioteka CGLIB, ponieważ Spring rozszerza klasy oznaczone adnotacją @Configuration i dodaje funkcje do metod oznaczonych adnotacją @Bean. Począwszy od Springa w wersji 3.2, klasy CGLIB są umieszczane w pliku JAR o nazwie *spring-core* i dlatego biblioteki tej nie trzeba wyraźnie definiować jako zależności projektu.

IMPORT **Rozdział06/ch06-bankapp-configuration.** (Ten projekt pokazuje aplikację MyBank, w której używane są adnotacje @Configuration i @Bean do przeprowadzenia programowej konfiguracji komponentów. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

W listingu 6.31 została przedstawiona klasa `BankAppConfiguration` używająca adnotacji @Configuration.

Listing 6.31. Klasa `BankAppConfiguration` przedstawiająca przykład użycia adnotacji @Configuration i @Bean

Projekt: `ch06-bankapp-configuration`

Położenie pliku: `src/main/java/sample/spring/chapter06/bankapp`

```
package sample.spring.chapter06.bankapp;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;
.....
@Configuration
public class BankAppConfiguration {
    .....
    @Bean(name = "customerRegistrationService")
    @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public CustomerRegistrationService customerRegistrationService() {
        return new CustomerRegistrationServiceImpl();
    }
    .....
}
```

Klasa `BankAppConfiguration` definiuje oznaczone adnotacjami @Bean metody, których wartością zwrótną są egzemplarze komponentów. Atrybut `name` adnotacji @Bean określa nazwę, pod jaką zwrócony egzemplarz komponentu zostanie zarejestrowany w kontenerze Springa. Adnotacja @Scope wymaga określenia zasięgu (singleton lub prototypu) dla zwróconego egzemplarza komponentu.



Adnotacja `@Scope` jest używana także w poziomie typu w celu wskazania zasięgu komponentu Springa. W listingu 6.14 pokazaliśmy użycie adnotacji `@Scope` na poziomie typu.

Przedstawiona w listingu 6.31 metoda `customerRegistrationService()` zwraca egzemplarz komponentu `CustomerRegistrationService`, który w kontenerze Springa jest zarejestrowany jako komponent o nazwie `customerRegistrationService` i zasięgu prototypu. Wykonanie metody `customerRegistrationService()` ma taki sam efekt jak umieszczenie w pliku XML kontekstu aplikacji poniższej definicji komponentu:

```
<bean id="customerRegistrationService" scope="prototype"
      class="sample.spring.chapter06.bankapp.service.CustomerRegistrationServiceImpl" />
```

W tabeli 6.3 wymieniliśmy atrybuty, których w adnotacji `@Bean` można używać do przeprowadzenia konfiguracji egzemplarza komponentu.

Tabela 6.3. Atrybuty pozwalające na konfigurację egzemplarza komponentu

Wartość atrybutu type	Opis
Autowire	Taki sam efekt jak w przypadku atrybutu <code>autowire</code> elementu <code><bean></code> (z podrozdziału 4.6 dowiesz się więcej na temat atrybutu <code>autowire</code>). Jeżeli komponent zwrócony przez metodę oznaczoną adnotacją <code>@Bean</code> jest zależny od innych komponentów, wówczas za pomocą atrybutu <code>autowire</code> można nakazać Springowi przeprowadzenie automatycznego wiązania zależności według nazwy lub typu.
initMethod	Taki sam efekt jak w przypadku atrybutu <code>init-method</code> elementu <code><bean></code> (z podrozdziału 5.2 dowiesz się więcej na temat atrybutu <code>init-method</code>).
destroyMethod	Taki sam efekt jak w przypadku atrybutu <code>destroy-method</code> elementu <code><bean></code> (z podrozdziału 5.2 dowiesz się więcej na temat atrybutu <code>destroy-method</code>).

Trzeba w tym miejscu koniecznie dodać, że metody oznaczone adnotacją `@Bean` mogą mieć również adnotacje `@Lazy`, `@DependsOn`, `@Primary` i `@Scope`. Wymienione adnotacje są stosowane dla egzemplarza obiektu zwracanego przez metodę oznaczoną adnotacją `@Bean`. Na przykład adnotacja `@DependsOn` pozwala na niejawne określenie zależności egzemplarza obiektu zwróconego przez metodę oznaczoną adnotacją `@Bean`. Ponadto jeśli egzemplarz komponentu zwróconego przez metodę oznaczoną adnotacją `@Bean` implementuje interfejs cyklu życiowego (taki jak `InitializingBean` i `DisposableBean`) oraz oferowane przez Spring interfejsy `*Aware` (takie jak `ApplicationContextAware`, `BeanNameAware`), wówczas będzie otrzymywał wywołania zwrotne z kontenera Springa.

W przedstawionych dotąd przykładach tworzyliśmy egzemplarz klasy `ClassPathXmlApplicationContext` (implementacja interfejsu `ApplicationContext`) w celu przedstawienia kontenera Springa. Jeżeli jako źródła komponentu używasz klasy oznaczonej adnotacją `@Configuration`, musisz utworzyć egzemplarz klasy `AnnotationConfigApplicationContext` (inna implementacja `ApplicationContext`) do przedstawienia kontenera Springa.

W listingu 6.32 została przedstawiona klasa `BankApp` tworząca egzemplarz klasy `AnnotationConfigApplicationContext`, a następnie pobierająca komponenty z nowo utworzonego egzemplarza `AnnotationConfigApplicationContext`.

Listing 6.32. Klasa *BankApp* przedstawiająca przykład użycia klasy *AnnotationConfigApplicationContext*Projekt: *ch06-bankapp-configuration*Położenie pliku: *src/main/java/sample/spring/chapter06/bankapp*

```

package sample.spring.chapter06.bankapp;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class BankApp {
    public static void main(String args[]) throws Exception {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(BankAppConfiguration.class);
        ....
        FixedDepositService fixedDepositService =
            context.getBean(FixedDepositService.class);
        FixedDepositService.createFixedDeposit(new FixedDepositDetails(1, 1000,
            12, "nazwa-uzytkownika@nazwa-domeny.pl"));
        ....
    }
}

```

W powyższym listingu klasa *BankAppConfiguration* jest przekazywana jako argument konstruktora klasy *AnnotationConfigApplicationContext*. Ponieważ klasa *AnnotationConfigApplicationContext* implementuje interfejs *ApplicationContext*, dostęp do zarejestrowanych komponentów odbywa się w dokładnie taki sam sposób jak w przypadku *ClassPathXmlApplicationContext*.

Konieczne zwróć uwagę na fakt, że metody oznaczone adnotacją *@Bean* mogą być również zdefiniowane w klasach oznaczonych adnotacją *@Component* i adnotacją JSR 330 *@Named*. W przypadku komponentów zdefiniowanych w wielu klasach oznaczonych adnotacjami *@Configuration*, *@Component* i *@Named* wszystkie klasy trzeba przekazać konstruktorowi *AnnotationConfigApplicationContext*.

W listingu 6.33 zostały przedstawione oznaczone adnotacjami *@Bean* metody klasy *BankAppConfiguration*, które zwracają implementacje *BeanFactoryPostProcessor* i *BeanPostProcessor*.

Listing 6.33. Klasa *BankAppConfiguration* definiująca komponenty *BeanFactoryPostProcessor* i *BeanPostProcessor*Projekt: *ch06-bankapp-configuration*Położenie pliku: *src/main/java/sample/spring/chapter06/bankapp*

```

package sample.spring.chapter06.bankapp;

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

public class BankAppConfiguration {
    ....
    @Bean
    public ExampleBeanPostProcessor exampleBeanPostProcessor() {
        return new ExampleBeanPostProcessor();
    }
}

```

```
@Bean
public static BeanNamePrinterBeanFactoryPostProcessor applicationConfigurer() {
    return new BeanNamePrinterBeanFactoryPostProcessor();
}
}
```

W powyższym listingu egzemplarz `ExampleBeanPostProcessor` przedstawia implementację `BeanPostProcessor` wyświetlającą w konsoli komunikat *przed* inicjalizacją nowo utworzonego egzemplarza komponentu i *po* oraz egzemplarz `BeanNamePrinterBeanFactoryPostProcessor` przedstawiający implementację `BeanFactoryPostProcessor` wyświetlającą nazwy wszystkich komponentów zarejestrowanych w kontenerze Springa.

Jeżeli przejdziesz do projektu *ch06-bankapp-configuration* i wykonasz metodę `main()` klasy `BankApp`, zauważysz, że egzemplarz `BeanNamePrinterBeanFactoryPostProcessor` jest wywoływany przed utworzeniem przez kontener Springa jakiegokolwiek komponentu zdefiniowanego przez klasę oznaczoną adnotacją `@Configuration`. Natomiast egzemplarz `ExampleBeanPostProcessor` jest wywoływany w trakcie tworzenia przez kontener Springa każdego nowego egzemplarza komponentu. To pokazuje, że niezależnie od sposobu konfiguracji komponentu — czyli deklaracyjnie (za pomocą pliku XML kontekstu aplikacji) lub programowo (za pomocą klasy oznaczonej adnotacją `@Configuration`) — komponenty implementujące interfejsy wywołań zwrotnych (np. `ApplicationContextAware`, `BeanNameAware`, `InitializingBean`, `DisposableBean`, `BeanFactoryPostProcessor`) otrzymują wywołania zwrotne z kontenera Springa.

6.12. Podsumowanie

W tym rozdziale poznałeś adnotacje, które można wykorzystać w celu ułatwienia zadania tworzenia aplikacji Springa. Dowiedziałeś się, jak desygnować klasę komponentu jako komponent Springa, używając do tego adnotacji `@Component`, `@Service`, `@Repository` i `@Controller`, jak przeprowadzać skanowanie ścieżki klas w celu automatycznej rejestracji komponentów Springa w kontenerze, jak weryfikować obiekty za pomocą API Validation i adnotacji zdefiniowanych w specyfikacji JSR 303, jak przeprowadzać wstrzykiwanie zależności za pomocą oferowanej przez Spring adnotacji `@Autowired`, adnotacji JSR 330 `@Inject` i adnotacji JSR 250 `@Resource`. Ponadto zobaczyłeś, jak użyć adnotacji `@Configuration` i `@Bean` do programowej konfiguracji komponentów. W kolejnym rozdziale przekonasz się, jak Spring ułatwia pracę z bazami danych.

Rozdział 7. Współpraca z bazą danych

7.1. Wprowadzenie

Spring ułatwia pracę z bazami danych, dostarczając warstwę abstrakcji opartą na interfejsie JDBC. Ułatwia także pracę z frameworkami ORM (ang. *object relational mapping* — mapowanie obiektowo relacyjne), takimi jak Hibernate (<http://hibernate.org/>) i MyBatis (<http://mybatis.org/>), przeznaczonymi do pracy z bazami danych. W tym rozdziale przeanalizujemy przykłady pokazujące, jak Spring ułatwia opracowywanie aplikacji współdziałających z bazami danych.

➔ Przykłady przedstawione w rozdziale korzystają z frameworku Hibernate 4. Jeżeli używasz Hibernate 3, wówczas w odpowiednich miejscach przykładowych fragmentów kodu będziesz musiał wprowadzić pewne zmiany.

Na początku rozdziału pokażemy przykładową aplikację używającą oferowanej przez Spring abstrakcji JDBC do współpracy z bazą danych MySQL. Następnie przystąpimy do opracowania tej samej aplikacji, ale opartej na oferowanej przez Spring obsłudze frameworku Hibernate. Na końcu rozdziału zobaczysz, jak Spring zapewnia obsługę zarządzania transakcjami, w sposób zarówno programowy, jak i deklaracyjny.

Najpierw zobaczymy, jakie wymagania stawia aplikacja MyBank, którą będziemy budować w tym rozdziale.

7.2. Wymagania aplikacji MyBank

MyBank to aplikacja internetowa umożliwiająca klientom banku wyświetlanie informacji szczegółowych o kontach, generowanie dokumentów bankowych, zakładanie lokat, żądanie wydania książeczek czekowych itd. Rysunek 7.1 przedstawia tabele `BANK_ACCOUNT_DETAILS` i `FIXED_DEPOSIT_DETAILS` ➔ `_DETAILS`, w których przechowywane są dane aplikacji MyBank.

Rysunek 7.1.

Tabele bazy danych używanej przez aplikację MyBank

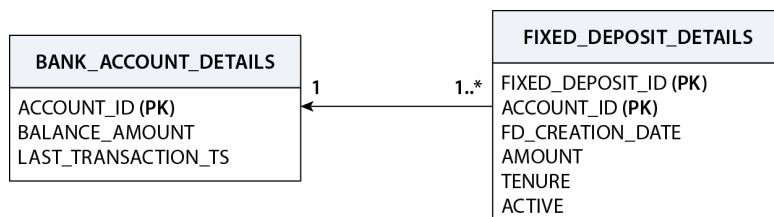


Tabela `BANK_ACCOUNT_DETAILS` zawiera informacje o kontach bankowych, natomiast w tabeli `FIXED_DEPOSIT_DETAILS` ➔ `_DETAILS` znajdują się informacje o lokatach założonych przez klienta. Na rysunku 7.1 wyraźnie widać istnienie wielu relacji typu „wiele do jednego” między tabelami `FIXED_DEPOSIT_DETAILS` ➔ `_DETAILS` i `BANK_ACCOUNT_DETAILS`. Kiedy klient banku zakłada nową lokatę, jej kwota jest odejmowana od wartości przechowywanej w kolumnie `BALANCE_AMOUNT` tabeli `BANK_ACCOUNT_DETAILS`, a informacje szczegółowe o lokacie są zapisywane w tabeli `FIXED_DEPOSIT_DETAILS`.

Oto kolumny tabeli `BANK_ACCOUNT_DETAILS`:

- `ACCOUNT_ID` — identyfikator konta, który pozwala na unikalną identyfikację klienta banku.
- `BALANCE_AMOUNT` — przechowuje bieżące saldo konta. Kiedy klient wydaje żądanie założenia nowej lokaty, wartość lokaty jest odejmowana od wysokości salda.
- `LAST_TRANSACTION_TS` — przechowuje datę i godzinę ostatniej transakcji przeprowadzonej na danym koncie.

Oto kolumny tabeli `FIXED_DEPOSIT_DETAILS`:

- `FIXED_DEPOSIT_ID` — identyfikator pozwalający na unikalną identyfikację lokaty. Kiedy klient zakłada nową lokatę, aplikacja MyBank generuje dla niej unikalny identyfikator, który będzie używany w przyszłych operacjach na tej lokacie. Wartość tej kolumny jest automatycznie generowana przez bazę danych MySQL.
- `ACCOUNT_ID` — klucz zewnętrzny identyfikujący konto bankowe, z którego została pobrana kwota równa wysokości założonej lokaty. Raz na kwartał odsetki od lokaty są przekazywane na konto bankowe wskazane przez tę kolumnę.
- `FD_CREATION_DATE` — data założenia lokaty.
- `AMOUNT` — wysokość założonej lokaty.
- `TENURE` — czas trwania lokaty (wyrażony w miesiącach). Lokatę można założyć na co najmniej 12 miesięcy, ale nie więcej niż na 60 miesięcy.
- `ACTIVE` — wskazuje, czy dana lokata jest aktualna. Jeżeli lokata jest aktualna, wówczas generuje odsetki przekazywane na konto wskazane w kolumnie `ACCOUNT_ID`.

Zobaczmy, jak można utworzyć aplikację MyBank w oparciu o oferowany przez Spring moduł JDBC.

7.3. Utworzenie aplikacji MyBank opartej na module JDBC

Dostępny w Springu moduł JDBC ułatwia tworzenie źródeł danych, ponieważ zajmuje się obsługą wszystkich szczegółów niskiego poziomu związanych z nawiązywaniem i zamykaniem połączeń, zarządzaniem transakcjami, przetwarzaniem wyjątków itd. W tym podrozdziale opracujemy aplikację MyBank opartą (jak wspomnieliśmy w poprzednim podrozdziale) na module JDBC oferowanym przez Spring. Dla zachowania prostoty zajmiemy się jedynie usługami i obiektami DAO aplikacji MyBank.

IMPORT **Rozdział107/ch07-bankapp-jdbc.** (Ten projekt pokazuje aplikację MyBank używającą oferowanego przez Spring modułu JDBC do współdziałania z bazą danych. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu. Przed wywołaniem tej metody musisz zainstalować bazę danych MySQL i wykonać skrypt SQL o nazwie `spring_bank_app_db.sql`, który znajdziesz w katalogu `sql` projektu `ch07-bankapp-jdbc`. Wykonanie skryptu `spring_bank_app_db.sql` powoduje utworzenie bazy danych `SPRING_BANK_APP_DB` wraz z tabelami `BANK_ACCOUNT_DETAILS` i `FIXED_DEPOSIT_DETAILS`. Ponadto zmodyfikuj plik `src/main/resources/META-INF/spring/database.properties` i wprowadź w nim odpowiednie informacje dotyczące używanej instalacji MySQL).

W aplikacji MyBank pierwszym zadaniem jest konfiguracja obiektu `javax.sql.DataSource` identyfikującego źródło danych, z którym będzie współdziałała aplikacja. Kolejny krok to implementacja obiektów DAO używających klas modułu JDBC do pracy ze źródłem danych. Teraz dokładnie omówimy wymienione powyżej kroki.

Konfiguracja źródła danych

Jeżeli używasz Springa do opracowania samodzielnej aplikacji, źródło danych możesz skonfigurować w pliku XML kontekstu aplikacji. Natomiast w przypadku budowania aplikacji korporacyjnej można zdefiniować źródło danych dołączone do serwera JNDI aplikacji, a następnie pobrać je z danych pliku XML kontekstu aplikacji w celu dalszego użycia w aplikacji. W przypadku projektu *ch07-bankapp-jdbc* źródło danych zostało skonfigurowane w pliku XML kontekstu aplikacji.

W listingu 7.1 przedstawiamy znajdującą się w pliku XML kontekstu aplikacji konfigurację źródła danych aplikacji MyBank.

Listing 7.1. Plik *applicationContext.xml* przedstawiający konfigurację źródła danych

Projekt: *ch07-bankapp-jdbc*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<context:property-placeholder location="classpath*:META-INF/spring/database.properties" />

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

W powyższym listingu element `<property-placeholder>` (więcej informacji na jego temat znajdziesz w podrozdziale 5.4) schematu context wczytuje właściwości z pliku *META-INF/spring/database.properties* i udostępnia je definicjom komponentów znajdującym się w pliku XML kontekstu aplikacji. Komponent `dataSource` przedstawia obiekt `javax.sql.DataSource` działający w charakterze fabryki do nawiązywania połączeń ze źródłem danych. Klasa `BasicDataSource` to implementacja interfejsu `javax.sql.DataSource` obsługującego funkcję puli połączeń. Klasa `BasicDataSource` jest częścią projektu Apache Commons DBCP (<http://commons.apache.org/proper/commons-dbcp/>) i obsługuje funkcję puli połączeń z bazą danych. Wartości właściwości `driverClassName`, `url`, `username` i `password` klasy `BasicDataSource` pochodzą z właściwości zdefiniowanych w pliku *database.properties*. Metoda `close()` klasy `BasicDataSource` powoduje zamknięcie wszystkich bezczynnych połączeń w puli. Ponieważ definicja komponentu dla klasy `BasicDataSource` zawiera przypisaną atrybutowi `destroy-method` wartość `close`, wszystkie bezczynne połączenia w puli są zamykane podczas usuwania przez kontener Springa egzemplarza komponentu `dataSource`.

Konfiguracja źródła danych w środowiskach Java EE

Jeżeli tworzysz aplikację korporacyjną, która będzie wdrożona w serwerze aplikacji, możesz użyć elementu `<jndi-lookup>` schematu jee oferowanego przez Spring i tym samym dołączone do JNDI źródło danych udostępnić w postaci komponentu Springa w `ApplicationContext`:

```
<jee:jndi-lookup jndi-name="java:comp/env/jdbc/bankAppDb" id="dataSource" />
```

W powyższym elemencie atrybut `jndi-name` podaje nazwę JNDI, do której odwołuje się obiekt `javax.sql.DataSource`, a atrybut `id` określa nazwę, pod jaką obiekt `javax.sql.DataSource` będzie zarejestrowany jako komponent w `ApplicationContext`.

Poznamy teraz pewne klasy modułu JDBC, których można używać w obiektach DAO do współdziałania z bazą danych.

Tworzenie obiektów DAO używających klas modułu JDBC

Moduł JDBC definiuje wiele klas ułatwiających współpracę z bazą danych. Na początek przyjrzymy się klasie `JdbcTemplate`, będącej sercem modułu JDBC w Springu. Inne klasy, które przeanalizujemy w tym podrozdziale, to `NamedParameterJdbcTemplate` i `SimpleJdbcInsert`. Aby dowiedzieć się więcej na temat pozostałych klas modułu JDBC, zajrzyj do dokumentacji Springa.

Klasa JdbcTemplate

Klasa `JdbcTemplate` zajmuje się zarządzaniem obiektami `Connection`, `Statement` i `ResultSet`, przechwytywaniem wyjątków JDBC i przekształcaniem ich na łatwe do zrozumienia wyjątki (takie jak `IncorrectResultSetColumnCountException` i `CannotGetJdbcConnectionException`), przeprowadzaniem operacji wsadowych itd. Programista aplikacji dostarcza jedynie polecenia SQL klasie `JdbcTemplate` i wyodrębnia wyniki po wykonaniu podanego kodu SQL.

Ponieważ klasa `JdbcTemplate` działa w charakterze opakowania dla obiektu `javax.sql.DataSource`, nie ma konieczności bezpośredniej pracy z obiektem `javax.sql.DataSource`. Egzemplarz `JdbcTemplate` jest najczęściej zainicjowany wraz z odwołaniem do obiektu `javax.sql.DataSource`, z którego musi pobrać połączenia, jak przedstawia listing 7.2.

Listing 7.2. Plik `applicationContext.xml` przedstawiający konfigurację klasy `JdbcTemplate`

Projekt: `ch07-bankapp-jdbc`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="dataSource" />
</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" . . . . .>
  . . . . .
</bean>
```

W powyższym listingu widać, że klasa `JdbcTemplate` definiuje właściwość `dataSource` odwołującą się do obiektu `javax.sql.DataSource`.

Jeżeli aplikacja używa źródła danych dołączonego do JNDI, można wykorzystać element `<jndi-lookup>` schematu jee do zarejestrowania takiego źródła danych jako komponentu w kontenerze Springa. Teraz klasa `JdbcTemplate` może odwoływać się do komponentu `javax.sql.DataSource` zarejestrowanego przez element `<jndi-lookup>`, jak przedstawia listing 7.3.

Listing 7.3. Konfiguracja klasy `JdbcTemplate` dla źródła danych dołączonego do JNDI

```
<beans .....
  xmlns:jee="http://www.springframework.org/schema/jee"
  xsi:schemaLocation=".....
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-4.0.xsd">

  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <jee:jndi-lookup jndi-name="java:comp/env/jdbc/bankAppDb" id="dataSource" />
  .....
</beans>
```

W powyższym listingu odwołanie do schematu jee znajduje się w pliku XML kontekstu aplikacji. Element `<jndi-lookup>` pobiera obiekt `javax.sql.DataSource` z JNDI, a następnie udostępnia jako komponent o nazwie `dataSource`, do którego odwołujemy się z poziomu klasy `JdbcTemplate`.

Egzemplarza `JdbcTemplate` zapewnia bezpieczeństwo wątków, co oznacza, że wiele obiektów DAO w aplikacji może współdzielić ten sam egzemplarz `JdbcTemplate` i używać go do pracy z bazą danych. W listingu 7.4 pokazujemy przykład użycia klasy `JdbcTemplate` przez metodę `createFixedDeposit()` klasy `FixedDepositDaoImpl`.

Listing 7.4. Klasa `FixedDepositDaoImpl` przedstawiająca zapisywanie danych za pomocą klasy `JdbcTemplate`

Projekt: `ch07-bankapp-jdbc`

Położenie pliku: `src/main/java/sample/spring/chapter07/bankapp/dao`

```
package sample.spring.chapter07.bankapp.dao;

import java.sql.*;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

@Repository(value = "FixedDepositDao")
public class FixedDepositDaoImpl implements FixedDepositDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;
    .....
    public int createFixedDeposit(final FixedDepositDetails fixedDepositDetails) {
        final String sql =
            "insert into fixed_deposit_details(account_id, fixedDeposit_creation_date,
            amount, tenure, active) values(?, ?, ?, ?, ?)";
```

```

    KeyHolder keyHolder = new GeneratedKeyHolder();

    jdbcTemplate.update(new PreparedStatementCreator() {

        @Override
        public PreparedStatement createPreparedStatement(Connection con)
            throws SQLException {
            PreparedStatement ps = con.prepareStatement(sql, new String[] {
                "fixed_deposit_id" });
            ps.setInt(1, fixedDepositDetails.getBankAccountId());
            ps.setDate(2, new java.sql.Date(fixedDepositDetails
                .getFixedDepositCreationDate().getTime()));
            ....
            return ps;
        }
    }, keyHolder);

    return keyHolder.getKey().intValue();
}
.....
}

```

W powyższym listingu klasa `FixedDepositDaoImpl` została oznaczona dostępną w Springu adnotacją `@Repository`, ponieważ reprezentuje klasę DAO. Egzemplarz `JdbcTemplate` skonfigurowany w pliku XML kontekstu aplikacji (patrz: listing 7.2) jest automatycznie wiązany w klasie `FixedDepositDaoImpl`. Metoda `update()` klasy `JdbcTemplate` akceptuje egzemplarze `PreparedStatementCreator` i `KeyHolder`. Klasa `PreparedStatementCreator` jest używana do przeprowadzania operacji wstawiania, uaktualniania i usuwania w bazie danych. Oferowany przez Spring interfejs `KeyHolder` to rodzaj pojemnika dla kluczy automatycznie generowanych podczas wykonywania zapytań SQL INSERT. Klasa `GeneratedKeyHolder` jest domyślną implementacją interfejsu `KeyHolder`.

Po zakończonym powodzeniem wykonaniu polecenia SQL INSERT automatycznie wygenerowane klucze są dodawane do egzemplarza `GeneratedKeyHolder`. Możesz je stamtąd pobrać za pomocą wywołania metody `getKey()`. W listingu 7.4 metoda `createFixedDeposit()` wstawia do tabeli `FIXED_DEPOSIT_DETAILS` informacje o lokacie i zwraca automatycznie wygenerowany klucz. Kod przedstawiony w listingu 7.4 pokazuje, że nie trzeba się przejmować przechwytywaniem wyjątku `SQLException`, który może być zgłoszony w trakcie wykonywania `PreparedStatement`. Wynika to z faktu, że klasa `JdbcTemplate` jest odpowiedzialna za przechwytywanie wyjątków `SQLExceptions` i ich obsługę.

Przechodzimy teraz do klasy `NamedParameterJdbcTemplate`.

Klasa `NamedParameterJdbcTemplate`

W listingu 7.4 pokazaliśmy, że jeżeli klasa `JdbcTemplate` jest używana do pracy z bazą danych, parametry przekazywane zapytaniu SQL są określane za pomocą miejsc zarezerwowanych `?`. Oferowana przez Spring klasa `NamedParameterJdbcTemplate` jest rodzajem opakowania dla egzemplarza `JdbcTemplate` i pozwala na użycie w zapytaniu SQL nazwanych parametrów zamiast miejsc zarezerwowanych `?`.

W listingu 7.5 przedstawiamy sposób konfiguracji klasy `NamedParameterJdbcTemplate` w pliku XML kontekstu aplikacji.

Listing 7.5. Plik `applicationContext.xml` przedstawiający konfigurację klasy `NamedParameterJdbcTemplate`

Projekt: `ch07-bankapp-jdbc`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="namedJdbcTemplate"
      class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
    <constructor-arg index="0" ref="dataSource" />
</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"....>
    ....
</bean>
```

W powyższym listingu pokazujemy, że klasa `NamedParameterJdbcTemplate` akceptuje obiekt `javax.sql.DataSource` jako argument konstruktora.

W listingu 7.6 została przedstawiona klasa `FixedDepositDaoImpl` używająca `NamedParameterJdbcTemplate` do pobrania z tabeli `FIXED_DEPOSIT_DETAILS` informacji szczegółowych o lokacie.

Listing 7.6. Klasa `FixedDepositDaoImpl` przedstawiająca przykład użycia klasy `NamedParameterJdbcTemplate`

Projekt: `ch07-bankapp-jdbc`

Położenie pliku: `src/main/java/sample/spring/chapter07/bankapp/dao`

```
package sample.spring.chapter07.bankapp.dao;

import java.sql.ResultSet;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
.....
@Repository(value = "FixedDepositDao")
public class FixedDepositDaoImpl implements FixedDepositDao {
    ....
    @Autowired
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
    ....
    public FixedDepositDetails getFixedDeposit(final int FixedDepositId) {
        final String sql = "select * from fixed_deposit_details where fixed_deposit_id
            = :FixedDepositId";

        SqlParameterSource namedParameters = new MapSqlParameterSource(
            "FixedDepositId", FixedDepositId);
        return namedParameterJdbcTemplate.queryForObject(sql, namedParameters,
            new RowMapper<FixedDepositDetails>() {
                public FixedDepositDetails mapRow(ResultSet rs, int rowNum) throws
                    SQLException {
                        FixedDepositDetails fixedDepositDetails = new FixedDepositDetails();
                        fixedDepositDetails.setActive(rs.getString("active"));
                        ....
                        return fixedDepositDetails;
                    }
            }
    }
}
```

```

        });
    }
}

```

Egzemplarz `NamedParameterJdbcTemplate` skonfigurowany w pliku XML kontekstu aplikacji (patrz: listing 7.5) jest automatycznie wiązany w klasie `FixedDepositDaoImpl`. W powyższym listingu zapytanie SQL przekazywane metodzie `queryForObject()` klasy `NamedParameterJdbcTemplate` zawiera nazwany parametr `FixedDepositId`. Wartości nazwanych parametrów są dostarczane za pomocą oferowanego przez Spring interfejsu `SqlParameterSource`. Klasa `MapSqlParameterSource` jest implementacją interfejsu `SqlParameterSource` przechowującą nazwane parametry (i ich wartości) w obiekcie `java.util.Map`. W omawianym listingu egzemplarz `MapSqlParameterSource` przechowuje wartość nazwanego parametru `FixedDepositId`. Metoda `queryForObject()` klasy `NamedParameterJdbcTemplate` wykonuje podane zapytanie SQL i zwraca pojedynczy obiekt. Obiekt `RowMapper` w Springu jest używany do przeprowadzenia mapowania każdego zwróconego rekordu na obiekt. W powyższym listingu zwrócony rekord w `ResultSet` jest przez `RowMapper` mapowany na obiekt `FixedDepositDetails`.

Przechodzimy teraz do omówienia klasy `SimpleJdbcInsert`.

Klasa `SimpleJdbcInsert`

W celu ułatwienia zadania utworzenia prostych zapytań SQL INSERT wydawanych tabeli klasa `SimpleJdbcInsert` używa metadanych znajdujących się w bazie danych.

W listingu 7.7 została przedstawiona klasa `BankAccountDaoImpl` używająca `SimpleJdbcInsert` do pobrania z tabeli `BANK_ACCOUNT_DETAILS` informacji szczegółowych dotyczących konta bankowego.

Listing 7.7. Klasa `BankAccountDaoImpl` przedstawiająca przykład użycia klasy `SimpleJdbcInsert`

Projekt: `ch07-bankapp-jdbc`

Położenie pliku: `src/main/java/sample/spring/chapter07/bankapp/dao`

```

package sample.spring.chapter07.bankapp.dao;

import javax.sql.DataSource;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
.....
@Repository(value = "bankAccountDao")
public class BankAccountDaoImpl implements BankAccountDao {
    private SimpleJdbcInsert insertBankAccountDetail;

    @Autowired
    private void setDataSource(DataSource dataSource) {
        this.insertBankAccountDetail = new SimpleJdbcInsert(dataSource)
            .withTableName("bank_account_details")
            .usingGeneratedKeyColumns("account_id");
    }

    @Override
    public int createBankAccount(final BankAccountDetails bankAccountDetails) {
        Map<String, Object> parameters = new HashMap<String, Object>(2);
        parameters.put("balance_amount", bankAccountDetails.getBalanceAmount());
        parameters.put("last_transaction_ts", new java.sql.Date(

```

```

        bankAccountDetails.getLastTransactionTimestamp().getTime()));

        Number key = insertBankAccountDetail.executeAndReturnKey(parameters);
        return key.intValue();
    }
    ....
}

```

Ponieważ metoda `setDataSource` jest oznaczona adnotacją `@Autowired`, obiekt `javax.sql.DataSource` jest przekazywany jako argument metody `setDataSource`. W metodzie `setDataSource()` następuje utworzenie egzemplarza `SimpleJdbcInsert` przez przekazanie konstruktorowi `SimpleJdbcInsert` odwołania do obiektu `javax.sql.DataSource`.

Metoda `withTableName()` klasy `SimpleJdbcInsert` określa nazwę tabeli, w której ma zostać wstawiony nowy rekord. Ponieważ informacje szczegółowe o koncie bankowym mają być wstawione do tabeli `BANK_ACCOUNT_DETAILS`, ciąg tekstowy `bank_account_details` jest przekazywany jako argument metodzie `withTableName()`. Metoda `usingGeneratedKeyColumns()` klasy `SimpleJdbcInsert` określa nazwy kolumn tabeli, które zawierają automatycznie generowane klucze. W przypadku tabeli `BANK_ACCOUNT_DETAILS` kolumna `ACCOUNT_ID` zawiera automatycznie generowane klucze. Dlatego ciąg tekstowy `account_id` jest przekazywany metodzie `usingGeneratedKeyColumns()`. Rzeczywista operacja wstawienia jest przeprowadzana przez wywołanie metody `executeAndReturnKey()` klasy `SimpleJdbcInsert`. Metoda `executeAndReturnKey()` akceptuje argument typu `java.util.Map` zawierający nazwy kolumn tabeli oraz wartości dla nich, natomiast wartością zwrótną tej metody jest wygenerowany klucz. Należy zwrócić uwagę, że do rzeczywistego wykonania zapytania SQL `INSERT` klasa `SimpleJdbcInsert` wewnętrznie używa klasy `JdbcTemplate`.

Jeżeli spojrzysz na klasę `BankAccountDaoImpl` w projekcie *ch07-bankapp-jdbc*, dostrzeżesz, że podczas pracy z bazą danych wykorzystuje ona zarówno `SimpleJdbcInsert`, jak i `JdbcTemplate`. Podobnie klasa `FixedDepositDaoImpl` w projekcie *ch07-bankapp-jdbc* wykorzystuje podczas pracy z bazą danych klasy zarówno `JdbcTemplate`, jak i `NamedParameterJdbcTemplate`. To pokazuje możliwość użycia podczas pracy z bazą danych połączenia różnych klas modułu JDBC oferowanego przez Spring.



Ponieważ projekt *ch07-bankapp-jdbc* używa oferowanego przez Spring modułu JDBC oraz (omówionej w podrozdziale 7.5) funkcji zarządzania transakcjami, plik *pom.xml* w projekcie *ch07-bankapp-jdbc* wskazuje zależność od plików JAR *spring-jdbc* i *spring-tx*.

Przechodzimy teraz do klasy `BankApp` projektu *ch07-bankapp-jdbc* odpowiedzialnej za utworzenie konta bankowego i założenie lokaty.

Klasa `BankApp`

Klasa `BankApp` w projekcie *ch07-bankapp-jdbc* pozwala na uruchomienie aplikacji `MyBank` jako samodzielnej aplikacji Javy. Metoda `main()` klasy `BankApp` tworzy w tabeli `BANK_ACCOUNT_DETAILS` rekord dla konta bankowego, a w tabeli `FIXED_DEPOSIT_DETAILS` rekord lokaty (dla nowo utworzonego konta bankowego).

W listingu 7.8 została przedstawiona klasa `BankApp`.

Listing 7.8. *Klasa BankApp*Projekt: *ch07-bankapp-jdbc*Położenie pliku: *src/main/java/sample/spring/chapter07/bankapp*

```

package sample.spring.chapter07.bankapp;
.....
public class BankApp {
    private static Logger logger = Logger.getLogger(BankApp.class);

    public static void main(String args[]) throws Exception {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");

        BankAccountService bankAccountService =
            context.getBean(BankAccountService.class);
        BankAccountDetails bankAccountDetails = new BankAccountDetails();
        .....
        int bankAccountId = bankAccountService.createBankAccount(bankAccountDetails);
        .....
        FixedDepositService fixedDepositService =
            context.getBean(FixedDepositService.class);
        FixedDepositDetails fixedDepositDetails = new FixedDepositDetails();
        .....
        int fixedDepositId =
            fixedDepositService.createFixedDeposit(fixedDepositDetails);
        .....
    }
}

```

W powyższym listingu obiekt `BankAccountService` współpracuje z egzemplarzem `BankAccountDaoImpl` (patrz: listing 7.7) w celu utworzenia konta bankowego, natomiast obiekt `FixedDepositService` współpracuje z obiektem `FixedDepositDaoImpl` (patrz: listingi 7.4 i 7.6) w celu założenia lokaty dla nowo utworzonego konta bankowego. Po wykonaniu metody `main()` klasy `BankApp` zauważysz, że do tabel `BANK_ACCOUNT_DETAILS` i `FIXED_DEPOSIT_DETAILS` został wstawiony nowy rekord.

W tym podrozdziale zobaczyleś, jak oferowany przez Spring moduł JDBC ułatwia uaktualnianie lub pobieranie danych z bazy danych. Moduł JDBC można wykorzystać także do innych celów:

- wykonywanie procedur składowanych i funkcji — do tego celu można użyć klasy `SimpleJdbcCall`;
- wykonywanie grupy zapytań preinterpretowanych;
- uzyskanie dostępu do relacyjnych baz danych w sposób zorientowany obiektowo, na przykład można rozszerzyć klasę `MappingSqlQuery` w celu utworzenia zapytania SQL i mapowania zwróconego obiektu `ResultSet` na obiekt domeny;
- przeprowadzenie konfiguracji egzemplarza osadzonej bazy danych, na przykład oferowanego przez Spring schematu `jdbc` można użyć do utworzenia egzemplarza bazy danych HSQL, H2 lub Derby, a następnie zarejestrować w kontenerze Springa egzemplarz bazy danych jako komponent typu `javax.sql.DataSource`.

Przejdźmy teraz do użycia oferowanej przez Spring obsługi frameworku Hibernate przeznaczonego do pracy z bazami danych.

7.4. Utworzenie aplikacji MyBank z użyciem Hibernate

Oferowany przez Spring moduł ORM zapewnia integrację z Hibernate, JPA (ang. *Java Persistence API*), MyBatis i JDO (ang. *Java Data Objects*). W tym podrozdziale przekonasz się, jak Spring ułatwia użycie frameworku Hibernate do pracy z bazą danych. Ponieważ framework jest dostawcą JPA, użyjemy adnotacji JPA do mapowania klas encji danych na tabele bazy danych.

IMPORT **Rozdział107/ch07-bankapp-hibernate.** (Ten projekt pokazuje aplikację MyBank używającą frameworku Hibernate do pracy z bazą danych. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Najpierw zobaczysz, jak przeprowadzana jest konfiguracja egzemplarza `SessionFactory` w Hibernate.

Konfiguracja `SessionFactory`

`SessionFactory` to fabryka przeznaczona do tworzenia obiektu `Session` frameworku Hibernate. Jest to obiekt `Session` używany przez DAO do przeprowadzania operacji tworzenia, odczytu, usuwania i uaktualniania w encji danych. Oferowana przez Spring `org.springframework.orm.hibernate4.LocalSessionFactoryBean` (implementacja `FactoryBean`) tworzy egzemplarz `SessionFactory`, który może być używany przez klasy DAO do pobrania egzemplarza `Session`.



Jeżeli podczas pracy z bazą danych w klasach DAO aplikacji chcesz użyć udostępnianego przez JPA `EntityManager`, musisz przeprowadzić konfigurację oferowanego przez Spring egzemplarza `LocalContainerEntityManagerFactoryBean` zamiast `org.springframework.orm.hibernate4.LocalSessionFactoryBean`.

W listingu 7.9 została przedstawiona konfiguracja klasy `LocalSessionFactoryBean` w pliku XML kontekstu aplikacji.

Listing 7.9. Plik `applicationContext.xml` przedstawiający konfigurację klasy `LocalSessionFactoryBean`

Projekt: `ch07-bankapp-hibernate`

Położenie pliku: `src/main/java/sample/spring/chapter07/bankapp`

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="packagesToScan" value="sample.spring" />
</bean>
```

Właściwość `dataSource` określa odwołanie do komponentu typu `javax.sql.DataSource`. Właściwość `packagesToScan` wskazuje pakiet lub pakiety, które będą przeszukiwane przez Spring pod kątem klas obsługujących trwale magazyny danych. Na przykład w omawianym listingu określono, że jeśli klasa odpowiedzialna za obsługę przechowywania danych będzie oznaczona oferowaną przez JPA adnotacją `@Entity` i będzie się znajdowała w pakiecie `sample.spring` package (bądź w jego subpakietach), wówczas zostanie automatycznie wykryta przez `org.springframework.orm.hibernate4.LocalSessionFactoryBean`. Alternatywnym rozwiązaniem dla użycia właściwości

`packagesToScan` jest wyraźne określenie za pomocą właściwości `annotatedClasses` wszystkich klas odpowiedzialnych za obsługę trwałego magazynu danych. Rozwiązanie alternatywne przedstawiamy w listingu 7.10.

Listing 7.10. Przykład użycia właściwości `annotatedClasses` klasy `LocalSessionFactoryBean`

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="annotatedClasses">
    <list>
      <value>sample.spring.chapter07.bankapp.domain.BankAccountDetails</value>
      <value>sample.spring.chapter07.bankapp.domain.FixedDepositDetails</value>
    </list>
  </property>
</bean>
```

W powyższym listingu właściwość `annotatedClasses` (typu `java.util.List`) zawiera informacje o wszystkich klasach odpowiedzialnych w aplikacji za obsługę trwałego magazynu danych.



Jeżeli korzystasz z Hibernate 3, to do utworzenia egzemplarza `SessionFactory` musisz użyć `org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean` zamiast `org.springframework.orm.hibernate4.LocalSessionFactoryBean`.

Skoro skonfigurowaliśmy `LocalSessionFactoryBean`, możemy teraz przejść do analizy klas DAO używających egzemplarza `SessionFactory` utworzonego przez `LocalSessionFactoryBean` do wykonywania operacji w bazie danych.

Utworzenie klas DAO używających do pracy z bazą danych API Hibernate

Podczas pracy z bazą danych obiekty DAO muszą uzyskiwać dostęp do oferowanego przez framework Hibernate obiektu `Session`. Aby uzyskać dostęp do obiektu `Session`, należy do obiektu DAO wstrzyknąć egzemplarz `SessionFactory` utworzony przez komponent `LocalSessionFactoryBean` (patrz: listing 7.9). Następnie wstrzyknięty egzemplarz `SessionFactory` można wykorzystać do uzyskania egzemplarza `Session`.

W listingu 7.11 została przedstawiona klasa `FixedDepositDaoImpl` używająca API Hibernate do zapisywania i pobierania encji `FixedDepositDetails`.

Listing 7.11. Klasa `FixedDepositDaoImpl` przedstawiająca użycie API Hibernate

Projekt: `ch07-bankapp-hibernate`

Położenie pliku: `src/main/java/sample/spring/chapter07/bankapp/dao`

```
package sample.spring.chapter07.bankapp.dao;

import org.hibernate.SessionFactory;
.....
@Repository(value = "FixedDepositDao")
public class FixedDepositDaoImpl implements FixedDepositDao {
```



```
@Autowired
private SessionFactory sessionFactory;

public int createFixedDeposit(final FixedDepositDetails fixedDepositDetails) {
    sessionFactory.getCurrentSession().save(fixedDepositDetails);
    return fixedDepositDetails.getFixedDepositId();
}

public FixedDepositDetails getFixedDeposit(final int FixedDepositId) {
    String hql = "from FixedDepositDetails as FixedDepositDetails where "
        + "FixedDepositDetails.FixedDepositId ="
        + FixedDepositId;

    return (FixedDepositDetails) sessionFactory.getCurrentSession()
        .createQuery(hql).uniqueResult();
}
```

W powyższym listingu widać, że egzemplarz `SessionFactory` jest automatycznie wiązany w egzemplarzu `FixedDepositDaoImpl`, który następnie będzie używany przez metody `createFixedDeposit()` i `getFixedDeposit()` do zapisu i pobrania encji `FixedDepositDetails`. Automatyczne wiązanie egzemplarza `SessionFactory` pokazuje, że możesz tę możliwość wykorzystać względem obiektów tworzonych przez implementację `FactoryBean`. Wystarczy po prostu zdefiniować typ tworzony przez `FactoryBean` i oznaczyć go adnotacją `@Autowired` (w podrozdziale 3.9 znajdziesz więcej informacji na temat oferowanego przez Spring interfejsu `FactoryBean`). Metody `createFixedDeposit()` i `getFixedDeposit()` wywołują metodę `getCurrentSession()` klasy `SessionFactory` w celu pobrania egzemplarza `Session`. Trzeba koniecznie zwrócić uwagę na to, że wywołanie metody `getCurrentSession()` zwraca obiekt `Session` powiązany z *bieżącą transakcją* lub *wątkiem*. Użycie metody `getCurrentSession()` okazuje się użyteczne, jeśli Spring ma zarządzać transakcjami, jak ma to miejsce w aplikacji `MyBank`.

Teraz poznasz możliwości Springa w zakresie programowego i deklaracyjnego zarządzania transakcjami.

7.5. Zarządzanie transakcjami za pomocą Springa

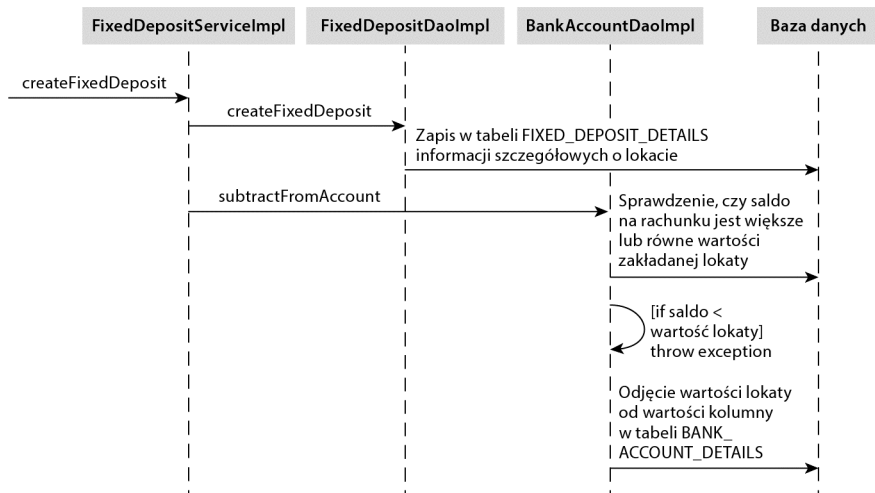
Spring obsługuje zarządzanie transakcjami w sposób zarówno programowy, jak i deklaracyjny. W przypadku programowego zarządzania transakcjami dostarczana przez Spring abstrakcja jest używana do wyraźnego rozpoczynania, kończenia i zatwierdzania transakcji. W przypadku zarządzania deklaracyjnego metody wykonywane w transakcji są oznaczane adnotacją `@Transactional`.

Najpierw poznamy wymagania w zakresie zarządzania transakcjami stawiane przez aplikację `MyBank` przedstawioną w podrozdziale 7.2.

Wymagania aplikacji `MyBank` w zakresie zarządzania transakcjami

W podrozdziale 7.2 wspomnieliśmy, że kiedy klient banku założy nową lokatę, jej wartość zostanie odjęta od wartości przechowywanej w kolumnie `BALANCE_AMOUNT` tabeli `BANK_ACCOUNT_DETAILS`, a następnie informacje szczegółowe o danej lokacie zostaną zapisane w tabeli `FIXED_DEPOSIT_DETAILS`.

Diagram na rysunku 7.2 pokazuje, że metoda `createFixedDeposit()` klasy `FixedDepositServiceImpl` zapisuje w tabeli `FIXED_DEPOSIT_DETAILS` informacje szczegółowe o lokacie oraz odejmuje jej wartość od wartości przechowywanej w odpowiedniej kolumnie rekordu klienta przechowywanego w tabeli `BANK_ACCOUNT_DETAILS`.



Rysunek 7.2. Sekwencja działań wykonywanych przez aplikację *MyBank*, gdy klient zakłada nową lokatę

Na rysunku 7.2 widać, że metoda `createFixedDeposit()` klasy `FixedDepositServiceImpl` wywołuje metodę `createFixedDeposit()` klasy `FixedDepositDaoImpl` i metodę `subtractFromAccount()` klasy `BankAccountDaoImpl`. Zadaniem metody `createFixedDeposit()` klasy `FixedDepositDaoImpl` jest zapisanie w tabeli `FIXED_DEPOSIT_DETAILS` informacji szczegółowych o lokacie. Metoda `subtractFromAccount()` klasy `BankAccountDaoImpl` najpierw sprawdza, czy klient ma wystarczającą ilość środków na założenie nowej lokaty w podanej wysokości. Jeżeli na rachunku klienta znajduje się wystarczająca ilość środków, metoda `subtractFromAccount()` odejmuje od dostępnych środków wartość lokaty. Natomiast w przypadku niewystarczającej ilości środków na koncie klienta metoda `subtractFromAccount()` klasy `BankAccountDaoImpl` powoduje zgłoszenie wyjątku. Jeżeli z jakiegokolwiek powodu działanie metody `createFixedDeposit()` klasy `FixedDepositDaoImpl` lub metody `subtractFromAccount()` klasy `BankAccountDaoImpl` zakończy się niepowodzeniem, system pozostanie w niespójnym stanie. Dlatego też obie metody muszą być wykonane w ramach *transakcji*.

Zobaczmy teraz, jak można użyć Springa do programowego zarządzania transakcjami w aplikacji *MyBank*.

Programowe zarządzanie transakcjami

Istnieje możliwość programowego zarządzania transakcjami za pomocą udostępnianej przez Spring klasy `TransactionTemplate` lub przez użycie oferowanego przez Spring interfejsu `PlatformTransactionManager`. Klasa `TransactionTemplate` ułatwia zarządzanie transakcjami, ponieważ zajmuje się *inicjalizacją* i *zatwierdzaniem* transakcji. Twoim zadaniem jest jedynie dostarczenie implementacji interfejsu `TransactionCallback` zawierającego kod wykonywany w ramach transakcji.

IMPORT **Rozdział07/ch07-bankapp-hibernate.** (Ten projekt pokazuje aplikację MyBank używającą udostępnianej przez Spring klasy `TransactionTemplate` do programowego zarządzania transakcjami. Aby uruchomić tę aplikację wystarczy wywołać metodę `main()` w klasie `BankApp` projektu. Utwórz bazę danych `SPRING_BANK_APP_DB`, a następnie tabele `BANK_ACCOUNT_DETAILS` i `FIXED_DEPOSIT_DETAILS`, tak jak to omówiliśmy dla projektu *ch07-bankapp-jdbc*).

W listingu 7.12 została przedstawiona znajdująca się w pliku XML kontekstu aplikacji konfiguracja klasy `TransactionTemplate`.

Listing 7.12. *Plik `applicationContext.xml` przedstawiający konfigurację `TransactionTemplate`*

Projekt: *ch07-bankapp-tx-jdbc*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"....>
    ....
</bean>

<bean id="txManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<bean id="transactionTemplate"
    class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="txManager"/>
    <property name="isolationLevelName" value="ISOLATION_READ_UNCOMMITTED" />
    <property name="propagationBehaviorName" value="PROPAGATION_REQUIRED" />
</bean>
```

Właściwość `transactionManager` interfejsu `TransactionTemplate` odwołuje się do implementacji `PlatformTransactionManager` odpowiedzialnej za zarządzanie transakcjami.

Właściwość `isolationLevelName` interfejsu `TransactionTemplate` określa poziom izolacji transakcji, jaki będzie przypisany transakcjom zarządzonym przez menedżera transakcji. Wartość właściwości `isolationLevelName` odwołuje się do stałej zdefiniowanej przez interfejs `TransactionDefinition` w Springu. Na przykład `ISOLATION_READ_UNCOMMITTED` to zdefiniowana przez interfejs `TransactionDefinition` stała wskazująca, że *niezatwierdzone* zmiany w transakcji mogą być odczytywane przez inne transakcje.

Właściwość `propagationBehaviorName` interfejsu `TransactionTemplate` określa zachowanie w zakresie propagacji transakcji. Wartość właściwości `propagationBehaviorName` odwołuje się do stałej zdefiniowanej przez interfejs `TransactionDefinition` w Springu. Na przykład `PROPAGATION_REQUIRED` to zdefiniowana przez interfejs `TransactionDefinition` stała wskazująca, że

- jeśli metoda *nie* została wywołana w transakcji, to menedżer transakcji rozpoczyna nową transakcję, a następnie w jej ramach wykonuje daną metodę;
- jeśli metoda została wywołana w transakcji, to menedżer transakcji wykonuje daną metodę w tej *samej* transakcji.

Spring oferuje kilka wbudowanych implementacji `PlatformTransactionManager` gotowych do użycia w zależności od rodzaju stosowanej w aplikacji technologii dostępu do danych. Na przykład implementacja `DataSourceTransactionManager` jest odpowiednia do zarządzania transakcjami w aplikacjach używających JDBC do pracy z bazą danych. Z kolei implementacja `HibernateTransactionManager` będzie właściwym wyborem, gdy do pracy z bazą danych używany jest framework Hibernate. Natomiast implementacja `JpaTransactionManager` to odpowiedni wybór, kiedy dostęp do danych jest obsługiwany za pomocą oferowanego przez JPA menedżera `EntityManager`. W omawianym listingu właściwość `transactionManager` interfejsu `TransactionTemplate` odwołuje się do egzemplarza `DataSourceTransactionManager`, ponieważ aplikacja `MyBank` w projekcie *ch07-bankapp-tx-jdbc* używa JDBC do uzyskania dostępu do danych. Kod przedstawiony w listingu 7.12 pokazuje, że właściwość `dataSource` klasy `DataSourceTransactionManager` odwołuje się do obiektu `javax.sql.DataSource` przedstawiającego bazę danych, której transakcje są zarządzane przez egzemplarz `DataSourceTransactionManager`.

W listingu 7.13 została przedstawiona klasa `FixedDepositServiceImpl` używająca do zarządzania transakcjami interfejsu `TransactionTemplate`.

Listing 7.13. Klasa `FixedDepositServiceImpl` używająca interfejsu `TransactionTemplate`

Projekt: *ch07-bankapp-tx-jdbc*

Położenie pliku: *src/main/java/sample/spring/chapter07/bankapp/service*

```
package sample.spring.chapter07.bankapp.service;

import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;
.....
@Service(value = "FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {

    @Autowired
    private TransactionTemplate transactionTemplate;
    ....
    @Override
    public int createFixedDeposit(final FixedDepositDetails fixedDepositDetails) throws
        Exception {
        transactionTemplate.execute(new TransactionCallback<FixedDepositDetails>() {
            public FixedDepositDetails doInTransaction(TransactionStatus status) {
                try {
                    myFixedDepositDao.createFixedDeposit(fixedDepositDetails);
                    bankAccountDao.subtractFromAccount(
                        fixedDepositDetails.getBankAccountId(),
                        fixedDepositDetails.getFixedDepositAmount()
                    );
                } catch (Exception e) { status.setRollbackOnly(); }
                return fixedDepositDetails;
            }
        });
        return fixedDepositDetails.getFixedDepositId();
    }
    ....
}
```

W powyższym listingu pokazaliśmy metodę `createFixedDeposit()` klasy `FixedDepositServiceImpl` (więcej informacji znajdziesz na rysunku 7.2), która w tabeli `FIXED_DEPOSIT_DETAILS` zapisuje informacje szczegółowe o lokacie, a także w rekordzie klienta w tabeli `BANK_ACCOUNT_DETAILS` odejmuje od odpowiedniej kolumny wartość równą wysokości zakładanej lokaty.

Istnieje możliwość utworzenia implementacji interfejsu `TransactionCallback` do zdefiniowania akcji, które mają być wykonywane w ramach transakcji. W ramach transakcji metoda `execute()` interfejsu `TransactionTemplate` wykonuje akcje zdefiniowane w egzemplarzu `TransactionCallback`. Interfejs `TransactionCallback` definiuje metodę `doInTransaction()`, którą należy zaimplementować w celu wskazania akcji przeznaczonych do wykonania w ramach transakcji. Metoda `doInTransaction()` interfejsu `TransactionCallback` jest w transakcji wywoływana przez metodę `execute()`. Metoda `doInTransaction()` akceptuje obiekt `TransactionStatus` używany do kontroli wyniku transakcji. W omawianym listingu metoda `doInTransaction()` zawiera wywołanie metody `createFixedDeposit()` klasy `FixedDepositDaoImpl` i metody `subtractFromAccount()` klasy `BankAccountDaoImpl`, gdyż chcemy wykonać obie w ramach pojedynczej transakcji. Ponieważ transakcja powinna zostać wycofana w przypadku niepowodzenia wykonania którejkolwiek z obu wymienionych metod, po zgłoszeniu wyjątku następuje wywołanie metody `setRollbackOnly()` klasy `TransactionStatus`. Jeżeli wywołasz metodę `setRollbackOnly()`, egzemplarz `TransactionTemplate` spowoduje wycofanie transakcji. Transakcja zostanie również automatycznie wycofana, jeśli wynikiem wykonania metody `doInTransaction()` będzie `java.lang.RuntimeException`.

Egzemplarz `TransactionCallback` akceptuje argument *typu ogólnego* odwołujący się do typu obiektu zwracanego przez metodę `doInTransaction()`. W listingu 7.13 metoda `doInTransaction()` zwraca obiekt `FixedDepositDetails`. Jeżeli nie chcesz, aby metoda `doInTransaction()` zwracała jakikolwiek obiekt, użyj klasy abstrakcyjnej `TransactionCallbackWithoutResult` implementującej interfejs `TransactionCallback`. Klasa `TransactionCallbackWithoutResult` pozwala na utworzenie implementacji `TransactionCallback`, w których metoda `doInTransaction()` nie zwraca wartości.

W listingu 7.14 została przedstawiona metoda `main()` klasy `BankApp` wywołująca metodę `createBankAccount()` klasy `BankAccountServiceImpl` w celu utworzenia konta i metodę `createFixedDeposit()` klasy `FixedDepositServiceImpl` w celu założenia lokaty dla nowo utworzonego konta.

Listing 7.14. Klasa `BankApp`

Projekt: `ch07-bankapp-tx-jdbc`

Położenie pliku: `src/main/java/sample/spring/chapter07/bankapp`

```
package sample.spring.chapter07.bankapp;

public class BankApp {
    ....
    public static void main(String args[]) throws Exception {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");

        BankAccountService bankAccountService =
            context.getBean(BankAccountService.class);
        FixedDepositService fixedDepositService =
            context.getBean(FixedDepositService.class);
    }
}
```

```
BankAccountDetails bankAccountDetails = new BankAccountDetails();
bankAccountDetails.setBalanceAmount(1000);
.....
int bankAccountId = bankAccountService.createBankAccount(bankAccountDetails);

FixedDepositDetails fixedDepositDetails = new FixedDepositDetails();
fixedDepositDetails.setFixedDepositAmount(1500);
fixedDepositDetails.setBankAccountId(bankAccountId);
.....
int FixedDepositId =
    FixedDepositService.createFixedDeposit(fixedDepositDetails);
.....
}
}
```

W powyższym listingu widać, że najpierw następuje utworzenie konta bankowego ze środkami w wysokości 1000, a następnie podejmowana jest próba założenia nowej lokaty na 1500. Ponieważ wartość lokaty jest większa niż ilość środków na koncie, metoda `subtractFromAccount()` klasy `BankAccountDaoImpl` zgłasza wyjątek (patrz: kod metody `subtractFromAccount()` lub rysunek 7.2).

Po wykonaniu metody `main()` klasy `BankApp` zobaczysz, że w tabeli `FIXED_DEPOSIT_DETAILS` *nie* pojawia się nowy rekord dla lokaty, a wartość 1500 nie zostaje odjęta od odpowiedniej kolumny w tabeli `BANK_ACCOUNT_DETAILS`. To pokazuje, że metody `createFixedDeposit()` klasy `FixedDepositDaoImpl` i `subtractFromAccount()` klasy `BankAccountDaoImpl` są wykonywane w ramach tej samej transakcji.

Zamiast używać klasy `TransactionTemplate` można bezpośrednio wykorzystać implementację `PlatformTransactionManager` do programowego zarządzania transakcjami. Podczas użycia implementacji `PlatformTransactionManager` konieczne jest wyraźne *inicjalizowanie* i *zatwierdzanie* (lub wycofanie) transakcji. Dlatego zaleca się stosowanie klasy `TransactionTemplate` zamiast bezpośredniego użycia implementacji `PlatformTransactionManager`.

Przechodzimy teraz do oferowanej przez Spring funkcji deklaracyjnego zarządzania transakcjami.

Deklaracyjne zarządzanie transakcjami

Programowe zarządzanie transakcjami powoduje powiązanie kodu aplikacji z klasami Springa. Natomiast deklaracyjny sposób zarządzania transakcjami wymaga jedynie zastosowania adnotacji `@Transactional` dla metod lub klas. Jeżeli dana metoda ma zostać wykonana w ramach transakcji, należy ją oznaczyć za pomocą adnotacji `@Transactional`. Gdy *wszystkie* metody klasy mają być wykonywane w transakcji, wtedy całą klasę trzeba oznaczyć adnotacją `@Transactional`.



Zamiast używać adnotacji `@Transactional` w trakcie deklaracyjnego zarządzania transakcjami można elementy oferowanego przez Spring schematu `tx` wykorzystać do identyfikacji metod transakcyjnych. Użycie schematu `tx` skutkuje znacznie większym plikiem XML kontekstu aplikacji. W tym rozdziale do deklaracyjnego zarządzania transakcjami będziemy jedynie używać adnotacji `@Transactional`.

IMPORT `Rozdział07/ch07-bankapp-jdbc` i **IMPORT** `Rozdział07/ch07-bankapp-hibernate`. (W projekcie *ch07-bankapp-jdbc* znajduje się aplikacja MyBank używająca do pracy z bazą danych oferowanego przez Spring modułu JDBC (więcej informacji na temat projektu *ch07-bankapp-jdbc* znajdziesz w podrozdziale 7.3). Z kolei projekt *ch07-bankapp-hibernate* zawiera aplikację MyBank używającą do pracy z bazą danych frameworku Hibernate (więcej informacji na temat projektu *ch07-bankapp-hibernate* znajdziesz w podrozdziale 7.4)).

Włączenie deklaracyjnego sposobu zarządzania transakcjami odbywa się za pomocą elementu `<annotation-driven>` schematu `tx`. W listingu 7.15 przedstawiamy przykład użycia elementu `<annotation-driven>` w projekcie *ch07-bankapp-jdbc*.

Listing 7.15. Plik *applicationContext.xml* przedstawiający przykład użycia elementu `<annotation-driven>`

Projekt: *ch07-bankapp-jdbc*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<beans ..... xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation=".....http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
  .....
  <tx:annotation-driven transaction-manager="txManager" />

  <bean id="txManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>
  .....
</beans>
```

W powyższym listingu znajduje się odwołanie do oferowanego przez Spring schematu `tx`, aby elementy były dostępne w pliku XML kontekstu aplikacji. Element `<annotation-driven>` pozwala na włączenie deklaracyjnego zarządzania transakcjami. Atrybut `transaction-manager` elementu `<annotation-driven>` wskazuje odwołanie do implementacji `PlatformTransactionManager` przeznaczonej do użycia dla zarządzania transakcjami. Omawiany listing pokazuje, że menedżerem transakcji w projekcie *ch07-bankapp-jdbc* jest implementacja `DataSourceTransactionManager`.

W listingu 7.16 pokazujemy, jak można zastosować deklaracyjne zarządzanie transakcjami w projekcie *ch07-bankapp-hibernate*, w którym do pracy z bazą danych używany jest framework Hibernate.

Listing 7.16. Plik *applicationContext.xml* przedstawiający użycie elementu `<annotation-driven>`

Projekt: *ch07-bankapp-hibernate*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<beans ..... xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation=".....http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
  .....
  <tx:annotation-driven transaction-manager="txManager" />

  <bean id="txManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>
  .....
</beans>
```

```

</bean>
.....
</beans>

```

Jeżeli porównasz kod przedstawiony w listingach 7.15 i 7.16, zauważysz, że jedyna różnica dotyczy implementacji PlatformTransactionManager wskazywanej przez atrybut transaction-manager elementu <annotation-driven>. W listingu 7.16 wyraźnie widać, że jeśli do pracy z bazą danych jest używany framework Hibernate, wówczas implementacją PlatformTransactionManager jest org.springframework.orm.hibernate4.HibernateTransactionManager.



Jeżeli korzystasz z Hibernate 3, to atrybutowi transaction-manager musisz przypisać wartość org.springframework.orm.hibernate3.HibernateTransactionManager zamiast org.springframework.orm.hibernate4.HibernateTransactionManager.

W listingu 7.17 pokazujemy, że klasa FixedDepositServiceImpl korzysta z deklaracyjnego zarządzania transakcjami.

Listing 7.17. Klasa FixedDepositServiceImpl, w której użyto adnotacji @Transactional

Projekt: ch07-bankapp-jdbc

Położenie pliku: src/main/java/sample/spring/chapter07/bankapp/service

```

package sample.spring.chapter07.bankapp.service;

import org.springframework.transaction.annotation.Transactional;
.....
@Service(value = "FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {
    .....
    @Transactional
    public int createFixedDeposit(FixedDepositDetails fixedDepositDetails) throws
        Exception {
        bankAccountDao.subtractFromAccount(fixedDepositDetails.getBankAccountId(),
            fixedDepositDetails.getFixedDepositAmount());
        return myFixedDepositDao.createFixedDeposit(fixedDepositDetails);
    }
    .....
}

```

W powyższym listingu metoda createFixedDeposit() została oznaczona adnotacją @Transactional. Oznacza to, że metoda createFixedDeposit() będzie wykonywana w ramach transakcji. Menedżer transakcji określony za pomocą atrybutu transaction-manager elementu <annotation-driven> (patrz: listingi 7.15 i 7.16) będzie używany do zarządzania transakcjami. Jeżeli w trakcie wykonywania metody createFixedDeposit() nastąpi zgłoszenie wyjątku java.lang.RuntimeException, transakcja zostanie automatycznie wycofana.

Adnotacja @Transactional definiuje atrybuty pozwalające na skonfigurowanie menedżera transakcji. Na przykład atrybut rollbackFor umożliwia określenie klas wyjątków, które będą powodowały wycofanie transakcji. Klasy wyjątków podane w atrybucie rollbackFor *muszą* być podklasami klasy java.lang.Throwable. Podobnie za pomocą atrybutu isolation można określić poziom izolacji transakcji.

W przypadku, gdy aplikacja definiuje wiele menedżerów transakcji, atrybutu `value` adnotacji `@Transactional` można użyć do podania nazwy komponentu implementacji `PlatformTransactionManager`, która ma być używana do zarządzania transakcjami. W listingu 7.18 pokazujemy, że w pliku XML kontekstu aplikacji zdefiniowano dwa menedżery przeznaczone do zarządzania transakcjami: `tx1` i `tx2`. Menedżer transakcji `tx1` jest używany przez metodę `methodA()` klasy `SomeServiceImpl`, natomiast menedżer `tx2` jest używany przez metodę `methodB()` klasy `SomeServiceImpl`.

Listing 7.18. Przykład użycia atrybutu `value` adnotacji `@Transactional`

```
----- Klasa SomeServiceImpl -----
@Service
public class SomeServiceImpl implements SomeService {
    ....
    @Transactional(value = "tx1")
    public int methodA() {....}

    @Transactional(value = "tx2")
    public int methodB() {....}
}
----- Plik XML kontekstu aplikacji -----

<tx:annotation-driven />

<bean id="tx1"
      class="org.springframework.orm.hibernate4.HibernateTransactionManager">
  <property name="sessionFactory1" ref="sessionFactory1"/>
</bean>

<bean id="tx2"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

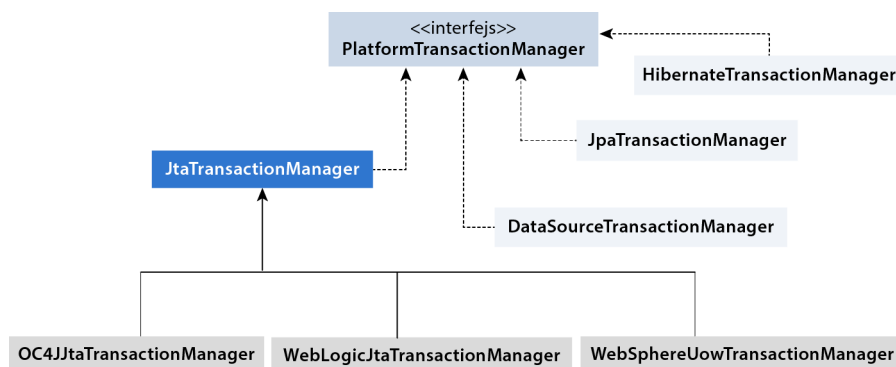
W powyższym listingu element `<annotation-driven>` schematu `tx` nie zawiera atrybutu `transaction-manager`, ponieważ używany menedżer transakcji został wskazany za pomocą adnotacji `@Transactional`. W omawianym listingu atrybut `value` adnotacji `@Transactional` wskazuje menedżer, który ma zarządzać transakcjami. Oznacza to, że metoda `methodA()` klasy `SomeServiceImpl` będzie wykonana przez menedżer `tx1`, natomiast metoda `methodB()` klasy `SomeServiceImpl` przez menedżer `tx2`.

Przechodzimy teraz do oferowanej przez Spring obsługi transakcji JTA (Java Transaction API).

Oferowana przez Spring obsługa JTA

Z rozdziału 1. dowiedziałeś się, że kiedy wiele źródeł transakcyjnych jest zaangażowanych w transakcję, wówczas do zarządzania nią używa się JTA. Spring dostarcza ogólną klasę `JtaTransactionManager` (implementację `PlatformTransactionManager`), której można użyć w aplikacji do zarządzania transakcjami JTA.

W większości środowisk aplikacji serwerowych stawiane przez nie wymagania spełni klasa `JtaTransactionManager`. Jednak Spring dostarcza także implementacje `PlatformTransactionManager` charakterystyczne dla wybranych producentów, które pozwalają na wykorzystanie funkcji dostępnych w poszczególnych serwerach podczas zarządzania transakcjami JTA. Dostarczane przez Spring implementacje charakterystyczne dla producentów to `OC4JTransactionManager` (dla Oracle OC4J), `WebLogicTransactionManager` (dla serwera aplikacji WebLogic), `WebSphereUowTransactionManager` (dla serwera aplikacji WebSphere). Na rysunku 7.3 pokazujemy, jak menedżery transakcji JTA i menedżery transakcji charakterystyczne dla producentów są powiązane z interfejsem `PlatformTransactionManager`. Na rysunku 7.3 widać, że interfejs `PlatformTransactionManager` jest implementowany przez klasy menedżerów transakcji zarówno JTA, jak i charakterystycznych dla producentów.



Rysunek 7.3. Menedżery transakcji JTA i menedżery charakterystyczne dla producentów implementują interfejs `PlatformTransactionManager`

Zobaczymy teraz, jak Spring ułatwia konfigurację menedżera transakcji JTA w pliku XML kontekstu aplikacji.

Konfiguracja menedżera transakcji JTA za pomocą elementu `<jta-transaction-manager>`

Oferowany przez Spring schemat tx zawiera element `<jta-transaction-manager>` automatycznie wykrywający serwer aplikacji, w którym została wdrożona dana aplikacja, a następnie konfiguruje odpowiedni menedżer transakcji JTA. W ten sposób programista unika konieczności wyraźnego konfigurowania w pliku XML kontekstu aplikacji menedżera transakcji JTA charakterystycznego dla użytego serwera. Na przykład jeśli aplikacja została wdrożona w serwerze WebSphere, element `<jta-transaction-manager>` skonfiguruje egzemplarz `WebSphereUowTransactionManager`. Jeżeli ta sama aplikacja zostanie wdrożona w serwerze WebLogic, element `<jta-transaction-manager>` skonfiguruje egzemplarz `WebLogicTransactionManager`. Natomiast w przypadku wdrożenia aplikacji w serwerze innym niż OC4J, WebSphere lub WebLogic element `<jta-transaction-manager>` skonfiguruje egzemplarz `JtaTransactionManager`.

7.6. Podsumowanie

W tym rozdziale zobaczyłeś, jak Spring obsługuje współpracę z bazą danych za pomocą JDBC i frameworku Hibernate ORM. Dowiedziałeś się, jak można wykorzystać Spring do zarządzania transakcjami zarówno programowo, jak i deklaracyjnie. W kolejnym rozdziale przekonasz się, że Spring ułatwia wysyłanie wiadomości e-mail i pracę z pośrednikami komunikacyjnymi oraz umożliwia przezroczyste buforowanie danych.

Rozdział 8. Komunikaty, wiadomości e-mail, asynchroniczne wykonywanie metod i buforowanie w Springu

8.1. Wprowadzenie

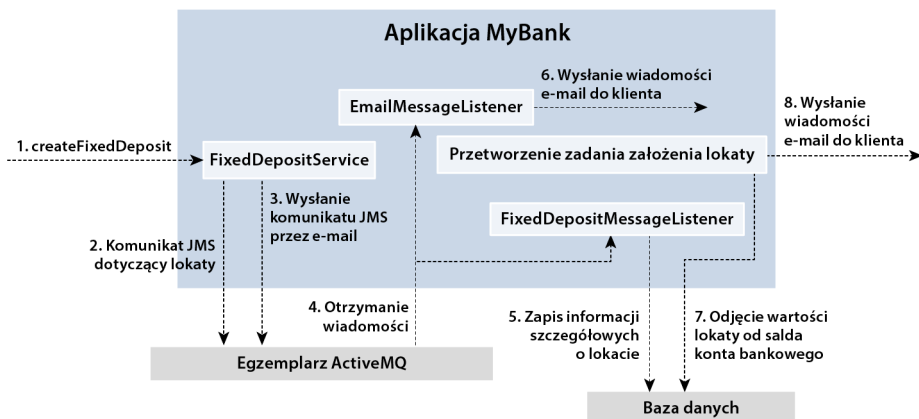
W poprzednim rozdziale zobaczyłeś, jak Spring ułatwia pracę z bazą danych. W kontekście aplikacji MyBank w tym rozdziale pójdziemy o krok dalej i przekonamy się, że Spring ułatwia także:

- wysyłanie i odbieranie komunikatów JMS od dostawcy JMS, takiego jak ActiveMQ;
- wysyłanie wiadomości e-mail;
- asynchroniczne wykonywanie metod;
- przechowywanie i pobieranie danych z bufora.

Najpierw zapoznamy się z wymaganiami aplikacji MyBank, które będziemy implementować w tym rozdziale.

8.2. Wymagania aplikacji MyBank

Aplikacja MyBank pozwala klientom na zakładanie nowych lokat i pobieranie informacji szczegółowych o już istniejących lokatach. Na rysunku 8.1 pokazujemy sekwencję zdarzeń występujących, gdy klient zażąda założenia nowej lokaty.



Rysunek 8.1. Działanie aplikacji MyBank, kiedy klient żąda założenia nowej lokaty

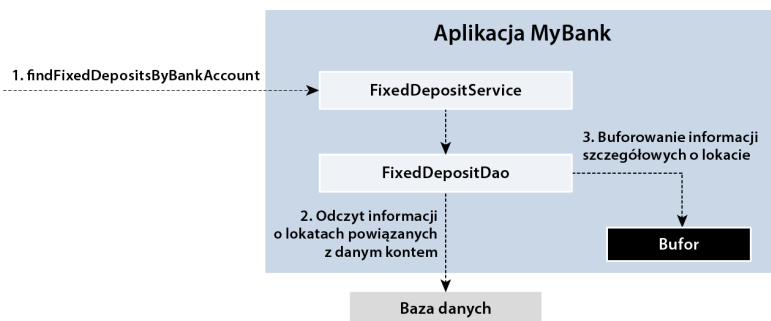
Przede wszystkim metoda `createFixedDeposit()` klasy `FixedDepositService` jest wywoływana w celu wysłania dwóch komunikatów JMS. Pierwszy zawiera informacje o adresie e-mail klienta, natomiast drugi — szczegóły dotyczące zakładanej lokaty. Egzemplarz `EmailMessageListener`

otrzymuje komunikat JMS wraz z adresem e-mail klienta i wysyła mu wiadomość o otrzymaniu żądania założenia nowej lokaty. Egzemplarz `FixedDepositMessageListener` otrzymuje komunikat JMS zawierający szczegóły nowej lokaty i umieszcza te informacje w bazie danych.

Zadanie harmonogramu jest wykonywane co 5 sekund i sprawdza, czy w bazie danych pojawiła się jakaś nowa lokata. Po znalezieniu nowej lokaty następuje odjęcie jej wartości od salda konta klienta oraz wysłanie klientowi wiadomości e-mail z informacją o założeniu nowej lokaty.

Diagram na rysunku 8.2 pokazuje zachowanie aplikacji MyBank podczas wywołania metody `findFixedDepositsByBankAccount()` klasy `FixedDepositService` w celu pobrania wszystkich lokat założonych dla danego klienta.

Rysunek 8.2.
Zachowanie aplikacji MyBank, kiedy klient żąda informacji o wszystkich posiadanych lokatach



Na rysunku 8.2 widać, że po wywołaniu metody `findFixedDepositsByBankAccount()` klasy `FixedDepositService` informacje o lokacie są pobierane z bazy danych, a następnie buforowane w pamięci. Jeżeli ponownie wywołasz metodę `findFixedDepositsByBankAccount()`, niezbędne informacje zostaną odczytane z bufora, a nie z bazy danych.

Zobaczmy teraz, jak używamy Springa w aplikacji MyBank w celu wysyłania komunikatów JMS do odbiorów JMS skonfigurowanych w ActiveMQ.

IMPORT Rozdział 108/ch08-bankapp. (Aby jak najwięcej skorzystać z materiału przedstawionego w tym rozdziale, zainstaluj bazę danych MySQL, a następnie wykonaj skrypt SQL o nazwie `spring_bank_app_db.sql`, który znajduje się w katalogu `sql` projektu `ch08-bankapp`. Wykonanie skryptu `spring_bank_app_db.sql` spowoduje utworzenie bazy danych `SPRING_BANK_APP_DB` wraz z tabelami `BANK_ACCOUNT_DETAILS` i `FIXED_DEPOSIT_DETAILS`. Odpowiednio zmodyfikuj plik `src/main/resources/META-INF/spring/database.properties` i wskaż w nim swoją instalację MySQL. Uzyskanie działającej funkcji wysyłania wiadomości e-mail wymaga zmodyfikowania pliku `src/main/resources/META-INF/spring/email.properties` i podania w nim informacji o serwerze i koncie poczty, które będą używane do wysyłania wiadomości e-mail. Zmodyfikuj klasę `BankApp`, aby zawierała adres e-mail klienta, do którego będą wysyłane wiadomości e-mail).

8.3. Wysyłanie komunikatów JMS

Spring ułatwia współpracę z dostawcami JMS za pomocą oferowanej warstwy abstrakcji zbudowanej na bazie API JMS. W kontekście aplikacji MyBank w tym rozdziale zobaczysz, jak za pomocą Springa *synchronicznie* i *asynchronicznie* wysyłać i otrzymywać wiadomości z brokera ActiveMQ.

Dla zachowania prostoty broker ActiveMQ został w projekcie *ch08-bankapp* skonfigurowany do działania w trybie *osadzonym*.



W Springu klasy odpowiedzialne za obsługę JMS są zdefiniowane w pliku JAR o nazwie *spring-jms*. Dlatego w aplikacji musisz zdefiniować zależność od pliku JAR *spring-jms*, aby móc używać oferowanej przez Spring obsługi JMS.

Konfiguracja brokera ActiveMQ do działania w trybie osadzonym

Osadzony broker ActiveMQ działa w *tej samej* wirtualnej maszynie Javy, w której została uruchomiona aplikacja. Schemat XML ActiveMQ można wykorzystać do konfiguracji osadzonego ActiveMQ w aplikacji Springa. W listingu 8.1 pokazujemy, jak schemat XML ActiveMQ jest używany do konfiguracji osadzonego brokera ActiveMQ w aplikacji MyBank.

Listing 8.1. Plik *applicationContext.xml* przedstawiający konfigurację osadzonego brokera ActiveMQ

Projekt: *ch08-bankapp*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<beans .....  
  xmlns:amq="http://activemq.apache.org/schema/core"  
  xsi:schemaLocation=".....http://activemq.apache.org/schema/core  
    http://activemq.apache.org/schema/core/activemq-core-5.7.0.xsd.....">  
  
  <amq:broker>  
    <amq:transportConnectors>  
      <amq:transportConnector uri="tcp://localhost:61616" />  
    </amq:transportConnectors>  
  </amq:broker>  
  .....  
</beans>
```

W powyższym listingu przestrzeń nazw *amq* odwołuje się do schematu XML ActiveMQ (*activemq-core-5.7.0.xsd*) pozwalającego na konfigurację osadzonego brokera ActiveMQ. Element *<broker>* konfiguruje osadzony broker ActiveMQ o nazwie *localhost*. Z kolei element *<transportConnectors>* pozwala na określenie połączeń, jakie będą mogły wykonywać klienci brokera ActiveMQ. W omawianym listingu element *<transportConnector>* zawiera element *<transportConnectors>* wskazujący, że klienci mogą nawiązywać połączenie z osadzonym brokerem ActiveMQ na porcie 61616, używając gniazda TCP.

Zobaczmy teraz, jak skonfigurować JMS *ConnectionFactory* w celu tworzenia połączeń do egzemplarza osadzonego ActiveMQ.

Konfiguracja JMS *ConnectionFactory*

W listingu 8.2 przedstawiamy sposób konfiguracji JMS *ConnectionFactory* w pliku XML kontekstu aplikacji.

Listing 8.2. Plik *applicationContext.xml* przedstawiający konfigurację *JMS ConnectionFactory*Projekt: *ch08-bankapp*Położenie pliku: *src/main/resources/META-INF/spring*

```

<beans .....
    xmlns:amq="http://activemq.apache.org/schema/core"
    xsi:schemaLocation=".....http://activemq.apache.org/schema/core
        http://activemq.apache.org/schema/core/activemq-core-5.7.0.xsd.....">
    .....
    <amq:connectionFactory brokerURL="vm://localhost" id="jmsFactory" />

    <bean class="org.springframework.jms.connection.CachingConnectionFactory"
        id="cachingConnectionFactory">
        <property name="targetConnectionFactory" ref="jmsFactory" />
    </bean>
    .....
</beans>

```

W powyższym listingu element `<connectionFactory>` schematu `amq` tworzy egzemplarz *JMS ConnectionFactory* używany do nawiązywania połączeń z egzemplarzem osadzonego *ActiveMQ* (patrz: listing 8.1). Atrybut `brokerUrl` określa adres URL służący do nawiązywania połączeń z brokerem *ActiveMQ*. Ponieważ używany jest osadzony broker *ActiveMQ*, atrybut `brokerUrl` określa, że podczas nawiązywania połączenia z egzemplarzem brokera *ActiveMQ* stosowany jest protokół *VM* (oznaczony przez `vm://`).

Udostępniana przez Spring klasa *CachingConnectionFactory* stanowi rodzaj adaptera dla egzemplarza *JMS ConnectionFactory* (określonego za pomocą właściwości `targetConnectionFactory`), który zapewnia dodatkowe funkcje buforowania egzemplarzy *JMS Session*, *MessageProducer* i *MessageConsumer*.

Zobaczmy teraz, jak można używać oferowanej przez Spring klasy *JmsTemplate* do wysyłania komunikatów *JMS*.

Wysyłanie komunikatów *JMS* za pomocą klasy *JmsTemplate*

Oferowana przez Spring klasa *JmsTemplate* ułatwia *synchroniczne* wysyłanie i otrzymywanie komunikatów *JMS*. Na potrzeby tego rozdziału skoncentrujemy się jedynie na sposobie wysyłania komunikatów *JMS* za pomocą klasy *JmsTemplate*. Podobnie jak w przypadku klas *TransactionTemplate* (patrz: podrozdział 7.5) i *JdbcTemplate* (patrz: podrozdział 7.3), także klasa *JmsTemplate* zapewnia warstwę aplikacji, dzięki której nie trzeba bezpośrednio pracować z działającym na niskim poziomie API *JMS*.

W listingu 8.3 pokazujemy, jak klasa *JmsTemplate* została skonfigurowana w pliku XML kontekstu aplikacji *MyBank* w celu wysyłania komunikatów do egzemplarza osadzonego *ActiveMQ*.

Listing 8.3. Plik *applicationContext.xml* przedstawiający konfigurację klasy *JmsTemplate*Projekt: *ch08-bankapp*Położenie pliku: *src/main/resources/META-INF/spring*

```

<beans .....
    xmlns:amq="http://activemq.apache.org/schema/core"
    xsi:schemaLocation=".....http://activemq.apache.org/schema/core

```



```

http://activemq.apache.org/schema/core/activemq-core-5.7.0.xsd.....">
.....
<bean class="org.springframework.jms.core.JmsTemplate" id="jmsTemplate">
  <property name="connectionFactory" ref="cachingConnectionFactory" />
  <property name="defaultDestination" ref="FixedDepositDestination" />
</bean>

<amq:queue id="FixedDepositDestination" physicalName="aQueueDestination" />
<amq:queue id="emailQueueDestination" physicalName="emailQueueDestination" />
.....
</beans>

```

Właściwość `connectionFactory` klasy `JmsTemplate` określa egzemplarz JMS `ConnectionFactory` używany do nawiązywania połączenia z dostawcą JMS. Właściwość `defaultDestination` klasy `JmsTemplate` odwołuje się do domyślnego odbiorcy JMS, do którego klasa `JmsTemplate` będzie wysyłała komunikaty JMS. W omawianym listingu właściwość `connectionFactory` odwołuje się do egzemplarza `CachingConnectionFactory` (patrz: listing 8.2), natomiast właściwość `defaultDestination` odwołuje się do kolejki odbiorców JMS utworzonej za pomocą elementu `<queue>` schematu `amq`.

Element `<queue>` schematu `amq` tworzy w ActiveMQ kolejkę odbiorców komunikatów JMS. W listingu 8.3 pierwszy element `<queue>` tworzy kolejkę o nazwie `aQueueDestination`, natomiast drugi element `<queue>` tworzy kolejkę o nazwie `emailQueueDestination`. Atrybut `physicalName` odwołuje się do nazwy, pod jaką kolejka odbiorców komunikatów JMS została utworzona w ActiveMQ. Z kolei atrybut `id` odwołuje się do nazwy, pod jaką kolejka odbiorców komunikatów JMS będzie dostępna w innych komponentach kontenera Springa. W przypadku kodu przedstawionego w listingu 8.3 właściwość `defaultDestination` klasy `JmsTemplate` odwołuje się do atrybutu `id` elementu `<queue>` tworzącego kolejkę `aQueueDestination`. Dlatego `aQueueDestination` jest domyślnym odbiorcą, do którego egzemplarz `JmsTemplate` będzie wysyłał komunikaty JMS.

Egzemplarz `JMS Session` używany przez klasę `JmsTemplate` ma ustawiony tryb *auto-acknowledge* i z natury *nie jest transakcyjny*. Jeżeli chcesz, aby klasa `JmsTemplate` używała *transakcyjnej* wersji egzemplarza `Sessions`, konieczne jest przypisanie wartości `true` właściwości `transacted` klasy `JmsTemplate`. W przypadku transakcyjnej wersji obiektu `Sessions` nowa transakcja rozpoczyna się wraz z utworzeniem egzemplarza `Session` przez aplikację lub po zatwierdzeniu bądź wycofaniu transakcji. Oznacza to, że transakcyjna wersja egzemplarza `JMS Session` *zawsze* jest powiązana z transakcją. Transakcyjnej wersji egzemplarza `JMS Session` można używać do wysyłania i obie-rania komunikatów JMS w transakcji. Jeżeli klasy `JmsTemplate` używasz wraz z oferowanym przez Spring egzemplarzem `JmsTransactionManager`, wówczas egzemplarz `JmsTemplate` *zawsze* pobierze transakcyjną wersję `JMS Session`.

Zobaczmy teraz, jak przedstawia się konfiguracja `JmsTransactionManager` i jak za pomocą klasy `JmsTemplate` komunikaty JMS są wysyłane w ramach transakcji.

Wysyłanie komunikatów JMS w transakcji

W rozdziale 7. zobaczyłeś, że Spring dostarcza kilka implementacji `PlatformTransactionManager` zapewniających zarządzanie transakcjami w sposób charakterystyczny dla zasobu. W aplikacjach JMS możesz użyć klasy `JmsTransactionManager` (jeszcze inna implementacja `PlatformTransactionManager`) do zarządzania transakcjami dla pojedynczego egzemplarza `JMS ConnectionFactory`. Ponieważ

JmsTransactionManager implementuje interfejs PlatformTransactionManager, można wykorzystać TransactionTemplate do programowego zarządzania transakcjami JMS lub też polegać na użyciu anotacji @Transactional i tym samym deklaracyjnie zarządzać transakcjami JMS.

W listingu 8.4 przedstawiamy konfigurację klasy JmsTransactionManager w pliku XML kontekstu aplikacji.

Listing 8.4. Plik *applicationContext.xml* przedstawiający konfigurację klasy *JmsTransactionManager*

Projekt: *ch08-bankapp*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<bean id="jmsTxManager"
      class="org.springframework.jms.connection.JmsTransactionManager">
    <property name="connectionFactory" ref="cachingConnectionFactory" />
</bean>
```

Właściwość *connectionFactory* klasy *JmsTransactionManager* określa odwołanie do egzemplarza JMS *ConnectionFactory*, dla którego *JmsTransactionManager* zarządza transakcjami. W powyższym listingu wartością właściwości *connectionFactory* jest odwołanie do komponentu *CachingConnectionFactory* ↗ Factory (patrz: listing 8.2). Ponieważ *CachingConnectionFactory* buforuje egzemplarz JMS *Sessions*, użycie *CachingConnectionFactory* wraz z *JmsTransactionManager* pozwala na zmniejszenie poziomu zużycia zasobów.

Jeżeli zachodzi potrzeba programowego zarządzania transakcjami JMS za pomocą klasy *TransactionTemplate*, wówczas w pliku XML kontekstu aplikacji należy skonfigurować klasę *TransactionTemplate*. Jeżeli chcesz użyć deklaracyjnego zarządzania transakcjami, skorzystaj z elementu *<annotation-driven>* schematu *tx* oferowanego przez Spring.

W listingu 8.5 została przedstawiona klasa *FixedDepositServiceImpl*, która używa klasy *JmsTemplate* w celu wysyłania komunikatów do osadzonego brokera ActiveMQ.

Listing 8.5. Klasa *FixedDepositServiceImpl* przedstawiająca przykład wysyłania komunikatów JMS przy użyciu *JmsTemplate*

Projekt: *ch08-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter08/bankapp/service*

```
package sample.spring.chapter08.bankapp.service;

import javax.jms.*;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

@Service(value = "FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {
    @Autowired
    private JmsTemplate jmsTemplate;
    ....
    @Override
    @Transactional("jmsTxManager")
    public void createFixedDeposit(final FixedDepositDetails fixedDepositDetails) throws
        Exception {
```

```

jmsTemplate.send("emailQueueDestination", new MessageCreator() {
    public Message createMessage(Session session) throws JMSException {
        TextMessage textMessage = session.createTextMessage();
        textMessage.setText(fixedDepositDetails.getEmail());
        return textMessage;
    }
});
// Ten komunikat JMS jest wysyłany do domyślnego odbiorcy skonfigurowanego dla klasy JmsTemplate.
jmsTemplate.send(new MessageCreator() {
    public Message createMessage(Session session) throws JMSException {
        ObjectMessage objectMessage = session.createObjectMessage();
        objectMessage.setObject(fixedDepositDetails);
        return objectMessage;
    }
});
}
.....
}

```

W powyższym listingu widać, że metoda `send()` klasy `JmsTemplate` jest używana w celu wysłania komunikatów do odbiorców zdefiniowanych przez kolejki `emailQueueDestination` i `aQueueDestination`. W listingu 8.3 możesz zobaczyć, w jaki sposób zdefiniowano odbiorców w pliku XML kontekstu aplikacji. Nazwa odbiorcy przekazywana metodzie `send()` klasy `JmsTemplate` jest zamieniana na rzeczywisty obiekt JMS Destination. Zamiana odbywa się za pomocą oferowanego przez Spring egzemplarza `DynamicDestinationResolver` (to implementacja dostarczanego przez Spring interfejsu `DestinationResolver`). Jeżeli w pliku XML kontekstu aplikacji odbiorców zdefiniowałeś za pomocą elementu `<queue>` (lub `<topic>`) schematu `amq`, wówczas nazwa odbiorcy przekazywana metodzie `send()` klasy `JmsTemplate` jest wartością atrybutu `id` elementu `<queue>` (lub `<topic>`) odpowiadającego odbiorcy, do którego mają być wysyłane komunikaty JMS.

W listingu 8.5 metoda `createFixedDeposit()` klasy `FixedDepositServiceImpl` została oznaczona adnotacją `@Transactional("jmsTxManager")`, która powoduje, że metoda `createFixedDeposit()` będzie wykonywana w ramach transakcji, a zarządzaniem transakcjami zajmuje się menedżer `jmsTxManager` (w listingu 8.4 możesz zobaczyć, jak skonfigurowano menedżer `jmsTxManager`). Metoda `send()` klasy `JmsTemplate` akceptuje nazwę odbiorcy komunikatów JMS oraz egzemplarz `MessageCreator`. W przypadku braku podanego odbiorcy metoda `send()` będzie wysyłać komunikaty JMS do odbiorcy *domyślnego*, który dla klasy `JmsTemplate` został skonfigurowany za pomocą właściwości `defaultDestination` (patrz: listing 8.3).

W metodzie `createMessage()` klasy `MessageCreator` tworzony jest komunikat JMS przeznaczony do wysłania. Nie trzeba bezpośrednio zajmować się obsługą wyjątków zgłaszanych przez API JMS, ponieważ to jest zadanie klasy `JmsTemplate`. W listingu 8.5 widać, że jeśli używasz klasy `JmsTemplate`, nie musisz bezpośrednio pobierać egzemplarza `Connection` z `ConnectionFactory`, tworzyć egzemplarza `Session` na podstawie `Connection` itd. w celu wysyłania komunikatów JMS. Dlatego też klasa `JmsTemplate` pozwala uniknąć konieczności pracy z działającym na niskim poziomie API JMS.

W listingu 8.5 egzemplarze `TextMessage` i `ObjectMessage` przedstawiają komunikaty JMS. Obie klasy (`TextMessage` i `ObjectMessage`) implementują interfejs `javax.jms.Message`. W aplikacji `MyBank` egzemplarz `TextMessage` jest używany do wysyłania adresu e-mail (prosta wartość w postaci ciągu

tekstowego) klienta, który zażądał założenia lokaty. Natomiast egzemplarz `ObjectMessage` jest używany w celu wysłania obiektu `FixedDepositDetails` (obiekt klasy `Serializable`) zawierającego informacje szczegółowe o lokacie. Ponieważ metoda `createFixedDeposit()` klasy `FixedDepositServiceImpl` jest wykonywana w ramach transakcji JMS, egzemplarzowi `ActiveMQ` wysyłane są oba wymienione komunikaty lub żaden.

Zamiast używać adnotacji `@Transactional` można w sposób programowy zarządzać transakcjami JMS za pomocą klasy `TransactionTemplate` (patrz: podrozdział 7.5). W listingu 8.6 przedstawiamy sposób konfiguracji klasy `TransactionTemplate` w celu użycia `JmsTransactionManager` do zarządzania transakcjami.

Listing 8.6. Konfiguracja klasy `TransactionTemplate`

```
<bean id="jmsTxManager"
    class="org.springframework.jms.connection.JmsTransactionManager">
    <property name="connectionFactory" ref="cachingConnectionFactory" />
</bean>

<bean id="transactionTemplate"
    class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="jmsTxManager" />
</bean>
```

W powyższym listingu właściwość `transactionManager` klasy `TransactionTemplate` odwołuje się do komponentu `JmsTransactionManager`.

Po przeprowadzeniu konfiguracji klasy `TransactionTemplate` można ją wykorzystać do zarządzania transakcjami JMS. W listingu 8.7 przedstawiamy odmianę metody `createFixedDeposit()` klasy `FixedDepositServiceImpl`, która do zarządzania transakcjami JMS używa klasy `TransactionTemplate`.

Listing 8.7. Programowe zarządzanie transakcjami JMS przy użyciu klasy `TransactionTemplate`

```
package sample.spring.chapter08.bankapp.service;

import javax.jms.*;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

@Service(value = "FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {
    @Autowired
    private JmsTemplate jmsTemplate;

    @Autowired
    private TransactionTemplate transactionTemplate;
    ....
    public void createFixedDeposit(final FixedDepositDetails fixedDepositDetails) throws
        Exception {

        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                jmsTemplate.send("emailQueueDestination", new MessageCreator()
                { ..... });
                jmsTemplate.send(new MessageCreator() { ..... });
            }
        });
    }
}
```

```

    });
}
.....
}

```

W powyższym listingu widać, że komunikaty JMS są wysyłane przez metodę `doInTransaction()` klasy `TransactionCallbackWithoutResult`, a więc w ramach tej samej transakcji JMS. Takie rozwiązanie jest podobne do programowego zarządzania transakcjami JDBC (patrz: podrozdział 7.5) za pomocą klasy `TransactionTemplate`.

Dotychczas przedstawiliśmy przykłady, w których klasa `JmsTemplate` była używana w celu wysyłania komunikatów do *prekonfigurowanych* odbiorców komunikatów JMS. Zobaczmy teraz, jak można skonfigurować klasę `JmsTemplate`, gdy aplikacja używa dynamicznych odbiorców komunikatów JMS.

Dynamiczni odbiorcy komunikatów JMS i konfiguracja klasy `JmsTemplate`

Jeżeli aplikacja używa dynamicznych odbiorców komunikatów JMS (to znaczy ustalanych w trakcie działania aplikacji), wówczas typ odbiorcy komunikatów JMS musi być określony za pomocą właściwości `pubSubDomain` klasy `JmsTemplate`. Właściwość `pubSubDomain` jest przez aplikację używana do ustalenia typu odbiorcy, do którego klasa `JmsTemplate` będzie wysyłała komunikaty JMS. W przypadku braku określonej właściwości `pubSubDomain` przyjmuje się założenie, że typem jest domyślna kolejka odbiorców komunikatów JMS.

W listingu 8.8 została przedstawiona klasa `JmsTemplate`, która wysyła komunikaty do dynamicznie ustalonego odbiorcy.

Listing 8.8. Użycie klasy `JmsTemplate` w celu wysyłania komunikatów do dynamicznie ustalanych odbiorców

```

----- applicationContext.xml -----

<bean class="org.springframework.jms.core.JmsTemplate" id="jmsTemplate">
  <property name="connectionFactory" ref="cachingConnectionFactory" />
  <property name="defaultDestination" ref="FixedDepositDestination" />
  <property name="pubSubDomain" value="true" />
</bean>

----- Dynamiczne ustalanie odbiorcy -----

jmsTemplate.send("dynamicTopic", new MessageCreator() {
    public Message createMessage(Session session) throws JMSException {
        session.createTopic("dynamicTopic");
        ObjectMessage objectMessage = session.createObjectMessage();
        objectMessage.setObject(someObject);
        return objectMessage;
    }
});

```

W powyższym listingu właściwość `pubSubDomain()` klasy `JmsTemplate` ma przypisaną wartość `true`, co oznacza, że jeśli dojdzie do użycia dynamicznych odbiorców komunikatów JMS, wówczas

Spring określa nazwę odbiorcy jako *topic*. Zwróć uwagę na fakt, że nazwa odbiorcy przekazywana metodzie `send()` klasy `JmsTemplate` to `dynamicTopic`, a ta sama nazwa jest używana również przez metodę `createMessage()` klasy `MessageCreator`. Ponieważ w pliku XML kontekstu aplikacji nie zdefiniowano odbiorcy `dynamicTopic`, Spring nie wie, czy `dynamicTopic` oznacza kolejkę czy odbiorcę. Wartością właściwości `pubSubDomain` klasy `JmsTemplate` jest `true`, więc egzemplarz `DynamicDestinationResolver` uznaje `dynamicTopic` jako odbiorcę dynamicznie tworzonego przez metodę `createMessage()` klasy `MessageCreator` w trakcie działania aplikacji. Jeżeli *nie* zostanie zdefiniowana właściwość `pubSubDomain` klasy `JmsTemplate`, wtedy egzemplarz `DynamicDestinationResolver` spróbuje uznać `dynamicTopic` jako kolejkę.

Zobaczmy teraz, jak klasa `JmsTemplate` ułatwia wysyłanie obiektów Javy jako komunikatów JMS.

Klasa `JmsTemplate` i konwersja komunikatu

Klasa `JmsTemplate` definiuje wiele metod `convertAndSend()` przeznaczonych do konwersji i wysyłania obiektów Javy jako komunikatów JMS. Domyślnie klasa `JmsTemplate` jest skonfigurowana wraz z egzemplarzem `SimpleMessageConverter` (implementacja interfejsu `MessageConverter` oferowana przez Spring), który konwertuje obiekty Javy na komunikaty JMS i na odwrót.

Interfejs `MessageConverter` definiuje następujące metody:

- `Object toMessage(Object object, Session session)` — konwertuje obiekt Javy (przedstawiany za pomocą argumentu `object`) na egzemplarz JMS `Message` za pomocą dostarczonego egzemplarza JMS `Session` (przedstawianego przez argument `session`).
- `Object fromMessage(Message message)` — przeprowadza konwersję argumentu `Message` na obiekt Javy.

Oferowana przez Spring klasa `SimpleMessageConverter` zapewnia możliwość przeprowadzania konwersji między typami `String` i `TextMessage` (JMS), `byte[]` i `BytesMessage` (JMS), `Map` i `MapMessage` (JMS) oraz obiektami `Serializable` i `ObjectMessage` (JMS). Jeżeli chcesz zmodyfikować obiekt JMS `Message` utworzony przez metodę `convertAndSend()` klasy `JmsTemplate`, wtedy do wprowadzenia modyfikacji możesz wykorzystać implementację `MessagePostProcessor`.

W listingu 8.9 przedstawiamy sytuację, w której implementacja `MessagePostProcessor` jest używana do modyfikacji komunikatu JMS utworzonego przez metodę `convertAndSend()` klasy `JmsTemplate`.

Listing 8.9. Przykład użycia metody `convertAndSend()` klasy `JmsTemplate`

```
jmsTemplate.convertAndSend("aDestination", "Witaj, świecie!",
    new MessagePostProcessor() {
        public Message postProcessMessage(Message message) throws JMSException {
            message.setBooleanProperty("printOnConsole", true);
            return message;
        }
    });
```

W powyższym listingu ciąg tekstowy `Witaj, świecie!` jest przekazywany metodzie `convertAndSend()`. Metoda ta tworzy egzemplarz JMS `TextMessage`, a następnie udostępnia go implementacji `MessagePostProcessor` w celu przeprowadzenia jakichkolwiek zmian w komunikacie, zanim zostanie

on wysłany. W omawianym listingu metoda `postProcessMessage()` klasy `MessagePostProcessor` ustawia wartość właściwości `printlnOnConsole` w komunikacie JMS przed jego wysłaniem do odbiorcy `aDestination`.

Do tej pory dowiedziałeś się, w jaki sposób można za pomocą klasy `JmsTemplate` wysyłać komunikaty JMS do odbiorców. Przekonajmy się teraz, jak odbierać komunikaty JMS za pomocą klasy `JmsTemplate`, a także oferowanych przez Spring kontenerów odbiorców komunikatów.

8.4. Odbieranie komunikatów JMS

Komunikaty JMS można odbierać *synchronicznie* za pomocą klasy `JmsTemplate` oraz *asynchronicznie* za pomocą oferowanych przez Spring kontenerów odbiorców komunikatów.

Synchroniczne odbieranie komunikatów JMS za pomocą klasy `JmsTemplate`

Klasa `JmsTemplate` definiuje wiele metod `receive()`, które można wykorzystać do *synchronicznego* odbierania komunikatów JMS. Trzeba koniecznie pamiętać, że wywołanie metody `receive()` klasy `JmsTemplate` powoduje zablokowanie danego wątku aż do chwili, gdy komunikat JMS zostanie odebrany. Aby zagwarantować, że wątek nie będzie zablokowany w nieskończoność, konieczne jest przypisanie odpowiedniej wartości właściwości `receiveTimeout` klasy `JmsTemplate`. Właściwość `receiveTimeout` określa wyrażony w milisekundach czas, po którego upływie zablokowany wątek powinien zostać zwolniony.

Klasa `JmsTemplate` definiuje również wiele metod `receiveAndConvert()` przeznaczonych do przeprowadzania automatycznej konwersji otrzymanej wiadomości JMS na obiekt Javy. Do przeprowadzania konwersji domyślnie klasa `JmsTemplate` używa implementacji `SimpleMessageConverter`.

Asynchroniczne odbieranie komunikatów JMS za pomocą kontenerów odbiorców komunikatów

Oferowane przez Spring kontenery odbiorców komunikatów można wykorzystać do *asynchronicznego* odbierania komunikatów JMS. Kontenery te zajmują się obsługą transakcji i aspektami związanymi z zarządzaniem zasobami. W ten sposób programista może skoncentrować się jedynie na utworzeniu logiki przetwarzania komunikatu.

Kontener odbiorców komunikatów otrzymuje komunikaty JMS, a następnie przekazuje je do implementacji `MessageListener` do dalszego przetworzenia. W listingu 8.10 przedstawiamy sposób konfiguracji kontenera odbiorców komunikatów za pomocą elementu `<listener-container>` schematu `jms` oferowanego przez Spring.

Listing 8.10. Plik `applicationContext.xml` przedstawiający konfigurację kontenera odbiorców komunikatów

Projekt: `ch08-bankapp`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans xmlns:jms="http://www.springframework.org/schema/jms"
      xsi:schemaLocation="http://www.springframework.org/schema/jms
```

```

http://www.springframework.org/schema/jms/spring-jms-4.0.xsd">
.....
<jms:listener-container connection-factory="cachingConnectionFactory"
    destination-type="queue" transaction-manager="jmsTxManager">

    <jms:listener destination="aQueueDestination" ref="FixedDepositMessageListener" />
    <jms:listener destination="emailQueueDestination" ref="emailMessageListener" />
</jms:listener-container>

<bean class="sample.spring.chapter08.bankapp.jms.EmailMessageListener"
    id="emailMessageListener" />

<bean class="sample.spring.chapter08.bankapp.jms.FixedDepositMessageListener"
    id="FixedDepositMessageListener" />
.....
</beans>

```

W powyższym listingu kod zawiera odwołanie do schematu jms, aby jego elementy były dostępne w pliku XML kontekstu aplikacji. Element `<listener-container>` jest przeznaczony do konfiguracji kontenera odbiorców komunikatów dla poszczególnych implementacji `MessageListener` zdefiniowanych przez elementy `<listener>`. Atrybut `connection-factory` określa odwołanie do komponentu JMS `ConnectionFactory`, który będzie używał kontenera do nawiązania połączenia z dostawcą JMS. Ponieważ w aplikacji `MyBank` używamy klasy `CachingConnectionFactory`, atrybut `connection-factory` odwołuje się do komponentu `cachingConnectionFactory` zdefiniowanego w pliku XML kontekstu aplikacji (patrz: listing 8.2). Atrybut `destination-type` określa typ odbiorcy, z którym powiązany jest kontener. Możliwe wartości atrybutu `destination-type` to: `queue`, `topic` i `durableTopic`.

Atrybut `transaction-manager` elementu `<listener-container>` wskazuje implementację Platform `TransactionManager` gwarantującą, że odbieranie i przetwarzanie przez `MessageListener` komunikatów JMS będzie zachodziło w ramach transakcji. W omawianym listingu wartość atrybutu `transaction-manager` odwołuje się do implementacji `JmsTransactionManager` (patrz: listing 8.4) skonfigurowanej dla aplikacji `MyBank`. Jeżeli implementacja `MessageListener` ma działać także z innymi zasobami transakcyjnymi, wówczas zamiast `JmsTransactionManager` rozważ użycie oferowanej przez Spring implementacji `JtaTransactionManager`. W samodzielnej aplikacji można wykorzystać osadzone menedżery transakcji, takie jak `Atomikos` (<http://www.atomikos.com/>), i za ich pomocą obsługiwać transakcje JTA w aplikacji.



Domyślnie element `<listener-container>` tworzy egzemplarz klasy Springa o nazwie `DefaultMessageListenerContainer` odpowiadającej poszczególnym implementacjom JMS `MessageListener` wskazanym przez elementy `<listener>`.

Poszczególne elementy `<listener>` określają implementacje JMS `MessageListener` wywoływane *asynchronicznie* przez kontener odbiorców komunikatów. Atrybut `destination` elementu `<listener>` określa nazwę odbiorcy, z którego implementacja `MessageListener` będzie pobierała komunikaty za pomocą kontenera odbiorców komunikatów. Z kolei atrybut `ref` elementu `<listener>` zawiera odwołanie do implementacji `MessageListener` odpowiedzialnej za przetwarzanie otrzymanych komunikatów JMS. W listingu 8.10 pokazano, że egzemplarz `FixedDepositMessageListener`

(implementacja `MessageListener`) jest odpowiedzialny za przetwarzanie komunikatów otrzymanych z `aQueueDestination`, natomiast `EmailMessageListener` (implementacja `MessageListener`) odpowiada za przetwarzanie komunikatów otrzymanych z `emailQueueDestination`.

Interfejs `MessageListener` definiuje metodę `onMessage()`, która jest *asynchronicznie* wywoływana przez kontener odbiorców komunikatów. Otrzymany od odbiorcy egzemplarz `JMS Message` jest przez ten kontener przekazywany metodzie `onMessage()`. Metoda `onMessage()` jest odpowiedzialna za przetworzenie otrzymanego komunikatu `JMS`. Listing 8.11 przedstawia używaną w aplikacji `MyBank` implementację `FixedDepositMessageListener`, której zadaniem jest pobieranie obiektu `FixedDepositDetails` z egzemplarza `JMS Message`, a następnie zapisanie w bazie danych informacji szczegółowych o lokacie znajdujących się w obiekcie `FixedDepositDetails`.

Listing 8.11. Klasa `FixedDepositMessageListener` przetwarzająca komunikat `JMS`

Projekt: `ch08-bankapp`

Położenie pliku: `src/main/java/sample/spring/chapter08/bankapp/jms`

```
package sample.spring.chapter08.bankapp.jms;

import javax.jms.MessageListener;
import javax.jms.ObjectMessage;
import sample.spring.chapter08.bankapp.domain.FixedDepositDetails;
.....
public class FixedDepositMessageListener implements MessageListener {
    @Autowired
    @Qualifier(value = "FixedDepositDao")
    private FixedDepositDao myFixedDepositDao;

    @Autowired
    private BankAccountDao bankAccountDao;

    @Transactional
    public int createFixedDeposit(FixedDepositDetails fixedDepositDetails) {
        bankAccountDao.subtractFromAccount(fixedDepositDetails.getBankAccountId(),
            fixedDepositDetails.getFixedDepositAmount());
        return myFixedDepositDao.createFixedDeposit(fixedDepositDetails);
    }

    @Override
    public void onMessage(Message message) {
        ObjectMessage objectMessage = (ObjectMessage) message;
        FixedDepositDetails fixedDepositDetails = null;
        try {
            fixedDepositDetails = (FixedDepositDetails) objectMessage.getObject();
        } catch (JMSException e) {
            e.printStackTrace();
        }
        if (fixedDepositDetails != null) {
            createFixedDeposit(fixedDepositDetails);
        }
    }
}
```

W powyższym listingu metoda `onMessage()` klasy `FixedDepositMessageListener` pobiera obiekt `FixedDepositDetails` z komunikatu `JMS`, a następnie w bazie danych zapisuje informacje szczegółowe

o lokacie. Za zapis tych informacji w bazie danych odpowiada metoda `createFixedDeposit()` klasy `FixedDepositMessageListener`. Ponieważ metoda `createFixedDeposit()` została oznaczona adnotacją `@Transactional`, będzie wykonana w ramach transakcji zarządzanej przez menedżer `DataSource` ➔ `TransactionManager` (przeanalizuj plik `applicationContext.xml` znajdujący się w projekcie `ch08-bankapp`). Kontener odbiorców komunikatów otrzymuje komunikat JMS i wykonuje metodę `onMessage()` klasy `FixedDepositMessageListener` w transakcji zarządzanej przez menedżer `JmsTransactionManager` (patrz: listing 8.10).

Metody `onMessage()` i `createFixedDeposit()` są wykonywane przez różne menedżery zarządzające transakcjami. Operacja uaktualnienia bazy danych *nie* zostanie wycofana, jeżeli z jakiegoś powodu transakcja JMS zakończy się niepowodzeniem. Komunikat JMS *nie* będzie dostarczony egzemplarzowi `MessageListener`, gdy uaktualnienie bazy danych zakończy się niepowodzeniem. Jeżeli odbiór (i przetworzenie) komunikatu JMS oraz operacja uaktualnienia bazy danych mają odbywać się w ramach tej samej transakcji, należy użyć transakcji JTA. Z tego podrozdziału dowiedziawsz się, jak wysyłać i odbierać komunikaty JMS za pomocą Springa. Zobaczmy teraz, jak Spring ułatwia wysyłanie wiadomości e-mail.

8.5. Wysyłanie wiadomości e-mail

Spring ułatwia wysyłanie wiadomości e-mail w aplikacji, ponieważ dostarcza warstwę abstrakcji opierającą się na API `JavaMail`. Automatycznie zajmuje się zarządzaniem zasobami i obsługą wyjątków, a więc programista może skoncentrować się na tworzeniu logiki niezbędnej do przygotowania wiadomości e-mail.

W celu wysłania wiadomości e-mail za pomocą Springa należy w pierwszej kolejności przeprowadzić konfigurację klasy `JavaMailSenderImpl` w pliku XML kontekstu aplikacji. Klasa ta działa w charakterze opakowania dla API `JavaMail`. Listing 8.12 przedstawia konfigurację klasy `JavaMail` ➔ `SenderImpl` w aplikacji `MyBank`.

Listing 8.12. Plik `applicationContext.xml` przedstawiający konfigurację klasy `JavaMailSenderImpl`

Projekt: `ch08-bankapp`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host" value="${email.host}" />
  <property name="protocol" value="${email.protocol}" />
  ....
  <property name="javaMailProperties">
    <props>
      <prop key="mail.smtp.auth">true</prop>
      <prop key="mail.smtp.starttls.enable">true</prop>
    </props>
  </property>
</bean>
```

Klasa `JavaMailSenderImpl` definiuje właściwości takie jak `host`, `port`, `protocol` itd., które zawierają informacje o serwerze poczty. Właściwość `javaMailProperties` określa informacje konfiguracyjne używane przez egzemplarz `JavaMailSenderImpl` podczas tworzenia obiektu `JavaMail Session`.

Właściwość `mail.smtp.auth` ma przypisaną wartość `true`, co oznacza użycie protokołu SMTP (ang. *Simple Mail Transfer Protocol*) do uwierzytelnienia użytkownika w serwerze poczty. Z kolei wartość `true` właściwości `mail.smtp.starttls.enable` oznacza użycie szyfrowania TLS w trakcie połączenia używanego do uwierzytelnienia użytkownika w serwerze poczty.

Kod przedstawiony w listingu 8.12 pokazuje, że wartości niektórych właściwości klasy `JavaMailSenderImpl` są podawane za pomocą miejsc zarezerwowanych dla właściwości. Na przykład wartością właściwości `host` jest `${email.host}`, natomiast wartość właściwości `protocol` podano jako `${email.protocol}`. Wartości miejsc zarezerwowanych są pobierane z pliku `email.properties` znajdującego się w katalogu `src/main/resources/META-INF/spring`. Listing 8.13 przedstawia zawartość pliku `email.properties`.

Listing 8.13. Zawartość pliku `email.properties`

Projekt: `ch08-bankapp`

Położenie pliku: `src/main/resources/META-INF/spring`

```
email.host=smtp.gmail.com
email.port=587
email.protocol=smtp
email.username=<podaj-nazwę-użytkownika>
email.password=<podaj-hasło>
```

W powyższym listingu wyraźnie widać, że plik `email.properties` zawiera informacje o serwerze poczty, protokole komunikacyjnym oraz koncie poczty, z którym ma zostać nawiązane połączenie. Właściwości zdefiniowane w pliku `email.properties` są używane do przeprowadzenia konfiguracji egzemplarza `JavaMailSenderImpl` (patrz: listing 8.12).



Klasy zapewniające warstwę abstrakcji utworzoną na podstawie API `JavaMail` są zdefiniowane w pliku JAR o nazwie `spring-context-support`. Dlatego też w celu wykorzystania oferowanej przez Spring funkcji wysyłania wiadomości e-mail aplikacja musi mieć określoną zależność od pliku JAR.

Udostępniana przez Spring klasa `SimpleMailMessage` przedstawia prostą wiadomość e-mail. Klasa `SimpleMailMessage` definiuje właściwości takie jak `to`, `cc`, `subject`, `text` itd., które można wykorzystywać podczas przygotowywania wiadomości e-mail wysyłanej z poziomu aplikacji.

W listingu 8.14 został przedstawiony plik XML kontekstu aplikacji, w którym zostały skonfigurowane dwa egzemplarze klasy `SimpleMailMessage` odpowiadające dwóm wiadomościom e-mail wysyłanym przez aplikację `MyBank`.

Listing 8.14. Plik `applicationContext.xml` przedstawiający konfigurację klasy `SimpleMailMessage`

Projekt: `ch08-bankapp`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<bean class="org.springframework.mail.SimpleMailMessage" id="requestReceivedTemplate">
  <property name="subject" value="${email.subject.request.received}" />
  <property name="text" value="${email.text.request.received}" />
</bean>
```

```
<bean class="org.springframework.mail.SimpleMailMessage" id="requestProcessedTemplate">
  <property name="subject" value="\${email.subject.request.processed}" />
  <property name="text" value="\${email.text.request.processed}" />
</bean>
```

W powyższym listingu komponent `requestReceivedTemplate` przedstawia wiadomość e-mail wysyłaną klientowi wraz z informacją o otrzymaniu żądania założenia nowej lokaty. Natomiast komponent `requestProcessedTemplate` przedstawia wiadomość e-mail, w której klient jest informowany o założeniu nowej lokaty. Właściwość `subject` klasy `SimpleMailMessage` określa temat wiadomości e-mail, z kolei właściwość `text` zawiera treść wiadomości e-mail. Wartości dla tych właściwości są zdefiniowane w pliku *emailtemplate.properties*, którego zawartość przedstawia listing 8.15.

Listing 8.15. Zawartość pliku *emailtemplate.properties*

Projekt: *ch08-bankapp*

Położenie pliku: *src/main/resources/META-INF/spring*

```
email.subject.request.received=Otrzymałmy żądanie założenia nowej lokaty
email.text.request.received=Informujemy, że otrzymaliśmy od Ciebie żądanie założenia nowej lokaty.

email.subject.request.processed=Nowa lokata została założona
email.text.request.processed=Informujemy, że Twoje żądanie założenia nowej lokaty zostało spełnione.
```

Do tej pory widziałeś, jak w pliku XML kontekstu aplikacji należy przeprowadzić konfigurację klas `JavaMailSenderImpl` i `SimpleMailMessage`. Przechodzimy teraz do tematu faktycznego wysyłania wiadomości e-mail.

W listingu 8.16 została przedstawiona znajdująca się w aplikacji *MyBank* klasa `EmailMessageListener` (to implementacja `JMS MessageListener`), która otrzymuje adres e-mail klienta, a następnie wysyła do niego wiadomość e-mail informującą o otrzymaniu żądania założenia nowej lokaty.

Listing 8.16. Klasa *EmailMessageListener* przedstawiająca wysyłanie wiadomości e-mail za pomocą interfejsu *MailSender*

Projekt: *ch08-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter08/bankapp/jms*

```
package sample.spring.chapter08.bankapp.jms;

import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;
.....
public class EmailMessageListener implements MessageListener {
    @Autowired
    private transient MailSender mailSender;

    @Autowired
    @Qualifier("requestReceivedTemplate")
    private transient SimpleMailMessage simpleMailMessage;

    public void sendEmail() {
        mailSender.send(simpleMailMessage);
    }

    public void onMessage(Message message) {
```

```

    TextMessage textMessage = (TextMessage) message;
    try {
        simpleMailMessage.setTo(textMessage.getText());
    } catch (Exception e) {
        e.printStackTrace();
    }
    sendEmail();
}
}

```

W powyższym listingu widać, że metoda `send()` klasy `MailSender` wysyła wiadomość e-mail przedstawioną za pomocą egzemplarza `SimpleMailMessage`. Ponieważ klasa `JavaMailSenderImpl` implementuje oferowany przez Spring interfejs `MailSender`, egzemplarz `JavaMailSenderImpl` (patrz: listing 8.12) jest automatycznie wiązany w egzemplarzu `EmailMessageListener`. Egzemplarz `SimpleMailMessage` o nazwie `requestReceivedTemplate` (patrz: listing 8.14) również jest automatycznie wiązany w egzemplarzu `EmailMessageListener`. Skoro właściwość to klasy `SimpleMailMessage` określa adres e-mail odbiorcy, metoda `onMessage()` otrzymuje z komunikatu JMS nazwę użytkownika adresu e-mail i przypisuje ją jako wartość właściwości to.

Oferowany przez Spring interfejs `MailSender` stanowi ogólny interfejs niezależny od API `JavaMail` i odpowiedni do wysyłania prostych wiadomości e-mail. Z kolei inny interfejs Springa o nazwie `JavaMailSender` (subinterfejs `MailSender`) jest zależny od API `JavaMail` i definiuje funkcjonalność niezbędną do wysyłania wiadomości MIME. Wiadomość MIME jest używana, jeśli zachodzi potrzeba wysłania wiadomości e-mail zawierającej osadzone obrazy, załączniki itd. Tego rodzaju wiadomość jest przedstawiana przez klasę `MimeMessage` w API `JavaMail`. Spring dostarcza klasy `MimeMessageHelper` i `MimeMessagePreparator`, za pomocą których można utworzyć i wypełnić danymi egzemplarz `MimeMessage`.

W listingu 8.17 została przedstawiona znajdująca się w aplikacji `MyBank` klasa `FixedDepositProcessorJob`, która odejmuje wartość lokaty od wysokości środków na koncie bankowym klienta, a następnie wysyła mu wiadomość e-mail informującą o spełnieniu żądania założenia nowej lokaty.

Listing 8.17. Klasa `FixedDepositProcessorJob` przedstawiająca użycie interfejsu `JavaMailSender`

Projekt: `ch08-bankapp`

Położenie pliku: `src/main/java/sample/spring/chapter08/bankapp/job`

```

package sample.spring.chapter08.bankapp.job;

import javax.mail.internet.MimeMessage;
import org.springframework.mail.javamail.JavaMailSender;

public class FixedDepositProcessorJob {
    ....
    @Autowired
    private transient JavaMailSender mailSender;

    @Autowired
    @Qualifier("requestProcessedTemplate")
    private transient SimpleMailMessage simpleMailMessage;

    private List<FixedDepositDetails> getInactiveFixedDeposits() {
        return myFixedDepositDao.getInactiveFixedDeposits();
    }
}

```

```

    }

    public void sendEmail() throws AddressException, MessagingException {
        List<FixedDepositDetails> inactiveFixedDeposits = getInactiveFixedDeposits();

        for (FixedDepositDetails fixedDeposit : inactiveFixedDeposits) {
            MimeMessage mimeMessage = mailSender.createMimeMessage();
            MimeMessageHelper mimeMessageHelper = new MimeMessageHelper(mimeMessage);
            mimeMessageHelper.setTo(fixedDeposit.getEmail());
            mimeMessageHelper.setSubject(simpleMailMessage.getSubject());
            mimeMessageHelper.setText(simpleMailMessage.getText());
            mailSender.send(mimeMessage);
        }
        myFixedDepositDao.setFixedDepositsAsActive(inactiveFixedDeposits);
    }
}

```

W powyższym listingu widać, że metoda `send()` klasy `JavaMailSender` jest używana do wysłania wiadomości MIME. Ponieważ egzemplarz `JavaMailSenderImpl` implementuje oferowany przez Spring interfejs `JavaMailSender`, egzemplarz `JavaMailSenderImpl` (patrz: listing 8.12) jest automatycznie wiązany w egzemplarzu `FixedDepositProcessorJob`. Egzemplarz `SimpleMailMessage` o nazwie `requestProcessedTemplate` (patrz: listing 8.14) również jest automatycznie wiązany w egzemplarzu `FixedDepositProcessorJob`. Zmienna egzemplarza `mailSender` jest zdefiniowana jako typu `JavaMailSender` (a nie `MailSender`), ponieważ egzemplarz `FixedDepositProcessorJob` tworzy i wysyła wiadomości MIME. Metoda `sendEmail()` klasy `FixedDepositProcessorJob` tworzy egzemplarz `MimeMessage`, używając do tego metody `createMimeMessage()` klasy `JavaMailSender`. Następnie udostępniana przez Spring klasa `MimeMessageHelper` jest używana do wypełnienia egzemplarza `MimeMessage` właściwościami `to`, `subject` i `text`.

W listingu 8.18 pokazujemy, jak można utworzyć metodę `sendEmail()` klasy `FixedDepositProcessorJob`, używając do tego interfejsu wywołań zwrotnych `MimeMessagePreparator` zamiast klasy `MimeMessageHelper`.

Listing 8.18. Przykład użycia interfejsu `MimeMessagePreparator`

```

import javax.mail.Message;
import javax.mail.internet.InternetAddress;
import org.springframework.mail.javamail.MimeMessagePreparator;

public class FixedDepositProcessorJob {
    .....
    public void sendEmail_() throws AddressException, MessagingException {
        List<FixedDepositDetails> inactiveFixedDeposits = getInactiveFixedDeposits();
        for (final FixedDepositDetails fixedDeposit : inactiveFixedDeposits) {
            mailSender.send(new MimeMessagePreparator() {
                @Override
                public void prepare(MimeMessage mimeMessage) throws Exception {
                    mimeMessage.setRecipient(Message.RecipientType.TO,
                        new InternetAddress(fixedDeposit.getEmail()));
                    mimeMessage.setSubject(simpleMailMessage.getText());
                    mimeMessage.setText(simpleMailMessage.getText());
                }
            });
        }
    }
}

```

```
    }  
    myFixedDepositDao.setFixedDepositsAsActive(inactiveFixedDeposits);  
  }  
}
```

W powyższym listingu widać, że egzemplarz `MimeMessagePreparator` jest przekazywany metodzie `send()` klasy `JavaMailSender` w celu przygotowania egzemplarza `MimeMessage` do wysłania. Metoda `prepare()` klasy `MimeMessagePreparator` dostarcza nowy egzemplarz `MimeMessage`, który trzeba wypełnić danymi. W omawianym listingu zwróć uwagę na fakt, że ustawienie właściwości `recipient` klasy `MimeMessage` wymaga pracy z działającym na niskim poziomie API `JavaMail`. W listingu 8.17 metoda `setTo()` klasy `MimeMessageHelper` akceptuje adres e-mail odbiorcy przekazany jako argument i służący do ustawienia właściwości `recipient` klasy `MimeMessage`. Dlatego też należy rozważyć użycie klasy `MimeMessageHelper` do wypełnienia egzemplarza `MimeMessage` przekazanego metodzie `prepare()` klasy `MimeMessagePreparator`.

Zobaczmy teraz, jak można wykorzystać `Spring` do asynchronicznego wykonywania zadań i tworzenia harmonogramu zadań do wykonania w przyszłości.

8.6. Harmonogram zadań i wykonywanie asynchroniczne

Istnieje możliwość asynchronicznego wykonywania zadań `java.lang.Runnable` za pomocą oferowanego przez `Spring` interfejsu `TaskExecutor`, a także tworzenia harmonogramu wykonywania zadań `java.lang.Runnable` z użyciem interfejsu `TaskScheduler`. Zamiast bezpośrednio pracować z interfejsami `TaskExecutor` i `TaskScheduler` można skorzystać z alternatywnego podejścia polegającego na użyciu adnotacji `@Async` i `@Scheduled` do, odpowiednio, asynchronicznego wykonywania metod i tworzenia harmonogramu wykonywania metod.

Najpierw przeanalizujemy interfejsy `TaskExecutor` i `TaskScheduler`.

Interfejs `TaskExecutor`

W `Javie 5` wprowadzono koncepcję tak zwanych *wykonawców* przeznaczonych do wykonywania zadań `java.lang.Runnable`. Egzekutor implementuje interfejs `java.util.concurrent.Executor` definiujący pojedynczą metodę `execute(Runnable runnable)`. Oferowany przez `Spring` interfejs `TaskExecutor` rozszerza interfejs `java.util.concurrent.Executor`. Ponadto `Spring` udostępnia kilka implementacji `TaskExecutor`, które można wybierać w zależności od wymagań aplikacji. W zależności od wybranej implementacji `TaskExecutor` zadanie `Runnable task` może być wykonane synchronicznie lub asynchronicznie, z użyciem puli wątków lub `CommonJ` itd. Pewne implementacje `TaskExecutor` dostarczane przez `Spring` to: `ThreadPoolTaskExecutor` (asynchroniczne wykonywanie zadań za pomocą wątku pobranego z puli), `SyncTaskExecutor` (synchroniczne wykonywanie zadań) i `SimpleAsyncTaskExecutor` (asynchroniczne wykonywanie każdego zadania w nowym wątku).

`ThreadPoolTaskExecutor` to najczęściej stosowana implementacja interfejsu `TaskExecutor`, która do wykonywania zadań używa wprowadzonego w `Javie 5` wykonawcy `ThreadPoolExecutor`. W listingu 8.19 przedstawiamy sposób konfiguracji egzemplarza `ThreadPoolTaskExecutor` w pliku XML kontekstu aplikacji.

Listing 8.19. Konfiguracja wykonawcy *ThreadPoolTaskExecutor*

```

<bean id="myTaskExecutor"
    class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
    <property name="corePoolSize" value="5" />
    <property name="maxPoolSize" value="10" />
    <property name="queueCapacity" value="15" />
    <property name="rejectedExecutionHandler" ref="abortPolicy"/>
</bean>

<bean id="abortPolicy" class="java.util.concurrent.ThreadPoolExecutor.AbortPolicy"/>

```

Właściwość `corePoolSize` określa minimalną liczbę wątków znajdujących się w puli wątków. Z kolei właściwość `maxPoolSize` definiuje maksymalną liczbę wątków, jakie mogą znaleźć się w puli wątków. Właściwość `queueCapacity` wskazuje maksymalną liczbę zadań, jakie mogą oczekiwać w kolejce, gdy wszystkie wątki z puli wątków są zajęte wykonywaniem zadań. Właściwość `rejectedExecutionHandler` określa procedurę obsługi dla zadań odrzuconych przez `ThreadPoolTaskExecutor`. Zadanie zostanie odrzucone przez `ThreadPoolTaskExecutor`, jeśli kolejka jest pełna i pula wątków nie zawiera żadnego wolnego wątku, który mógłby wykonać dane zadanie. Właściwość `rejectedExecutionHandler` odwołuje się do egzemplarza obiektu `java.util.concurrent.RejectedExecutionHandler`.

W listingu 8.19 właściwość `rejectedExecutionHandler` odwołuje się do egzemplarza `java.util.concurrent.ThreadPoolExecutor.AbortPolicy`, który zawsze zgłasza wyjątek `RejectedExecutionException`. Inne procedury obsługi dostępne dla odrzuconych zadań to `java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy` (odrzucone zadanie będzie wykonane w wątku komponentu, który je wywołuje), `java.util.concurrent.ThreadPoolExecutor.DiscardOldestPolicy` (wyrzuca najstarsze zadanie z kolejki i próbuje wykonać odrzucone) i `java.util.concurrent.ThreadPoolExecutor.DiscardPolicy` (usuwa odrzucone zadanie).

Element `<executor>` schematu `task` oferowanego przez Spring ułatwia konfigurację egzemplarza `ThreadPoolTaskExecutor`, jak przedstawia listing 8.20.

Listing 8.20. Konfiguracja egzemplarza *ThreadPoolTaskExecutor* za pomocą schematu *task*

```

<beans xmlns:task="http://www.springframework.org/schema/task"
    xsi:schemaLocation="....http://www.springframework.org/schema/task
        http://www.springframework.org/schema/task/spring-task-4.0.xsd">

    <task:executor id=" myTaskExecutor" pool-size="5-10"
        queue-capacity="15" rejection-policy="ABORT" />
</beans>

```

W powyższym listingu element `<executor>` konfiguruje egzemplarz `ThreadPoolTaskExecutor`. Atrybut `pool-size` określa podstawową i maksymalną wielkość puli wątków. W omawianym przykładzie podstawowa wielkość puli to 5 wątków, a maksymalna to 10 wątków. Atrybut `queue-capacity` powoduje ustawienie właściwości `queueCapacity`, natomiast atrybut `rejection-policy` określa procedurę obsługi dla odrzuconych zadań. Możliwe do użycia wartości atrybutu `rejection-policy` to `ABORT`, `CALLER_RUNS`, `DISCARD_OLDEST` i `DISCARD`.

Po przeprowadzeniu konfiguracji egzemplarza `ThreadPoolTaskExecutor` przez jego wyraźne skonfigurowanie jako komponentu Springa (patrz: listing 8.19) lub za pomocą schematu `task` oferowanego przez Spring (patrz: listing 8.20) można wreszcie wstrzyknąć egzemplarz `ThreadPoolTaskExecutor` do komponentów, które mają asynchronicznie wykonywać zadania `java.lang.Runnable`, jak to przedstawia listing 8.21.

Listing 8.21. *Wykonywanie zadań za pomocą egzemplarza wykonawcy `ThreadPoolTaskExecutor`*

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.task.TaskExecutor;

@Component
public class Sample {
    @Autowired
    private TaskExecutor taskExecutor;

    public void executeTask(Runnable task) {
        taskExecutor.execute(task);
    }
}
```

W powyższym listingu egzemplarz `ThreadPoolTaskExecutor` jest automatycznie wiązany w klasie `Sample`, a następnie jest używany przez metodę `executeTask()` klasy `Sample` w celu wykonania zadania `java.lang.Runnable`.

Egzemplarz `TaskExecutor` wykonuje zadanie `java.lang.Runnable` natychmiast po jego wystąpieniu, a samo zadanie będzie wykonane tylko raz. Jeżeli zadanie `java.lang.Runnable` ma zostać wykonane w innym terminie i ma być wykonywane okresowo, można wykorzystać implementację interfejsu `TaskScheduler`.

Interfejs `TaskScheduler`

Oferowany przez Spring interfejs `TaskScheduler` zapewnia abstrakcję pozwalającą na tworzenie harmonogramów zadań `java.lang.Runnable` do wykonania. Udostępniany przez Spring interfejs `Trigger` zapewnia abstrakcję czasu, w którym ma zostać wykonane zadanie `java.lang.Runnable`. Egzemplarz `TaskScheduler` trzeba powiązać z egzemplarzem `Trigger` i tym samym przygotować harmonogram dla zadań `java.lang.Runnable`. Egzemplarz `PeriodicTrigger` (implementacja interfejsu `Trigger`) jest używany, gdy zadania mają być wykonywane *okresowo*. `CronTrigger` (inna implementacja `Trigger`) akceptuje *wyrażenie cron*, które wskazuje datę i godzinę wykonania zadania.

`ThreadPoolTaskScheduler` to jedna z najczęściej stosowanych implementacji `TaskScheduler`, która wewnętrznie używa wprowadzonego w Javie 5 wykonawcy `ScheduledThreadPoolExecutor` (implementacja dostępnego w Javie 5 interfejsu `ScheduledExecutorService`) do tworzenia harmonogramów zadań do wykonania. Istnieje możliwość przeprowadzenia konfiguracji implementacji `ThreadPoolTaskScheduler` i powiązania jej z implementacją `Trigger`, tworząc tym samym harmonogram do wykonania zadania. Listing 8.22 przedstawia sposób konfiguracji i użycia egzemplarza `ThreadPoolTaskExecutor`.

Listing 8.22. Przykład konfiguracji i użycia egzemplarza *ThreadPoolTaskExecutor*

```

----- Konfiguracja ThreadPoolTaskScheduler -----

<bean id="myScheduler"
      class="org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler">
  <property name="poolSize" value="5"/>
</bean>

----- Użycie ThreadPoolTaskScheduler -----

import org.springframework.scheduling.TaskScheduler;
import org.springframework.scheduling.support.PeriodicTrigger;

@Component
public class Sample {
  @Autowired
  @Qualifier("myScheduler")
  private TaskScheduler taskScheduler;

  public void executeTask(Runnable task) {
    taskScheduler.schedule(task, new PeriodicTrigger(5000));
  }
}

```

W powyższym listingu właściwość `poolSize` klasy `ThreadPoolTaskScheduler` określa liczbę wątków w puli wątków. Aby utworzyć harmonogram zadania do wykonania, następuje wywołanie metody `schedule()` klasy `ThreadPoolTaskScheduler` i przekazanie jej zadania `java.lang.Runnable` i egzemplarza `Trigger`. W omawianym listingu metodzie `schedule()` jest przekazywany egzemplarz `PeriodicTrigger`. Argumentem konstruktora `PeriodicTrigger` jest (wyrażony w milisekundach) odstęp czasu między wykonaniem kolejnych zadań.

Element `<scheduler>` schematu `task` oferowanego przez Spring ułatwia konfigurację egzemplarza `ThreadPoolTaskScheduler`. Egzemplarz `ThreadPoolTaskScheduler` utworzony przez element `<scheduler>` może być następnie użyty przez element `<scheduled-tasks>` schematu `task` w celu przygotowania harmonogramu wykonywania metod komponentu. Listing 8.23 pokazuje sposób użycia elementów `<scheduler>` i `<scheduled-tasks>` w aplikacji `MyBank` w celu wykonywania co 5 sekund metody `sendEmail()` klasy `FixedDepositProcessorJob`.

Listing 8.23. Przykład użycia elementów `<scheduler>` i `<scheduled-tasks>`

Projekt: `ch08-bankapp`

Położenie pliku: `src/main/java/sample/spring/chapter08/bankapp/job`

```

<task:scheduler id="emailScheduler" pool-size="10" />

<task:scheduled-tasks scheduler="emailScheduler">
  <task:scheduled ref="FixedDepositProcessorJob" method="sendEmail"
    fixed-rate="5000" />
</task:scheduled-tasks>

<bean id="FixedDepositProcessorJob"
      class="sample.spring.chapter08.bankapp.job.FixedDepositProcessorJob" />

```

W powyższym listingu element `<scheduler>` przeprowadza konfigurację egzemplarza `ThreadPoolTaskScheduler`. Atrybut `id` elementu `<scheduler>` określa nazwę, pod jaką egzemplarz `ThreadPoolTaskScheduler` będzie dostępny dla innych komponentów w kontenerze Springa. Atrybut `scheduler` elementu `<scheduled-tasks>` wskazuje odwołanie do egzemplarza `ThreadPoolTaskScheduler` używanego do utworzenia harmonogramu wykonywania metod komponentu. W omawianym listingu egzemplarz `ThreadPoolTaskScheduler` utworzony przez element `<scheduler>` jest wskazywany przez atrybut `scheduler` elementu `<scheduled-tasks>`.

Element `<scheduled-tasks>` zawiera jeden lub więcej elementów `<scheduled>`. W elemencie `<scheduled>` są przechowywane informacje o metodach komponentu przeznaczonych do wykonania, a także o wyzwalaczu dla tych metod. Atrybut `ref` określa odwołanie do komponentu Springa, atrybut `method` określa metodę komponentu, do której odnosi się atrybut `ref`, natomiast atrybut `fixed-rate` (wyzwalacz oparty na przedziale czasu) wskazuje odstęp czasu między kolejnymi wywołaniami metod. W listingu 8.23 element `<scheduled>` określa, że metoda `sendEmail()` klasy `FixedDepositProcessorJob` będzie wykonywana co 5 sekund.

Zamiast atrybutu `fixed-rate` elementu `<scheduled>` można użyć atrybutu `fixed-delay` (wyzwalacz oparty na przedziale czasu), `cron` (wyzwalacz oparty na mechanizmie cron) lub `trigger` (odwołanie do implementacji `Trigger`) i tym samym określić wyzwalacz dla metod.

Przechodzimy teraz do oferowanych przez Spring adnotacji `@Async` i `@Scheduled`.

Adnotacje `@Async` i `@Scheduled`

Jeżeli metoda komponentu zostanie oznaczona adnotacją `@Async`, wówczas będzie asynchronicznie wykonana przez Spring. Z kolei oznaczenie metody komponentu adnotacją `@Scheduled` spowoduje przygotowanie harmonogramu dla jej wykonania.

Użycie adnotacji `@Async` i `@Scheduled` jest włączane za pomocą elementu `<annotation-driven>` schematu `task`, jak przedstawia listing 8.24.

Listing 8.24. Włączenie użycia adnotacji `@Async` i `@Scheduled`

```
<task:annotation-driven executor="anExecutor" scheduler="aScheduler"/>

<task:executor id="anExecutor"/>

<task:scheduled-tasks scheduler="aScheduler">
  <task:scheduled ref="sampleJob" method="doSomething" fixed-rate="5000" />
</task:scheduled-tasks>
```

Atrybut `executor` elementu `<annotation-driven>` określa odwołanie do oferowanego przez Spring egzemplarza `TaskExecutor` (lub wprowadzonego w Javie 5 `Executor`) używanego do wykonywania metod oznaczonych adnotacją `@Async`. Atrybut `scheduler` wskazuje odwołanie do oferowanego przez Spring egzemplarza `TaskScheduler` (lub wprowadzonego w Javie 5 `ScheduledExecutorService`) używanego do wykonywania metod oznaczonych adnotacją `@Scheduled`.

Teraz dokładnie przeanalizujemy adnotację `@Async`.

Adnotacja @Async

W listingu 8.25 przedstawiamy pewne ważne aspekty, które trzeba uwzględnić podczas użycia adnotacji @Async.

Listing 8.25. Przykład użycia adnotacji @Async

```
import java.util.concurrent.Future;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.AsyncResult;
import org.springframework.stereotype.Component;

@Component
public class Sample {
    @Async
    public void doA() { ..... }

    @Async(value="someExecutor")
    public void doB(String str) { ..... }

    @Async
    public Future<String> doC() {
        return new AsyncResult<String>("Witaj");
    }
}
```

Atrybut value adnotacji @Async określa oferowany przez Spring egzemplarz TaskExecutor (lub wprowadzony w Javie 5 Executor) używany do asynchronicznego wykonania metody. Ponieważ adnotacja @Async w metodzie doA() nie wskazuje wykonawcy, do asynchronicznego wykonania metody doA() zostanie użyty oferowany przez Spring wykonawca SimpleAsyncTaskExecutor. Z kolei adnotacja @Async w metodzie doB() ma przypisaną wartość someExecutor dla atrybutu value. Oznacza to, że komponent o nazwie someExecutor (typu TaskExecutor lub wprowadzonego w Javie 5 Executor) zostanie użyty do asynchronicznego wykonania metody doB(). Metody oznaczone adnotacją @Async mogą akceptować argumenty, podobnie jak miało to miejsce w przypadku metody doB() w omawianym listingu. Wartością zwrótną metody oznaczonej adnotacją @Async może być void (podobnie jak w metodach doA() i doB()) lub Future (podobnie jak w metodzie doC()). Aby wartością zwrótną był egzemplarz Future, wartość przeznaczona do zwrotu powinna być opakowana obiektem AsyncResult, który następnie jest zwracany przez metodę.

Teraz dokładnie przeanalizujemy adnotację @Scheduled.

Adnotacja @Scheduled

W listingu 8.26 przedstawiamy pewne ważne aspekty, które trzeba uwzględnić podczas użycia adnotacji @Scheduled.

Listing 8.26. Przykład użycia adnotacji @Scheduled

```
import org.springframework.scheduling.annotation.Scheduled;

@Component
public class Sample {
    @Scheduled(cron="0 0 9-17 * * MON-FRI")
}
```

```
public void doA() { ..... }

@Scheduled(fixedRate = 5000)
public void doB() { ..... }
}
```

Wartością zwrótną metody oznaczonej adnotacją `@Scheduled` *musi* być `void`, a ponadto metoda *nie może* akceptować żadnych argumentów. *Konieczne* jest określenie atrybutu `cron`, `fixedRate` lub `fixedDelay` adnotacji `@Scheduled`.

Trzeba zwrócić uwagę na jeszcze jeden aspekt. Jeżeli adnotacja `@Async` (lub `@Scheduled`) zostanie użyta dla jednej lub więcej metod klasy, wówczas w ścieżce dostępu zdefiniowanej przez zmienną środowiskową `CLASSPATH` w aplikacji musi się znaleźć plik JAR `CGLIB`. Jeżeli adnotacja `@Async` (lub `@Scheduled`) będzie używana jedynie dla metod zdefiniowanych w interfejsie, wtedy nie ma konieczności dołączania pliku JAR `CGLIB`. Począwszy od wydania Spring 3.2, klasy `CGLIB` znajdują się w pliku JAR o nazwie *spring-core*. Dlatego nie trzeba wyraźnie definiować zależności projektu od pliku JAR `CGLIB`.



Jeżeli w aplikacji Spring chcesz zastosować Quartz Scheduler (<http://quartz-scheduler.org/>), możesz wykorzystać udostępniane przez Spring klasy integracji, które ułatwiają użycie Quartz Scheduler.

Spring ułatwia buforowanie w aplikacji, dostarczając warstwy abstrakcji zbudowane na podstawie `java.util.concurrent.ConcurrentMap` i `Ehcache` (<http://ehcache.org/>).

8.7. Buforowanie

Jeżeli w aplikacji chcesz skorzystać z buforowania, rozważ użycie warstwy abstrakcji oferowanej przez Spring. Warstwa ta pozwala programistom uniknąć konieczności bezpośredniej pracy z API implementacji buforowania. Począwszy od wydania Spring 3.2, warstwa abstrakcji buforowania jest dostępna w standardzie dla `java.util.concurrent.ConcurrentMap`, `Ehcache` oraz dla rozwiązań z zakresu buforowania implementujących specyfikację JSR 107, czyli API Java Temporary Caching (znane pod nazwą `JCACHE`).

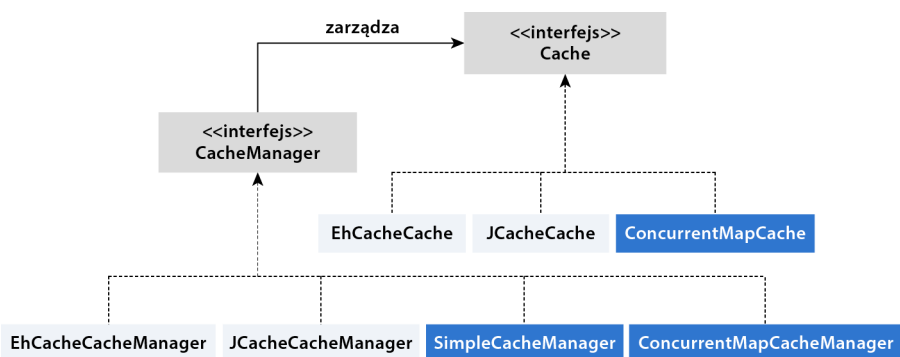


Jeżeli używasz rozwiązania z zakresu buforowania, które aktualnie nie jest obsługiwane przez warstwę abstrakcji w Springu, możesz bezpośrednio wykorzystać API buforowania lub utworzyć adaptery mapujące oferowaną przez Spring warstwę abstrakcji na rozwiązanie z zakresu buforowania.

Spring dostarcza interfejs `CacheManager` definiujący metody przeznaczone do zarządzania kolekcją egzemplarzy `Cache`. Egzemplarz `CacheManager` działa w charakterze opakowania dla menedżera buforowania dostarczanego przez zastosowane rozwiązanie z zakresu buforowania. Na przykład `EhCacheCacheManager` jest opakowaniem dla rozwiązania `net.sf.ehcache.CacheManager` oferowanego przez `Ehcache`, `JCacheCacheManager` to opakowanie dla oferowanej przez specyfikację JSR 107 implementacji dostawcy `javax.cache.CacheManager` itd. Egzemplarz `Cache` jest opakowaniem

dla używanego bufora i dostarcza metody przeznaczone do pracy z nim. Na przykład EhCacheCache (implementacja Cache) jest opakowaniem dla `net.sf.ehcache.Ehcache`, natomiast JCacheCache (implementacja Cache) to opakowanie dla oferowanego przez specyfikację JSR 107 egzemplarza dostawcy `javax.cache.Cache`.

Spring dostarcza również egzemplarz `ConcurrentMapCacheManager` gotowy do zastosowania, jeśli chcesz użyć `java.util.concurrent.ConcurrentMap` jako bufora. Egzemplarz Cache zarządzany przez `ConcurrentMapCacheManager` jest typu `ConcurrentMapCache`. Diagram na rysunku 8.3 pokazuje związki zachodzące między interfejsami `CacheManager` i `Cache` dostarczonymi przez warstwę abstrakcji buforowania w Springu.



Rysunek 8.3. Implementacja `CacheManager` działa w charakterze opakowania dla rozwiązania z zakresu buforowania, natomiast implementacja `Cache` dostarcza operacje przeznaczone do pracy z zastosowanym buforowaniem

➔ Jeżeli chcesz użyć oferowanej przez Spring warstwy abstrakcji buforowania dla rozwiązania, które aktualnie nie jest obsługiwane przez Spring, musisz dostarczyć implementacje `CacheManager` i `Cache` dla rozwiązania z zakresu buforowania.

Z rysunku 8.3 wynika, że `CacheManager` zarządza egzemplarzami `Cache`. Natomiast `EhCacheCacheManager` zarządza egzemplarzami `EhCacheCache` (zastosowanym magazynem bufora jest `Ehcache`), `JCacheCacheManager` zarządza egzemplarzami `JCacheCache` (zastosowanym magazynem bufora jest rozwiązanie z zakresu buforowania implementujące specyfikację JSR 107), `ConcurrentMapCacheManager` zarządza egzemplarzami `ConcurrentMapCache` (zastosowanym magazynem bufora jest `java.util.concurrent.ConcurrentMap`) itd.

Na rysunku 8.3 pokazano klasę `SimpleCacheManager` implementującą interfejs `CacheManager`. Klasa `SimpleCacheManager` jest użyteczna w prostych rozwiązaniach z zakresu buforowania oraz do celów testowych. Na przykład jeśli chcesz użyć `java.util.concurrent.ConcurrentMap` jako magazynu bufora, do zarządzania buforem możesz wykorzystać `SimpleCacheManager` zamiast `ConcurrentMapCacheManager`.

Zobaczmy teraz, jak interfejs `CacheManager` jest konfigurowany w pliku XML kontekstu aplikacji.

Konfiguracja interfejsu CacheManager

W aplikacji MyBank kolekcja egzemplarzy `java.util.concurrent.ConcurrentMap` jest używana jako magazyn buforowania. Dlatego klasę `SimpleCacheManager` wykorzystano do zarządzania buforem.

W listingu 8.27 pokazujemy, jak egzemplarz `SimpleCacheManager` został skonfigurowany w aplikacji MyBank.

Listing 8.27. Konfiguracja klasy `SimpleCacheManager`

Projekt: `ch08-bankapp`

Położenie pliku: `src/main/resources/META-INF/spring/`

```
<bean id="myCacheManager"
      class="org.springframework.cache.support.SimpleCacheManager">
  <property name="caches">
    <set>
      <bean
        class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean">
          <property name="name" value="FixedDepositList" />
        </bean>
      <bean
        class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean">
          <property name="name" value="FixedDeposit" />
        </bean>
    </set>
  </property>
</bean>
```

Właściwość `caches` klasy `SimpleCacheManager` określa kolekcję buforów zarządzanych przez egzemplarz `SimpleCacheManager`. `ConcurrentMapCacheFactoryBean` to implementacja `FactoryBean` ułatwiająca konfigurację egzemplarza `ConcurrentMapCache`, czyli obiektu `Cache` używającego egzemplarza `java.util.concurrent.ConcurrentHashMap` (implementacja interfejsu `java.util.concurrent.ConcurrentMap`) jako magazynu bufora. Właściwość `name` klasy `ConcurrentMapCacheFactoryBean` określa nazwę bufora. W omawianym listingu bufory `FixedDepositList` i `FixedDeposit` są zarządzane przez egzemplarz `SimpleCacheManager`.

Zobaczmy teraz, jak w aplikacjach można używać oferowanych przez Spring adnotacji buforowania.

Adnotacje buforowania `@Cacheable`, `@CacheEvict` i `@CachePut`

Po skonfigurowaniu odpowiedniej implementacji `CacheManager` dla aplikacji trzeba zdecydować się, w jaki sposób korzystać z oferowanej przez Spring abstrakcji buforowania. Do dyspozycji masz adnotacje buforowania (takie jak `@Cacheable`, `@CacheEvict` i `@CachePut`) oraz schemat `cache`. Ponieważ użycie oferowanego przez Spring schematu `cache` oznacza znaczne zwiększenie pliku XML kontekstu aplikacji, w tym rozdziale przeanalizujemy jedynie użycie adnotacji buforowania zapewniających obsługę buforowania deklaracyjnego.

W celu użycia adnotacji buforowania konieczne jest skonfigurowanie elementu `<annotation-driven>` schematu `cache`, jak przedstawia listing 8.28.

Listing 8.28. *Użycie elementu <annotation-driven> do włączenia adnotacji buforowania*Projekt: *ch08-bankapp*Położenie pliku: *src/main/resources/META-INF/spring/*

```

<beans xmlns:cache="http://www.springframework.org/schema/cache"
  xsi:schemaLocation=".....
  http://www.springframework.org/schema/cache
  http://www.springframework.org/schema/cache/spring-cache.xsd">

  <cache:annotation-driven cache-manager="myCacheManager"/>
  .....
</beans>

```

W powyższym listingu oferowany przez Spring schemat cache został dołączony, aby jego elementy były dostępne w pliku XML kontekstu aplikacji. Atrybut `cache-manager` elementu `<annotation-driven>` odwołuje się do komponentu `CacheManager` używanego do zarządzania buforem. Nie ma konieczności określania atrybutu `cache-manager`, jeżeli komponent `CacheManager` ma nazwę `cacheManager`.

Po włączeniu obsługi adnotacji buforowania możemy zapoznać się z różnymi dostępnymi adnotacjami buforowania.

Adnotacja @Cacheable

Adnotacja `@Cacheable` użyta dla metody oznacza, że wartość zwracana przez tę metodę będzie buforowana. Klasa `DefaultKeyGenerator` jest domyślnie używana do wygenerowania klucza, pod którym w buforze będzie przechowywana wartość zwrótna metody. Do wygenerowania klucza klasa `DefaultKeyGenerator` używa sygnatury metody i jej argumentów. Istnieje możliwość wykorzystania własnego generatora klucza przez dostarczenie implementacji interfejsu `KeyGenerator` oferowanego przez Spring oraz podanie nazwy tej implementacji jako wartości atrybutu `key-generator` elementu `<annotation-driven>`.

W listingu 8.29 przedstawiamy przykład użycia adnotacji `@Cacheable` do buforowania wartości zwrótniej metody `findFixedDepositsByBankAccount()` klasy `FixedDepositService` w aplikacji `MyBank`.

Listing 8.29. *Przykład użycia adnotacji @Cacheable*Projekt: *ch08-bankapp*Położenie pliku: *src/main/java/sample/spring/chapter08/bankapp/service*

```

package sample.spring.chapter08.bankapp.service;

import org.springframework.cache.annotation.Cacheable;
.....
@Service(value = "FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {
  .....
  @Cacheable(value = { "FixedDepositList" })
  public List<FixedDepositDetails> findFixedDepositsByBankAccount(int bankAccountId) {
    logger.info("Wywołano metodę findFixedDepositsByBankAccount.");
    return myFixedDepositDao.findFixedDepositsByBankAccount(bankAccountId);
  }
}

```


Atrybut `value` adnotacji `@Cacheable` określa region w buforze, w którym będzie buforowana wartość zwrrotna. W listingu 8.27 dla aplikacji `MyBank` utworzono w buforze region o nazwie `FixedDepo` ↪ `sitList`. W listingu tym adnotacja `@Cacheable` określa, że wartość zwracana przez metodę `find` ↪ `FixedDepositsByBankAccount()` będzie przechowywana w buforze `FixedDepositList`. Trzeba koniecznie zwrócić uwagę na fakt, że metoda oznaczona adnotacją `@Cacheable` nie jest wywoływana, jeśli metodzie będzie przekazany ten sam zestaw wartości argumentów. Jednak metoda oznaczona adnotacją `@Cacheable` zostanie wywołana w przypadku przekazania innej wartości dla co najmniej jednego z argumentów metody.

Adnotacja `@CacheEvict`

Jeżeli chcesz usunąć dane z bufora po wywołaniu metody, oznacz ją adnotacją `@CacheEvict`. W aplikacji `MyBank` po założeniu nowej lokaty informacje o lokacie buforowane przez metodę `findFixedDepo` ↪ `sitsByBankAccount()` klasy `FixedDepositServiceImpl` muszą być usunięte z bufora. To gwarantuje, że w trakcie kolejnego wywołania metody `findFixedDepositsByBankAccount()` informacje o nowo założonej lokacie również zostaną pobrane z bazy danych. W listingu 8.30 przedstawiamy przykład użycia adnotacji `@CacheEvict`.

Listing 8.30. Przykład użycia adnotacji `@CacheEvict`

Projekt: `ch08-bankapp`

Położenie pliku: `src/main/java/sample/spring/chapter08/bankapp/service`

```
package sample.spring.chapter08.bankapp.service;

import org.springframework.cache.annotation.CacheEvict;
.....
@Service(value = "FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {
    .....
    @Transactional("jmsTxManager")
    @CacheEvict(value = { "FixedDepositList" }, allEntries=true,
        beforeInvocation = true)
    public void createFixedDeposit(final FixedDepositDetails fixedDepositDetails) throws
        Exception { ..... }
    .....
}
```

W powyższym listingu adnotacja `@CacheEvict` w metodzie `createFixedDeposit()` nakazuje Springowi usunięcie wszystkich buforowanych elementów w regionie bufora o nazwie `FixedDepositList`. Atrybut `value` określa nazwę regionu bufora, z którego mają być usunięte buforowane elementy. Z kolei atrybut `allEntries` określa, czy usunięte mają być wszystkie elementy ze wskazanego regionu bufora. Jeżeli chcesz usunąć tylko konkretny element, wówczas użyj atrybutu `key` i podaj klucz buforowanego elementu. Istnieje również możliwość zastosowania warunkowego usunięcia elementów poprzez wykorzystanie w tym celu atrybutu `condition`. Atrybuty `condition` i `key` pozwalają na podanie wartości za pomocą wyrażeń SpEL (więcej informacji na temat SpEL znajdziesz w podrozdziale 6.8), a tym samym na przeprowadzanie skomplikowanych operacji usuwania elementów bufora. Atrybut `beforeInvocation` określa, czy operacja usunięcia elementów

bufora będzie przeprowadzana *przed* wykonaniem metody czy po. Ponieważ wartością atrybutu `beforeInvocation` jest `true`, elementy zostaną usunięte z bufora przed wywołaniem metody `create` ➔ `FixedDeposit()`.

Adnotacja `@CachePut`

Spring oferuje adnotację `@CachePut` wskazującą, że dana metoda *zawsze* będzie wywoływana, a wartość zwrrotna tej metody będzie umieszczana w buforze. Adnotacja `@CachePut` różni się od adnotacji `@Cacheable` pod tym względem, że `@Cacheable` nakazuje Springowi pominięcie wywołania metody, jeśli ma zostać wywołana z tym samym zestawem wartości argumentów.

W listingu 8.31 przedstawiamy przykład użycia adnotacji `@CachePut` dla klasy `FixedDepositServiceImpl` aplikacji `MyBank`.

Listing 8.31. Przykład użycia adnotacji `@CachePut`

Projekt: `ch08-bankapp`

Położenie pliku: `src/main/java/sample/spring/chapter08/bankapp/service`

```
package sample.spring.chapter08.bankapp.service;

import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
.....
@Service(value = "FixedDepositService")
public class FixedDepositServiceImpl implements FixedDepositService {
    ....
    @CachePut(value="FixedDeposit", key="#FixedDepositId")
    public FixedDepositDetails getFixedDeposit(int FixedDepositId) {
        logger.info("Wywołanie metody getFixedDeposit wraz z FixedDepositId " +
            FixedDepositId);
        return myFixedDepositDao.getFixedDeposit(FixedDepositId);
    }

    @Cacheable(value={"FixedDeposit"}, key="#FixedDepositId")
    public FixedDepositDetails getFixedDepositFromCache(int FixedDepositId) {
        logger.info("Wywołanie metody getFixedDepositFromCache wraz FixedDepositId "
            + FixedDepositId);
        throw new RuntimeException("Ta metoda zgłosiła wyjątek, ponieważ "
            + "obiekt FixedDepositDetails musi pochodzić z bufora");
    }
    ....
}
```

W powyższym listingu metoda `getFixedDeposit()` jest oznaczona adnotacją `@CachePut`, co powoduje, że będzie zawsze wywoływana, a zwracany przez nią obiekt `FixedDepositDetails` będzie przechowywany w buforze o nazwie `FixedDeposit`. Atrybut `value` określa nazwę bufora przeznaczonego do przechowywania obiektu `FixedDepositDetails`. Atrybut `key` wskazuje klucz używany do przechowywania w buforze obiektu `FixedDepositDetails` zwróconego przez metodę. Jak możesz zauważyć, atrybut `key` określa klucz za pomocą wyrażenia SpEL. Wartość `#FixedDepositId` argumentu `key` odwołuje się do argumentu `FixedDepositId` przekazywanego metodzie `getFixedDeposit()`.

Podsumowując: obiekt `FixedDepositDetails` zwrócony przez metodę `getFixedDeposit()` jest przechowywany w buforze o nazwie `FixedDeposit`, a wartość argumentu `FixedDepositId` metody jest używana jako klucz.

W kodzie przedstawionym w listingu 8.31 metoda `getFixedDepositFromCache()` klasy `FixedDepositServiceImpl` pobiera z bufora obiekt `FixedDepositDetails`, opierając się na wartości atrybutu `key` podanej w adnotacji `@Cacheable`. Zwróć uwagę, że działanie metody `getFixedDepositFromCache()` sprowadza się do zgłoszenia wyjątku `RuntimeException`. Wartość atrybutu `key` odwołuje się do argumentu `FixedDepositId` przekazywanego metodzie `getFixedDepositFromCache()`. Jeżeli obiekt `FixedDepositDetails` nie zostanie znaleziony w buforze, wówczas nastąpi wywołanie metody `getFixedDepositFromCache()`, która zgłosi wyjątek `RuntimeException`.

Zobaczmy teraz, co się stanie po uruchomieniu aplikacji `MyBank` w projekcie *ch08-bankapp*.

8.8. Aplikacja MyBank

Klasa `BankApp` aplikacji `MyBank` definiuje metodę `main()` aplikacji. Metoda ta uzyskuje dostęp do metod egzemplarzy `FixedDepositService` i `BankAccountService` w celu zademonstrowania różnych funkcji omówionych w rozdziale.

W listingu 8.32 została przedstawiona klasa `BankApp` aplikacji `MyBank`.

Listing 8.32. Klasa `BankApp`

Projekt: *ch08-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter08/bankapp*

```
package sample.spring.chapter08.bankapp;

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

public class BankApp {
    public static void main(String args[]) throws Exception {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");

        BankAccountService bankAccountService =
            context.getBean(BankAccountService.class);

        BankAccountDetails bankAccountDetails = new BankAccountDetails();
        ....
        int bankAccountId = bankAccountService.createBankAccount(bankAccountDetails);

        FixedDepositService fixedDepositService =
            context.getBean(FixedDepositService.class);
        FixedDepositDetails fixedDepositDetails = new FixedDepositDetails();
        ....
        fixedDepositDetails.setEmail("nazwauser@nazwadomeny.pl");
        fixedDepositService.createFixedDeposit(fixedDepositDetails);
        ....
        fixedDepositService.findFixedDepositsByBankAccount(bankAccountId);
        fixedDepositService.findFixedDepositsByBankAccount(bankAccountId);
    }
}
```

```

FixedDepositService.createFixedDeposit(fixedDepositDetails);
.....
List<FixedDepositDetails> FixedDepositDetailsList = FixedDepositService
    .findFixedDepositsByBankAccount(bankAccountId);

for (FixedDepositDetails detail : FixedDepositDetailsList) {
    FixedDepositService.getFixedDeposit(detail.getFixedDepositId());
}

for (FixedDepositDetails detail : FixedDepositDetailsList) {
    FixedDepositService.getFixedDepositFromCache(detail.getFixedDepositId());
}
.....
}
}

```

W powyższym listingu metoda `main()` wykonuje następujące zadania:

1. Na początku następuje utworzenie w tabeli `BANK_ACCOUNT_DETAILS` rekordu dla nowego konta bankowego. W tym celu wywoływana jest metoda `createBankAccount()` klasy `BankAccountService`.
2. W tabeli `FIXED_DEPOSIT_DETAILS` dla nowo utworzonego konta bankowego następuje założenie nowej lokaty przez wywołanie metody `createFixedDeposit()` klasy `FixedDepositService`. Należy się upewnić, że właściwość `email` obiektu `FixedDepositDetails` zawiera adres e-mail, na który mają być wysyłane wiadomości. Metoda `createFixedDeposit()` wysyła dwa komunikaty JMS (patrz: listing 8.5). Pierwszy komunikat zawiera informacje o adresie e-mail podanym we właściwości `email` obiektu `FixedDepositDetails` i jest przetwarzany przez egzemplarz `EmailMessageListener` (patrz: listing 8.16) odpowiedzialny za wysyłanie wiadomości e-mail klientom. Drugi komunikat JMS jest przetwarzany przez egzemplarz `FixedDepositMessageListener` (patrz: listing 8.11) odpowiedzialny za zapis w tabeli `FIXED_DEPOSIT_DETAILS` informacji szczegółowych o lokacie. W tym miejscu warto zwrócić uwagę na fakt, że metoda `createFixedDeposit()` klasy `FixedDepositServiceImpl` jest oznaczona adnotacją `@CacheEvict` (patrz: listing 8.30). Skutkiem będzie usunięcie wszystkich elementów buforowanych w `FixedDepositList`.
3. Metoda `findFixedDepositsByBankAccount()` klasy `FixedDepositService` jest wywoływana w celu pobrania informacji o lokatach założonych dla konta bankowego, które zostało utworzone w kroku 1. Ponieważ metoda `findFixedDepositsByBankAccount()` została oznaczona adnotacją `@Cacheable` (patrz: listing 8.29), zwrócone przez metodę `findFixedDepositsByBankAccount()` informacje o lokatach są przechowywane w buforze o nazwie `FixedDepositList`. W listingu 8.29 widać, że metoda `findFixedDepositsByBankAccount()` wyświetla w konsoli komunikat o treści Wywołano metodę `findFixedDepositsByBankAccount`. W omawianym listingu metoda `findFixedDepositsByBankAccount()` jest wywoływana dwukrotnie dla tego samego argumentu `bankAccountId`, ale w konsoli został wyświetlony tylko jeden komunikat o treści Wywołano metodę `findFixedDepositsByBankAccount`. Wynika to z faktu, że drugie wywołanie metody `findFixedDepositsByBankAccount()` pobiera niezbędne informacje z bufora o nazwie `FixedDepositList`, a sama metoda `findFixedDepositsByBankAccount()` *nie* jest w rzeczywistości wywoływana.
4. W tabeli `FIXED_DEPOSIT_DETAILS` następuje zapisanie informacji o kolejnej lokacie założonej dla konta utworzonego w kroku 1. Lokata jest zakładana za pomocą wywołania metody

`createFixedDeposit()` klasy `FixedDepositService`. Teraz metoda `createFixedDeposit()` klasy `FixedDepositServiceImpl` jest oznaczona adnotacją `@CacheEvict` (patrz: listing 8.30), co powoduje usunięcie wszystkich elementów buforowanych w `FixedDepositList`.

5. Metoda `findFixedDepositsByBankAccount()` klasy `FixedDepositService` jest ponownie wywoływana. Tym razem nastąpi rzeczywiste wykonanie metody `findFixedDepositsByBankAccount()`, ponieważ poprzednie wywołanie metody `createFixedDeposit()` w kroku 4. spowodowało usunięcie wszystkich elementów z bufora `FixedDepositList`. Tym razem w konsoli znów zostanie wyświetlony komunikat o treści Wywołanie metody `findFixedDepositsByBankAccount`. Zwrócone przez metodę `findFixedDepositsByBankAccount()` informacje o lokatach będą buforowane w `FixedDepositList`, ponieważ wymieniona metoda jest oznaczona adnotacją `@Cacheable`.
6. Dla każdej lokaty pobranej w kroku 5. następuje wywołanie metody `getFixedDeposit()` klasy `FixedDepositService` (patrz: listing 8.31). Metoda `getFixedDeposit()` akceptuje identyfikator lokaty, a jej wartością zwrótną są pobrane z bazy danych informacje szczegółowe o danej lokacie. Metoda `getFixedDeposit()` jest oznaczona adnotacją `@CachePut`, co powoduje, że *zawsze* będzie wywoływana. Zwrócone przez metodę `getFixedDeposit()` informacje o lokacie będą buforowane w `FixedDeposit`.
7. Dla każdej lokaty pobranej w kroku 5. następuje wywołanie metody `getFixedDepositFromCache()` klasy `FixedDepositService` (patrz: listing 8.31). Metoda `getFixedDepositFromCache()` akceptuje identyfikator lokaty i zgłasza wyjątek `RuntimeException`. Metoda `getFixedDepositFromCache()` jest oznaczona adnotacją `@Cacheable` i będzie wykonywana tylko wtedy, gdy informacje o lokacie nie zostaną znalezione w buforze `FixedDeposit`. Ponieważ informacje o wszystkich lokatach są w kroku 6. buforowane przez metodę `getFixedDeposit()`, metoda `getFixedDepositFromCache()` nigdy nie zostanie wykonana.
8. Co 5 sekund egzemplarz `FixedDepositProcessorJob` (patrz: listing 8.17) sprawdza, czy w bazie danych pojawiły się informacje o jakiejś nowej lokacie. Jeżeli w bazie danych zostaną znalezione informacje o nowej lokacie, `FixedDepositProcessorJob` aktywuje tę lokatę, a następnie wysyła klientowi wiadomość e-mail z potwierdzeniem o założeniu nowej lokaty.

8.9. Podsumowanie

W tym rozdziale zajęliśmy się niektórymi z najczęściej używanych funkcji Springa. Zobaczyłeś, jak Spring może ułatwić wysyłanie i odbieranie komunikatów JMS, wysyłanie wiadomości e-mail, asynchroniczne wywoływanie metod komponentu, tworzenie harmonogramów wywoływania metod komponentu i buforowanie danych. W następnym rozdziale poznasz oferowaną przez Spring obsługę programowania zorientowanego aspektowego (ang. *aspect-oriented programming*, AOP).

Rozdział 9. Programowanie zorientowane aspektowo

9.1. Wprowadzenie

Programowanie zorientowane aspektowo (ang. *aspect-oriented programming*, AOP) to rodzaj podejścia programistycznego, w którym obowiązki rozproszone między wieloma klasami są hermetyzowane w oddzielnej klasie określanej mianem *aspektu*. Natomiast obowiązki rozproszone między wieloma klasami są nazywane *zagadnieniami przekrojowymi* (ang. *cross-cutting concern*). Rejestrowanie danych, zarządzanie transakcjami, buforowanie, zapewnienie bezpieczeństwa itd. to przykłady zadań przekrojowych.

Spring oferuje framework AOP używany wewnętrznie przez Spring do implementacji usług deklarycyjnych, na przykład zarządzania transakcjami (patrz: rozdział 7.) i buforowania (patrz: rozdział 8.). Zamiast stosować oferowany przez Spring framework AOP rozważ użycie AspectJ (<http://www.eclipse.org/aspectj/>) jako frameworku AOP dla tworzonej aplikacji. Ponieważ możliwości frameworku Spring AOP są wystarczające w większości sytuacji, a ponadto zapewnia on integrację z kontenerem Springa, w tym rozdziale skoncentrujemy się właśnie na użyciu frameworku AOP dostarczanego przez Spring.

Rozpoczynamy od przykładu użycia AOP.

9.2. Prosty przykład AOP

Przyjmujemy założenie, że na potrzeby audytu konieczne jest przechwytywanie argumentów przekazywanych w klasach, które zostały zdefiniowane w warstwie usługi w aplikacji MyBank. Proste podejście w zakresie rejestracji danych o argumentach metody polega na umieszczeniu w każdej metodzie logiki odpowiedzialnej za rejestrację danych. Takie rozwiązanie oznacza jednak, że każda metoda otrzymuje *dodatkowe* zadanie, jakim jest rejestracja szczegółów dotyczących argumentów metody. Ponieważ zadanie rejestracji danych o argumentach metody pojawia się w wielu klasach i metodach, przedstawia *zagadnienie przekrojowe*.

Przygotowanie obsługi zagadnienia przekrojowego za pomocą AOP wymaga wykonania następujących kroków:

- utworzenie klasy Javy (określanej mianem aspektu);
- dodanie do nowo utworzonej klasy implementacji zagadnienia przekrojowego;
- użycie wyrażenia regularnego w celu określenia metod, do których ma zastosowanie przygotowane zagadnienie przekrojowe.

W terminologii programowania zorientowanego aspektowego metody aspektu implementującego zagadnienia przekrojowe są określane mianem *rady* (ang. *advice*). Każda rada jest powiązana z *punktem przecięcia* (ang. *pointcut*) identyfikującym metody, do których ma zastosowanie rada. Metody, do których stosowana będzie rada, są nazywane *punktami złączenia* (ang. *join points*).

W oferowanym przez Spring frameworku AOP masz możliwość opracowania aspektu za pomocą *stylu adnotacji* (AspectJ) lub *stylu schematu* (XML). W przypadku stylu adnotacji udostępniane przez AspectJ adnotacje takie jak `@Aspect`, `@Pointcut` i `@Before` są używane do przygotowania aspektu. Natomiast w stylu schematu elementy schematu aop służą do konfiguracji komponentu Springa jako aspektu.

IMPORT **Rozdział09/ch09-simple-aop.** (Ten projekt pokazuje aplikację MyBank, w której framework Spring AOP został wykorzystany do rejestracji informacji o argumentach metod przekazanych metodom zdefiniowanym przez klasy w warstwie usługi aplikacji MyBank. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

W listingu 9.1 przedstawiamy przykład aspektu rejestrowania danych, który zapisuje informacje o argumentach przekazanych metodom usług w aplikacji MyBank.

Listing 9.1. Klasa `LoggingAspect`

Projekt: `ch09-simple-aop`

Położenie pliku: `src/main/java/sample/spring/chapter09/bankapp/aspects`

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {
    private Logger logger = Logger.getLogger(LoggingAspect.class);
    @Before(value = "execution(*
        sample.spring.chapter09.bankapp.service.*Service.*(..))")
    public void log(JoinPoint joinPoint) {
        logger.info("Wejście do metody "
            + joinPoint.getSignature().getName() + " klasy "
            + joinPoint.getTarget().getClass().getSimpleName());

        Object[] args = joinPoint.getArgs();
        for (int i = 0; i < args.length; i++) {
            logger.info("args[" + i + "] --> " + args[i]);
        }
    }
}
```

W listingu 9.1:

- Adnotacja `@Aspect` na poziomie typu oznacza, że klasa `LoggingAspect` jest *aspektem* AOP.
- Adnotacja `@Before` na poziomie metody oznacza, że metoda `log()` przedstawia *radę* stosowaną, *zanim* nastąpi wykonanie metod dopasowanych przez atrybut `value`. W podrozdziale 9.5 znajdziesz więcej informacji o różnych typach rad, które możesz zdefiniować.
- Atrybut `value` adnotacji `@Before` określa *wyrażenie punktu przecięcia*, które jest przez framework Spring AOP używane w celu identyfikacji metod (nazywanych *metodami docelowymi*) stosujących zdefiniowane rady. W podrozdziale 9.4 znajdziesz dokładne omówienie wyrażeń

punktu przecięcia. W tym miejscu możesz przyjąć następujące założenie: wyrażenie punktu przecięcia `execution(* sample.spring.chapter09.bankapp.service.*Service.*(..))` określa, że metoda `log()` klasy `LoggingAspect` ma zastosowanie dla *wszystkich* metod publicznych zdefiniowanych przez klasy (lub interfejsy) w pakiecie `sample.spring.chapter09.bankapp`.
→ `service` i których nazwa kończy się na `Service`.

- Argument `JoinPoint` metody `log()` przedstawia metodę docelową, w której zostanie zastosowana rada. Metoda `log()` używa egzemplarza `JoinPoint` w celu pobrania informacji o argumentach przekazanych metodzie docelowej. W listingu 9.1 metoda `getArgs()` egzemplarza `JoinPoint` jest wywoływana i pobiera argumenty przekazane metodzie docelowej.

Aspekt trzeba zarejestrować w kontenerze Springa, aby udostępniany przez Spring framework AOP wiedział o istnieniu aspektu. W listingu 9.1 klasa `LoggingAspect` została oznaczona adnotacją `@Component`, a więc jest automatycznie rejestrowana w kontenerze Springa.

W listingu 9.2 została przedstawiona klasa `BankApp` wywołująca metody klas `BankAccountService` → `ServiceImpl` (implementuje interfejs `BankAccountService`) i `FixedDepositServiceImpl` (implementuje interfejs `FixedDepositService`) aplikacji `MyBank`.

Listing 9.2. Klasa `BankApp`

Projekt: `ch09-simple-aop`

Położenie pliku: `src/main/java/sample/spring/chapter09/bankapp`

```
package sample.spring.chapter09.bankapp;
.....
public class BankApp {
    public static void main(String args[]) throws Exception {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");

        BankAccountService bankAccountService =
            context.getBean(BankAccountService.class);
        BankAccountDetails bankAccountDetails = new BankAccountDetails();
        bankAccountDetails.setBalanceAmount(1000);
        bankAccountDetails.setLastTransactionTimestamp(new Date());
        bankAccountService.createBankAccount(bankAccountDetails);

        FixedDepositService fixedDepositService =
            context.getBean(FixedDepositService.class);
        fixedDepositService.createFixedDeposit(new FixedDepositDetails(1, 1000, 12,
            "nazwa-uzytkownika@nazwa-domeny.pl"));
    }
}
```

W powyższym listingu przez metodę `main()` klasy `BankApp` wywoływane są metoda `createBankAccount()` klasy `BankAccountService` i metoda `createFixedDeposit()` klasy `FixedDepositService`. Jeżeli więc wywołasz metodę `main()` klasy `BankApp`, w konsoli zostaną wyświetlone następujące dane wyjściowe:

```
INFO LoggingAspect - Wejście do metody createBankAccount klasy BankAccountServiceImpl
INFO LoggingAspect - args[0] -->BankAccountDetails [accountId=0, balanceAmount=1000,
↳lastTransactionTimestamp=Sat Oct 27 16:48:11 IST 2012]
INFO BankAccountServiceImpl - Wywołanie metody createBankAccount
```

```
INFO LoggingAspect - Wejście do metody createFixedDeposit klasy FixedDepositServiceImpl
INFO LoggingAspect - args[0] -->id :1, wysokość lokaty : 1000.0, czas trwania : 12, e-mail :
↳nazwa-uzytkownika@nazwa-domeny.pl
INFO FixedDepositServiceImpl - Wywołanie metody createFixedDeposit
```

Powyższe dane wyjściowe pokazują, że metoda `log()` klasy `LoggingAspect` jest wykonywana przed metodą `createBankAccount()` klasy `BankAccountService` oraz metodą `createFixedDeposit()` klasy `FixedDepositService`.

W kontekście `LoggingAspect` zobaczmy teraz, jak działa oferowany przez Spring framework AOP.

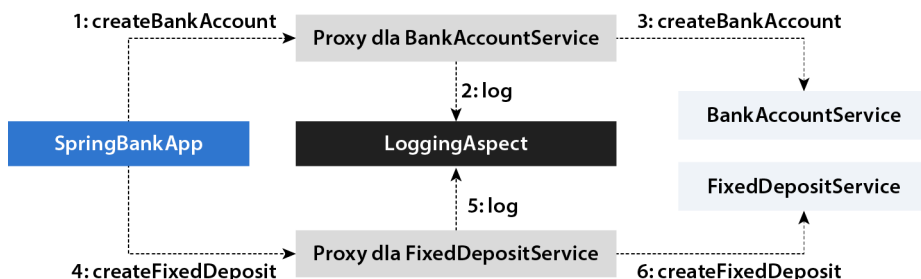


W projekcie *ch09-simple-aop* zdefiniowano w celu użycia stylu adnotacji `AspectJ` zależność od plików JAR o nazwach *spring-aop*, *aopalliance*, *aspectjrt* i *aspectjweaver*. Więcej informacji na ten temat znajdziesz w pliku *pom.xml* znajdującym się w projekcie *ch09-simple-aop*.

9.3. Oferowany przez Spring framework AOP

Framework AOP w Springu jest *oparty na proxy*, a więc *obiekt proxy* jest tworzony dla obiektów będących celem rady. Proxy to obiekt pośredniczący, wprowadzony przez framework AOP i znajdujący się między obiektem wywołującym i obiektem docelowym. W trakcie działania aplikacji wywołania do obiektu docelowego są przechwytywane przez proxy, a rady zdefiniowane dla metod docelowych są wykonywane przez proxy. We frameworku AOP obiekt docelowy jest egzemplarzem komponentu zarejestrowanym w kontenerze Springa.

Diagram na rysunku 9.1 pokazuje metodę `log()` klasy `LoggingAspect` (patrz: listing 9.1) zastosowaną do metod klas `BankAccountService` i `FixedDepositService` (patrz: listing 9.2).



Rysunek 9.1. Obiekt proxy jest odpowiedzialny za przechwytywanie wywołań metod w obiekcie docelowym oraz wykonywanie rad mających zastosowanie dla metody docelowej

Na rysunku 9.1 widać, że proxy jest tworzone dla obiektów `BankAccountService` i `FixedDepositService`. Proxy dla obiektu `BankAccountService` przechwytuje wywołania metody `createBankAccount()`, natomiast proxy dla obiektu `FixedDepositService` przechwytuje wywołania metody `createFixedDeposit()`. Proxy dla obiektu `BankAccountService` najpierw wykonuje metodę `log()` klasy `LoggingAspect`, a następnie metodę `createBankAccount()` klasy `BankAccountService`. Podobnie proxy dla obiektu `FixedDepositService` najpierw wykonuje metodę `log()` klasy `LoggingAspect`, a następnie metodę `createFixedDeposit()` klasy `FixedDepositService`.

Czas wykonania rady (takiej jak metoda `log()` aspektu `LoggingAspect`) zależy od *typu* rady. W stylu adnotacji `AspectJ` typ rady jest określany za pomocą adnotacji `AspectJ`. Na przykład adnotacja `@Before` oznacza wykonanie rady *przed* wywołaniem metody docelowej, natomiast adnotacja `@After` oznacza wykonanie rady *po* wywołaniu metody docelowej. Adnotacja `@Around` oznacza wykonanie rady zarówno *przed*, jak i *po* wykonaniu metody docelowej. Ponieważ metoda `log()` klasy `LoggingAspect` została oznaczona adnotacją `@Before`, metoda `log()` będzie wykonywana *przed* wykonaniem metody obiektu docelowego.

Zobaczmy teraz, jak framework AOP w Springu tworzy obiekt proxy.

Tworzenie proxy

Podczas użycia frameworku Spring AOP masz możliwość jawnego tworzenia proxy za pomocą `ProxyFactoryBean` (patrz: pakiet `org.springframework.aop.framework`) lub pozostawienia Springowi zadania automatycznego tworzenia proxy AOP, co jest określane mianem *autoproxying*.

Jeżeli do tworzenia aspektów chcesz zastosować adnotacje w stylu `AspectJ`, musisz włączyć obsługę tego rodzaju adnotacji, co wymaga użycia elementu `<aspectj-autoproxy>` schematu aop oferowanego przez Spring. Element `<aspectj-autoproxy>` nakazuje także frameworkowi Spring AOP automatyczne tworzenie proxy AOP dla obiektów docelowych. Listing 9.3 przedstawia przykład użycia elementu `<aspectj-autoproxy>` w projekcie *ch09-simple-aop*.

Listing 9.3. Plik `applicationContext.xml` przedstawiający użycie elementu `<aspectj-autoproxy>`

Projekt: *ch09-simple-aop*

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans .....
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation=".....http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">

  <context:component-scan base-package="sample.spring" />
  <aop:aspectj-autoproxy proxy-target-class="false" expose-proxy="true"/>
</beans>
```

Atrybut `proxy-target-class` elementu `<aspectj-autoproxy>` określa rodzaj proxy (oparte na `Javie SE` lub na `CGLIB`) tworzonych dla obiektów docelowych. Z kolei atrybut `expose-proxy` określa, czy proxy AOP będzie dostępne dla obiektu docelowego. Jeżeli wartością atrybutu `expose-proxy` jest `true`, wówczas metoda obiektu docelowego może uzyskać dostęp do proxy AOP przez wywołanie metody statycznej `currentProxy()` klasy `AopContext`.

Framework Spring AOP tworzy proxy oparte na `CGLIB` lub `Javie SE`. Jeżeli obiekt docelowy nie implementuje żadnego interfejsu, wówczas Spring AOP tworzy proxy oparte na `CGLIB`. Natomiast w przypadku, gdy obiekt docelowy implementuje jeden lub więcej interfejsów, Spring AOP tworzy proxy oparte na `Javie SE`. Jeżeli wartością atrybutu `proxy-target-class` elementu `<aspectj-autoproxy>` będzie `false`, wówczas Spring AOP tworzy proxy oparte na `Javie SE`, gdy

obiekt docelowy implementuje jeden bądź więcej interfejsów. W przypadku przypisania atrybutowi `proxy-target-class` wartości `true` Spring AOP tworzy proxy oparte na CGLIB, nawet jeśli obiekt implementuje jeden lub więcej interfejsów.



Począwszy od wersji Spring 3.2, klasy CGLIB znajdują się w pliku JAR o nazwie *spring-core*. Dlatego nie trzeba w aplikacji wyraźnie wskazywać pliku JAR CGLIB, aby umożliwić frameworkowi Spring AOP tworzenie dla obiektów docelowych proxy opartych na CGLIB.

Przechodzimy teraz do przypadku, w którym preferowane będzie przypisanie wartości `true` atrybutowi `expose-proxy` elementu `<aspectj-autoproxy>`.

IMPORT Rozdział 109/ch09-aop-proxy. (Ten projekt pokazuje aplikację *MyBank*, w której metoda `currentProxy()` klasy `AopProxy` została użyta przez metodę docelową do pobrania obiektu proxy AOP utworzonego przez framework AOP w Springu. Aby uruchomić tę aplikację, wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Atrybut `expose-proxy`

W listingu 9.4 została przedstawiona zmodyfikowana wersja klasy `BankAccountServiceImpl`, w której metoda `createBankAccount()` wywołuje metodę `isDuplicateAccount()` w celu sprawdzenia, czy w systemie istnieje już konto bankowe charakteryzujące się podanymi cechami.

Listing 9.4. Klasa `BankAccountServiceImpl`

```
@Service(value = "bankAccountService")
public class BankAccountServiceImpl implements BankAccountService {
    @Autowired
    private BankAccountDao bankAccountDao;

    @Override
    public int createBankAccount(BankAccountDetails bankAccountDetails) {
        if(!isDuplicateAccount(bankAccountDetails)) {
            return bankAccountDao.createBankAccount(bankAccountDetails);
        } else {
            throw new BankAccountAlreadyExistsException("Konto już istnieje.");
        }
    }

    @Override
    public boolean isDuplicateAccount(BankAccountDetails bankAccountDetails) { .... }
}
```

W powyższym listingu widać, że metoda `createBankAccount()` wywołuje metodę `isDuplicateAccount()` w celu sprawdzenia, czy konto bankowe o podanych parametrach istnieje już w systemie.

W tym momencie może pojawić się pytanie, czy metoda `log()` klasy `LoggingAspect` (patrz: listing 9.1) zostanie wykonana, gdy nastąpi wywołanie metody `isDuplicateAccount()` przez metodę `createBankAccount()`. Nawet pomimo dopasowania przez metodę `isDuplicateAccount()` wyrażenia punktu przecięcia określonego za pomocą adnotacji `@Before` metody `log()` klasy `LoggingAspect` (patrz: listing 9.1) metoda `log()` *nie* zostanie wywołana. Wynika to z faktu, że względem metod

wywoływanych przez obiekt docelowy *nie* jest stosowane proxy przez AOP. Ponieważ wywołanie metody nie przechodzi przez obiekt proxy AOP, jakkolwiek rada powiązana z metodą docelową *nie* zostanie wykonana.

Aby zagwarantować, że wywołanie metody `isDuplicateAccount()` będzie do obiektu docelowego przeprowadzone poprzez proxy AOP, w metodzie `createBankAccount()` należy pobrać obiekt proxy AOP, a następnie wywołać metodę `isDuplicateAccount()` w tym obiekcie. Listing 9.5 przedstawia przykład pobrania obiektu proxy AOP w metodzie `createBankAccount()`.

Listing 9.5. Klasa `BankAccountServiceImpl`

Projekt: *ch09-aop-proxy*

Położenie pliku: *src/main/java/sample/spring/chapter09/bankapp/service*

```
package sample.spring.chapter09.bankapp.service;

import org.springframework.aop.framework.AopContext;
.....
@Service(value = "bankAccountService")
public class BankAccountServiceImpl implements BankAccountService {
    .....
    @Override
    public int createBankAccount(BankAccountDetails bankAccountDetails) {
        // Pobranie proxy i wywołanie za jego pomocą metody isDuplicateAccount().
        boolean isDuplicateAccount =
            ((BankAccountService)AopContext.currentProxy()).
                isDuplicateAccount(bankAccountDetails);

        if(!isDuplicateAccount) { ..... }
        .....
    }

    @Override
    public boolean isDuplicateAccount(BankAccountDetails bankAccountDetails) { ..... }
}
```

W powyższym listingu wywołanie metody `currentProxy()` klasy `AopContext` zwraca proxy AOP, które wykonuje wywołanie metody `createBankAccount()`. Jeżeli metoda `createBankAccount()` nie zostanie wykonana za pomocą frameworku Spring AOP lub wartością atrybutu `expose-proxy` elementu `<aspectj-autoproxy>` jest `false`, wówczas wywołanie metody `currentProxy()` zakończy się zgłoszeniem wyjątku `java.lang.IllegalStateException`. Ponieważ proxy AOP implementuje ten sam interfejs co obiekt docelowy, w omawianym listingu zwrócone przez metodę `currentProxy()` proxy AOP jest rzutowane na typ `BankAccountService` i następuje wywołanie metody `isDuplicateAccount()` klasy `BankAccountService`.

Jeżeli przejdziesz do projektu *ch09-aop-proxy* i wykonasz metodę `main()` klasy `BankApp`, zauważysz, że metoda `log()` klasy `LoggingAspect` jest wykonywana, gdy nastąpi wywołanie metody `isDuplicateAccount()` przez metodę `createBankAccount()`.

Teraz przechodzimy do dokładnego omówienia wyrażen punktu przecięcia.

9.4. Wyrażenia punktu przecięcia

Podczas użycia frameworku Spring AOP wyrażenie punktu przecięcia identyfikuje *punkty połączeń*, do których zastosowanie mają rady. W przypadku Springa AOP punkt połączenia *zawsze* jest metodą komponentu. Jeżeli chcesz zastosować rady dla właściwości, konstruktorów, metod niepublicznych oraz obiektów niebędących komponentami Springa, powinieneś użyć AspectJ zamiast frameworku Spring AOP. Jeśli masz zamiar opracować aspekty za pomocą adnotacji stylu AspectJ, wyrażenie punktu przecięcia możesz podać, używając adnotacji AspectJ @Pointcut lub adnotacji AspectJ @Before, @After itd. określających typ rady.

Wyrażenia punktów przecięcia używają tak zwanych *desygnatorów punktu przecięcia*, takich jak execution, args, within, this itd., w celu dopasowania metod, do których ma być zastosowana rada. Na przykład w listingu 9.1 adnotacja @Before zawiera desygnator execution, a tym samym wyszukuje metody, do których zastosowanie będzie miała metoda log() klasy LoggingAspect.

Teraz zobaczysz, jak wyrażenia punktów przecięcia są określone za pomocą adnotacji @Pointcut.

IMPORT **Rozdział 109/ch09-aop-pointcut.** (Ten projekt pokazuje aplikację MyBank, w której użyto adnotacji AspectJ @Pointcut w celu określenia wyrażenia punktu przecięcia. Aby uruchomić tę aplikację, wystarczy wywołać metodę main() w klasie BankApp projektu).

Adnotacja @Pointcut

Atrybut value adnotacji @Pointcut określa wyrażenie punktu przecięcia. Aby użyć adnotacji @Pointcut, utwórz *pustą* metodę i oznacz ją adnotacją @Pointcut. Pusta metoda *musi* być zdefiniowana w taki sposób, aby zwracała void. Rada odwołująca się do metody oznaczonej adnotacją @Pointcut będzie zastosowana w metodach dopasowanych przez wyrażenie punktu przecięcia wskazane przez adnotację @Pointcut.



Użycie adnotacji @Pointcut okazuje się szczególnie użyteczne, gdy wyrażenie punktu przecięcia jest współdzielone przez wiele rad w tym samym aspekcie lub w różnych aspektach.

W listingu 9.6 została przedstawiona zmodyfikowana wersja klasy LoggingAspect (patrz: listing 9.1) używająca adnotacji @Pointcut.

Listing 9.6. Klasa LoggingAspect

Projekt: ch09-aop-pointcuts

Położenie pliku: src/main/java/sample/spring/chapter09/bankapp/aspects

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
@Component
public class LoggingAspect {
```

```

@Pointcut(value = "execution(*
    sample.spring.chapter09.bankapp.service.*Service.*(..)")
private void invokeServiceMethods() { }

@Before(value = "invokeServiceMethods()")
public void log(JoinPoint joinPoint) {
    logger.info("Wejście do metody "
        + joinPoint.getSignature().getName() + " klasy "
        + joinPoint.getTarget().getClass().getSimpleName();
        ....
    }
}

```

W powyższym listingu metoda `invokeServiceMethods` została oznaczona adnotacją `@Pointcut`, a atrybut `value` adnotacji `@Before` odwołuje się do metody `invokeServiceMethods()`. Oznacza to, że metoda `log()` będzie zastosowana względem metod dopasowanych przez wyrażenie punktu przecięcia określone za pomocą adnotacji `@Pointcut` w metodzie `invokeServiceMethods()`.

Ponieważ `execution` i `args` są najczęściej używanymi desygntorami punktu przecięcia, teraz dokładnie je przeanalizujemy.

Desygntory `execution` i `args`

Desygntor `execution` punktu przecięcia ma następujący format:

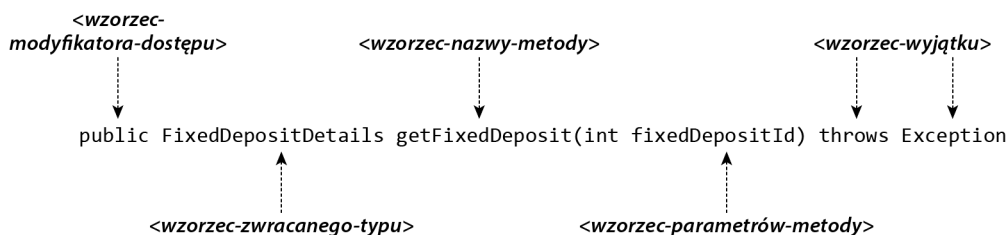
```

execution(<wzorzec-modyfikatora-dostępu> <wzorzec-zwracanego-typu> <wzorzec-deklarowanego-typu>
↳ <wzorzec-nazwy-metody>(<wzorzec-parametrów-metody>) <wzorzec-wyjątku>)

```

Jeżeli porównasz wyrażenie `execution` z deklaracją metody, zauważysz występujące między nimi podobieństwa.

Na rysunku 9.2 pokazujemy, jak poszczególne fragmenty wyrażenia `execution` można mapować na deklarację metody.



Rysunek 9.2. Mapowanie poszczególnych fragmentów wyrażenia `execution` na różne części deklaracji metody

Framework Spring AOP dopasowuje poszczególne fragmenty wyrażenia `expression` do różnych części deklaracji metody (patrz: rysunek 9.2) w celu znalezienia metod, do których ma zastosowanie dana rada. Na rysunku nie pokazano `<wzorca-deklarowanego-typu>`, ponieważ jest on używany tylko wtedy, gdy zachodzi potrzeba odwołania się do metod, które znajdują się w określonym pakiecie lub są określonego typu.

W tabeli 9.1 wymieniamy poszczególne fragmenty tworzące wyrażenie `execution`.

Tabela 9.1. Elementy tworzące wyrażenie `execution`

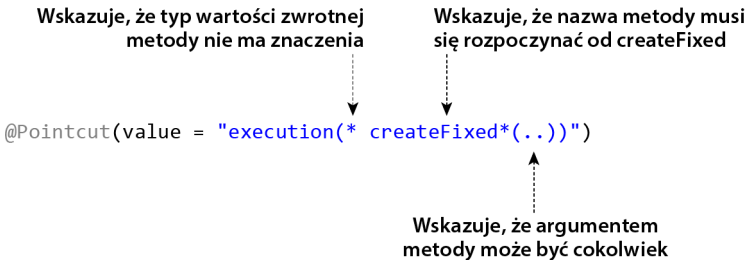
Fragment wyrażenia	Opis
wzorzec-modyfikatora-dostępu	Określa modyfikator dostępu metody docelowej. W Springu AOP jedyną wartością, jaką można określić dla tej części wyrażenia, to <code>public</code> . Ta część wyrażenia <code>execution</code> jest <i>opcjonalna</i> .
wzorzec-zwracanego-typu	Określa w pełni kwalifikowaną nazwę typu zwrótnego metody docelowej. Wartość <code>*</code> oznacza, że typ zwrótny metody docelowej nie ma znaczenia.
wzorzec-deklarowanego-typu	Określa w pełni kwalifikowaną nazwę typu zwrótnego zawierającego metodę docelową. Ta część wyrażenia <code>execution</code> jest <i>opcjonalna</i> . Wartość <code>*</code> oznacza, że wszystkie typy (klasy i interfejsy) w aplikacji są uwzględniane przez wyrażenie punktu przecięcia.
wzorzec-nazwy-metody	Określa wzorzec nazwy metody. Na przykład wartość <code>save*</code> oznacza, że metody o nazwach rozpoczynających się od <code>save</code> będą celem dla danej rady.
wzorzec-parametrów-metody	Określa wzorzec parametru metody. Jeżeli wartością będzie <code>(..)</code> , oznacza to, że metoda docelowa może zawierać dowolną liczbę argumentów lub żaden.
wzorzec-wyjątku	Określa wyjątek (lub wyjątki) zgłaszany przez metodę docelową. Ta część wyrażenia <code>execution</code> jest <i>opcjonalna</i> .

Desygnator `args` określa argumenty, jakie muszą być akceptowane przez metodę docelową *w trakcie działania aplikacji*. Na przykład jeśli wyrażenie punktu przecięcia przeznaczone jest do wyszukiwania metod akceptujących egzemplarz `java.util.List`, wówczas desygnator `args` ma postać `args(java.util.List)`. W dalszej części podrozdziału zobaczysz, jak desygnator `args` może być użyty do udostępniania radzie argumentów przekazywanych metodzie docelowej.

Teraz przeanalizujemy kilka przykładów wyrażań punktu przecięcia używających desygnatorów `execution` i `args`.

Przykład 1.

Na rysunku 9.3 zostało pokazane wyrażenie używające wzorca nazwy metody.

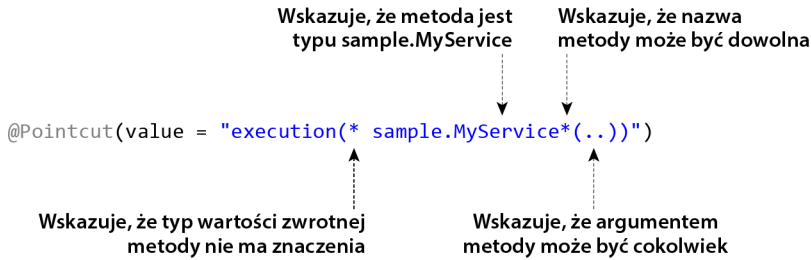


Rysunek 9.3. Wyrażenie punktu przecięcia używające wzorca nazwy metody

Przedstawione wyrażenie punktu przecięcia powoduje dopasowanie metod o nazwach rozpoczynających się od `createFixed`. Jako typ wartości zwrótniej podano `*`, a więc metoda docelowa może zwracać wartość dowolnego typu. Zapis `(..)` oznacza, że metoda może akceptować zero lub więcej argumentów.

Przykład 2.

Na rysunku 9.4 zostało pokazane wyrażenie określające typ metody.

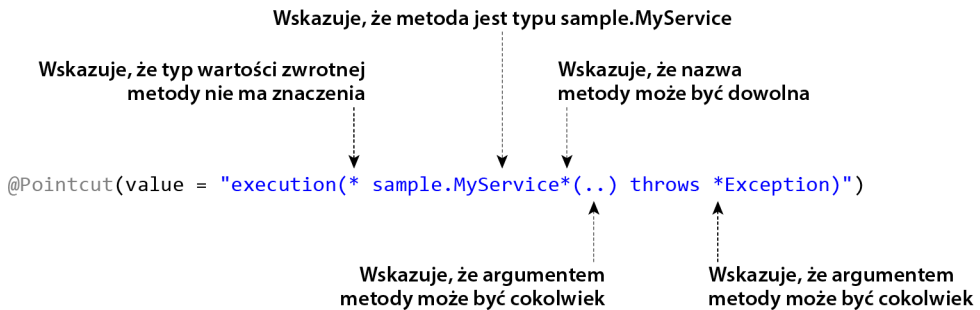


Rysunek 9.4. Wyrażenie punktu przecięcia wskazujące typ (klasa lub interfejs) zawierający metodę docelową

Przedstawione wyrażenie punktu przecięcia powoduje dopasowanie metod typu `MyService` zdefiniowanych w pakiecie `sample`.

Przykład 3.

Na rysunku 9.5 zostało pokazane wyrażenie określające wzorzec metody.

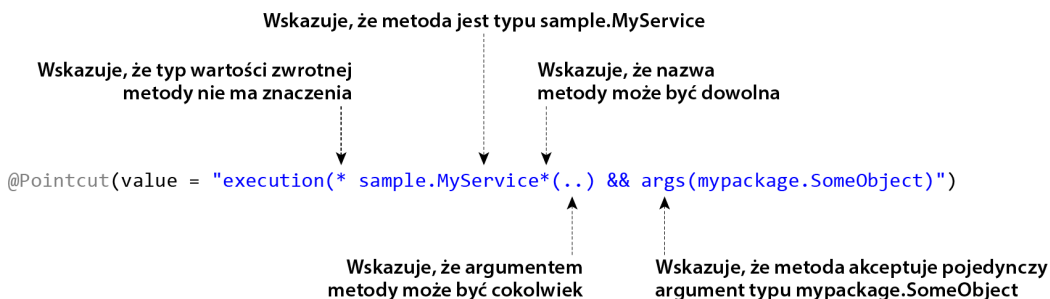


Rysunek 9.5. Wyrażenie punktu przecięcia wskazujące wzorzec metody

Przedstawione wyrażenie punktu przecięcia powoduje dopasowanie metod typu `sample.MyService` zawierających klauzulę `throws`.

Przykład 4.

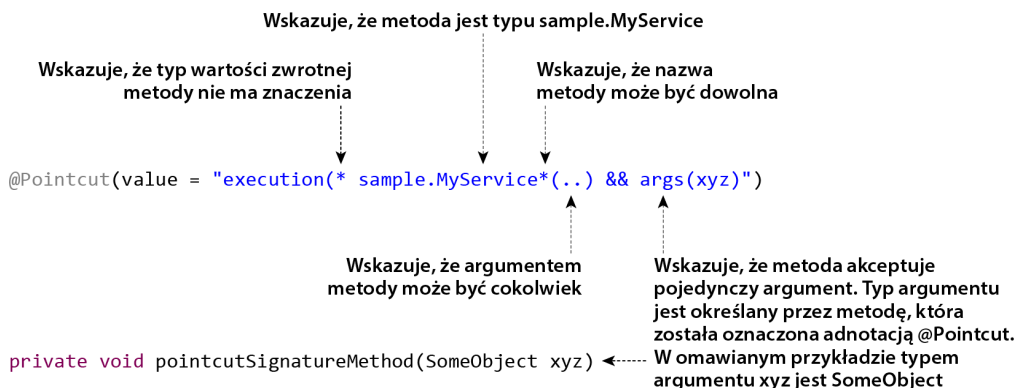
Na rysunku 9.6 zostało pokazane wyrażenie określające egzemplarz obiektu przekazywany metodzie docelowej.



Rysunek 9.6. Wyrażenie punktu przecięcia wskazujące egzemplarz obiektu przekazywany metodzie docelowej

Przedstawione wyrażenie punktu przecięcia pokazuje użycie desygnatorów `execution` i `args`. Jeżeli chcesz jednocześnie użyć więcej niż jednego desygnatora, możesz je łączyć za pomocą operatorów `&&` i `||`, tworząc w ten sposób skomplikowane wyrażenia punktu przecięcia. Powyższe wyrażenie dopasowuje metody zdefiniowane w pakiecie `sample.MyService` akceptujące egzemplarz `SomeObject`. Operator `&&` w omawianym wyrażeniu punktu przecięcia oznacza, że metoda docelowa *musi* być dopasowana w wyrażeniach określonych przez desygnatory `execution` i `args`.

Jeżeli rada ma uzyskać dostęp do jednego lub więcej argumentów przekazanych metodzie docelowej, wówczas nazwy argumentów należy podać w wyrażeniu `args`, jak pokazuje rysunek 9.7.



Rysunek 9.7. Desygnator `args` określa argumenty metody docelowej, które muszą być dostępne dla rady

W pokazanym na rysunku 9.7 wyrażeniu punktu przecięcia desygnator `args` określa, że metoda docelowa *musi* akceptować argument typu `SomeObject`, a sam argument jest dostępny dla rady za pomocą parametru `xyz`. Przechodzimy teraz do praktycznego przykładu użycia tej funkcji w celu przekazania argumentów radzie.

Przekazanie do rady argumentów metody docelowej

W listingu 9.7 została przedstawiona zmodyfikowana wersja klasy `LoggingAspect`, której metoda `log()` jest wykonywana tylko wtedy, gdy argument przekazany metodzie docelowej jest egzemplarzem `FixedDepositDetails`, a egzemplarz `FixedDepositDetails` również jest dostępny dla metody `log()`.

Listing 9.7. Klasa *LoggingAspect* przedstawiająca przekazanie radzie argumentów metody docelowej

```
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
@Component
public class LoggingAspect {
    ....
    @Pointcut(value =
        "execution(* sample.spring.chapter09.bankapp.service.*Service.*(..))
        && args(FixedDepositDetails) ")

    private void invokeServiceMethods(FixedDepositDetails FixedDepositDetails) {
    }

    @Before(value = "invokeServiceMethods(FixedDepositDetails)")
    public void log(JoinPoint joinPoint, FixedDepositDetails FixedDepositDetails) {
        ....
    }
}
```

W powyższym listingu wyrażenie `args` określa, że egzemplarz `FixedDepositDetails` przekazywany metodzie docelowej jest dostępny dla metody `log()` — czyli rady — za pomocą parametru `FixedDepositDetails`. Ponieważ wyrażenie `args` dostarcza metodę `log()` wraz z egzemplarzem obiektu `FixedDepositDetails`, metoda `log()` została zmodyfikowana w taki sposób, aby akceptowała dodatkowy argument typu `FixedDepositDetails`.

Desygatory punktu przecięcia, takie jak `execution`, `args`, `within`, `this`, `target` itd., są zdefiniowane przez AspectJ. Spring AOP definiuje charakterystyczny dla tego frameworku desygator `bean`. Warto więc przeanalizować pokrótce desygator `bean` punktu przecięcia.

Desygator `bean`

Desygator `bean` służy do ograniczenia metod docelowych do określonego identyfikatora (lub nazwy) komponentu. Istnieje możliwość podania dokładnego identyfikatora lub nazwy komponentu, ewentualnie można wskazać wzorzec. Spójrz teraz na kilka przykładów użycia desygatora `bean`.

Przykład 1.

Na rysunku 9.8 został pokazany desygator `bean` wskazujący nazwę komponentu, którego metody są celem dla zdefiniowanej rady.

Nazwa lub identyfikator komponentu, którego
metody są celem dla zdefiniowanej rady

↓

```
@Pointcut(value = "bean(someBean)")
```

Rysunek 9.8. Desygator `bean` określa nazwę komponentu zawierającego metody, w których będzie stosowana rada

Powyższe wyrażenie dopasuje metody zdefiniowane przez komponent o nazwie `someBean`.

Przykład 2.

Na rysunku 9.9 został pokazany desygator bean określający, że rada będzie zastosowana w metodach komponentów, których nazwa rozpoczyna się od `someBean`.

Rada będzie zastosowana w metodach komponentów,
których nazwa rozpoczyna się od `someBean`

↓

```
@Pointcut(value = "bean(someBean*)")
```

Rysunek 9.9. *Desygator bean wskazujący, że rada będzie zastosowana w metodach komponentów, których nazwa rozpoczyna się od `someBean`*

Powyższe wyrażenie określa, że rada będzie zastosowana w metodach komponentów, których nazwa rozpoczyna się od `someBean`.



Podobnie jak w przypadku innych desygatorów punktu przecięcia, istnieje możliwość połączenia desygatora bean z innymi za pomocą operatorów `&&` i `||`. Takie połączenie umożliwia tworzenie skomplikowanych wyrażeń.

Teraz przechodzimy do desygatorów, które przeprowadzają dopasowanie na podstawie adnotacji.

Desygatory oparte na adnotacjach

AspectJ dostarcza desygatory punktów przecięcia takie jak `@annotation`, `@target`, `@within` i `@args`, których można użyć w połączeniu ze Springiem AOP do wyszukania metod docelowych. Spójrz na kilka przykładów użycia tego rodzaju desygatorów.

Przykład 1.

Na rysunku 9.10 pokazujemy przykład użycia desygatora `@annotation`.

Rada będzie zastosowana dla metod oznaczonych
oferowaną przez Spring adnotacją `@Cacheable`

↓

```
@Pointcut(value = "@annotation(org.springframework.cache.annotation.Cacheable)")
```

Rysunek 9.10. *Przedstawiony desygator `@annotation` określa, że rada będzie zastosowana dla metod oznaczonych oferowaną przez Spring adnotacją `@Cacheable`*

Powyższe wyrażenie spowoduje dopasowanie metod oznaczonych oferowaną przez Spring adnotacją `@Cacheable`.

Przykład 2.

Na rysunku 9.11 pokazujemy przykład użycia desygatora `@target`.

Rada będzie zastosowana dla metod znajdujących się w obiekcie oznaczonym oferowaną przez Spring adnotacją @Component

`@Pointcut(value = "@target(org.springframework.stereotype.Component)")`

Rysunek 9.11. Przedstawiony desygnotor `@target` określa, że rada będzie zastosowana dla metod oznaczonych oferowaną przez Spring adnotacją `@Component`

Powyższe wyrażenie spowoduje dopasowanie metod znajdujących się w obiekcie oznaczonym oferowaną przez Spring adnotacją `@Component`.

W tym podrozdziale przeanalizowaliśmy wybrane desygnotory zdefiniowane przez AspectJ. Trzeba koniecznie zwrócić uwagę na fakt, że *nie* wszystkie desygnotory zdefiniowane przez AspectJ są obsługiwane przez framework Spring AOP. Jeżeli w wyrażeniu punktu przecięcia użyjesz nieobsługiwanego desygnotora, wówczas Spring AOP zgłosi wyjątek `java.lang.IllegalArgumentException`. Na przykład użycie desygnotorów `call`, `set` i `get` w wyrażeniu punktu przecięcia spowoduje, że Spring AOP zgłosi wyjątek `java.lang.IllegalArgumentException`.

Przechodzimy teraz do analizy różnych typów rad oraz sposobów ich tworzenia.

9.5. Typy rad

Dotychczas w tym rozdziale przedstawiliśmy przykłady rad typu *before*. Tego rodzaju rada jest tworzona przez określenie metody aspektu adnotacją `@Before` (patrz: listingi 9.1, 9.6 i 9.7). Inne typy rad, które można utworzyć, to *after*, *after returning*, *after throwing* i *around*.

IMPORT **Rozdział09/ch09-aop-advice**. (Ten projekt pokazuje aplikację MyBank, w której użyto różnego rodzaju rad. Aby uruchomić tę, aplikację wystarczy wywołać metodę `main()` w klasie `BankApp` projektu).

Przechodzimy teraz do przeanalizowania funkcji i sposobów tworzenia różnego rodzaju rad.

Rada before

Rada *before* jest wykonywana przed wykonaniem metody docelowej. Jeżeli rada *before* nie zgłosi wyjątku, wówczas metoda docelowa *zawsze* będzie wywołana. Istnieje możliwość kontrolowania, czy metoda docelowa ma zostać wykonana — do tego celu używa się rady *around*, którą omówimy w dalszej części rozdziału.

Jak wcześniej wspomnieliśmy, adnotacja AspectJ `@Before` jest używana do wskazania rady jako typu *before*.

Metoda oznaczona adnotacją `@Before` może zdefiniować jej pierwszy argument jako typu `JoinPoint`. Argument typu `JoinPoint` można użyć wewnątrz rady do pobrania informacji o metodzie docelowej. Na przykład w listingu 9.1 pokazano, że egzemplarz `JoinPoint` może być użyty do pobrania nazwy klasy obiektu docelowego oraz argumentów przekazanych metodzie docelowej.

Rada after returning

Rada *after returning* jest wykonywana *po* zakończeniu działania metody docelowej. Warto zwrócić uwagę, że nie zostanie wykonana, jeśli metoda docelowa zgłosi wyjątek. Rada ta jest oznaczona adnotacją AspectJ `@AfterReturning`. Może uzyskać dostęp do wartości zwrotnej metody docelowej, a następnie zmodyfikować ją przed zwrotem tej wartości do obiektu wywołującego radę.

Klasa `SampleAspect` projektu *ch09-aop-advice* przedstawia aspekt AOP. W listingu 9.8 pokazano, że klasa definiuje radę *after returning* wyświetlającą wartość zwróconą przez metodę `createBankAccount()` klasy `BankAccountService`.

Listing 9.8. Klasa `SampleAspect` przedstawiająca użycie rady *after returning*

Projekt: *ch09-aop-advice*

Położenie pliku: *src/main/java/sample/spring/chapter09/bankapp/aspects*

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.annotation.AfterReturning;
.....
@Aspect
public class SampleAspect {
    private Logger logger = Logger.getLogger(SampleAspect.class);

    @Pointcut(value = "execution(*
        sample.spring..BankAccountService.createBankAccount(..)")
    private void createBankAccountMethod() {}

    @AfterReturning(value = "createBankAccountMethod()", returning = "aValue")
    public void afterReturningAdvice(JoinPoint joinPoint, int aValue) {
        logger.info("Wartość zwrócona przez metodę "
            + joinPoint.getSignature().getName()
            + " wynosi " + aValue);
    }
    .....
}
```

W powyższym listingu metoda `afterReturningAdvice()` przedstawia radę typu *after returning*. Wyrażenie punktu przecięcia określone przed adnotacją `@Pointcut` definiuje punkt złączenia z metodą `createBankAccount()` klasy `BankAccountService`. Dwie kropki (`..`) w wyrażeniu `execution` wskazują, że pakiet `sample.spring` i znajdujące się w nim pakiety będą przeszukiwane w celu znalezienia typu `BankAccountService`.

W listingu 9.8 metoda `afterReturningAdvice()` klasy `SampleAspect` jest wywoływana po metodzie `createBankAccount()` klasy `BankAccountService`. Atrybut `returning` adnotacji `@AfterReturning` określa nazwę, pod którą wartość zwrotna metody docelowej będzie dostępna dla rady. W omawianym listingu wartość zwrotna metody `createBankAccount()` jest dostępna dla metody `afterReturningAdvice()` za pomocą argumentu `aValue`. Typ argumentu `aValue` został ustalony jako `int`, ponieważ metoda `createBankAccount()` zwraca wartość w postaci liczby całkowitej. Trzeba w tym miejscu dodać, że po zdefiniowaniu atrybutu `returning` rada będzie stosowana jedynie do metod

zwracających wartość wskazanego typu. Jeżeli rada *after returning* zostanie zastosowana w metodach zwracających różne typy wartości (w tym także void), wówczas typ argumentu wartości zwrotnej można określić jako Object.

Jak widać w listingu 9.8, metoda oznaczona adnotacją @AfterReturning może zdefiniować jej pierwszy argument jako typu JoinPoint, aby uzyskać dostęp do informacji o metodzie docelowej.

Rada after throwing

Rada *after throwing* jest wykonywana, gdy metoda docelowa zgłosi wyjątek. Może uzyskać dostęp do wyjątku zgłaszanego przez metodę docelową. Rada ta jest oznaczana adnotacją AspectJ @After ➔Throwing.

W listingu 9.9 przedstawiamy przykład rady *after throwing* wykonywanej po zgłoszeniu wyjątku przez metodę docelową.

Listing 9.9. Klasa SampleAspect przedstawiająca użycie rady after throwing

Projekt: ch09-aop-advice

Położenie pliku: src/main/java/sample/spring/chapter09/bankapp/aspects

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.annotation.AfterThrowing;
.....
@Aspect
public class SampleAspect {
    private Logger logger = Logger.getLogger(SampleAspect.class);
    .....
    @Pointcut(value = " execution(* sample.spring..FixedDepositService.*(..)) ")
    private void exceptionMethods() {}
    .....
    @AfterThrowing(value = "exceptionMethods()", throwing = "exception")
    public void afterThrowingAdvice(JoinPoint joinPoint, Throwable exception) {
        logger.info("Wyjątek zgłoszony przez " + joinPoint.getSignature().getName()
            + " Typ zgłoszonego wyjątku : " + exception);
    }
}
```

W powyższym listingu metoda afterThrowingAdvice() klasy SampleAspect przedstawia radę *after throwing*. Metoda afterThrowingAdvice() jest wykonywana, gdy zostanie zgłoszony wyjątek przez dowolną metodę obiektu FixedDepositService. W omawianym listingu atrybut throwing adnotacji @AfterThrowing określa nazwę, pod jaką wyjątek zgłoszony przez metodę docelową będzie dostępny dla metody afterThrowingAdvice(). Ponieważ wartością atrybutu throwing jest exception, wyjątek będzie przekazywany metodzie afterThrowingAdvice() za pomocą argumentu o nazwie exception. Zwróć uwagę na fakt, że typ argumentu exception to java.lang.Throwable. Oznacza to, że metoda afterThrowingAdvice() będzie wykonywana dla wszystkich wyjątków zgłoszonych przez metodę docelową. Jeżeli chcesz, aby metoda afterThrowingAdvice() była wykonywana jedynie po zgłoszeniu przez metodę docelową wyjątku określonego typu, zmień typ argumentu exception.

Na przykład jeśli metoda `afterThrowingAdvice()` ma być wykonywana jedynie po zgłoszeniu przez metodę docelową wyjątku `java.lang.IllegalStateException`, wówczas należy podać `java.lang.↳IllegalStateException` jako typ argumentu `exception`.

Jak widać w listingu 9.9, metoda oznaczona adnotacją `@AfterThrowing` może zdefiniować jej pierwszy argument jako typu `JoinPoint`, aby uzyskać dostęp do informacji o metodzie docelowej.

Rada after

Rada *after* jest wykonywana po metodzie docelowej niezależnie od sposobu jej zakończenia — normalnie lub zgłoszeniem wyjątku. Rada ta jest oznaczana adnotacją `AspectJ @After`.

W listingu 9.10 przedstawiamy przykład rady *after* wykonywanej dla metody `createBankAccount()` klasy `BankAccountService` oraz dla metod zdefiniowanych przez interfejs `FixedDepositService`.

Listing 9.10. Klasa `SampleAspect` przedstawiająca użycie rady *after*

Projekt: `ch09-aop-advice`

Położenie pliku: `src/main/java/sample/spring/chapter09/bankapp/aspects`

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.annotation.After;
.....
@Aspect
public class SampleAspect {
    private Logger logger = Logger.getLogger(SampleAspect.class);
    @Pointcut(value = "execution(*
        sample.spring..BankAccountService.createBankAccount(..)")
    private void createBankAccountMethod() {}

    @Pointcut(value = "execution(* sample.spring..FixedDepositService.*(..)")
    private void exceptionMethods() {}
    .....
    @After(value = "exceptionMethods() || createBankAccountMethod()")
    public void afterAdvice(JoinPoint joinPoint) {
        logger.info("Rada after wykonana dla metody "
            + joinPoint.getSignature().getName());
    }
}
```

W powyższym listingu metoda `afterAdvice()` klasy `SampleAspect` przedstawia radę *after*. Metoda `afterAdvice()` jest wykonywana dopiero po zakończeniu działania metody docelowej. Zwróć uwagę na fakt, że atrybut `value` adnotacji `@After` używa operatora `||` w celu połączenia wyrażeń punktu przecięcia przedstawianych przez metody `createBankAccountMethod()` i `exceptionMethods()`. W ten sposób powstaje nowe wyrażenie punktu przecięcia.

Jak widać w listingu 9.10, metoda oznaczona adnotacją `@After` może zdefiniować jej pierwszy argument jako typu `JoinPoint`, aby uzyskać dostęp do informacji o metodzie docelowej.

Rada around

Rada *around* jest wykonywana zarówno *przed*, jak i *po* wykonaniu metody docelowej. W przeciwieństwie do innych rad może kontrolować, czy metoda docelowa w ogóle zostanie wykonana. Rada ta jest oznaczona adnotacją `AspectJ @Around`.

W listingu 9.11 przedstawiamy przykład rady *around* zdefiniowanej przez klasę `SampleAspect` w projekcie *ch09-aop-advice*.

Listing 9.11. Klasa `SampleAspect` przedstawiająca użycie rady *around*

Projekt: *ch09-aop-advice*

Położenie pliku: *src/main/java/sample/spring/chapter09/bankapp/aspects*

```
package sample.spring.chapter09.bankapp.aspects;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.springframework.util.StopWatch;
.....
@Aspect
public class SampleAspect {
    .....
    @Around(value = "execution(* sample.spring.*Service.*(..))")
    public Object aroundAdvice(ProceedingJoinPoint pjp) {
        Object obj = null;
        StopWatch watch = new StopWatch();
        watch.start();
        try {
            obj = pjp.proceed();
        } catch (Throwable throwable) {
            // Przeprowadzenie wszelkich niezbędnych działań.
        }
        watch.stop();
        logger.info(watch.prettyPrint());
        return obj;
    }
}
```

W powyższym listingu metoda `aroundAdvice()` przedstawia radę *around*. Argument `ProceedingJoinPoint` metody `aroundAdvice()` jest przeznaczony do kontrolowania, czy metoda docelowa w ogóle zostanie wykonana. Trzeba koniecznie zwrócić uwagę na fakt, że argument `ProceedingJoinPoint` *musi* być pierwszym argumentem przekazany radzie *around*. Kiedy wykonujesz metodę `proceed()` klasy `ProceedingJoinPoint`, metoda docelowa będzie wywołana. Oznacza to, że jeśli nie wywołasz metody `proceed()` klasy `ProceedingJoinPoint`, wówczas metoda docelowa *nie* zostanie wykonana. Po przekazaniu `Object[]` metodzie `proceed()` wartości znajdujące się w `Object[]` są przekazywane jako argumenty metody docelowej. Gdy rada *nie* wywoła metody docelowej, sama rada może zwrócić wartość.

Ponieważ metoda docelowa jest wywoływana jedynie po wywołaniu metody `proceed()` klasy `ProceedingJoinPoint`, rada *around* pozwala na wykonywanie zadań *przed* wywołaniem metody docelowej i *po* nim oraz na współdzielenie informacji między wspomnianymi zadaniami. W listingu 9.11 metoda `aroundAdvice()` rejestruje ilość czasu niezbędnego do wykonania metody docelowej.

Metoda `aroundAdvice()` uruchamia stoper (przedstawiany przez obiekt `StopWatch` klasy Springa) przed wywołaniem metody `proceed()` klasy `ProceedingJoinPoint`. Zatrzymanie stopera następuje po wywołaniu metody `proceed()` klasy `ProceedingJoinPoint`. Metoda `prettyPrint()` klasy `StopWatch` jest używana do wyświetlenia czasu potrzebnego na wykonanie metody docelowej.

Jeżeli zachodzi potrzeba modyfikacji wartości zwróconej przez metodę docelową, należy wartość zwrótną metody `proceed()` klasy `ProceedingJoinPoint` rzutować na typ zwrrotny metody docelowej, a następnie ją zmodyfikować. Metoda wywołująca będzie widziała wartość zwracaną przez metodę. Dlatego typ wartości zwrótniej metody rady *trzeba* określić jako `Object` lub jako typ wartości zwracanej przez metodę docelową. Metoda rady ma możliwość zwrócenia wartości zwracanej przez metodę docelową lub zwrócenia zupełnie innej wartości. Na przykład zamiast wywoływać metodę docelową rada *around* może przeanalizować argument (lub argumenty) przekazywany metodzie docelowej i zwrócić wartość pochodzącą z bufora, jeśli istnieje w nim element odpowiadający przekazanemu zestawowi argumentów.

Dotychczas analizowaliśmy przykłady pokazujące tworzenie aspektów w stylu adnotacji `AspectJ`. Przechodzimy teraz do przykładów użycia zwykłych komponentów Springa jako aspektów AOP.

9.6. Spring AOP, czyli styl schematu XML

W schemacie w stylu XML zwykły komponent Springa działa w charakterze aspektu. Metoda zdefiniowana w aspekcie jest za pomocą oferowanego przez Spring schematu aop powiązana z typem rady i wyrażeniem punktu przecięcia.

IMPORT **Rozdział 09/ch09-aop-xml-schema.** (Ten projekt jest taki sam jak przedstawiony wcześniej *ch09-aop-advice*, z wyjątkiem faktu, że klasa `SampleAspect` w projekcie *ch09-aop-xml-schema* jest prostą klasą Javy, która nie używa adnotacji `AspectJ`).

W listingu 9.12 została przedstawiona klasa `SampleAspect` projektu *ch09-aop-xml-schema* definiującego rady.

Listing 9.12. Klasa `SampleAspect`

Projekt: *ch09-aop-xml-schema*

Położenie pliku: *src/main/java/sample/spring/chapter09/bankapp/aspects*

```
package sample.spring.chapter09.bankapp.aspects;
.....
public class SampleAspect {
    .....
    public void afterReturningAdvice(JoinPoint joinPoint, int aValue) {
        logger.info("Wartość zwrócona przez metodę "
            + joinPoint.getSignature().getName()
            + " ma wartość " + aValue);
    }

    public void afterThrowingAdvice(JoinPoint joinPoint, Throwable exception) {
        logger.info("Wyjątek zgłoszony przez " + joinPoint.getSignature().getName()
            + " Typ wyjątku to : " + exception);
    }
    .....
}
```

W powyższym listingu widać, że klasa `SampleAspect` definiuje metody przedstawiające rady AOP. Zwróć uwagę na fakt, że klasa `SampleAspect` *nie* jest oznaczona adnotacją `@Aspect`, a jej metody nie są oznaczone adnotacjami takimi jak `@After`, `@AfterReturning` itd.

Zobaczmy, jak element `<config>` schematu aop oferowanego przez Spring jest używany do konfiguracji zwykłego komponentu Springa jako aspektu AOP.

Konfiguracja aspektu AOP

W stylu schematu XML konfiguracja dotycząca AOP znajduje się w elemencie `<config>` schematu aop oferowanego przez Spring. Sam aspekt AOP jest konfigurowany za pomocą elementu `<aspect>` znajdującego się w elemencie `<config>`.

W listingu 9.13 pokazujemy sposób konfiguracji klasy `SampleAspect` za pomocą elementów `<config>` i `<aspect>`.

Listing 9.13. Plik `applicationContext.xml` przedstawiający przykład użycia schematu aop

Projekt: `ch09-aop-xml-schema`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans ..... xmlns:aop="http://www.springframework.org/schema/aop" ..... >
  .....
  <bean id="sampleAspect"
        class="sample.spring.chapter09.bankapp.aspects.SampleAspect" />

  <aop:config proxy-target-class="false" expose-proxy="true">
    <aop:aspect id="sampleAspect" ref="sampleAspect">
      .....
    </aop:aspect>
  </aop:config>
</beans>
```

Ponieważ element `<config>` opiera się na automatycznym proxy, element `<config>` definiuje atrybuty `proxy-target-class` i `expose-proxy`. Jak zapewne pamiętasz, te same atrybuty są definiowane przez element `<aspectj-autoproxy>` schematu aop oferowanego przez Spring. W podrozdziale 9.3 znajdziesz więcej informacji na temat atrybutów `proxy-target-class` i `expose-proxy`.

W przedstawionym powyżej listingu 9.13 definicja komponentu `sampleAspect` określa klasę `SampleAspect` jako komponent. Element `<aspect>` konfiguruje komponent `sampleAspect` jako aspekt AOP. Atrybut `id` elementu `<aspect>` określa unikalny identyfikator aspektu, natomiast atrybut `ref` wskazuje komponent Springa, który ma zostać skonfigurowany jako aspekt AOP.

Skoro mamy już skonfigurowany aspekt AOP, zobaczmy teraz, jak można mapować metody zdefiniowane w aspekcie AOP na różne typy rad i wyrażen punktów przecięcia.

Konfiguracja rady

Radę można skonfigurować za pomocą jednego z następujących elementów znajdujących się w elemencie `<aspect>`: `<before>` (do konfiguracji rady typu *before*), `<after-returning>` (do konfiguracji rady typu *after returning*), `<after-throwing>` (do konfiguracji rady typu *after throwing*), `<after>` (do konfiguracji rady typu *after*) i `<around>` (do konfiguracji rady typu *around*).

Zobaczmy więc, jak rady zdefiniowane w klasie `SampleAspect` projektu *ch09-aop-xml-schema* zostały skonfigurowane w pliku XML kontekstu aplikacji.

Konfiguracja rady after returning

Rysunek 9.12 pokazuje, jak metoda `afterReturningAdvice()` klasy `SampleAspect` została skonfigurowana jako rada typu *after returning* za pomocą elementu `<after-returning>`.

```
public void afterReturningAdvice(JoinPoint joinPoint, int aValue) {
    logger.info("Value returned by " + joinPoint.getSignature().getName()
        + " method is " + aValue);
}
↓ ↓
<aop:after-returning method="afterReturningAdvice" returning="aValue"
pointcut="execution(*sample.spring..BankAccountService.createBankAccount(..))" />
```

Rysunek 9.12. Metoda `afterReturningAdvice()` klasy `SampleAspect` została skonfigurowana jako rada typu *after returning* z użyciem elementu `<after-returning>`

Atrybut `method` elementu `<after-returning>` określa nazwę metody, która ma być skonfigurowana jako rada typu *after returning*. Atrybut `returning` służy do tego samego celu co atrybut `returning` adnotacji `@AfterReturning` — wartość zwrótną metody docelowej udostępnia radzie. Atrybut `pointcut` wskazuje wyrażenie punktu przecięcia używane do wyszukania metod, dla których będzie zastosowana rada.

Konfiguracja rady after throwing

Rysunek 9.13 pokazuje, jak metoda `afterThrowingAdvice()` klasy `SampleAspect` została skonfigurowana jako rada typu *after throwing* za pomocą elementu `<after-throwing>`.

```
public void afterThrowingAdvice(JoinPoint joinPoint, Throwable exception) {
    logger.info("Exception thrown by " + joinPoint.getSignature().getName()
        + " Exception type is : " + exception);
}
↓ ↓
<aop:after-throwing method="afterThrowingAdvice" throwing="exception"
pointcut="execution(* sample.spring..FixedDepositService.*(..))" />
```

Rysunek 9.13. Metoda `afterThrowingAdvice()` klasy `SampleAspect` została skonfigurowana jako rada typu *after throwing* z użyciem elementu `<after-throwing>`

Atrybut `method` elementu `<after-throwing>` określa nazwę metody, która ma być skonfigurowana jako rada typu *after throwing*. Atrybut `throwing` służy do tego samego celu co atrybut `throwing` adnotacji `@AfterThrowing` — wyjątek zgłoszony przez metodę docelową zostaje udostępniony radzie. Atrybut `pointcut` wskazuje wyrażenie punktu przecięcia używane do wyszukania metod, dla których będzie zastosowana rada.

Inne typy rad (*before*, *after* i *around*) są konfigurowane w dokładnie taki sam sposób jak przedstawiliśmy powyżej dla rad *after returning* i *after throwing*.

Zobaczmy teraz różne sposoby powiązania wyrażenia punktu przecięcia z radą.

Powiązanie wyrażenia punktu przecięcia z radą

Elementy `<after>`, `<after-returning>`, `<after-throwing>`, `<before>` i `<around>` schematu `aop` oferowanego przez Spring definiują atrybut `pointcut`, którego można użyć do określenia wyrażenia punktu przecięcia powiązanego z radą. Jeżeli to samo wyrażenie punktu przecięcia ma być współdzielone przez wiele różnych rad, wówczas do jego zdefiniowania można użyć elementu `<pointcut>` znajdującego się w elemencie `<config>`.

W listingu 9.14 przedstawiamy przykład elementu `<pointcut>` użytego do zdefiniowania wyrażenia punktu przecięcia.

Listing 9.14. Plik `applicationContext.xml` przedstawiający użycie elementu `<pointcut>`

```
<beans ..... xmlns:aop="http://www.springframework.org/schema/aop" ..... >
    .....
    <bean id="sampleAspect"
        class="sample.spring.chapter09.bankapp.aspects.SampleAspect" />

    <aop:config proxy-target-class="false" expose-proxy="true">
        <aop:pointcut expression="execution(* sample.spring.*Service.*(..)"
            id="services" />

        <aop:aspect id="sampleAspect" ref="sampleAspect">
            <aop:after method="afterAdvice" pointcut-ref="services" />
            <aop:around method="aroundAdvice" pointcut-ref="services"/>
        </aop:aspect>
    </aop:config>
</beans>
```

W powyższym listingu element `<pointcut>` określa wyrażenie punktu przecięcia. Atrybut `expression` wskazuje wyrażenie, natomiast atrybut `id` określa unikalny identyfikator dla tego wyrażenia. Zdefiniowane przez element `<pointcut>` wyrażenie punktu przecięcia jest używane przez atrybut `pointcut-ref` elementów `<after>`, `<after-returning>` itd. Na przykład w omawianym listingu elementy `<after>` i `<around>` używają atrybutu `pointcut-ref` w celu odwołania się do wyrażenia punktu przecięcia nazwanego `services`.

9.7. Podsumowanie

W tym rozdziale przedstawiliśmy koncepcję programowania zorientowanego aspektowo i pokazaliśmy, jak można użyć frameworku AOP w Springu do realizacji zagadnień przekrojowych w aplikacjach Springa. Zobaczyłeś, jak można tworzyć aspekty za pomocą stylu adnotacji AspectJ i schematu XML. Przeanalizowaliśmy temat tworzenia i konfiguracji różnego typu rad. Poruszyliśmy także temat wyrażen punktu przecięcia, które można stosować do dopasowania metod w aplikacji. Więcej wyczerpujących informacji na temat oferowanego przez Spring frameworku AOP znajdziesz w dokumentacji Springa. W kolejnym rozdziale dowiesz się, jak tworzyć aplikacje sieciowe za pomocą modułu Spring Web MVC.

Rozdział 10. Podstawy modułu Spring Web MVC

10.1. Wprowadzenie

Framework Spring zawiera moduł Spring Web MVC, który dostarcza framework typu MVC (ang. *model-view-controller*, czyli model – widok – kontroler) przeznaczony do tworzenia aplikacji sieciowych opartych na serwetach. Spring Web MVC to *nieinwazyjny* framework pozwalający na zachowanie *podziału zadań* między obiektami aplikacji tworzącymi warstwę sieciową. Na przykład obiekt *kontrolera* jest używany do przetwarzania żądania, obiekt *weryfikatora* jest odpowiedzialny za sprawdzanie poprawności, obiekt *polecenia* służy do przechowywania danych formularza itd. Trzeba w tym miejscu koniecznie wspomnieć, że żaden z wymienionych obiektów aplikacji nie implementuje ani nie rozszerza jakiegokolwiek klasy bądź interfejsu Springa.

W tym rozdziale zaczniemy od przeanalizowania struktury katalogów stosowanej przez wszystkie projekty *sieciowe*, które zostaną tutaj omówione. Następnie przejdziemy do prostej aplikacji sieciowej typu Witaj, świecie! opracowanej za pomocą modułu Spring Web MVC. W pozostałej części rozdziału zajmiemy się pewnymi adnotacjami Spring Web MVC w kontekście aplikacji MyBank. Natomiast na koniec pokrótce przyjrzymy się nieco bardziej zaawansowanym funkcjom Spring Web MVC, które zostaną omówione w następnym rozdziale.

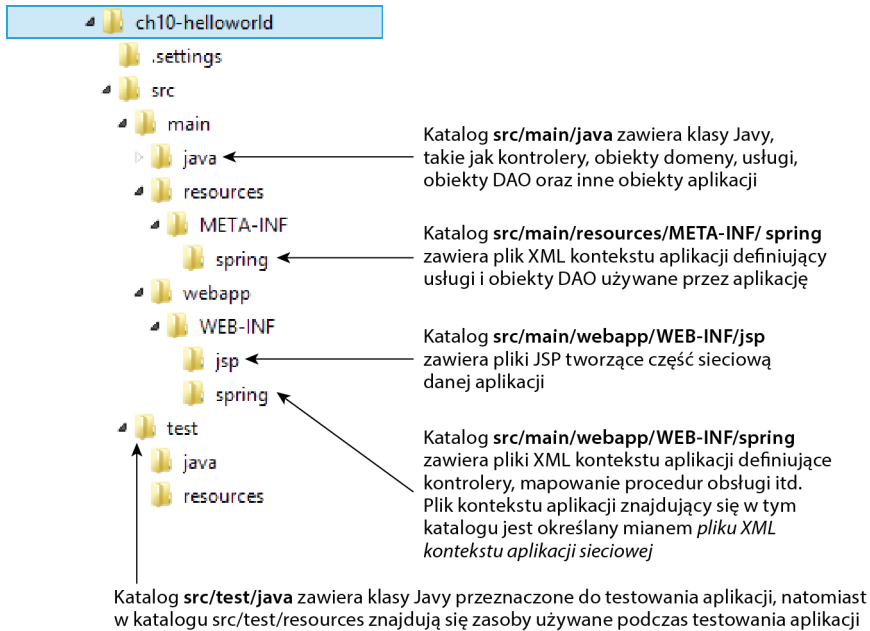
IMPORT **Rozdział10/ch10-helloworld.** (Ten projekt pokazuje prostą aplikację sieciową typu Witaj, świecie! opracowaną za pomocą modułu Spring Web MVC. W dodatku A znajdziesz informacje dotyczące wdrażania przykładowych projektów sieciowych w serwerze Tomcat. Po wdrożeniu aplikacji przejdź pod adres URL <http://localhost:8080/ch10-helloworld/helloworld/sayhello>. Jeżeli wdrożenie aplikacji zakończy się pomyślnie, otrzymasz wygenerowany komunikat Witaj, świecie!).

10.2. Struktura katalogów przykładowych projektów

Na rysunku 10.1 pokazujemy najważniejsze katalogi projektu *ch10-helloworld*. Poniżej wymieniamy niektóre z katalogów, o których koniecznie trzeba pamiętać.

- Katalog *src/main/resources/META-INF/spring* zawiera *główny plik XML kontekstu aplikacji sieciowej*. Plik ten definiuje komponenty współdzielone przez wszystkie serwlety i filtry aplikacji sieciowej. Główny plik XML kontekstu aplikacji sieciowej zwykle definiuje źródła danych, usługi, DAO, menedżery transakcji itd. Za wczytanie tego pliku odpowiada egzemplarz *ContextLoader* ↪ *Listener* (implementacja *javax.servlet.ServletContextListener*). Z podrozdziału 10.10 dowiesz się więcej na temat konfiguracji *ContextLoaderListener* w pliku *web.xml*.
- Katalog *src/main/webapp/WEB-INF/spring* zawiera *plik XML kontekstu aplikacji sieciowej* definiujący komponenty będące częścią warstwy *sieciowej* aplikacji. Plik ten zwykle definiuje *kontrolery* (nazywane także *procedurami obsługi*), *mapowanie procedur obsługi*, *komponenty odpowiedzialne za obsługę widoków*, *wyjątków* itd. Więcej informacji na temat tych obiektów znajdziesz w dalszej części rozdziału.

- Komponenty zdefiniowane w głównym pliku XML kontekstu aplikacji sieciowej są *dostępne* dla komponentów zdefiniowanych w pliku XML kontekstu aplikacji sieciowej. Oznacza to, że komponent zdefiniowany w pliku XML kontekstu aplikacji sieciowej może być zależny od komponentu zdefiniowanego w głównym pliku XML kontekstu aplikacji sieciowej, ale *nie* na odwrót.



Rysunek 10.1. Struktura katalogów projektu *ch10-helloworld*

Przechodzimy teraz do analizy plików konfiguracyjnych i klas w projekcie *ch10-helloworld*.

10.3. Poznajemy przykładową aplikację

Jeżeli w Eclipse IDE klikniesz prawym przyciskiem myszy projekt *ch10-helloworld* i wybierzesz opcję *Build Path/Configure Build Path*, zobaczysz, że zależnościami aplikacji są następujące plik JAR: *spring-beans*, *spring-context*, *spring-core*, *spring-expression*, *spring-web* i *spring-webmvc*. Są one niezbędne do zbudowania podstawowej aplikacji Spring Web MVC.

W tabeli 10.1 wymieniamy pliki konfiguracyjne i pliki kodu źródłowego Javy znajdujące się w projekcie *ch10-helloworld*. W dalszej części podrozdziału dokładniej przyjrzymy się tym plikom i klasom.

Poza plikami wymienionymi w tabeli 10.1 projekt *ch10-helloworld* zawiera także plik *log4j.properties* z konfiguracją Log4j oraz plik *pom.xml* z danymi wejściowymi dla narzędzia Maven. Jeżeli chcesz wiedzieć więcej na temat tych plików, zajrzyj do dokumentacji poświęconej Log4j (<http://logging.apache.org/log4j/1.2/>) i narzędziu Maven (<http://maven.apache.org/index.html>).

Przechodzimy teraz do dokładniejszej analizy plików wymienionych w tabeli 10.1.

Tabela 10.1. Pliki konfiguracyjne i pliki kodu źródłowego w przykładowym projekcie

Nazwa	Opis
<i>HelloWorldController.java</i>	Kontroler w Spring Web MVC odpowiedzialny za obsługę żądań. Znajdziesz go w pakiecie <code>sample.spring.chapter10.web</code> w katalogu <code>src/main/java</code> .
<i>helloworld.jsp</i>	Plik JSP odpowiedzialny za wyświetlenie komunikatu Witaj, świecie!. Znajdziesz go w katalogu <code>src/main/webapp/WEB-INF/jsp</code> .
<i>myapp-config.xml</i>	Plik XML kontekstu aplikacji sieciowej zawierającej definicje komponentów dla kontrolerów, mapowania procedur obsługi itd. Znajdziesz go w katalogu <code>src/main/webapp/WEB-INF/spring</code> .
<i>web.xml</i>	Deskryptor wdrażania aplikacji sieciowej. Znajdziesz go w katalogu <code>src/main/webapp/WEB-INF</code> .

Plik HelloWorldController.java, czyli klasa kontrolera aplikacji sieciowej Hello World

W aplikacjach Spring Web MVC logika odpowiedzialna za przetwarzanie żądania znajduje się w klasie kontrolera. W listingu 10.1 została przedstawiona klasa `HelloWorldController` projektu `ch10-helloworld`.

Listing 10.1. Klasa `HelloWorldController`

Projekt: `ch10-helloworld`

Położenie pliku: `src/main/java/sample/spring/chapter10/web`

```
package sample.spring.chapter10.web;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
.....
public class HelloWorldController implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        Map<String, String> modelData = new HashMap<String, String>();
        modelData.put("msg", "Witaj, świecie!");
        return new ModelAndView("helloworld", modelData);
    }
}
```

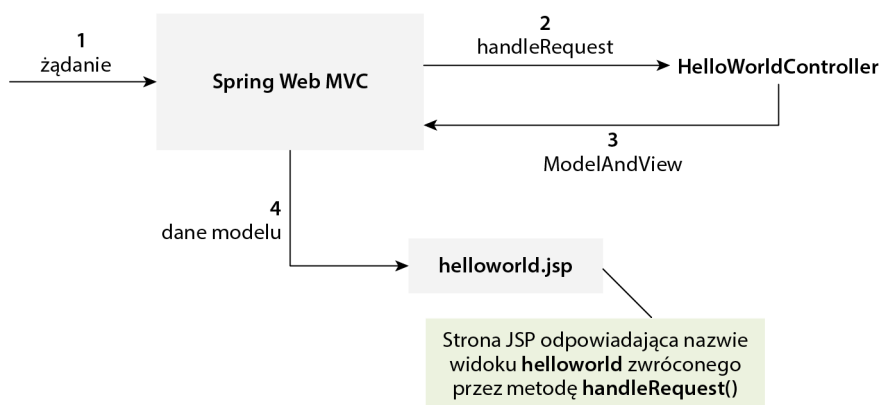
W powyższym listingu widać, że klasa `HelloWorldController` implementuje oferowany przez Spring interfejs `Controller`. Z kolei interfejs `Controller` definiuje metodę `handleRequest()`, którą wykorzystamy w celu zaimplementowania logiki przeznaczonej do obsługi żądania. Metoda `handleRequest()` zwraca obiekt `ModelAndView` zawierający następujące informacje:

- dane (określane mianem *modelu*) przeznaczone do wyświetlenia użytkownikowi;
- nazwa logiczna strony JSP (określana mianem *widoku*) wyświetlająca dane modelu.

Dane modelu są zwykle przedstawiane przez obiekt typu `java.util.Map`, a poszczególne elementy obiektu przedstawiają tak zwany *atrybut modelu*. Nazwa widoku (strona JSP) przeznaczona do wyświetlenia użytkownikowi jest podawana w postaci wartości typu `String`.

Listing 10.1 pokazuje, że metoda `handleRequest()` klasy `HelloWorldController` zwraca obiekt `ModelAndView` zawierający `helloworld` (wartość typu `String`) podaną jako nazwa widoku oraz `modelData` (obiekt typu `java.util.Map`) jako dane modelu. Egzemplarz `modelData` zawiera atrybut `msg`, którego wartość to komunikat Witaj, świecie!. Wkrótce zobaczysz, że atrybut modelu `msg` jest używany przez widok `helloworld` (strona JSP) do wyświetlenia użytkownikowi komunikatu Witaj, świecie!.

Diagram na rysunku 10.2 pokazuje, jak metoda `handleRequest()` klasy `HelloWorldController` generuje stronę JSP.



Rysunek 10.2. Framework Spring Web MVC wywołuje metodę `handleRequest()` klasy `HelloWorldController` i używa jej wartości zwrotnej (obiekt `ModelAndView`) do wygenerowania strony `helloworld.jsp`

Z rysunku 10.2 wynika, że framework Spring Web MVC przechwytuje przychodzące żądania HTTP i wywołuje metodę `handleRequest()` klasy `HelloWorldController`. Wartością zwrótną metody `handleRequest()` jest obiekt `ModelAndView` zawierający dane modelu i informacje o widoku. Po otrzymaniu obiektu `ModelAndView` z metody `handleRequest()` framework Spring Web MVC przekazuje żądanie HTTP do strony `helloworld.jsp` i udostępnia jej atrybuty modelu, które teraz są określane mianem *atrybutów żądania*.



Atrybuty modelu są przez framework Spring Web MVC udostępniane widokowi (np. JSP i Velocity) w formacie odpowiednim dla widoku. Na przykład jeśli używasz strony JSP jako widoku, wówczas atrybuty modelu są udostępniane stronie JSP w postaci atrybutów żądania.

Plik `helloworld.js`, czyli strona JSP wyświetlająca komunikat

W listingu 10.2 została przedstawiona strona `helloworld.jsp` zdefiniowana w projekcie `ch10-helloworld`.

Listing 10.2. Strona JSP o nazwie *helloworld.jsp*Projekt: *ch10-helloworld*Położenie pliku: *src/main/webapp/WEB-INF/jsp*

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<c:out value="${msg}" />
```

W powyższym listingu element `<c:out>` wyświetla wartość atrybutu żądania `msg`. Atrybut ten odwołuje się do atrybutu modelu `msg` zwróconego przez metodę `handleRequest()` klasy `HelloWorldController` (patrz: listing 10.1). Ponieważ wartością atrybutu modelu `msg` jest Witaj, świecie!, strona *helloworld.jsp* po prostu wyświetla komunikat o treści Witaj, świecie!.

Plik *myapp-config.xml*, czyli plik XML kontekstu aplikacji sieciowej

W listingu 10.3 zostały przedstawione komponenty skonfigurowane w pliku *myapp-config.xml* projektu *ch10-helloworld*.

Listing 10.3. Zawartość pliku *myapp-config.xml*Projekt: *ch10-helloworld*Położenie pliku: *src/main/webapp/WEB-INF/spring*

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

  <bean name="helloWorldController"
    class="sample.spring.chapter10.web.HelloWorldController" />

  <bean id="handlerMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="urlMap">
      <map>
        <entry key="/sayhello" value-ref="helloWorldController" />
      </map>
    </property>
  </bean>

  <bean id="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
  </bean>
</beans>
```

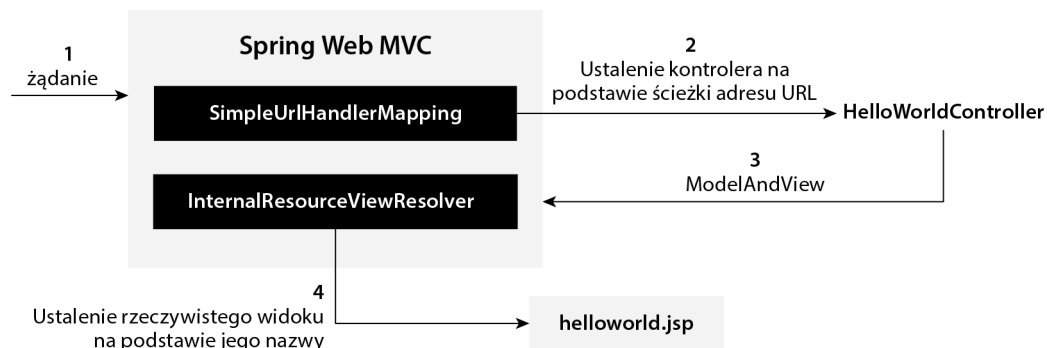
W powyższym listingu widać, że poza `HelloWorldController` w pliku *myapp-config.xml* skonfigurowane zostały także komponenty Springa o nazwach `SimpleUrlHandlerMapping` i `InternalResourceViewResolver`.

Komponent `SimpleUrlHandlerMapping` (implementacja interfejsu `HandlerMapping` oferowanego przez Spring) mapuje przychodzące żądanie HTTP na kontroler odpowiedzialny za obsługę żądań.

W celu mapowania żądania na kontroler komponent `SimpleUrlHandlerMapping` używa ścieżki dostępu adresu URL. Właściwość `urlMap` (typu `java.util.Map`) określa mapowanie ścieżki dostępu adresu URL na kontroler komponentu. W listingu 10.3 ścieżka dostępu adresu URL w postaci `/sayhello` (określona przez atrybut `key`) jest mapowana na komponent `HelloWorldController` (określony przez atrybut `value-ref`). Ścieżka dostępu adresu URL wskazana przez atrybut `key` jest *względna* dla adresu URL, na który `DispatcherServlet` (serwlet) jest mapowany w deskrypcie wdrożenia aplikacji sieciowej. Do `DispatcherServlet` powrócimy jeszcze w dalszej części tego podrozdziału.

Komponent `InternalResourceViewResolver` (implementacja interfejsu `ViewResolver` oferowanego przez Spring) wyszukuje rzeczywisty widok (np. stronę JSP lub serwlet) w oparciu o nazwę widoku przechowywaną w obiekcie `ModelAndView`. Rzeczywisty widok jest wyszukiwany przez *poprzedzenie* nazwy widoku wartością właściwości `prefix` oraz *dołączenie* do nazwy widoku wartości właściwości `suffix`. W listingu 10.3 widać, że wartością właściwości `prefix` jest `/WEB-INF/jsp`, natomiast wartością właściwości `suffix` jest `.jsp`. Ponieważ metoda `handleRequest()` klasy `HelloWorldController` zwraca obiekt `ModelAndView` zawierający `helloworld` jako nazwę widoku, rzeczywistym plikiem widoku jest `/WEB-INF/jsp/helloworld.jsp` (ten ciąg tekstowy został ustalony przez poprzedzenie nazwy widoku `helloworld` wartością `/WEB-INF/jsp` oraz dołączenie `.jsp` na jej końcu).

Na rysunku 10.3 pokazujemy, jaką rolę w aplikacji Hello World odgrywają komponenty `SimpleUrlHandlerMapping` i `InternalResourceViewResolver`.



Rysunek 10.3. Komponent `SimpleUrlHandlerMapping` wyszukuje kontroler do wywołania, natomiast `InternalResourceViewResolver` ustala rzeczywisty widok do wyświetlenia na podstawie jego nazwy

Komponenty `SimpleUrlHandlerMapping` i `InternalResourceViewResolver` są *automatycznie* wykrywane przez Spring Web MVC i używane do wyszukania kontrolera przeznaczonego, odpowiednio, do obsługi żądania i ustalenia faktycznego widoku do wyświetlenia.

Plik `web.xml`, czyli deskryptor wdrożenia aplikacji sieciowej

W aplikacjach Spring Web MVC żądania są przechwytywane przez `DispatcherServlet` (serwlet dostarczany przez Spring Web MVC) odpowiedzialny za przekazywanie żądań do odpowiednich kontrolerów.

W listingu 10.4 została przedstawiona konfiguracja DispatcherServlet w pliku *web.xml* projektu *ch10-helloworld*.

Listing 10.4. Zawartość pliku *web.xml*

Projekt: *ch10-helloworld*

Położenie pliku: *src/main/webapp/WEB-INF/spring*

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">

  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/myapp-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

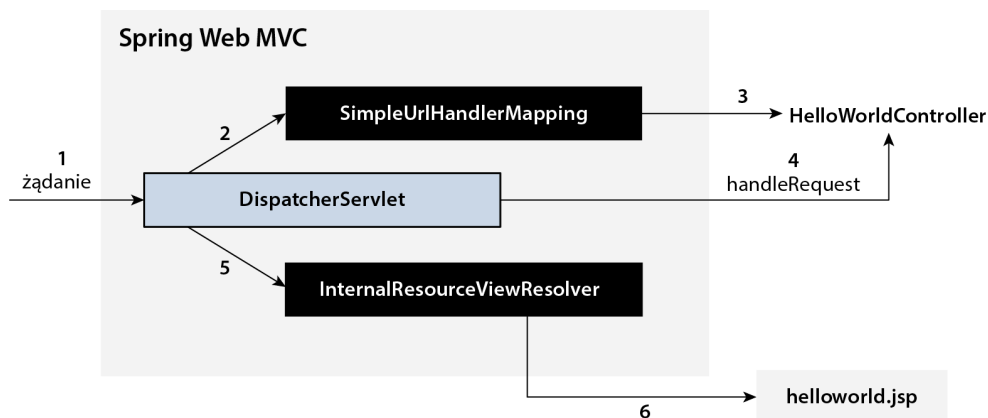
  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/helloworld/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Serwlet DispatcherServlet jest powiązany z plikiem XML kontekstu aplikacji sieciowej, w którym jest identyfikowany przez parametr inicjalizacyjny serwletu o nazwie *contextConfigLocation*. W omawianym listingu parametr inicjalizacyjny *contextConfigLocation* odwołuje się do pliku *myapp-config.xml* (patrz: listing 10.3).

Jeżeli nie wskażesz parametru *contextConfigLocation*, serwlet DispatcherServlet będzie szukał pliku XML kontekstu aplikacji sieciowej o nazwie *<nazwa-dla-DispatcherServlet>-servlet.xml* w katalogu *WEB-INF* aplikacji sieciowej. W omawianym przykładzie *<nazwa-dla-DispatcherServlet>* to nazwa serwletu zdefiniowana w elemencie *<servlet-name>* znajdującym się w elemencie *<servlet>*, który konfiguruje DispatcherServlet. Na przykład jeśli *nie* podamy parametru *contextConfigLocation* w listingu 10.3, wówczas serwlet DispatcherServlet będzie szukał w katalogu *WEB-INF* pliku o nazwie *hello-servlet.xml*.

Zdefiniowane w pliku XML kontekstu aplikacji komponenty *HandlerMapping* i *ViewResolver* są używane przez serwlet DispatcherServlet do przetwarzania żądania. DispatcherServlet korzysta z implementacji *HandlerMapping* w celu wyszukania odpowiedniego kontrolera dla żądania i używa implementacji *ViewResolver* do ustalenia rzeczywistego widoku na podstawie nazwy widoku podanej przez kontroler.

W kontekście prezentowanej tutaj aplikacji sieciowej typu „Witaj, świecie!” diagram na rysunku 10.4 podsumowuje rolę odgrywaną przez serwlet DispatcherServlet podczas przetwarzania żądania.



Rysunek 10.4. Serwlet *DispatcherServlet* używa komponentów *HandlerMapping* i *ViewResolver* do przetworzenia żądania

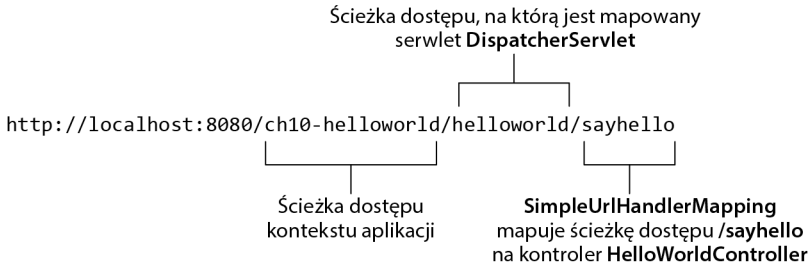
Na podstawie rysunku 10.4 można stwierdzić, że mamy do czynienia z następującą sekwencją zadań przeprowadzanych przez Spring Web MVC podczas przetwarzania żądania:

- Żądanie jest najpierw przechwytywane przez serwlet *DispatcherServlet*.
- Serwlet *DispatcherServlet* używa komponentu *HandlerMapping* (w przypadku aplikacji sieciowej typu „Witaj, świecie!” jest to implementacja komponentu *SimpleUrlHandlerMapping*) w celu wyszukania odpowiedniego kontrolera do obsługi żądania.
- Serwlet *DispatcherServlet* wywołuje przeznaczoną do obsługi żądania metodę kontrolera (w przypadku omawianej aplikacji sieciowej jest to *handleRequest()* klasy *HelloWorldController*).
- Zwróconą przez kontroler nazwę widoku serwlet *DispatcherServlet* przekazuje komponentowi *ViewResolver* (w przypadku omawianej aplikacji sieciowej jest to komponent *InternalResourceViewResolver*) w celu odszukania rzeczywistego widoku (JSP lub serwlet) do wygenerowania.
- Serwlet *DispatcherServlet* przekazuje żądanie do właściwego żądania (JSP lub serwlet). Dane modelu zwrócone przez kontroler są udostępniane widokowi w postaci atrybutów żądania.

Serwlet *DispatcherServlet* naszej aplikacji sieciowej jest mapowany na wzorzec `/helloworld/*` (patrz: listing 10.4), a komponent *SimpleUrlHandlerMapping* mapuje adres URL `/sayhello` na komponent *HelloWorldController* (patrz: listing 10.3). Jeżeli przejdziesz pod adres URL `http://localhost:8080/ch10-helloworld/helloworld/sayhello`, skutkiem będzie wywołanie metody *handleRequest()* kontrolera *HelloWorldController*. Rysunek 10.5 pokazuje, jak Spring Web MVC mapuje adres URL `http://localhost:8080/ch10-helloworld/helloworld/sayhello` na kontroler *HelloWorldController*.

Na rysunku 10.5 fragment `/ch10-helloworld` w adresie URL przedstawia ścieżkę dostępu kontekstu omawianej aplikacji sieciowej. Fragment `/helloworld` adresu URL jest mapowany na serwlet *DispatcherServlet* (patrz: listing 10.4), natomiast fragment `/sayhello` adresu URL jest mapowany na kontroler *HelloWorldController* (patrz: listing 10.3).

W tym podrozdziale zobaczyłeś, jak za pomocą Spring Web MVC można opracować prostą aplikację sieciową typu „Witaj, świecie!”. Przyjrzymy się teraz dokładniej serwletowi *DispatcherServlet*, który działa w charakterze wzorca *front controller* w aplikacji Spring Web MVC.



Rysunek 10.5. Sposób mapowania adresu URL `http://localhost:8080/ch10-helloworld/helloworld/sayhello` na kontroler `HelloWorldController` w Spring Web MVC

10.4. Serwet DispatcherServlet

Z poprzedniego podrozdziału dowiedziałeś się, że serwet `DispatcherServlet` działa w charakterze wzorca *front controller* współpracującego z komponentami `HandlerMapping` i `ViewResolver` zdefiniowanymi w pliku XML kontekstu aplikacji sieciowej i przeznaczonymi do przetwarzania żądań. W tym podrozdziale poznasz sposób działania serwetu `DispatcherServlet`.

W trakcie inicjalizacji serwet `DispatcherServlet` wczytuje odpowiedni plik XML kontekstu aplikacji sieciowej (można go wskazać za pomocą parametru inicjalizacyjnego `contextConfigLocation` lub w postaci pliku o nazwie `<nazwa-dla-DispatcherServlet>-servlet.xml` umieszczonego w katalogu `WEB-INF`), a następnie tworzy egzemplarz obiektu `WebApplicationContext` udostępnianego przez Spring. `WebApplicationContext` to subinterfejs `ApplicationContext` dostarczający funkcje przydatne podczas tworzenia aplikacji sieciowych. Na przykład komponenty w `WebApplication` `Context` mogą mieć zdefiniowane dodatkowe zasięgi, takie jak `request` i `session`. Obiekt `WebApplicationContext` możesz potraktować jako obiekt przedstawiający egzemplarz kontenera Springa w aplikacji Spring Web MVC.

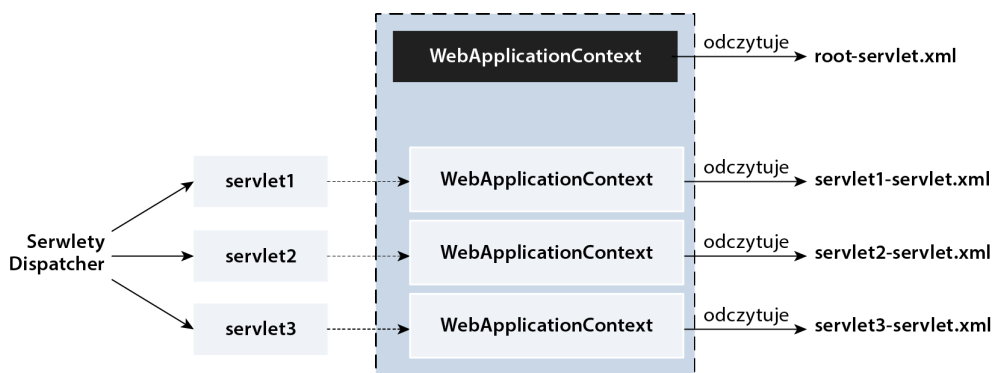
W tabeli 10.2 wymieniamy dodatkowe zasięgi, jakie można określić komponentom skonfigurowanym w pliku XML kontekstu aplikacji sieciowej.

Jeżeli aplikacja sieciowa składa się z wielu modułów, można zdefiniować serwet `DispatcherServlet` dla każdego modułu w pliku `web.xml`. W takim przypadku poszczególne serwety `DispatcherServlet` mają własne pliki XML kontekstu aplikacji sieciowej zawierające informacje o komponentach (np. kontrolerach, widokach itd.) charakterystycznych dla danego modułu. Trzeba w tym miejscu dodać, że wspomniane komponenty nie są współdzielone przez egzemplarze `DispatcherServlet`. Komponenty współdzielone między egzemplarzami `DispatcherServlet` są zdefiniowane w głównym pliku XML kontekstu aplikacji sieciowej. Jak wcześniej wspomnieliśmy, główny plik XML kontekstu aplikacji sieciowej definiuje źródła danych, usługi, obiekty DAO itd., które najczęściej są współdzielone przez różne moduły aplikacji sieciowej. Z podrozdziału 10.10 dowiesz się więcej o wczytywaniu głównego pliku XML aplikacji sieciowej.

Na rysunku 10.6 przedstawiamy relacje zachodzące między komponentami zdefiniowanymi przez plik XML kontekstu aplikacji sieciowej powiązanego z `DispatcherServlet` i komponentami zdefiniowanymi przez główny plik XML kontekstu aplikacji sieciowej.

Tabela 10.2. Dodatkowe zasięgi, jakie można wskazać komponentom

Zasięg komponentu	Opis
request	Kontener Springa tworzy nowy egzemplarz komponentu dla każdego żądania HTTP. Egzemplarz komponentu jest usuwany przez kontener Springa po zakończeniu żądania HTTP. Ten zasięg jest prawidłowy jedynie dla implementacji <code>ApplicationContext</code> odpowiednich w aplikacjach sieciowych. Na przykład jeśli używasz <code>XmlWeb</code> ↪ <code>ApplicationContext</code> lub <code>AnnotationConfigWebApplicationContext</code> , tylko wtedy możesz użyć zasięgu <code>request</code> dla komponentu.
session	Kontener Springa tworzy nowy egzemplarz komponentu po utworzeniu egzemplarza HTTP Session. Egzemplarz komponentu jest usuwany przez kontener Springa po usunięciu HTTP Session. Ten zasięg jest prawidłowy jedynie dla implementacji <code>ApplicationContext</code> odpowiednich w aplikacjach sieciowych. Na przykład jeśli używasz <code>XmlWeb</code> ↪ <code>ApplicationContext</code> lub <code>AnnotationConfigWebApplicationContext</code> , tylko wtedy możesz użyć zasięgu <code>session</code> dla komponentu.
globalSession	Ten zasięg jest odpowiedni jedynie dla aplikacji portlet.

Rysunek 10.6. Komponenty w głównym `WebApplicationContext` są dziedziczone przez egzemplarz `WebApplicationContext` powiązany z `DispatcherServlet`

Na rysunku 10.6 `servlet1`, `servlet2` i `servlet3` to nazwy egzemplarzy `DispatcherServlet` skonfigurowanych w pliku `web.xml`. Z kolei `servlet1-servlet.xml`, `servlet2-servlet.xml` i `servlet3-servlet.xml` to nazwy plików XML kontekstu aplikacji sieciowej wczytywanych przez, odpowiednio, `servlet1`, `servlet2` i `servlet3`. Podczas inicjalizacji egzemplarzy `DispatcherServlet` następuje tworzenie egzemplarza `WebApplicationContext` odpowiadającego poszczególnym plikom `servlet1-servlet.xml`, `servlet2-servlet.xml` i `servlet3-servlet.xml` i powiązywanie go z egzemplarzem `DispatcherServlet`. Egzemplarz `WebApplicationContext` jest również tworzony dla głównego pliku XML aplikacji sieciowej `root-servlet.xml`. Komponenty znajdujące się w głównym egzemplarzu `WebApplicationContext` są dostępne dla wszystkich egzemplarzy `WebApplicationContext` powiązanych z `DispatcherServlets`.

Przechodzimy teraz do analizy, jak kontroler lub inny komponent Springa zdefiniowany w pliku XML kontekstu aplikacji sieciowej może uzyskać dostęp do obiektów `ServletContext` i `Servlet` ↪ `Config`.

Uzyskanie dostępu do obiektów ServletContext i ServletConfig

W pewnych sytuacjach komponenty zdefiniowane w pliku XML kontekstu aplikacji sieciowej mogą wymagać uzyskania dostępu do obiektu ServletContext lub ServletConfig powiązanego z aplikacją sieciową.

ServletContext to obiekt API Servlet używany przez komponent do komunikacji z kontenerem serwletu. Na przykład obiekt ten można wykorzystać do pobierania i ustawiania atrybutów kontekstu, pobierania parametrów inicjalizacyjnych kontekstu itd. Jeżeli klasa komponentu implementuje oferowany przez Spring interfejs ServletContextAware (interfejs wywołań zwrotnych), wówczas kontener Springa dostarcza egzemplarz komponentu wraz z egzemplarzem obiektu ServletContext.

ServletConfig to obiekt API Servlet używany przez komponent w celu pobrania informacji konfiguracyjnych o serwlecie DispatcherServlet, który przechwycił żądanie. Na przykład można go wykorzystać do pobrania parametrów inicjalizacyjnych przekazywanych serwletowi DispatcherServlet oraz do pobrania nazwy, pod którą serwlet DispatcherServlet został skonfigurowany w pliku *web.xml*. Jeżeli klasa komponentu implementuje oferowany przez Spring interfejs ServletConfigAware (interfejs wywołań zwrotnych), wówczas kontener Springa dostarcza egzemplarz komponentu wraz z egzemplarzem obiektu ServletConfig.

W listingu 10.5 została przedstawiona klasa komponentu implementująca interfejsy ServletContextAware i ServletConfigAware.

Listing 10.5. Przykład użycia interfejsów ServletContextAware i ServletConfigAware

```
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import org.springframework.web.context.ServletConfigAware;
import org.springframework.web.context.ServletContextAware;

public class ABean implements ServletContextAware, ServletConfigAware {
    private ServletContext servletContext;
    private ServletConfig servletConfig;

    @Override
    public void setServletContext(ServletContext servletContext) {
        this.servletContext = servletContext;
    }

    @Override
    public void setServletConfig(ServletConfig servletConfig) {
        this.servletConfig = servletConfig;
    }

    public void doSomething() {
        // Użycie obiektów ServletContext i ServletConfig.
    }
}
```

W powyższym listingu widać, że klasa komponentu ABean implementuje interfejsy ServletContextAware i ServletConfigAware. Interfejs ServletContextAware definiuje metodę setServletContext()

wywoływaną przez kontener Springa w celu dostarczenia egzemplarza komponentu ABean wraz z egzemplarzem obiektu ServletContext. Z kolei interfejs ServletConfigAware definiuje metodę `setServletConfig()` wywoływaną przez kontener Springa w celu dostarczenia egzemplarza ABean wraz z egzemplarzem obiektu ServletConfig.

Wcześniej poznałeś sposób tworzenia kontrolera przez implementację interfejsu Controller. Teraz zapoznasz się z adnotacjami `@Controller` i `@RequestMapping`, które ułatwiają tworzenie kontrolerów.

10.5. Opracowanie kontrolerów za pomocą adnotacji `@Controller` i `@RequestMapping`

Spring Web MVC dostarcza klasy, takie jak `MultiActionController`, `UrlFilenameViewController`, `AbstractController` itd., które można rozszerzać w celu tworzenia implementacji kontrolerów. Jeżeli rozszerzysz klasę Springa lub zaimplementujesz oferowany przez Spring interfejs do utworzenia kontrolera, wówczas klasa kontrolera jest ściśle powiązana ze Springiem. W wydaniu Spring 2.5 wprowadzono adnotacje takie jak `@Controller`, `@RequestMapping`, `@ModelAttribute` itd., pozwalające na tworzenie adnotacji za pomocą elastycznych sygnatur metod. W tym podrozdziale poznasz różne adnotacje Spring Web MVC przeznaczone do tworzenia kontrolerów oznaczonych adnotacjami.

Na początek przeanalizuj aplikację sieciową typu „Witaj, świecie!” wykorzystującą kontroler oznaczony adnotacją do wyświetlenia komunikatu Witaj, świecie!.

IMPORT **Rozdział10/ch10-annotation-helloworld.** (Ten projekt pokazuje prostą aplikację sieciową typu „Witaj, świecie!” wykorzystującą kontroler oznaczony adnotacją do wyświetlenia komunikatu Witaj, świecie!. Po wdrożeniu aplikacji w serwerze Tomcat przejdź pod adres URL <http://localhost:8080/ch10-annotation-helloworld/helloworld/saySomething/sayhello>. Wynikiem będzie wyświetlenie komunikatu Witaj, świecie!).

Opracowanie aplikacji sieciowej za pomocą kontrolera oznaczonego adnotacją

Projekt *ch10-annotation-helloworld* jest podobny do omówionego wcześniej *ch10-helloworld*, z wyjątkiem faktu, że w projekcie *ch10-annotation-helloworld* został wykorzystany kontroler oznaczony adnotacją do wyświetlenia komunikatu Witaj, świecie!. Pliki *web.xml* i *helloworld.jsp* w obu projektach są dokładnie takie same, natomiast inne są pliki *HelloWorldController.java* i *myapp-config.xml*. Z tego powodu w tym miejscu analiza zostanie ograniczona do plików *HelloWorldController.java* i *myapp-config.xml*.

Na początek zajmiemy się utworzeniem kontrolera za pomocą adnotacji `@Controller` i `@RequestMapping`.

Adnotacje `@Controller` i `@RequestMapping`

Klasę można określić jako kontroler Spring Web MVC poprzez oznaczenie jej adnotacją `@Controller`. Natomiast adnotacja `@RequestMapping` jest używana do mapowania żądania przychodzącego na odpowiednią metodę kontrolera.

W listingu 10.6 została przedstawiona klasa `HelloWorldController` używająca adnotacji `@Controller` i `@RequestMapping`.

Listing 10.6. Klasa `HelloWorldController`, w której użyto adnotacji `@Controller` i `@RequestMapping`

```
package sample.spring.chapter10.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
.....
@Controller(value="sayHelloController")
@RequestMapping("/saySomething")
public class HelloWorldController {

    @RequestMapping("/sayhello")
    public ModelAndView sayHello() {
        Map<String, String> modelData = new HashMap<String, String>();
        modelData.put("msg", "Witaj, świecie!");
        return new ModelAndView("helloworld", modelData);
    }
}
```

W powyższym listingu klasa `HelloWorldController` została oznaczona adnotacjami `@Controller` i `@RequestMapping`, natomiast metoda `sayHello()` adnotacją `@RequestMapping`. Adnotacja `@Controller` to odmiana adnotacji `@Component` (patrz: rozdział 6.) wskazująca, że `HelloWorldController` to komponent kontrolera.

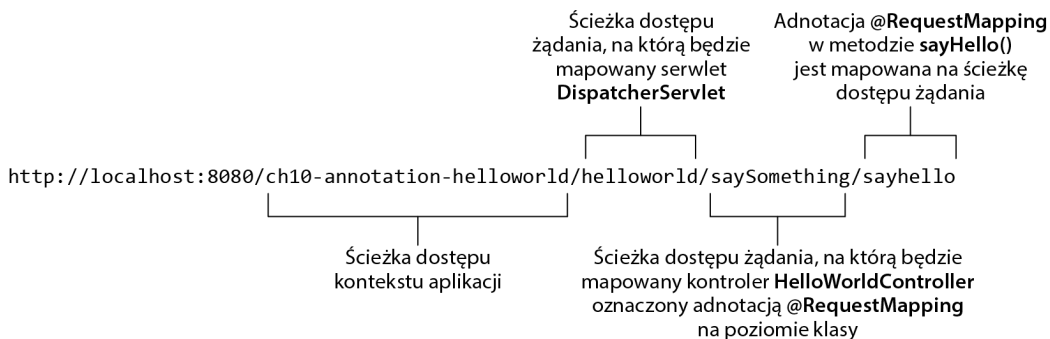
Podobnie jak klasy oznaczone adnotacjami `@Service` (patrz: rozdział 6.) i `@Repository` (patrz: rozdział 7.), także klasy z adnotacjami `@Controller` są automatycznie rejestrowane w kontenerze Springa. Nie ma więc konieczności wyraźnego definiowania w pliku XML kontekstu aplikacji sieciowej klasy oznaczonej adnotacją `@Controller`. Atrybut `value` adnotacji `@Controller` określa nazwę, pod którą klasa będzie zarejestrowana w kontenerze Springa. Atrybut `value` służy więc do tego samego celu co atrybut `id` elementu `<bean>`. Jeżeli atrybut `value` nie zostanie podany, wówczas do zarejestrowania klasy w kontenerze Springa zostanie użyta nazwa klasy (pierwsza litera zostanie zmieniona na małą).

Adnotacja `@RequestMapping` mapuje przychodzące żądania sieciowe na odpowiednie kontrolery i/lub metody kontrolera. Adnotacja `@RequestMapping` na poziomie typu mapuje żądanie do odpowiedniego kontrolera. Na przykład adnotacja `@RequestMapping("/saySomething")` w klasie `HelloWorldController` wskazuje, że wszystkie żądania prowadzące do ścieżki dostępu `/saySomething` będą obsługiwane przez kontroler `HelloWorldController`.

Adnotacja `@RequestMapping` na poziomie metody zawęża adnotację `@RequestMapping` na poziomie typu do określonej metody w klasie kontrolera. Na przykład adnotacja `@RequestMapping("/sayhello")` w metodzie `sayHello()` przedstawionej w listingu 10.6 określa, że metoda `sayHello()` będzie wywoływana, gdy żądanie dotyczy ścieżki dostępu `/saySomething/sayhello`. Zauważ, że metoda `sayHello()` klasy `HelloWorldController` nie akceptuje żadnych argumentów, a jej wartością zwrótną jest obiekt `ModelAndView`. To jest możliwe, ponieważ kontrolery oznaczone adnotacją mogą mieć elastyczne

sygnatury metod. W podrozdziale 10.7 poznasz możliwe argumenty i typy wartości zwrrotnych metod oznaczonych adnotacją `@RequestMapping`.

Adnotacja `@RequestMapping` na poziomie typu zwykle określa ścieżkę dostępu żądania lub wzorzec ścieżki dostępu. Ponadto adnotacja `@RequestMapping` na poziomie metody zwykle wskazuje metodę HTTP lub parametr żądania, aby jeszcze bardziej zawęzić mapowania określone przez adnotację `@RequestMapping` na poziomie typu. Na rysunku 10.7 pokazujemy, jak adres URL `http://localhost:8080/ch10-annotation-helloworld/helloworld/saySomething/sayhello` będzie mapowany na wywołanie metody `sayHello()` klasy `HelloWorldController`.



Rysunek 10.7. Sposób mapowania adresu URL żądania na odpowiednią metodę kontrolera oznaczoną adnotacją `@RequestMapping`

Na rysunku 10.7 widać, jak konkretny adres URL żądania jest mapowany na wywołanie metody `sayHello()` klasy `HelloWorldController`.

Teraz zobaczysz, jak w aplikacji Spring Web MVC włączyć możliwość tworzenia kontrolerów za pomocą adnotacji.

Włączenie adnotacji Spring Web MVC

W celu użycia w aplikacji Spring Web MVC kontrolerów oznaczonych adnotacjami konieczne jest włączenie adnotacji za pomocą elementu `<annotation-driven>` schematu mvc oferowanego przez Spring, jak przedstawia listing 10.7.

Listing 10.7. Przykład użycia elementu `<annotation-driven>` w pliku `myapp-config.xml`

Projekt: `ch10-annotation-helloworld`
 Położenie pliku: `src/main/webapp/WEB-INF/spring`

```

<beans .....
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xsi:schemaLocation=".....http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd.....">

  <mvc:annotation-driven />
  <context:component-scan base-package="sample.spring.chapter10.web" />

  <bean id="viewResolver"
  
```

```
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/jsp/" />
  <property name="suffix" value=".jsp" />
</bean>
</beans>
```

W powyższym listingu element `<mvc:annotation-driven>` schematu `mvc` włącza możliwość użycia adnotacji Spring Web MVC podczas implementacji kontrolerów. Ponadto element `<component-scan>` (więcej informacji na jego temat znajdziesz w podrozdziale 6.2) schematu `context` został użyty w celu automatycznej rejestracji w kontenerze Springa klas oznaczonych adnotacją `@Controller`.

W tym podrozdziale zobaczyłeś, jak opracować prostą aplikację sieciową typu „Witaj, świecie!” za pomocą adnotacji `@Controller` i `@RequestMapping`. Przechodzimy teraz do wymagań aplikacji sieciowej MyBank, którą opracujemy w tym rozdziale. Aplikacja ta będzie korzystać z adnotacji Spring Web MVC.

10.6. Wymagania aplikacji sieciowej MyBank

Na rysunku 10.8 pokazujemy stronę główną aplikacji sieciowej MyBank wyświetlającą listę aktualnych lokat założonych w systemie.

ID	Wysokość lokaty	Czas trwania	E-mail	Operacja
1	10000	24	a1email@nazwa-domeny.pl	Zamknij Edytuj
2	20000	36	a2email@nazwa-domeny.pl	Zamknij Edytuj
3	30000	36	a3email@nazwa-domeny.pl	Zamknij Edytuj
4	50000	36	a4email@nazwa-domeny.pl	Zamknij Edytuj
5	15000	36	a5email@nazwa-domeny.pl	Zamknij Edytuj

Załącz nową lokatę

Rysunek 10.8. Strona główna aplikacji sieciowej MyBank wyświetlająca informacje szczegółowe o lokatach (strona oferuje możliwość zamykania, modyfikowania i tworzenia lokat)

Na rysunku 10.8 kolumna *ID* zawiera unikalny identyfikator lokaty. Wartość identyfikatora jest przypisywana lokacie podczas jej zakładania przez użytkownika. Łąca *Zamknij* i *Edytuj* umożliwiają użytkownikowi usunięcie lokaty lub jej edycję. Z kolei przycisk *Załącz nową lokatę* powoduje wyświetlenie formularza, w którym użytkownik wprowadza informacje szczegółowe dotyczące zakładanej lokaty. Formularz ten widoczny jest na rysunku 10.9.

W formularzu pokazanym na rysunku 10.9 kliknięcie przycisku *Zapisz* powoduje zapisanie w magazynie danych informacji szczegółowych o lokacie. Z kolei kliknięcie łącza *Wróć* przenosi użytkownika z powrotem na stronę wyświetlającą listę lokat (patrz: rysunek 10.8). Jak widać na rysunku, wprowadzenie nieodpowiednich danych w polach powoduje wyświetlenie komunikatów błędów.

Załącz nową lokatę

Wartość (w PLN):	100	Wartość musi wynosić co najmniej 1000.
Czas trwania (w miesiącach):	6	Czas trwania musi wynosić co najmniej 12.
E-mail:	xyz	To nie jest prawidłowy adres.

[Wróć](#)

Rysunek 10.9. Formularz przeznaczony do zakładania lokat (wymagane jest wypełnienie wszystkich pól)

Po kliknięciu łącza *Edytuj* pokazanego na rysunku 10.8 nastąpi wyświetlenie formularza podobnego do widocznego na rysunku 10.9 i wyświetlającego informacje szczegółowe o wybranej lokacie. Naciśnięcie łącza *Zamknij* widocznego na rysunku 10.8 powoduje usunięcie wybranej lokaty z listy.

Skoro znamy wymagania aplikacji sieciowej MyBank, możemy przejść do analizy sposobu jej implementacji za pomocą adnotacji Spring Web MVC.

10.7. Adnotacje Spring Web MVC: @RequestMapping i @RequestParam

Z podrozdziału 10.5 dowiedziałeś się, że adnotacje @Controller i @RequestMapping można wykorzystać do utworzenia prostego kontrolera. W tym rozdziale dokładniej przeanalizujemy @RequestMapping oraz inne adnotacje Spring Web MVC ułatwiające tworzenie kontrolerów oznaczonych adnotacjami.

IMPORT **Rozdział10/ch10-bankapp.** (Ten projekt pokazuje aplikację sieciową MyBank, w której użytkownik ma możliwość zarządzania lokatami. Jeżeli wdrożysz projekt w serwerze Tomcat i przejdziesz pod adres <http://localhost:8080/ch10-bankapp>, zobaczysz listę lokat założonych w systemie, podobną do pokazanej na rysunku 10.8).

Rozpoczynamy od analizy adnotacji @RequestMapping.

Mapowanie żądań na kontrolery lub metody kontrolerów za pomocą adnotacji @RequestMapping

W podrozdziale 10.5 zobaczyłeś, że adnotacja @RequestMapping jest stosowana na poziomie *typu* i *metody* w celu mapowania żądań na kontrolery i ich metody. Z tego podrozdziału dowiesz się, jak Spring Web MVC mapuje żądanie sieciowe na konkretną metodę kontrolera, używając do tego adnotacji @RequestMapping. Następnie przejdziemy do atrybutów adnotacji @RequestMapping, a także do typów argumentów i wartości zwrotnych, jakie mogą mieć metody oznaczone adnotacją @RequestMapping.

Adnotacja @RequestMapping i RequestMappingHandlerMapping

W listingu 10.8 przedstawiamy przykład użycia adnotacji @RequestMapping w klasie SomeController (to kontroler Spring Web MVC).

Listing 10.8. Klasa *SomeController* przedstawiająca użycie adnotacji *@RequestMapping*

```

@Controller
@RequestMapping("/type_Level_Url")
public class SomeController {

    @RequestMapping("/methodA_Url")
    public ModelAndView methodA() { ..... }

    @RequestMapping("/methodB_Url")
    public ModelAndView methodB() { ..... }
}

```

Element <annotation-driven> schematu mvc oferowanego przez Spring tworzy egzemplarz *RequestMappingHandlerMapping* (implementacja *HandlerMapping*) odpowiedzialny za mapowanie żądania sieciowego na odpowiednią metodę oznaczoną adnotacją *@RequestMapping*. Egzemplarz *RequestMappingHandlerMapping* uznaje metody kontrolera za punkty końcowe i odpowiada za *unikalne* mapowanie żądania na metodę kontrolera, opierając się na adnotacjach *@RequestMapping* stosowanych na poziomie typu i metody. W przypadku kontrolera *SomeController*, jeżeli ścieżka dostępu żądania ma postać */type_Level_Url/methodA_Url*, wówczas nastąpi wywołanie metody *methodA()*. Natomiast ścieżka dostępu żądania w postaci */type_Level_Url/methodB_Url* oznacza wywołanie metody *methodB()*. Warto pamiętać, że jeśli żądanie nie będzie mogło zostać unikalnie zmapowane na metodę kontrolera, wartością zwrotną żądania będzie kod stanu HTTP 404 (oznaczający brak zasobu).

Atrybuty adnotacji *@RequestMapping* są używane w celu zawężenia mapowania do żądania określonego kontrolera lub jego metody. Istnieje możliwość podania tych atrybutów na poziomie zarówno typu, jak i metody adnotacji *@RequestMapping*. Przechodzimy teraz do atrybutów adnotacji *@RequestMapping*.

Mapowanie żądań na podstawie ścieżki dostępu żądania

Atrybut *value* adnotacji *@RequestMapping* określa ścieżkę dostępu żądania, na którą będzie mapowany kontroler lub jego metoda. Istnieje możliwość podania ścieżki dostępu żądania bez wyraźnego podawania atrybutu *value* w adnotacji *@RequestMapping*. Na przykład zamiast *@RequestMapping(value = "/type_Level_Url")* można użyć krótszej formy *@RequestMapping("/type_Level_Url")*.

Wartością atrybutu *value* może być również wzorec w stylu Ant. Na przykład można podać wzorec w postaci takiej jak */myUrl/**, */myUrl/*** i */myUrl/*.do*. W listingu 10.9 przedstawiamy przykład adnotacji *@RequestMapping* używającej */myUrl/*** jako wzorca dopasowania ścieżki dostępu.

W powyższym listingu adnotacja *@RequestMapping("/myUrl/**")* na poziomie typu oznacza, że kontroler *SomeController* zajmuje się obsługą wszystkich żądań, których ścieżka dostępu rozpoczyna się od */myUrl*. Dlatego żądania do adresów URL */myUrl/abc*, */myUrl/xyz* i */myUrl/123/something* będą obsługiwane przez kontroler *SomeController*.

Listing 10.9. Klasa *SomeController* przedstawiająca przykład użycia dla ścieżki dostępu żądania wzorca w stylu Ant

```

@Controller
@RequestMapping("/myUrl/**")
public class SomeController { ..... }

```

Mapowanie żądań na podstawie metod HTTP

Atrybut `method` adnotacji `@RequestMapping` umożliwia wskazanie metody HTTP, która będzie obsługiwana przez kontroler lub jego metodę. Dlatego jeśli wartością atrybutu `method` będzie metoda HTTP GET, wówczas wskazany kontroler lub jego metoda obsługują *jedynie* żądania HTTP GET.

W listingu 10.10 przedstawiamy metodę `listFixedDeposits()` klasy `FixedDepositController` przeznaczoną do wygenerowania listy lokat założonych w systemie.

Listing 10.10. Użycie atrybutu metody `@RequestMapping`

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter10/web*

```
package sample.spring.chapter10.web;

import org.springframework.web.bind.annotation.RequestMethod;
.....
@Controller
@RequestMapping(value="/fixedDeposit")
public class FixedDepositController {
    .....
    @RequestMapping(value = "/list", method = RequestMethod.GET)
    public ModelAndView listFixedDeposits() { ..... }
    .....
}
```

W powyższym listingu adnotacja `@RequestMapping` w metodzie `listFixedDeposits()` określa wartość atrybutu `method` jako `RequestMethod.GET`. Element `RequestMethod` to typ wyliczeniowy (enum) definiujący metody HTTP, na przykład GET, POST, PUT, DELETE itd. Ponieważ wartość atrybutu `method` wynosi `RequestMethod.GET`, metoda `listFixedDeposits()` zostanie wywołana tylko wtedy, gdy żądanie HTTP GET będzie skierowane pod adres URL */fixedDeposit/list*. Na przykład po wykonaniu żądania HTTP POST pod adres URL */fixedDeposit/list* aplikacja zwróci kod stanu HTTP 405 (nieobsługiwana metoda HTTP).

Wartością atrybutu `method` może być również tablica metod HTTP, jak zostało pokazane w listingu 10.11.

Listing 10.11. Wskazanie wielu metod HTTP jako wartości atrybutu `method`

```
@Controller
@RequestMapping(value="/sample")
public class MyController {

    @RequestMapping(value = "/action" method={ RequestMethod.GET, RequestMethod.POST })
    public ModelAndView action() { ..... }
}
```

W powyższym przykładzie metoda `action()` została oznaczona adnotacją `@RequestMapping`, której atrybut `method` ma wartość `{ RequestMethod.GET, RequestMethod.POST }`. Oznacza to, że metoda `action()` zostanie wywołana, gdy żądanie HTTP GET lub POST zostanie wysłane pod adres URL */sample/action*.

Mapowanie żądań na podstawie parametrów żądania

Atrybut `params` adnotacji `@RequestMapping` zwykle określa nazwę i wartość parametru żądania, który *musi* być obecny w żądaniu. Listing 10.12 przedstawia metodę `showOpenFixedDepositForm()` klasy `FixedDepositController` odpowiedzialną za wyświetlenie formularza umożliwiającego założenie nowej lokaty.

Listing 10.12. Przykład użycia atrybutu `param` adnotacji `@RequestMapping`

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter10/web*

```
package sample.spring.chapter10.web;

import org.springframework.web.bind.annotation.RequestMethod;
.....
@Controller
@RequestMapping(value="/fixedDeposit")
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=createFDForm", method = RequestMethod.POST)
    public ModelAndView showOpenFixedDepositForm() { ..... }
    .....
}
```

W powyższym listingu adnotacja `@RequestMapping` w metodzie `showOpenFixedDepositForm()` określa wartość atrybutu `params` jako `fdAction=createFDForm`. Ponieważ kontroler `FixedDepositController` jest mapowany na adres URL */fixedDeposit*, metoda `showOpenFixedDepositForm()` zostanie wywołana, gdy żądanie HTTP POST zawierające parametr o nazwie `fdAction` i wartości `createFDForm` zostanie wykonane na adres URL */fixedDeposit*.

Jeżeli chcesz mapować żądania na kontroler lub jego metodę, opierając się na wartościach wielu parametrów żądania, możesz jako wartość atrybutu `params` podać tablicę par nazwa – wartość będących parametrami żądania. Przykład takiego rozwiązania przedstawiamy w listingu 10.13.

Listing 10.13. Określenie wielu parametrów żądania (w postaci par nazwa – wartość) jako wartości atrybutu `params`

```
@RequestMapping(params = { "x=a", "y=b" })
public void perform() { ..... }
```

W powyższym listingu metoda `perform()` jest wywoływana tylko wtedy, gdy żądanie zawiera parametry o nazwach `x` i `y` oraz wartościach odpowiednio `a` i `b`.

Istnieje również możliwość mapowania żądań na kontroler lub jego metody w oparciu o obecność w żądaniu parametru żądania. W tym celu trzeba jedynie podać nazwę parametru żądania jako wartość atrybutu `params`. Na przykład metoda `perform()` przedstawiona w listingu 10.14 jest wywoływana niezależnie od wartości parametru żądania `x`.

Listing 10.14. Metoda `perform()` jest wywoływana, gdy parametr żądania zostanie znaleziony

```
@RequestMapping(params = "x")
public void perform() { ..... }
```

Aby mapować żądania na kontroler lub jego metodę, jeśli parametr żądania *nie* istnieje, należy użyć operatora `!`. Na przykład przedstawiona w listingu 10.15 metoda `perform()` jest wywoływana, gdy w żądaniu *nie* zostanie znaleziony parametr żądania o nazwie `x`.

Listing 10.15. Metoda `perform()` jest wywoływana, gdy parametr żądania `x` nie został odnaleziony

```
@RequestMapping(params = "!x")
public void perform() { ..... }
```

Istnieje możliwość użycia operatora `!=` w celu mapowania żądań na kontroler lub jego metodę, gdy wartość parametru żądania *nie* jest równa podanej wartości. Przykład takiego rozwiązania przedstawiono w listingu 10.16.

Listing 10.16. Metoda `perform()` jest wywoływana, jeśli parametr żądania `x` nie jest równy `a`

```
@RequestMapping(params = "x != a")
public void perform() { ..... }
```

W powyższym listingu metoda `perform()` jest wywoływana tylko wtedy, gdy żądanie zawiera parametr o nazwie `x`, a wartość `x` jest inna niż `a`.

Mapowanie żądań na podstawie typu MIME żądania

Nagłówek żądania `Content-Type` określa typ MIME żądania. Atrybut `consumes` adnotacji `@RequestMapping` określa typ MIME żądania obsługiwany przez kontroler lub jego metodę. Dlatego jeśli wartość atrybutu `consumes` zostanie dopasowana do wartości nagłówka żądania `Content-Type`, żądanie będzie mapowane na określony kontroler lub jego metodę.

W listingu 10.17 pokazujemy, że metoda `perform()` jest wywoływana, gdy wartością nagłówka `Content-Type` jest `application/json`.

Listing 10.17. Metoda `perform()` jest wywoływana, jeśli wartość wartości nagłówka `Content-type` jest `application/json`

```
@RequestMapping(consumes = "application/json")
public void perform() { ..... }
```

Podobnie jak w przypadku atrybutu `params` istnieje możliwość użycia operatora `!` w celu określenia wartości, jakiej nagłówek `Content-Type` *nie* powinien mieć. Na przykład przedstawiona w listingu 10.18 metoda `perform()` jest wykonywana, gdy wartość nagłówka `Content-Type` jest inna niż `application/json`.

Listing 10.18. Metoda `perform()` jest wywoływana, jeśli wartość nagłówka `Content-Type` jest inna niż `application/json`

```
@RequestMapping(consumes = "!application/json")
public void perform() { ..... }
```

Istnieje możliwość podania tablicy wartości dla atrybutu `consumes`. W takim przypadku żądanie będzie mapowane na kontroler lub metodę kontrolera, jeśli wartość nagłówka `Content-Type` zostanie

dopasowana do dowolnej wartości przypisanej atrybutowi `consumes`. W listingu 10.19 metoda `perform()` jest wywoływana, gdy wartością nagłówka `Content-Type` jest `application/json` lub `text/plain`.

Listing 10.19. *Metoda `perform()` jest wywoływana, jeśli wartością nagłówka `Content-Type` jest `application/json` lub `text/plain`*

```
@RequestMapping(consumes = { "application/json", "text/plain" })
public void perform() { ..... }
```

Mapowanie żądań w oparciu o akceptowane typy MIME odpowiedzi

Nagłówek żądania `Accept` określa akceptowane typy MIME odpowiedzi. Atrybut `produces` adnotacji `@RequestMapping` pozwala na zdefiniowanie akceptowanych typów MIME odpowiedzi. Dlatego po dopasowaniu wartości atrybutu `produces` do wartości nagłówka żądania `Accept` żądanie to będzie mapowane na konkretny kontroler lub metodę kontrolera.

W listingu 10.20 pokazujemy, że metoda `perform()` jest wywoływana, gdy wartość nagłówka `Accept` wynosi `application/json`.

Listing 10.20. *Metoda `perform()` jest wywoływana, jeśli wartością nagłówka `Accept` jest `application/json`*

```
@RequestMapping(produces = "application/json")
public void perform() { ..... }
```

Podobnie jak w przypadku atrybutu `consumes` istnieje możliwość użycia operatora `!` w celu wskazania, jakiej wartości nagłówek `Accept` *nie* powinien mieć. Jeżeli atrybutowi `produces` przypiszesz tablicę wartości, żądanie będzie mapowane na kontroler lub jego metodę, gdy wartość nagłówka `Accept` zostanie dopasowana do dowolnej z wartości określonych przez atrybut `produces`.

Mapowanie żądania na podstawie wartości nagłówka żądania

W celu mapowania żądań w oparciu o nagłówki żądania można użyć atrybutu `headers` adnotacji `@RequestMapping`. W listingu 10.21 pokazujemy, że żądanie będzie mapowane na metodę `perform()`, jeśli wartością nagłówka `Content-Type` będzie `text/plain`.

Listing 10.21. *Metoda `perform()` jest wywoływana, jeśli wartość nagłówka `Content-Type` wynosi `text/plain`*

```
@RequestMapping(produces = "application/json")
public void perform() { ..... }
```

Podobnie jak w przypadku atrybutu `params` istnieje możliwość użycia operatorów `!` i `!=` podczas określania wartości atrybutu `headers`. Na przykład w listingu 10.22 pokazujemy, że żądanie jest mapowane na metodę `perform()`, jeśli wartość nagłówka `Content-Type` *nie* wynosi `application/json`, nagłówek `Cache-Control` *nie* znajduje się w żądaniu, a nagłówek `From` *istnieje* i ma dowolną wartość.

Listing 10.22. *Użycie operatorów `!` i `!=` w celu określenia wartości atrybutu `headers`*

```
@RequestMapping(headers = { "Content-Type != application/json", "!Cache-Control", "From" })
public void perform() { ..... }
```

Po przeanalizowaniu atrybutów adnotacji `@RequestMapping` przechodzimy do atrybutów, które można przekazać metodom oznaczonym adnotacją `@RequestMapping`.

Argumenty metod oznaczonych adnotacją `@RequestMapping`

Metody oznaczone adnotacją `@RequestMapping` mogą mieć elastyczne sygnatury metod. Typy argumentów przekazywanych do metod oznaczonych adnotacją `@RequestMapping` zawierają `HttpServletRequest`, `HttpSession`, `java.security.Principal`, `org.springframework.validation.BindingResult`, `org.springframework.web.bind.support.SessionStatus`, `org.springframework.ui.Model` itd. Pełną listę argumentów, które mogą być przekazane metodzie oznaczonej adnotacją `@RequestMapping`, znajdziesz w dokumentacji Javadoc poświęconej tej adnotacji.

Ponieważ w tej książce omawiamy różne funkcje Spring Web MVC, niewątpliwie spotkamy się z sytuacjami, które wymagają przekazania różnych typów argumentów metodom oznaczonym adnotacją `@RequestMapping`. W tym momencie skoncentrujemy się na przypadku, gdy jako argument ma być wysłany obiekt `HttpServletRequest`.

W listingu 10.23 została przedstawiona metoda `viewFixedDepositDetails()` klasy `FixedDepositController` akceptująca argument typu `HttpServletRequest`.

Listing 10.23. Klasa `FixedDepositController`, której przekazywany jest argument typu `HttpServletRequest`

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter10/web*

```
package sample.spring.chapter10.web;

import javax.servlet.http.HttpServletRequest;
.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=view", method = RequestMethod.GET)
    public ModelAndView viewFixedDepositDetails(HttpServletRequest request) {
        FixedDepositDetails fixedDepositDetails = fixedDepositService
            .getFixedDeposit(Integer.parseInt(request.getParameter("fixedDepositId")));
        .....
    }
    .....
}
```

Metoda `viewFixedDepositDetails()` jest wywoływana po kliknięciu łącza *Edytuj* odpowiadającego lokacie (patrz: listing 10.8). Egzemplarz `HttpServletRequest` jest przez metodę `viewFixedDepositDetails()` używany do pobrania parametru żądania `fixedDepositId`, który unikalnie identyfikuje lokatę w systemie.

Przechodzimy teraz do typów wartości zwrotnych obsługiwanych przez metody oznaczone adnotacją `@RequestMapping`.

Typy wartości zwrrotnych metod oznaczonych adnotacją @RequestMapping

Typy wartości zwrrotnych obsługiwane przez metody oznaczone adnotacją @RequestMapping obejmują ModelAndView, org.springframework.web.servlet.View, String, java.util.concurrent.Callable, void itd. Pełną listę typów wartości zwrrotnych obsługiwanych przez metody oznaczone adnotacją @RequestMapping znajdziesz w dokumentacji Javadoc poświęconej adnotacji @RequestMapping.

Ponieważ w tej książce omawiamy różne funkcje Spring Web MVC, niewątpliwie spotkamy się z sytuacjami, w których metody oznaczone adnotacją @RequestMapping będą musiały mieć różne typy wartości zwrrotnych. W tym podrozdziale przedstawimy przykłady pokazujące metody, które zwracają wartości typu String i ModelAndView.

W listingu 10.24 została przedstawiona metoda showOpenFixedDepositForm() klasy FixedDepositController odpowiedzialna za wygenerowanie formularza HTML umożliwiającego założenie nowej lokaty (patrz: rysunek 10.9).

Listing 10.24. Klasa FixedDepositController, której wartość zwrrotna jest typu ModelAndView

Projekt: ch10-bankapp

Położenie pliku: src/main/java/sample/spring/chapter10/web

```
package sample.spring.chapter10.web;

import org.springframework.ui.ModelMap;
.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=createFDForm", method = RequestMethod.POST)
    public ModelAndView showOpenFixedDepositForm() {
        FixedDepositDetails fixedDepositDetails = new FixedDepositDetails();
        fixedDepositDetails.setEmail("Musisz podać poprawny adres e-mail");
        ModelMap modelData = new ModelMap();
        modelData.addAttribute(fixedDepositDetails);
        return new ModelAndView("createFixedDepositForm", modelData);
    }
    .....
}
```

W powyższym listingu showOpenFixedDepositForm() zwraca obiekt ModelAndView zawierający egzemplarz FixedDepositDetails jako atrybut modelu i ciąg tekstowy createFixedDepositForm jako nazwę widoku.

Jeżeli omawiany listing porównasz z listingami 10.1 i 10.6, zauważysz, że do przechowywania atrybutów modelu metoda showOpenFixedDepositForm() używa oferowanego przez Spring obiektu ModelMap zamiast java.util.Map. Obiekt ModelMap to implementacja interfejsu java.util.Map pozwalająca na przechowywanie atrybutów modelu bez konieczności wyraźnego podawania ich nazw. ModelMap automatycznie generuje nazwę atrybutu modelu na podstawie wcześniej ustalonej strategii. Na przykład po dodaniu własnego obiektu Javy jako atrybutu modelu rozpoczynająca się małą literą nazwa klasy obiektu zostanie użyta jako nazwa atrybutu modelu. W omawianym listingu

po dodaniu egzemplarza `FixedDepositDetails` do `ModelMap` będzie on tam przechowywany pod nazwą `fixedDepositDetails`.

Kiedy metoda oznaczona adnotacją `@RequestMapping` zwraca wartość w postaci ciągu tekstowego, wartość ta jest przez skonfigurowany dla aplikacji sieciowej egzemplarz `ViewResolver` uznawana za nazwę rzeczywistego widoku (np. stronę JSP lub serwet). W listingu 10.25 przedstawiamy konfigurację egzemplarza `InternalResourceViewResolver` w projekcie *ch10-bankapp*.

Listing 10.25. Plik *bankapp-config.xml* przedstawiający konfigurację `ViewResolver`

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/webapp/WEB-INF/spring*

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/jsp/" />
  <property name="suffix" value=".jsp" />
</bean>
```

Powyższa konfiguracja sugeruje, że kiedy wartością zwrótną będzie `xyz`, zostanie ona uznana za plik */WEB-INF/jsp/xyz.jsp*. Więcej informacji dotyczących `InternalResourceViewResolver` znajdziesz w podrozdziale 10.3.

Jeżeli ciąg tekstowy zwrócony przez metodę oznaczoną adnotacją `@RequestMapping` ma prefiks `redirect:`, wówczas jest traktowany jako adres URL przekierowania, a *nie* jako nazwa widoku. Listing 10.26 przedstawia metodę `closeFixedDeposit()` klasy `FixedDepositController` odpowiedzialną za zamknięcie lokaty, gdy użytkownik kliknie przycisk *Zamknij* (patrz: rysunek 10.9).

Listing 10.26. Klasa `FixedDepositController` przedstawiająca przykład wartości zwrótej typu `String`

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter10/web*

```
@RequestMapping(params = "fdAction=close", method = RequestMethod.GET)
public String closeFixedDeposit(..... int fdId) {
    fixedDepositService.closeFixedDeposit(fdId);
    return "redirect:/fixedDeposit/list";
}
```

Metoda `closeFixedDeposit()` klasy `FixedDepositController` zamyka lokatę wskazaną przez argument `fdId` i zwraca wartość w postaci ciągu tekstowego `redirect:/fixedDeposit/list`. Ponieważ zwrócony ciąg tekstowy zawiera prefiks `redirect:`, użytkownik zostanie przekierowany na stronę pod adresem */fixedDeposit/list*, która wyświetla listę lokat (patrz: rysunek 10.8).

Przechodzimy teraz do adnotacji `@RequestParam` pozwalającej na przypisanie argumentowi metody kontrolera wartości parametru żądania.

Przekazanie parametrów żądania do metod kontrolera z użyciem adnotacji `@RequestParam`

W listingu 10.23 zobaczyłeś, że obiekt `HttpServletRequest` można przekazać metodzie kontrolera i użyć go do pobrania parametrów żądania. Zamiast przekazywać obiekt `HttpServletRequest` meto-

dzie kontrolera istnieje także możliwość oznaczenia argumentu metody adnotacją `@RequestParam` w celu przypisania mu wartości parametru żądania.



Zwróć uwagę, że adnotacja `@RequestParam` może być użyta tylko wtedy, gdy metoda została oznaczona adnotacją `@RequestMapping` lub `@ModelAttribute` (druga z wymienionych adnotacji zostanie omówiona w rozdziale 11.).

W listingu 10.27 została przedstawiona metoda `closeFixedDeposit()` klasy `FixedDepositController` wywoływana, gdy użytkownik kliknie przycisk *Zamknij* (patrz: rysunek 10.8), aby zamknąć lokatę.

Listing 10.27. Klasa `FixedDepositController` przedstawiająca przykład użycia adnotacji `@RequestParam`

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter10/web*

```
package sample.spring.chapter10.web;

import org.springframework.web.bind.annotation.RequestParam;
.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=close", method = RequestMethod.GET)
    public String closeFixedDeposit(@RequestParam(value = "fixedDepositId") int fdId) {
        fixedDepositService.closeFixedDeposit(fdId);
        return "redirect:/fixedDeposit/list";
    }
    .....
}
```

Atrybut `value` adnotacji `@RequestParam` określa nazwę parametru żądania, którego wartość jest przypisywana argumentowi metody. W omawianym listingu adnotacja `@RequestParam` została użyta do przypisania wartości parametru żądania `fixedDepositId` argumentowi metody `fdId`. Ponieważ typem argumentu `fdId` jest `int`, Spring będzie odpowiedzialny za konwersję parametru żądania `fixedDepositId` na typ `int`. Domyślnie Spring automatycznie oferuje konwersję prostych typów Javy, takich jak `int`, `long`, `java.util.Date` itd. W celu przeprowadzenia konwersji parametrów na własny typ Javy (np. `Address`) konieczne jest zarejestrowanie własnego edytora `Property` ↪ `Editors` wraz z oferowanym przez Spring egzemplarzem `WebDataBinder` lub `org.springframework.format.Formatter` wraz z oferowanym przez Spring egzemplarzem `FormattingConversionService`. Więcej informacji na temat `WebDataBinder` znajdziesz w rozdziale 11., natomiast o `Formatter` i `FormattingConversionService` — w rozdziale 13.

Przekonajmy się teraz, jak z poziomu metody kontrolera można uzyskać dostęp do wszystkich parametrów żądania.

Przekazanie wszystkich parametrów żądania metodzie kontrolera

W celu przekazania *wszystkich* parametrów żądania metodzie kontrolera należy zdefiniować argument typu `Map<String, String>` lub `MultiValueMap<String, String>` (dostarczony przez Spring obiekt implementujący interfejs `java.util.Map`), a następnie oznaczyć go adnotacją `@RequestParam`.

W listingu 10.28 została przedstawiona metoda `openFixedDeposit()` klasy `FixedDepositController` odpowiedzialna za utworzenie nowej lokaty, gdy użytkownik wprowadzi informacje szczegółowe o lokacie, a następnie kliknie przycisk *Zapisz* w formularzu służącym do zakładania nowych lokat (patrz: rysunek 10.9).

Listing 10.28. Klasa `FixedDepositController` przedstawiająca przykład uzyskania dostępu do wszystkich parametrów żądania

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter10/web*

```
package sample.spring.chapter10.web;

import java.util.Map;
.....
@RequestMapping(value = "/fixedDeposit")
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
    public ModelAndView openFixedDeposit(@RequestParam Map<String, String> params) {
        String depositAmount = params.get("depositAmount");
        String tenure = params.get("tenure");
        .....
    }
}
```

W powyższym listingu argument `params` typu `Map<String, String>` został oznaczony adnotacją `@RequestParam`. Zwróć uwagę na fakt, że atrybut `value` adnotacji `@RequestParam` *nie* został określony. Jeżeli atrybut `value` adnotacji `@RequestParam` nie jest podany, a typem argumentu metody jest `Map<String, String>` lub `MultiValueMap<String, String>`, wówczas Spring kopiuje wszystkie parametry żądania do argumentu metody. Każda wartość parametru żądania jest przechowywana w obiekcie `Map` (lub `MultiValueMap`), a nazwa parametru żądania jest używana jako klucz.

W listingu 10.29 została przedstawiona metoda `editFixedDeposit()` klasy `FixedDepositController` odpowiedzialna za wprowadzanie zmian w istniejących lokatach.

Listing 10.29. Klasa `FixedDepositController` przedstawiająca przykład uzyskania dostępu do wszystkich parametrów żądania

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter10/web*

```
package sample.spring.chapter10.web;
import org.springframework.util.MultiValueMap;
.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=edit", method = RequestMethod.POST)
    public ModelAndView editFixedDeposit(@RequestParam MultiValueMap<String, String>
        params) {
        String depositAmount = params.get("depositAmount").get(0);
        String tenure = params.get("tenure").get(0);
        .....
    }
}
```


Użycie atrybutu `defaultValue` do określenia wartości domyślnej dla parametru żądania

Atrybut `defaultValue` adnotacji `@RequestParam` określa wartość domyślną parametru żądania. Jeżeli parametr żądania określony przez atrybut `value` adnotacji `@RequestParam` nie zostanie znaleziony w żądaniu, wówczas wartość atrybutu `defaultValue` zostanie przypisana argumentowi metody. W listingu 10.32 przedstawiamy przykład użycia atrybutu `defaultValue`.

Listing 10.32. Przykład użycia atrybutu `defaultValue` adnotacji `@RequestParam`

```
@RequestMapping(....)
public String perform(@RequestParam(value = "location", defaultValue = "earth")
    String param) {
    ....
}
```

W powyższym listingu, jeżeli parametr żądania o nazwie `location` nie zostanie znaleziony w żądaniu, argumentowi metody `param` zostanie przypisana wartość `earth`.

W tym podrozdziale przeanalizowaliśmy adnotacje `@RequestMapping` i `@RequestParam` użyte podczas tworzenia kontrolera `FixedDepositController` w aplikacji `MyBank`. Przechodzimy teraz do sposobu weryfikacji danych formularza sieciowego przeprowadzanej przez klasę `FixedDepositController`.

10.8. Weryfikacja

We wcześniejszej części rozdziału zobaczyłeś, że metoda `showOpenFixedDepositForm()` klasy `FixedDepositController` (patrz: listing 10.24) generuje stronę JSP o nazwie `createFixedDepositForm.jsp`, która wyświetla formularz umożliwiający założenie nowej lokaty. Po wysłaniu formularza sieciowego wprowadzone w nim dane są weryfikowane przez metodę `openFixedDeposit()` klasy `FixedDepositController` (patrz: listing 10.28). Jeżeli w trakcie weryfikacji zostaną wykryte błędy, strona `createFixedDepositForm.jsp` zostanie wygenerowana ponownie wraz z komunikatami błędów oraz początkowymi danymi wprowadzonymi przez użytkownika (patrz: rysunek 10.9).

W listingu 10.33 został przedstawiony element `<form>` zdefiniowany na stronie `createFixedDepositForm.jsp`.

W powyższym listingu atrybut `method` elementu `<form>` wskazuje POST jako metodę HTTP, natomiast atrybut `action` określa `/fixedDeposit?fdAction=create` jako adres URL, na który zostanie wysłany formularz sieciowy po kliknięciu przez użytkownika przycisku *Zapisz*. Wysłanie formularza skutkuje wywołaniem metody `openFixedDeposit()` klasy `FixedDepositController`.

Listing 10.33. Zawartość pliku `createFixedDepositForm.jsp` wraz z elementem `<form>`

Projekt: `ch10-bankapp`
Położenie pliku: `src/main/webapp/WEB-INF/jsp`

```
<form name="createFixedDepositForm" method="POST"
    action="${pageContext.request.contextPath}/fixedDeposit?fdAction=create">
    ....
    <input type="submit" value="Zapisz" />
</form>
```

W listingu 10.34 przedstawiamy kod odpowiedzialny w metodzie `openFixedDeposit()` za przeprowadzenie weryfikacji. Pokazujemy również, jak początkowe dane wprowadzone przez użytkownika są ponownie wyświetlane, gdy w trakcie procesu weryfikacji zostaną wykryte błędy.

Listing 10.34. Metoda `openFixedDeposit()` klasy `FixedDepositController`

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter10/web*

```
package sample.spring.chapter10.web;

.....
import org.apache.commons.lang3.math.NumberUtils;
@RequestMapping(value = "/fixedDeposit")
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
    public ModelAndView openFixedDeposit(@RequestParam Map<String, String> params) {
        String depositAmount = params.get("depositAmount");
        .....
        Map<String, Object> modelData = new HashMap<String, Object>();

        if (!NumberUtils.isNumber(depositAmount)) {
            modelData.put("error.depositAmount", "Podaj prawidłową liczbę.");
        } else if (NumberUtils.toInt(depositAmount) < 1000) {
            modelData.put("error.depositAmount",
                "Wartość musi wynosić co najmniej 1000.");
        }
        .....
        FixedDepositDetails fixedDepositDetails = new FixedDepositDetails();
        fixedDepositDetails.setDepositAmount(depositAmount);
        .....
        if (modelData.size() > 0) { //To oznacza wykrycie błędów w trakcie weryfikacji.
            modelData.put("fixedDepositDetails", fixedDepositDetails);
            return new ModelAndView("createFixedDepositForm", modelData);
        } else {
            fixedDepositService.saveFixedDeposit(fixedDepositDetails);
            return new ModelAndView("redirect:/fixedDeposit/list");
        }
    }
    .....
}
```

Metoda `openFixedDeposit()` przeprowadza sprawdzenie informacji podanych przez użytkownika, czyli wysokości lokaty, czasu jej trwania oraz adresu e-mail. Zauważ, że dla ułatwienia procesu weryfikacji wykorzystano klasę `NumberUtils` biblioteki Apache Commons Lang (<http://commons.apache.org/proper/commons-lang/>). Zmienna `modelData` jest obiektem `java.util.Map` przechowującym atrybuty modelu, które mają być przekazane na stronę *createFixedDepositForm.jsp* w przypadku wykrycia błędów podczas weryfikacji.

Ponieważ w przypadku wykrycia błędów chcemy wyświetlić zarówno komunikaty błędów, jak i początkowe dane wprowadzone przez użytkownika, wymienione informacje są przechowywane w `modelData`. Na przykład jeśli użytkownik wprowadzi nieprawidłową wysokość lokaty, zostanie wyświetlony komunikat błędu przechowywany w obiekcie `modelData` pod nazwą `error.depositAmount`. Wartości wprowadzone przez użytkownika są stosowane w nowym egzemplarzu obiektu

FixedDepositDetails. Po wystąpieniu błędów w trakcie weryfikacji nowo utworzony egzemplarz FixedDepositDetails zostaje dodany do modelData pod nazwą fixedDepositDetails, a następnie generowana jest strona *createFixedDepositForm.jsp*. Natomiast w przypadku braku błędów podczas weryfikacji nowo utworzony obiekt FixedDepositDetails zostaje zapisany w źródle danych i generowana jest strona wyświetlająca listę wszystkich założonych lokat.

Ponieważ obiektu FixedDepositDetails używamy do przechowywania początkowych danych wprowadzonych przez użytkownika, wszystkie atrybuty FixedDepositDetails zostały zdefiniowane jako typu String, jak przedstawia listing 10.35.

Listing 10.35. Klasa FixedDepositDetails

Projekt: ch10-bankapp

Położenie pliku: src/main/java/sample/spring/chapter10/domain

```
package sample.spring.chapter10.domain;

public class FixedDepositDetails {
    private long id; //Wartość id jest ustawiana przez system.
    private String depositAmount;
    private String tenure;
    private String email;

    //Metody getter i setter dla właściwości.
    .....
}
```

Ponieważ właściwości depositAmount i tenure zostały zdefiniowane jako typu String, konieczne jest przygotowanie dodatkowej logiki odpowiedzialnej za ich konwersję na wartości liczbowe, aby można było przeprowadzić porównanie liczb. W rozdziale 11. dowiesz się, jak Spring Web MVC ułatwia dołączanie danych formularza do *obsługujących* go obiektów (np. FixedDepositDetails) oraz jak ponownie wyświetlić początkowe dane w przypadku wystąpienia błędów podczas weryfikacji.

W listingu 10.36 przedstawiamy kod strony *createFixedDepositForm.jsp*, który pokazuje, jak komunikaty błędów i początkowe dane wprowadzone w formularzu są wyświetlane w aplikacji MyBank.

Listing 10.36. Zawartość pliku createFixedDepositForm.jsp

Projekt: ch10-bankapp

Położenie pliku: src/main/webapp/WEB-INF/jsp

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<form name="createFixedDepositForm" method="POST"
    action="${pageContext.request.contextPath}/fixedDeposit?fdAction=create">
    .....
    <td class="td"><b>Wartość (w PLN):</b></td>
    <td class="td">
        <input type="text" name="depositAmount"
            value="${requestScope.fixedDepositDetails.depositAmount}"/>
        <font style="color: #C11B17;">
            <c:out value="${requestScope['error.depositAmount']}" /></font>
        </td>
        .....
        <input type="submit" value="Zapisz" />
    </form>
```

W powyższym listingu wartość pola `depositAmount` w formularzu została określona jako `${requestScope.fixedDepositDetails.depositAmount}`. W metodzie `openFixedDeposit()` (patrz: listing 10.34) egzemplarz `FixedDepositDetails` jest dodawany jako atrybut modelu o nazwie `fixedDepositDetails`. Dlatego wyrażenie `${requestScope.fixedDepositDetails.depositAmount}` wyświetla początkową wartość wprowadzoną przez użytkownika w polu `depositAmount`.

Wyrażenie `${requestScope['error.depositAmount']}` odwołuje się do atrybutu żądania `error.depositAmount`. W metodzie `openFixedDeposit()` (patrz: listing 10.34) widać, że `error.depositAmount` zawiera wyświetlany użytkownikowi komunikat błędu dotyczący nieprawidłowej wysokości lokaty. Dlatego element `<c:out value= "${requestScope['error.depositAmount']}" />` wyświetla użytkownikowi wspomniany komunikat błędu w przypadku podania przez niego nieprawidłowej wysokości lokaty.

Przechodzimy teraz do obsługi wyjątków w aplikacjach Spring Web MVC.

10.9. Obsługa wyjątków za pomocą adnotacji `@ExceptionHandler`

Adnotacja `@ExceptionHandler` jest używana w oznaczonym adnotacją kontrolerze w celu wskazania metod odpowiedzialnych za obsługę wyjątków zgłaszanych przez kontroler. Udostępniany przez Spring egzemplarz `HandlerExceptionResolver` jest przeznaczony do mapowania wyjątku na odpowiednią metodę kontrolera odpowiedzialną za obsługę wyjątku. Warto zwrócić uwagę, że element `<annotation-driven>` schematu `mvc` konfiguruje egzemplarz `ExceptionHandlerExceptionResolver` (implementacja `HandlerExceptionResolver`) mapujący wyjątek na odpowiednią metodę oznaczoną adnotacją `@ExceptionHandler`.

W listingu 10.37 przedstawiamy użycie adnotacji `@ExceptionHandler` w projekcie *ch10-bankapp*.

Listing 10.37. Przykład użycia adnotacji `@ExceptionHandler`

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter10/web*

```
package sample.spring.chapter10.web;

import org.springframework.web.bind.annotation.ExceptionHandler;
.....
@Controller
@RequestMapping(value = "/fixedDeposit")
public class FixedDepositController {
    .....
    @ExceptionHandler
    public String handleException(Exception ex) {
        return "error";
    }
}
```

W powyższym listingu widać, że metoda `handleException()` klasy `FixedDepositController` jest oznaczona adnotacją `@ExceptionHandler`. Metoda ta będzie więc wywoływana przez Spring Web MVC do obsługi wyjątków zgłaszanych przez kontroler `FixedDepositController`. Metoda oznaczona adnotacją `@ExceptionHandler` najczęściej generuje stronę wyświetlającą informacje szczegółowe o błędzie. Atrybut `value` adnotacji `@ExceptionHandler` określa listę wyjątków obsługiwanych

przez daną metodę oznaczoną adnotacją `@ExceptionHandler`. Jeżeli atrybut *value* *nie* zostanie podany, wówczas przez daną metodę będą obsługiwane wyjątki o typach wskazanych jako argumenty metody. W omawianym listingu metoda `handleException()` obsługuje wyjątki typu `java.lang.Exception`.

Podobnie jak metody oznaczone adnotacją `@RequestMapping`, także metody oznaczone adnotacją `@ExceptionHandler` mogą mieć elastyczne sygnatury. Typy wartości zwrrotnych obsługiwane przez metody oznaczone adnotacją `@ExceptionHandler` obejmują między innymi `ModelAndView`, `View`, `String`, `void`, `Model`. Natomiast typy argumentów obsługiwanych przez metody oznaczone adnotacją `@ExceptionHandler` obejmują między innymi `HttpServletRequest`, `HttpServletResponse`, `HttpSession`. Pełną listę obsługiwanych argumentów i typów wartości zwrrotnych znajdziesz w dokumentacji Javadoc dla adnotacji `@ExceptionHandler`.

W celu wyświetlenia informacji zwróconych przez metodę oznaczoną adnotacją `@ExceptionHandler` jest używany serwet `DispatcherServlet` odpowiedzialny za wygenerowanie odpowiedniej strony błędu. Na przykład w listingu 10.37 zwrócony przez metodę `handleException()` ciąg tekstowy zawierający komunikat błędu jest używany przez serwet `DispatcherServlet` do wygenerowania strony `/WEB-INF/jsp/error.jsp`. Jeżeli metoda oznaczona adnotacją `@ExceptionHandler` nie zwraca żadnych informacji o widoku (to znaczy typem jej wartości zwrótniej jest `void` lub `Model`), wówczas oferowana przez Spring klasa `RequestToViewNameTranslator` (więcej informacji na jej temat znajdziesz w podrozdziale 11.2) jest używana do ustalenia widoku, który powinien być wyświetlony.

Istnieje możliwość zdefiniowania w klasie kontrolera wielu metod oznaczonych adnotacją `@ExceptionHandler` przeznaczonych do obsługi różnego typu wyjątków. Atrybut *value* adnotacji `@ExceptionHandler` pozwala na określenie typu wyjątku obsługiwanego przez daną metodę. W listingu 10.38 pokazano, że metoda `myExceptionHandler()` obsługuje wyjątki typu `IOException` i `FileNotFoundException`, natomiast metoda `myOtherExceptionHandler()` obsługuje wyjątki typu `TimeoutException`.

Listing 10.38. Określenie typu wyjątków obsługiwanych przez metodę oznaczoną adnotacją `@ExceptionHandler`

```
@Controller
.....
public class MyController {
    .....
    @ExceptionHandler(value = {IOException.class, FileNotFoundException.class})
    public String myExceptionHandler() {
        return "someError";
    }

    @ExceptionHandler(value = TimeoutException.class)
    public String myOtherExceptionHandler() {
        return "otherError";
    }
}
```

Jeżeli kontroler `MyController` zgłosi wyjątek typu `IOException` lub `FileNotFoundException` (bądź wyjątek będący podtypem `IOException` lub `FileNotFoundException`), do jego obsługi zostanie wywołana metoda `myExceptionHandler()`. Natomiast jeśli kontroler `MyController` zgłosi wyjątek

typu `TimeoutException` (bądź wyjątek będący jego podtypem), do jego obsługi zostanie wywołana metoda `myOtherExceptionHandler()`.

Teraz omówimy sposób użycia przez Spring egzemplarza `ContextLoaderListener` do wczytania głównego pliku (lub plików) XML kontekstu aplikacji sieciowej.

10.10. Wczytanie głównego pliku XML kontekstu aplikacji sieciowej

Jak wspomnieliśmy na początku rozdziału, główny plik XML kontekstu aplikacji sieciowej definiuje komponenty współdzielone przez wszystkie serwlety i filtry aplikacji sieciowej. W listingu 10.39 przedstawiamy konfigurację egzemplarza `ContextLoaderListener`.

Listing 10.39. Konfiguracja egzemplarza `ContextLoaderListener`

Projekt: *ch10-bankapp*

Położenie pliku: *src/main/webapp/WEB-INF/web.xml*

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath*:META-INF/spring/applicationContext.xml</param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
```

W powyższym listingu element `<listener>` konfiguruje egzemplarz `ContextLoaderListener` (implementacja `ServletContextListener`) odpowiedzialny za wczytanie głównego pliku (lub plików) XML kontekstu aplikacji sieciowej wskazanego przez parametr inicjalizacyjny kontekstu serwletu `contextConfigLocation`. Element `<context-param>` jest używany do wskazania parametru inicjalizacyjnego kontekstu serwletu. Egzemplarz `ContextLoaderListener` tworzy główny egzemplarz `WebApplicationContext`, za pomocą którego rejestrowane są komponenty wczytane z głównego pliku (lub plików) XML kontekstu aplikacji sieciowej.

W omawianym listingu parametr `contextConfigLocation` wskazuje */META-INF/spring/applicationContext.xml* jako główny plik XML kontekstu aplikacji sieciowej. Istnieje możliwość wskazania wielu plików XML kontekstu aplikacji, trzeba je wówczas rozdzielić przecinkami, średnikami, znakami nowego wiersza lub odstępami. Jeżeli parametr `contextConfigLocation` nie zostanie podany, wtedy `ContextLoaderListener` potraktuje */WEB-INF/applicationContext.xml* jako główny plik XML kontekstu aplikacji sieciowej.

10.11. Podsumowanie

W tym rozdziale przedstawiliśmy niektóre z najważniejszych obiektów występujących w prostej aplikacji Spring Web MVC. Dowiedziałeś się również, jak używać adnotacji `@Controller`, `@RequestMapping`, `@RequestParam` i `@ExceptionHandler` w celu utworzenia kontrolerów oznaczonych adnotacjami. W kolejnym rozdziale zobaczysz, jak Spring *dołącza* parametry żądania do obiektów obsługujących formularza i przeprowadzających weryfikację danych.

Rozdział 11. Weryfikacja i dołączanie danych w Spring Web MVC

11.1. Wprowadzenie

W poprzednim rozdziale analizowaliśmy aplikację sieciową MyBank utworzoną za pomocą adnotacji `@Controller`, `@RequestMapping` i `@RequestParam`. Zobaczyłeś, że dane formularza były pobierane z żądania (patrz: listingi 10.23, 10.28 i 10.29) oraz *wyraźnie* przypisywane obsługującemu go obiektowi (tutaj `FixedDepositDetails`). Ponadto logika odpowiedzialna za weryfikację danych znajdowała się w samej metodzie kontrolera.

W tym rozdziale przeanalizujemy następujące zagadnienia:

- adnotacje `@ModelAttribute` i `@SessionAttributes`, które są użyteczne podczas pracy z atrybutami modelu;
- użycie oferowanego przez Spring egzemplarza `WebDataBinder` w celu łatwiejszego dołączania danych formularza do obsługującego go obiektu;
- weryfikacja obiektów obsługujących formularz za pomocą API Validation dostępnego w Springu, a także za pomocą adnotacji ograniczeń zdefiniowanych w specyfikacji JSR 303.
- Użycie oferowanej przez Spring biblioteki form ułatwiającej tworzenie stron JSP.

Zaczynamy od analizy adnotacji `@ModelAttribute`, która jest używana w celu dodawania i pobierania atrybutów modelu do obiektu `Model` istniejącego w Spring i z tego obiektu.

11.2. Dodawanie i pobieranie atrybutów modelu z pomocą adnotacji `@ModelAttribute`

W poprzednim rozdziale zobaczyłeś, że metoda oznaczona adnotacją `@RequestMapping` przechowuje atrybuty modelu w egzemplarzu `HashMap` (lub `ModelMap`), natomiast zwraca te atrybuty za pomocą obiektu `ModelAndView`. Atrybuty modelu zwrócone przez metodę oznaczoną adnotacją `@RequestMapping` są przechowywane w oferowanym przez Spring modelu `Model`.

Atrybut modelu może przedstawiać obiekt obsługujący formularz lub *dane referencyjne*. Obiekt `FixedDepositDetails` w aplikacji sieciowej MyBank jest przykładem obiektu obsługującego formularz. Po wysłaniu formularza umożliwiającego założenie nowej lokaty znajdujące się w nim informacje będą przechowywane w obiekcie `FixedDepositDetails`. Jako obiekty obsługujące formularz w aplikacji najczęściej są stosowane obiekty *domeny* lub encje. Pojęcie *dane referencyjne* odwołuje się do informacji dodatkowych (innych niż pochodzące z obiektu obsługującego formularz) wymaganych przez widok. Na przykład po dodaniu do każdej lokaty kategorii użytkownika (np. wojskowy, emeryt) formularz przeznaczony do zakładania nowych lokat powinien wyświetlać listę rozwijaną zawierającą dostępne kategorie. Lista kategorii to *dane referencyjne* niezbędne do wyświetlenia formularza umożliwiającego założenie nowej lokaty.

Adnotacja `@ModelAttribute` jest używana w metodach i argumentach metod do, odpowiednio, przechowywania i pobierania atrybutów modelu z oferowanego przez Spring obiektu `Model`. Adnotacja `@ModelAttribute` w metodzie wskazuje, że dana metoda dodaje jeden lub więcej atrybutów modelu do obiektu `Model`. Ponadto adnotacja `@ModelAttribute` w argumencie metody jest używana do pobierania atrybutu modelu z obiektu `Model` i przypisania go argumentowi metody.

IMPORT **Rozdział 11/ch11-bankapp.** (Ten projekt pokazuje aplikację sieciową `MyBank`, w której użyto adnotacji `@ModelAttribute` i oferowanej przez Spring biblioteki znaczników o nazwie `form`. Funkcjonalność oferowana przez aplikację sieciową `MyBank` w projektach `ch10-bankapp` i `ch11-bankapp` jest taka sama. Po wdrożeniu aplikacji w serwerze Tomcat przejdź pod adres URL `http://localhost:8080/ch11-bankapp`, a zobaczysz wyświetloną listę lokat w systemie).

Najpierw zajmiemy się metodami oznaczonymi atrybutem `@ModelAttribute`.

Dodanie atrybutów modelu za pomocą adnotacji `@ModelAttribute` stosowane na poziomie metody

W listingu 11.1 została przedstawiona metoda `getNewFixedDepositDetails()` klasy `FixedDepositController` oznaczona adnotacją `@ModelAttribute`.

Listing 11.1. Przykład użycia adnotacji `@ModelAttribute` na poziomie metody

Projekt: `ch11-bankapp`

Położenie pliku: `src/main/java/sample/spring/chapter11/web`

```
package sample.spring.chapter11.web;

import org.springframework.web.bind.annotation.ModelAttribute;
import sample.spring.chapter11.domain.FixedDepositDetails;
.....
@Controller
@RequestMapping(value = "/fixedDeposit")
.....
public class FixedDepositController {
    private static Logger logger = Logger.getLogger(FixedDepositController.class);
    .....
    @ModelAttribute(value = "newFixedDepositDetails")
    public FixedDepositDetails getNewFixedDepositDetails() {
        FixedDepositDetails fixedDepositDetails = new FixedDepositDetails();
        fixedDepositDetails.setEmail("Musisz podać poprawny adres e-mail.");
        logger.info("Metoda getNewFixedDepositDetails(): zwrot nowego egzemplarza
            FixedDepositDetails");
        return fixedDepositDetails;
    }
    .....
}
```

Metoda `getNewFixedDepositDetails()` tworzy i zwraca nowy egzemplarz obiektu `FixedDepositDetails`. Ponieważ metoda `getNewFixedDepositDetails()` jest oznaczona adnotacją `@ModelAttribute`, zwracany przez nią egzemplarz `FixedDepositDetails` jest dodawany do obiektu `Model`. Atrybut `value` adnotacji `@ModelAttribute` określa, że zwrócony przez tę metodę obiekt `FixedDepositDetails` jest w obiekcie `Model` przechowywany pod nazwą `newFixedDepositDetails`. Zauważ, że metoda `getNew`

- ↳ `FixedDepositDetails()` wyświetla w konsoli komunikat o treści Metoda `getNewFixedDepositDetails()`:
- ↳ `Details()`: zwrot nowego egzemplarza `FixedDepositDetails`.



Zwróć uwagę na fakt, że zasięgiem dla atrybutów modelu jest *żądanie*. Oznacza to utratę atrybutów modelu po zakończeniu żądania lub po jego *przekierowaniu*.

W dalszej części rozdziału dowiesz się, jak strona `createFixedDepositForm.jsp` (patrz: plik `src/main/webapp/WEB-INF/jsp/createFixedDepositForm.jsp`) w projekcie `ch11-bankapp` używa oferowanej przez Spring biblioteki form w celu uzyskania dostępu do obiektu `FixedDepositDetails` o nazwie `newFixedDepositDetails` przechowywanego w obiekcie `Model`.

Jeżeli atrybut `value` adnotacji `@ModelAttribute` nie zostanie podany, to zwrócony obiekt będzie przechowywany w obiekcie `Model` pod taką samą nazwą jak nazwa *typu* zwracanego obiektu. W listingu 11.2 obiekt `Sample` zwracany przez metodę `getSample()` jest w obiekcie `Model` przechowywany pod nazwą `sample`.

Listing 11.2. Przykład użycia adnotacji `@ModelAttribute`, gdy atrybut `value` nie jest określony

```
import org.springframework.ui.Model;
.....
public class SampleController {

    @ModelAttribute
    public Sample getSample() {
        return new Sample();
    }
}
```

Metoda oznaczona adnotacją `@ModelAttribute` akceptuje ten sam typ argumentów co metoda oznaczona adnotacją `@RequestMapping`. W listingu 11.3 została przedstawiona metoda oznaczona adnotacją `@ModelAttribute` i akceptująca argument typu `HttpServletRequest`.

Listing 11.3. Przykład użycia oznaczonej adnotacją `@ModelAttribute` metody, która akceptuje argument w postaci `HttpServletRequest`

```
@ModelAttribute(value = "myObject")
public SomeObject doSomething(HttpServletRequest request) { ..... }
```

W rozdziale 10. zobaczyłeś, jak adnotacja `@RequestParam` jest używana w celu przekazania parametrów żądania do metody oznaczonej adnotacją `@RequestMapping`. Adnotacja `@RequestParam` może być używana także do przekazywania parametrów żądania do metody oznaczonej adnotacją `@ModelAttribute`, jak przedstawia listing 11.4.

Listing 11.4. Przekazanie parametrów żądania do metody oznaczonej adnotacją `@ModelAttribute`

```
@ModelAttribute(value = "myObject")
public SomeObject doSomething(@RequestParam("someArg") String myarg) { ..... }
```

Ponieważ metody oznaczone adnotacjami `@RequestMapping` i `@ModelAttribute` mogą akceptować argument w postaci obiektu `Model`, istnieje możliwość *bezpośredniego* dodania atrybutów modelu

do obiektu `Model` w metodzie oznaczonej adnotacją `@ModelAttribute` lub `@RequestMapping`. Listing 11.5 przedstawia oznaczoną adnotacją `@ModelAttribute` metodę, która bezpośrednio dodaje atrybuty modelu do obiektu `Model`.

Listing 11.5. Bezpośrednie dodawanie atrybutów modelu do obiektu `Model`

```
import org.springframework.ui.Model;
.....
public class SampleWebController {

    @ModelAttribute
    public void doSomething(Model model) {
        model.addAttribute("myobject", new MyObject());
        model.addAttribute("otherobject", new OtherObject());
    }
}
```

W powyższym listingu obiekt `Model` jest przekazywany jako argument metodzie `doSomething()`, która bezpośrednio dodaje atrybuty modelu do obiektu `Model`. Dlatego typ wartości zwrótej metody `doSomething()` został określony jako `void`, a atrybut `value` adnotacji `@ModelAttribute` *nie* został użyty.

Istnieje możliwość oznaczenia metody zarówno adnotacją `@RequestMapping`, jak i `@ModelAttribute`. W listingu 11.6 widać, że metoda `listFixedDeposits()` klasy `FixedDepositController` jest jednocześnie oznaczona adnotacjami `@RequestMapping` i `@ModelAttribute`.

Listing 11.6. Przykład użycia adnotacji `@ModelAttribute` i `@RequestMapping` w tej samej metodzie

Projekt: *ch11-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter11/web*

```
package sample.spring.chapter11.web;
.....
@Controller
@RequestMapping(value = "/fixedDeposit")
.....
public class FixedDepositController {
    private static Logger logger = Logger.getLogger(FixedDepositController.class);

    @RequestMapping(value = "/list", method = RequestMethod.GET)
    @ModelAttribute(value = "fdList")
    public List<FixedDepositDetails> listFixedDeposits() {
        logger.info("Metoda listFixedDeposits(): pobranie listy lokat.");
        return fixedDepositService.getFixedDeposits();
    }
    .....
}
```

Metoda `listFixedDeposits()` generuje stronę *list.jsp* (patrz: plik *src/main/webapp/WEB-INF/jsp/fixedDeposit/list.jsp* w projekcie *ch11-bankapp*) wyświetlającą listę lokat w systemie. Kiedy metoda jest jednocześnie oznaczona adnotacjami `@RequestMapping` i `@ModelAttribute`, wówczas jej wartość zwrótna jest uznawana za atrybut modelu, a *nie* nazwę widoku. W takiej sytuacji nazwa widoku będzie ustalona za pomocą oferowanej przez Spring klasy `RequestMappingHandlerMethod`, która określa widok do wygenerowania, opierając się na adresie URI przychodzącego żądania.

W dalszej części rozdziału dokładniej przeanalizujemy klasę `RequestToViewNameTranslator`. W listingu 11.6 zwróć uwagę, że metoda `listFixedDeposits()` wyświetla komunikat o treści Metoda `listFixedDeposits()`: pobranie listy lokat.

Warto pamiętać o możliwości zdefiniowania w kontrolerze wielu metod oznaczonych adnotacją `@ModelAttribute`. Kiedy żądanie jest przekazane do oznaczonej adnotacją `@RequestMapping` metody kontrolera, wtedy *wszystkie* metody oznaczone adnotacją `@ModelAttribute` są przez kontroler wykonywane *przed* wywołaniem metody oznaczonej adnotacją `@RequestMapping`. W listingu 11.7 został przedstawiony kontroler definiujący metody oznaczone adnotacjami `@RequestMapping` i `@ModelAttribute`.

Listing 11.7. Metoda oznaczona adnotacją `@RequestMapping` jest wywoływana po wykonaniu wszystkich metod oznaczonych adnotacją `@ModelAttribute`

```
@RequestMapping("/mycontroller")
public class MyController {

    @RequestMapping("/perform")
    public String perform() { ..... }

    @ModelAttribute(value = "a")
    public A getA() { ..... }

    @ModelAttribute(value = "b")
    public B getB() { ..... }
}
```

W powyższym listingu, jeśli żądanie będzie mapowane na metodę `perform()` kontrolera `MyController`, wówczas Spring Web MVC najpierw wykona metody `getA()` i `getB()`, a następnie wywoła metodę `perform()`.

Jeżeli metoda jest jednocześnie oznaczona adnotacjami `@RequestMapping` i `@ModelAttribute`, wówczas zostanie wykonana tylko raz podczas przetwarzania żądania. W listingu 11.8 został przedstawiony kontroler definiujący metodę oznaczoną adnotacjami `@RequestMapping` i `@ModelAttribute`.

Listing 11.8. Metoda oznaczona adnotacjami `@RequestMapping` i `@ModelAttribute` jest wywoływana tylko raz podczas przetwarzania żądania

```
@RequestMapping("/mycontroller")
public class MyController {

    @RequestMapping("/perform")
    @ModelAttribute
    public String perform() { ..... }

    @ModelAttribute(value = "a")
    public A getA() { ..... }

    @ModelAttribute(value = "b")
    public B getB() { ..... }
}
```

W powyższym listingu jeśli żądanie jest mapowane na metodę `perform()` kontrolera `MyController`, wówczas Spring Web MVC najpierw wykona metody `getA()` i `getB()`, a następnie wywoła metodę `perform()`. Ponieważ metoda `perform()` jest oznaczona adnotacjami `@RequestMapping` i `@ModelAttribute` ➔bute, oferowana przez Spring klasa `RequestToViewNameTranslator` jest używana do ustalenia nazwy widoku, który powinien być wygenerowany po wykonaniu metody `perform()`.

Jeżeli projekt *ch11-bankapp* wdrożysz w serwerze Tomcat i przejdiesz pod adres URL *http://localhost:8080/ch11-bankapp/fixedDeposit/list*, zobaczysz stronę internetową wyświetlającą listę lokat. Ponadto w konsoli znajdują się następujące komunikaty:

```
INFO sample.spring.chapter11.web.FixedDepositController – Metoda getNewFixedDepositDetails():  
➔zwrot nowego egzemplarza FixedDepositDetails.  
INFO sample.spring.chapter11.web.FixedDepositController – Metoda listFixedDeposits(): pobranie  
➔listy lokat.
```

Powyższe dane wyjściowe pokazują, że metoda `getNewFixedDepositDetails()` oznaczona adnotacją `@ModelAttribute` jest wywoływana jako pierwsza, a dopiero później jest wykonywana metoda `listFixedDeposits()` oznaczona adnotacjami `@ModelAttribute` i `@RequestMapping`.

Zobaczmy teraz, jak atrybuty modelu są pobierane z obiektu `Model` za pomocą adnotacji `@ModelAttribute` w metodzie.

Pobieranie atrybutów modelu za pomocą adnotacji `@ModelAttribute`

Istnieje możliwość użycia adnotacji `@ModelAttribute` w argumentach metody oznaczonej adnotacją `@RequestMapping` w celu pobrania atrybutów modelu z obiektu `Model`.

W listingu 11.9 została przedstawiona metoda `openFixedDeposit()` klasy `FixedDepositController` używająca adnotacji `@ModelAttribute` do pobrania obiektu `newFixedDepositDetails` z obiektu `Model`.

Listing 11.9. Przykład użycia adnotacji `@ModelAttribute` w argumencie metody

Projekt: *ch11-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter11/web*

```
package sample.spring.chapter11.web;  
.....  
@Controller  
@RequestMapping(value = "/fixedDeposit")  
.....  
public class FixedDepositController {  
    .....  
    @ModelAttribute(value = "newFixedDepositDetails")  
    public FixedDepositDetails getNewFixedDepositDetails() {  
        .....  
        logger.info("Metoda getNewFixedDepositDetails(): zwrot nowego egzemplarza  
            FixedDepositDetails");  
        .....  
    }  
    .....  
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)  
    public String openFixedDeposit(  

```

```

    @ModelAttribute(value = "newFixedDepositDetails")
    FixedDepositDetails fixedDepositDetails,....) {
        ....
        fixedDepositService.saveFixedDeposit(fixedDepositDetails);
        logger.info("Metoda openFixedDeposit(): zapisano informacje o lokacie.
            Przekierowanie w celu wyświetlenia listy lokat.");
        ....
    }
}
.....
}

```

W powyższym listingu oznaczona adnotacją `@ModelAttribute` metoda `getNewFixedDepositDetails()` jest wywoływana *przed* oznaczoną adnotacją `@RequestMapping` metodą `openFixedDeposit()`. Po wywołaniu metody `getNewFixedDepositDetails()` zwrócony przez nią egzemplarz `FixedDepositDetails` jest przechowywany w obiekcie `Model` pod nazwą `newFixedDepositDetails`. Teraz argument `fixedDepositDetails` metody `openFixedDeposit()` jest oznaczony adnotacją `@ModelAttribute` (value="newFixedDepositDetails"). Dlatego obiekt `newFixedDepositDetails` zostanie pobrany z obiektu `Model` i przypisany argumentowi `fixedDepositDetails`.

Jeżeli przyjrzyś się metodzie `openFixedDeposit()` klasy `FixedDepositController`, zauważysz *brak* logiki odpowiedzialnej za pobranie z żądania danych (wysokość lokaty, czas jej trwania i adres e-mail klienta) przeznaczonych do wypełniania egzemplarza `newFixedDepositDetails`. Wynika to z faktu, że oferowany przez Spring obiekt `WebDataBinder` (dokładnie omówiony w dalszej części rozdziału) jest odpowiedzialny za pobranie wymienionych parametrów żądania i wypełnienie nimi odpowiednich (o dopasowanych nazwach) właściwości egzemplarza `newFixedDepositDetails`. Na przykład po znalezieniu w żądaniu parametru o nazwie `tenure` obiekt `WebDataBinder` przypisze właściwości `tenure` nowego egzemplarza `newFixedDepositDetails` wartość parametru żądania `tenure`.

Na rysunku 11.1 pokazano podsumowanie sekwencji działań podejmowanych przez Spring, gdy żądanie jest przekazywane metodzie `openFixedDeposit()` klasy `FixedDepositController`.

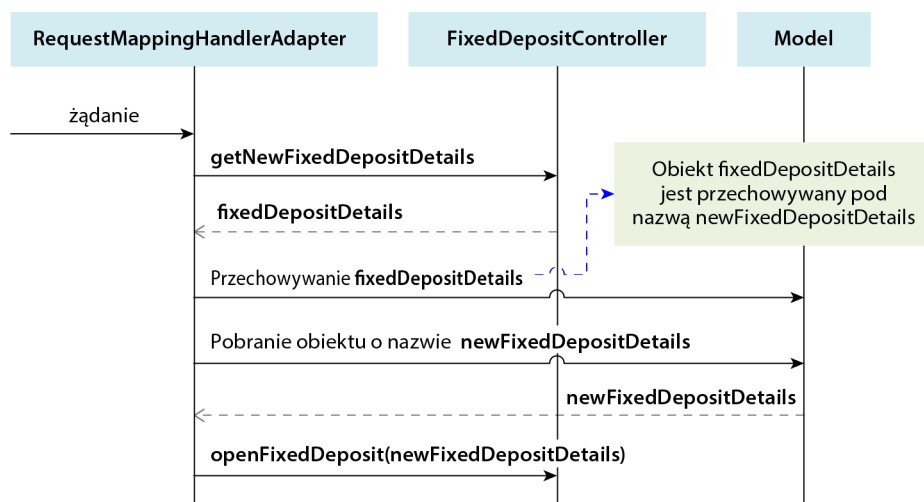
Na rysunku 11.1 udostępniany przez Spring Web MVC obiekt `RequestMappingHandlerAdapter` jest odpowiedzialny za wywołanie metod kontrolera oznaczonych adnotacjami `@ModelAttribute` i `@RequestMapping`. Najpierw wywoływana jest metoda `getNewFixedDepositDetails()`, a zwrócony przez nią egzemplarz `FixedDepositDetails` jest przechowywany w obiekcie `Model` pod nazwą `newFixedDepositDetails`. Następnie egzemplarz `newFixedDepositDetails` jest pobierany z obiektu `Model` i przekazywany jako argument metodzie `openFixedDeposit()`.

Przekonajmy się teraz, kiedy podczas przetwarzania żądania jest wywołana metoda oznaczona adnotacją `@ModelAttribute`.

Przetwarzanie żądania i metody oznaczone adnotacją `@ModelAttribute`

W listingu 11.6 widziałeś, że wykonanie metody `listFixedDeposits()` powoduje wyświetlenie w konsoli następującego komunikatu:

```
Metoda listFixedDeposits(): pobranie listy lokat.
```



Rysunek 11.1. Kolejność wywołania w kontrolerze *FixedDepositController* metod oznaczonych adnotacjami *@ModelAttribute* i *@RequestMapping*

W listingu 11.9 widziałeś, że wykonanie metody `getNewFixedDepositDetails()` powoduje wyświetlenie w konsoli następującego komunikatu:

Metoda `getNewFixedDepositDetails()`: zwrot nowego egzemplarza `FixedDepositDetails`.

Z kolei metoda `openFixedDeposit()` wyświetla poniższy komunikat:

Metoda `openFixedDeposit()`: zapisano informacje o lokacie. Przekierowanie w celu wyświetlenia ↪ listy lokat.

Aby poznać kolejność wywoływania metod `listFixedDeposits()`, `getNewFixedDepositDetails()` i `openFixedDeposit()`, przeprowadź wdrożenie projektu *ch11-bankapp*, a następnie wykonaj następujące kroki:

1. Przejdź do adresu URL `http://localhost:8080/ch11-bankapp/fixedDeposit/list`. Powinieneś zobaczyć listę lokat w systemie oraz przycisk *Załącz nową lokatę* (patrz: rysunek 10.8).
2. Kliknij przycisk *Załącz nową lokatę*. Spowoduje to wyświetlenie formularza HTML umożliwiającego założenie nowej lokaty (patrz: rysunek 10.9).
3. Podaj szczegóły lokaty i kliknij przycisk *Zapisz*. Jeżeli wprowadzone dane nie zawierają żadnych błędów, nowa lokata zostanie zapisana. Następnie ponownie będzie wyświetlona lista lokat znajdujących się w systemie (nowa lokata również znajdzie się na liście).

W tabeli 11.1 wymieniamy operacje wykonywane przez użytkownika i odpowiadające im komunikaty, które są przez aplikację *MyBank* wyświetlane w konsoli.

W tabeli 11.1 widać, że oznaczona adnotacją `@ModelAttribute` metoda `getNewFixedDepositDetails()` jest wywołana *przed wszystkimi* wywołaniami metod oznaczonych adnotacją `@RequestMapping` w klasie `FixedDepositController`. Ponieważ metoda `getNewFixedDepositDetails()` tworzy nowy egzemplarz obiektu `FixedDepositDetails`, obiekt ten jest tworzony za każdym razem, gdy żądanie jest obsługiwane przez kontroler `FixedDepositController`.

Tabela 11.1. Operacje wykonywane przez użytkownika i odpowiadające im komunikaty

Operacja	Komunikaty wyświetlane w konsoli
Przejdźcie do adresu URL <i>http://localhost:8080/ ch11-bankapp/fixedDeposit/list</i>	Metoda <code>getNewFixedDepositDetails()</code> — zwrot nowego egzemplarza <code>FixedDepositDetails</code> . Metoda <code>listFixedDeposits()</code> — pobranie listy założonych lokat.
Kliknięcie przycisku <i>Założ nową lokatę</i>	Metoda <code>getNewFixedDepositDetails()</code> — zwrot nowego egzemplarza <code>FixedDepositDetails</code> . Metoda <code>showOpenFixedDepositForm()</code> — wyświetlenie formularza umożliwiającego założenie nowej lokaty.
Wprowadzenie informacji szczegółowych o lokacie i kliknięcie przycisku <i>Zapisz</i>	Metoda <code>getNewFixedDepositDetails()</code> — zwrot nowego egzemplarza <code>FixedDepositDetails</code> . Metoda <code>openFixedDeposit()</code> — zapisano informacje o lokacie; przekierowanie w celu wyświetlenia listy lokat. Metoda <code>getNewFixedDepositDetails()</code> — zwrot nowego egzemplarza <code>FixedDepositDetails</code> . Metoda <code>listFixedDeposits()</code> — pobranie listy założonych lokat.

Jeżeli oznaczona adnotacją `@ModelAttribute` metoda wykonuje zapytanie SQL lub wywołuje zewnętrzną usługę sieciową w celu wypełnienia atrybutu modelu zwróconego przez metodę, wówczas wiele wywołań metody oznaczonej adnotacją `@ModelAttribute` niewątpliwie będzie miało negatywny wpływ na wydajność aplikacji. W dalszej części rozdziału zobaczysz, że dzięki adnotacji `@SessionAttributes` można uniknąć wielu wywołań metody oznaczonej adnotacją `@ModelAttribute`. Adnotacja `@SessionAttributes` nakazuje Springowi buforowanie obiektu zwróconego przez metodę oznaczoną adnotacją `@ModelAttribute`.

Przechodzimy teraz do omówienia sytuacji, w której atrybut modelu wskazywany przez oznaczony adnotacją `@ModelAttribute` argument metody *nie* znajduje się w obiekcie `Model`.

Zachowanie argumentów metody oznaczonych adnotacją `@ModelAttribute`

Z poprzednich podrozdziałów dowiedziałeś się, że adnotacja `@ModelAttribute` może być użyta w argumencie metody do pobrania atrybutu modelu z obiektu `Model`. Jeżeli atrybut modelu określony przed adnotacją `@ModelAttribute` *nie* znajduje się w obiekcie `Model`, Spring automatycznie tworzy nowy egzemplarz typu argumentu metody, przypisuje mu argument metody i umieszcza w obiekcie `Model`. Aby umożliwić Springowi utworzenie egzemplarza typu argumentu metody, klasa Javy typu argumentu metody *musi* zawierać bezargumentowy konstruktor.

Spójrz na przedstawiony w listingu 11.10 kontroler `SomeController` definiujący pojedynczą metodę `doSomething()` oznaczoną adnotacją `@RequestMapping`.

Listing 11.10. Argument oznaczony adnotacją `@ModelAttribute` nie jest dostępny w obiekcie `Model`

```
@Controller
@RequestMapping(value = "/some")
public class SomeController {
    ....
}
```

```
@RequestMapping("/do")
public void doSomething(@ModelAttribute("myObj") MyObject myObject) {
    logger.info(myObject);
    .....
}
}
```

W powyższym listingu widać, że klasa `SomeController` nie definiuje żadnej oznaczonej adnotacją `@ModelAttribute` metody, która dodaje obiekt o nazwie `myObj` i typie `MyObject` do obiektu `Model`. Dlatego po otrzymaniu żądania do metody `doSomething()` Spring tworzy egzemplarz `MyObject`, przypisuje mu argument `myObject`, a następnie nowo utworzony egzemplarz `MyObject` umieszcza w obiekcie `Model`.

Przechodzimy teraz do analizy obiektu `RequestToViewNameTranslator` udostępnianego przez Spring.

Obiekt `RequestToViewNameTranslator`

Obiekt `RequestToViewNameTranslator` ustala widok do wygenerowania, gdy metoda oznaczona adnotacją `@RequestMapping` nie podaje wyraźnie nazwy widoku, który powinien zostać wygenerowany.

Wcześniej w tym rozdziale zobaczyłeś, że kiedy metoda oznaczona adnotacją `@RequestMapping` ma również adnotację `@ModelAttribute`, wówczas jej wartość zwrrotna jest uznawana za atrybut modelu. W takiej sytuacji obiekt `RequestToViewNameTranslator` staje się odpowiedzialny za ustalenie na podstawie przychodzącego żądania sieciowego nazwy widoku przeznaczonego do wygenerowania. Podobnie jeśli oznaczona adnotacją `@RequestMapping` metoda zwraca `void`, `org.springframework.ui.Model` lub `java.util.Map`, wtedy obiekt `RequestToViewNameTranslator` ustala widok przeznaczony do wygenerowania.

`DefaultRequestToViewNameTranslator` to implementacja `RequestToViewNameTranslator` domyślnie używana przez serwet `DispatcherServlet` w celu ustalenia widoku do wygenerowania, gdy metoda oznaczona adnotacją `@RequestMapping` wyraźnie nie wskazuje takiego widoku. W celu ustalenia nazwy widoku logicznego do wygenerowania egzemplarz `DefaultRequestToViewNameTranslator` używa adresu URI. Z użytego adresu URI egzemplarz `DefaultRequestToViewNameTranslator` usuwa ukośniki znajdujące się na początku i na końcu adresu oraz rozszerzenie pliku, w ten sposób ustalając nazwę widoku. Na przykład jeśli adres URL ma postać `http://localhost:8080/doSomething.htm`, nazwą widoku jest `doSomething`.

W przypadku aplikacji sieciowej `MyBank` metoda `listFixedDeposits()` klasy `FixedDepositController` (patrz: listing 11.6 lub plik `FixedDepositController.java` w projekcie `ch11-bankapp`) jest oznaczona adnotacjami `@RequestMapping` i `@ModelAttribute`. Dlatego egzemplarz `RequestToViewNameTranslator` jest używany przez serwet `DispatcherServlet` w celu ustalenia widoku do wygenerowania. Ponieważ metoda `listFixedDeposits()` jest mapowana na adres URI `/fixedDeposit/list`, egzemplarz `RequestToViewNameTranslator` zwraca `/fixedDeposit/list` jako nazwę widoku. Egzemplarz `ViewResolver` skonfigurowany w pliku XML konspektu aplikacji sieciowej `MyBank` (patrz: plik `bankapp-config.xml` w projekcie `ch11-bankapp`) mapuje nazwę widoku `/fixedDeposit/list` na stronę `/WEB-INF/jsp/fixedDeposit/list.jsp`.

Przechodzimy teraz do omówienia adnotacji `@SessionAttributes`.

11.3. Buforowanie atrybutów modelu za pomocą adnotacji `@SessionAttributes`

W poprzednim podrozdziale zobaczyłeś, że *wszystkie* metody kontrolera oznaczone adnotacją `@ModelAttribute` są *zawsze* wywoływane przed metodami oznaczonymi adnotacją `@RequestMapping`. Takie zachowanie może nie być akceptowane w sytuacjach, w których metody oznaczone adnotacją `@ModelAttribute` pobierają dane z bazy danych lub zewnętrznej usługi sieciowej, aby wypełnić atrybut modelu. W takim przypadku klasę kontrolera można oznaczyć adnotacją `@SessionAttributes` oznaczającą, że atrybuty modelu są między żądaniami przechowywane w obiekcie `HttpSession`.

Gdy dla klasy użyta jest adnotacja `@SessionAttributes`, wówczas metoda oznaczona adnotacją `@ModelAttribute` będzie wywoływana tylko wtedy, gdy atrybut oznaczony adnotacją `@ModelAttribute` *nie* znajduje się w obiekcie `HttpSession`. Ponadto adnotacja `@ModelAttribute` w argumencie metody spowoduje utworzenie nowego egzemplarza atrybutu modelu tylko wtedy, gdy atrybut modelu nie znajduje się w obiekcie `HttpSession`.

IMPORT **Rozdział11/ch11-session-attributes.** (Ten projekt pokazuje zmodyfikowaną wersję projektu *ch11-bankapp* używającą adnotacji `@SessionAttributes` do tymczasowego przechowywania atrybutów modelu w `HttpSession`. Funkcjonalność oferowana przez aplikację sieciową *MyBank* w projektach *ch10-bankapp* i *ch11-session-attributes* jest taka sama. Po wdrożeniu aplikacji w serwerze Tomcat przejdź pod adres URL <http://localhost:8080/ch11-session-attributes>, a zobaczysz wyświetloną listę lokat w systemie).

W listingu 11.11 przedstawiamy przykład użycia adnotacji `@SessionAttributes` w projekcie *ch11-session-attributes* do tymczasowego przechowywania atrybutów modelu `newFixedDepositDetails` i `editableFixedDepositDetails` w obiekcie `HttpSession`.

Listing 11.11. Przykład użycia adnotacji `@SessionAttributes`

Projekt: *ch11-session-attributes*

Położenie pliku: *src/main/java/sample/spring/chapter11/web*

```
package sample.spring.chapter11.web;

import org.springframework.web.bind.annotation.SessionAttributes;
.....
@SessionAttributes(value = { "newFixedDepositDetails", "editableFixedDepositDetails" })
public class FixedDepositController {
    .....
    @ModelAttribute(value = "newFixedDepositDetails")
    public FixedDepositDetails getNewFixedDepositDetails() {
        FixedDepositDetails fixedDepositDetails = new FixedDepositDetails();
        fixedDepositDetails.setEmail("Należy podać prawidłowy adres e-mail.");
        return fixedDepositDetails;
    }
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
    public String openFixedDeposit(
        @ModelAttribute(value = "newFixedDepositDetails") FixedDepositDetails
```

```

        fixedDepositDetails, ..... ) { ..... }
    .....
    @RequestMapping(params = "fdAction=view", method = RequestMethod.GET)
    public ModelAndView viewFixedDepositDetails(
        @RequestParam(value = "fixedDepositId") int fixedDepositId) {
        FixedDepositDetails fixedDepositDetails = fixedDepositService
            .getFixedDeposit(fixedDepositId);
        Map<String, Object> modelMap = new HashMap<String, Object>();
        modelMap.put("editableFixedDepositDetails", fixedDepositDetails);
        .....
        return new ModelAndView("editFixedDepositForm", modelMap);
    }
}

```

Atrybut `value` adnotacji `@SessionAttributes` określa *nazwy* atrybutów modelu tymczasowo przechowywanych w obiekcie `HttpSession`. W powyższym listingu atrybuty modelu o nazwach `newFixedDepositDetails` i `editableFixedDepositDetails` są między zadaniami przechowywane w obiekcie `HttpSession`. Atrybut modelu `newFixedDepositDetails` jest zwracany przez oznaczoną adnotacją `@ModelAttribute` metodę `getNewFixedDepositDetails()`, natomiast atrybut modelu `editableFixedDepositDetails` jest zwracany przez oznaczoną adnotacją `@RequestMapping` metodę `viewFixedDepositDetails()`.

Kontroler dostarcza atrybuty modelu za pomocą metod oznaczonych adnotacją `@ModelAttribute`, metod oznaczonych adnotacją `@RequestMapping` (ich wartościami zwrótnymi są obiekty `ModelAndView`, `Model` lub `Map`), a także przez bezpośrednie dodawanie atrybutów modelu do obiektu `Model`. Dostarczane przez kontroler atrybuty modelu za pomocą *dowolnego* z wymienionych sposobów są kandydatami do przechowywania ich w obiekcie `HttpSession` przez użycie adnotacji `@SessionAttributes`.

Podczas użycia adnotacji `@SessionAttributes` trzeba się upewnić, że atrybuty modelu przechowywane w obiekcie `HttpSession` zostaną z niego usunięte, gdy nie będą już potrzebne. Na przykład atrybut modelu `newFixedDepositDetails` przedstawia egzemplarz `FixedDepositDetails` używany przez formularz służący do zakładania nowej lokaty do wyświetlenia komunikatu. Należy podać prawidłowy adres e-mail (patrz: metoda `getNewFixedDepositDetails()` w listingu 11.11). Ponadto po kliknięciu przez użytkownika przycisku *Zapisz* w formularzu zakładania nowej lokaty dane wprowadzone w formularzu zostaną przypisane egzemplarzowi `newFixedDepositDetails` (patrz: metoda `openFixedDeposit()` w listingu 11.11). Po założeniu nowej lokaty egzemplarz `newFixedDepositDetails` nie jest już potrzebny i dlatego powinien zostać usunięty z obiektu `HttpSession`. Podobnie atrybut modelu `editableFixedDepositDetails` nie jest potrzebny po zakończonej powodzeniem modyfikacji informacji szczegółowych o lokacie.

Można nakazać Springowi usunięcie *wszystkich* atrybutów modelu przechowywanych w obiekcie `HttpSession`, co wymaga wywołania metody `setComplete()` udostępnianego przez Spring obiektu `SessionStatus`. W listingu 11.12 widać, jak metody `openFixedDeposit()` i `editFixedDeposit()` klasy `FixedDepositController` wywołują metodę `setComplete()` klasy `SessionStatus` po zakończonej powodzeniem operacji zakładania lub modyfikacji lokaty.

W powyższym listingu widać, że metody `openFixedDeposit()` i `editFixedDeposit()` zostały zdefiniowane tak, aby akceptować argument typu `SessionStatus`. Kiedy metoda oznaczona adnotacją

Listing 11.12. Usunięcie atrybutów modelu z obiektu *HttpSession* za pomocą obiektu *HttpSession*Projekt: *ch11-session-attributes*Położenie pliku: *src/main/java/sample/spring/chapter11/web*

```

package sample.spring.chapter11.web;
import org.springframework.web.bind.support.SessionStatus;
.....
@SessionAttributes(value = { "newFixedDepositDetails", "editableFixedDepositDetails" })
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
    public String openFixedDeposit(
        @ModelAttribute(value = "newFixedDepositDetails") FixedDepositDetails
        fixedDepositDetails, ....., SessionStatus sessionStatus) {
        fixedDepositService.saveFixedDeposit(fixedDepositDetails);
        sessionStatus.setComplete();
    }

    @RequestMapping(params = "fdAction=edit", method = RequestMethod.POST)
    public String editFixedDeposit(
        @ModelAttribute("editableFixedDepositDetails") FixedDepositDetails
        fixedDepositDetails, ....., SessionStatus sessionStatus) {
        fixedDepositService.editFixedDeposit(fixedDepositDetails);
        sessionStatus.setComplete();
        .....
    }
}
.....
}

```

`@RequestMapping` określa argument typu `SessionStatus`, Spring dostarcza takiej metodzie egzemplarz `SessionStatus`. Wywołanie metody `setComplete()` nakazuje Springowi usunięcie z obiektu `HttpSession` atrybutów modelu *bieżącego* kontrolera.

W listingach 11.11 i 11.12 widzieliśmy, że atrybut `value` adnotacji `@SessionAttributes` pozwala na podanie nazw atrybutów modelu, które mają być tymczasowo przechowywane w `HttpSession`. Jeżeli obiekt `HttpSession` ma przechowywać jedynie określone *typy* atrybutów modelu, wówczas można użyć atrybutu `types` adnotacji `@SessionAttributes`. Na przykład przedstawiona poniżej adnotacja `@SessionAttributes` określa, że atrybuty o nazwach *x* i *y* oraz *wszystkie* atrybuty modelu typu `MyObject` będą tymczasowo przechowywane w `HttpSession`.

```
@SessionAttributes(value = { "x", "y" }, types = { MyObject.class })
```

Kolejność wykonywania metod `listFixedDeposits()`, `getNewFixedDepositDetails()` i `openFixedDeposit()` możesz zobaczyć po wdrożeniu projektu *ch11-session-attributes* i wykonaniu operacji wymienionych w tabeli 11.2.

W projekcie *ch11-bankapp* widziałeś, że oznaczona adnotacją `@ModelAttribute` metoda `getNewFixedDepositDetails()` klasy `FixedDepositController` była wywoływana za każdym razem, gdy żądanie docierało do kontrolera `FixedDepositController`. Z tabeli 11.2 wynika, że metoda `getNewFixedDepositDetails()` jest wywoływana, gdy żądanie jest po raz pierwszy obsługiwane przez kontroler

Tabela 11.2. Wykonywane operacje i generowane przez nie komunikaty

Operacja	Komunikaty wyświetlane w konsoli
Przejdźcie do adresu URL <i>http://localhost:8080/ch11-session-attributes/fixedDeposit/list</i>	Metoda <code>getNewFixedDepositDetails()</code> — zwrot nowego egzemplarza <code>FixedDepositDetails</code> . Metoda <code>listFixedDeposits()</code> — pobranie listy założonych lokat.
Kliknięcie przycisku <i>Założ nową lokatę</i>	Metoda <code>showOpenFixedDepositForm()</code> — wyświetlenie formularza umożliwiającego założenie nowej lokaty.
Wprowadzenie informacji szczegółowych o lokacie i kliknięcie przycisku <i>Zapisz</i>	Metoda <code>openFixedDeposit()</code> — zapisanie informacji o lokacie; przekierowanie w celu wyświetlenia listy lokat. Metoda <code>getNewFixedDepositDetails()</code> — zwrot nowego egzemplarza <code>FixedDepositDetails</code> . Metoda <code>listFixedDeposits()</code> — pobranie listy założonych lokat.

`FixedDepositController`. Ponieważ metoda `openFixedDeposit()` usuwa atrybuty modelu przechowywane w obiekcie `HttpSession`, żądanie do metody `listFixedDeposits()` skutkuje ponownym wywołaniem metody `getNewFixedDepositDetails()`.

Skoro dowiedziałeś się, jak można używać adnotacji `@ModelAttribute` i `@SessionAttributes`, teraz przejdźmy do tematu *dołączania danych* w aplikacjach Spring Web MVC.

11.4. Dołączanie danych

Kiedy w aplikacji Spring Web MVC zostanie wysłany formularz sieciowy, parametry żądania *automatycznie* spowodują ustawienie *atrybutu modelu* działającego w charakterze obiektu obsługującego formularz. Proces ustawiania parametrów żądania w obiekcie obsługującym formularz nosi nazwę *dołączania danych*. W tym podrozdziale przeanalizujemy oferowany przez Spring egzemplarz `WebDataBinder`, który łączy parametry żądania do obiektu obsługującego formularz sieciowy.

IMPORT **Rozdział11/ch11-data-binding.** (Ten projekt pokazuje zmodyfikowaną wersję projektu *ch11-session-attributes* i pokazuje sposób rejestracji w kontenerze Springa implementacji `PropertyEditor`. Po wdrożeniu aplikacji w serwerze Tomcat przejdź pod adres URL *http://localhost:8080/ch11-data-binding*, a zobaczysz wyświetloną listę lokat w systemie).

W listingu 11.13 została przedstawiona klasa `FixedDepositDetails`, która znajduje się w projekcie *ch11-session-attributes*.

Listing 11.13. Klasa `FixedDepositDetails`

Projekt: *ch11-data-binding*

Położenie pliku: *src/main/java/sample/spring/chapter11/web*

```
package sample.spring.chapter11.domain;

import java.util.Date;

public class FixedDepositDetails {
    .....
    private long depositAmount;
```

```
private Date maturityDate;
.....
public void setDepositAmount(long depositAmount) {
    this.depositAmount = depositAmount;
}
public void setMaturityDate(Date maturityDate) {
    this.maturityDate = maturityDate;
}
.....
}
```

W powyższym listingu widać, że właściwości `depositAmount` i `maturityDate` są typu odpowiednio `long` i `java.util.Date`. Wartości tych właściwości są ustawiane po wysłaniu formularza *Załącz nową lokatę* w projekcie *ch11-data-binding*. Rysunek 11.2 przedstawia znajdujący się w projekcie *ch11-data-binding* formularz, który umożliwia założenie nowej lokaty.

Rysunek 11.2.

Formularz

umożliwiający założenie
nowej lokaty

Załącz nową lokatę

Wartość (w PLN):	1200
Czas trwania:	01-27-2013
E-mail:	nazwa@nazwa-domeny.pl
Zapisz Wróć	

Na rysunku 11.2 pola *Wartość (w PLN)* i *Czas trwania* odpowiadają właściwościom odpowiednio `depositAmount` i `maturityDate` klasy `FixedDepositDetails` (patrz: listing 11.13). Trzeba zwrócić uwagę na jeden bardzo ważny aspekt pokazanego formularza. Pole *Czas trwania* akceptuje datę w formacie MM-dd-yyyy, na przykład 01-27-2013. Ponieważ właściwość `depositAmount` jest typu `long`, natomiast `maturityDate` typu `java.util.Date`, wbudowany w Spring mechanizm dołączania danych staje się odpowiedzialny za przeprowadzenie konwersji typu ze `String` na zdefiniowany przez egzemplarz `FixedDepositDetails`.

W listingu 11.14 pokazujemy, że metoda `openFixedDeposit()` klasy `FixedDepositController` jest wywoływana, gdy użytkownik wypełni formularz zakładania nowej lokaty, a następnie kliknie przycisk *Zapisz* (patrz: rysunek 11.2).

Listing 11.14. Klasa `FixedDepositController` pokazująca przykład automatycznego dołączania danych

Projekt: *ch11-data-binding*

Położenie pliku: *src/main/java/sample/spring/chapter11/web*

```
package sample.spring.chapter11.web;

@Controller
.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
    public String openFixedDeposit(
```

```

    @ModelAttribute(value = "newFixedDepositDetails") FixedDepositDetails
    fixedDepositDetails, BindingResult bindingResult, SessionStatus sessionStatus) {
        ....
    }
    .....
}

```

W powyższym listingu oznaczony adnotacją `@ModelAttribute` argument `FixedDepositDetails` przedstawia obiekt obsługujący formularz. Parametry żądania są ustawiane po wysłaniu formularza sieciowego. Oferowany przez Spring egzemplarz `WebDataBinder` dołącza parametry żądania do egzemplarza `FixedDepositDetails`.

Teraz dowiesz się, jak egzemplarz `WebDataBinder` przeprowadza operację dołączania danych.

WebDataBinder, czyli dołączanie danych dla parametrów żądania sieciowego

Egzemplarz `WebDataBinder` używa nazwy parametru żądania, aby wyszukać w obiekcie obsługującym formularz odpowiadającą mu metodę typu setter w stylu `JavaBean`. Jeżeli metoda ta zostanie znaleziona, wówczas `WebDataBinder` wywołuje ją i przekazuje wartość parametru żądania jako argument metody typu setter. Gdy metoda typu setter została zdefiniowana do akceptacji argumentu typu innego niż `String`, wtedy `WebDataBinder` używa odpowiedniego edytora `PropertyEditor` do przeprowadzenia konwersji typu.

W listingu 11.15 pokazujemy, że w aplikacji klasa `MyObject` działa w charakterze obiektu obsługującego formularz sieciowy.

Listing 11.15. Klasa `MyObject` przedstawiająca obiekt obsługujący formularz sieciowy

```

public class MyObject {
    private String x;
    private N y;
    ....
    public void setX(String x) {
        this.x = x;
    }
    public void setY(N y) {
        this.y = y;
    }
}

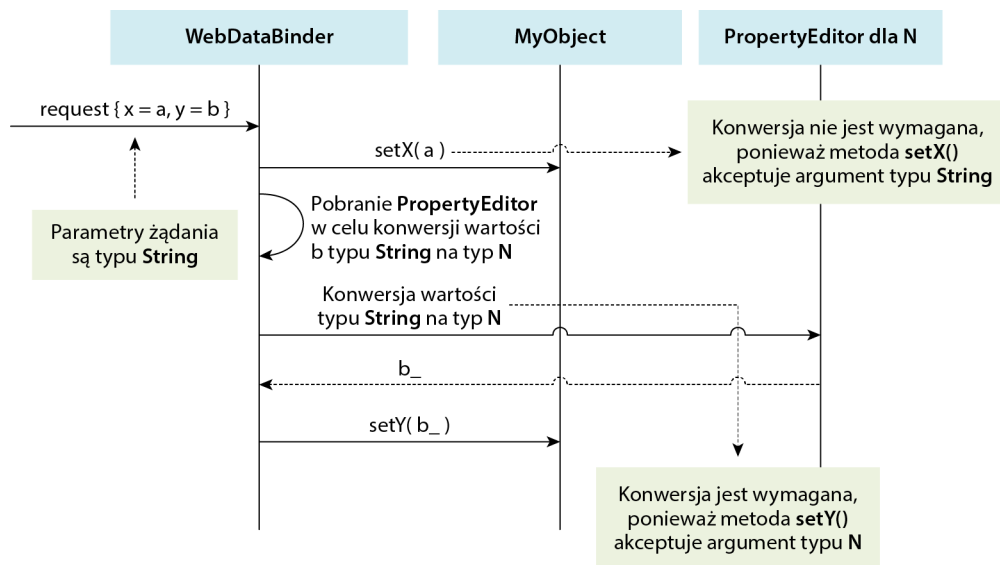
```

Z powyższego listingu wynika, że klasa `MyObject` definiuje właściwości o nazwach `x` i `y` oraz typach odpowiednio `String` i `N`.

Na rysunku 11.3 pokazujemy, jak egzemplarz `WebDataBinder` dołącza parametry żądania `x` i `y` do właściwości `x` i `y` egzemplarza `MyObject`.

Z rysunku 11.3 wynika, że egzemplarz `WebDataBinder` używa edytora `PropertyEditor` do przeprowadzenia konwersji wartości `b` typu `String` na typ `N`. Konwersja jest przeprowadzana przed wywołaniem metody `setY()` egzemplarza `MyObject`.

Spring dostarcza kilka wbudowanych implementacji `PropertyEditor`, które są używane przez `WebDataBinder` do konwersji wartości parametru żądania typu `String` na typ zdefiniowany przez



Rysunek 11.3. Egzemplarz *WebDataBinder* dołącza dane przez użycie zarejestrowanego edytora *PropertyEditor* do przeprowadzenia konwersji typu

obiekt obsługujący formularz sieciowy. Na przykład *CustomNumberEditor*, *FileEditor* i *CustomDateEditor* to tylko niektóre z implementacji *PropertyEditors* dostarczanych przez Spring. Pełną listę wbudowanych implementacji *PropertyEditors* znajdziesz w pakiecie `org.springframework.beans.propertyeditors`.

Implementacja *CustomNumberEditor* jest stosowana do konwersji wartości typu `String` na typ `java.lang.Number`; podobna sytuacja dotyczy typów `Integer`, `Long`, `Double` itd. Implementacja *CustomDateEditor* jest używana do przeprowadzania konwersji wartości typu `String` na typ `java.util.Date`. Istnieje możliwość przekazania egzemplarza `java.text.DateFormat` do implementacji *CustomDateEditor* w celu określenia formatu daty, jaki ma być stosowany podczas przetwarzania i wyświetlania dat. Obie wymienione implementacje *PropertyEditors* są wymagane w projekcie *ch11-data-binding*, ponieważ musimy mieć możliwość konwersji wartości parametrów żądania na wartości właściwości `depositAmount` (typu `long`) i `maturityDate` (typu `java.util.Date`). Implementacja *CustomNumberEditor* jest rejestrowana wraz z egzemplarzem *WebDataBinder*, natomiast trzeba wyraźnie przeprowadzić rejestrację implementacji *CustomDateEditor*.

Teraz przechodzimy do tematu konfiguracji egzemplarza *WebDataBinder* oraz rejestracji implementacji *PropertyEditor*.

Konfiguracja egzemplarza *WebDataBinder*

Konfiguracja egzemplarza *WebDataBinder* odbywa się przez:

- zdefiniowanie w klasie kontrolera metody oznaczonej adnotacją `@InitBinder`;
- przeprowadzenie w pliku XML kontekstu aplikacji sieciowej konfiguracji egzemplarza *WebBinding* `Initializer`;

- zdefiniowanie metody oznaczonej adnotacją `@InitBinder` w klasie oznaczonej adnotacją `@ControllerAdvice`.

Wymienione powyżej podejścia przeanalizujemy teraz, konfigurując egzemplarz `WebDataBinder` i rejestrując w nim implementację `PropertyEditor`.

Zdefiniowanie w klasie kontrolera metody oznaczonej adnotacją `@InitBinder`

Metoda oznaczona adnotacją `@InitBinder` w klasie kontrolera oznacza, że dana metoda inicjuje egzemplarz `WebDataBinder`, który będzie stosowany przez kontroler podczas dołączania danych. Atrybut `value` adnotacji `@InitBinder` określa nazwę (lub nazwy) atrybutu modelu, do którego zastosowanie ma zainicjowany egzemplarz `WebDataBinder`.

W listingu 11.16 została przedstawiona metoda `initBinder_New()` klasy `FixedDepositController` oznaczona adnotacją `@InitBinder`.

Listing 11.16. Klasa `FixedDepositController` przedstawiająca użycie adnotacji `@InitBinder`

Projekt: `ch11-data-binding`

Położenie pliku: `src/main/java/sample/spring/chapter11/web`

```
package sample.spring.chapter11.web;

import java.text.SimpleDateFormat;
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;

@Controller
.....
public class FixedDepositController {
    .....
    @ModelAttribute(value = "newFixedDepositDetails")
    public FixedDepositDetails getNewFixedDepositDetails() { ..... }
    @InitBinder(value = "newFixedDepositDetails")
    public void initBinder_New(WebDataBinder webDataBinder) {
        webDataBinder.registerCustomEditor(Date.class,
            new CustomDateEditor(new SimpleDateFormat("MM-dd-yyyy"), false));
    }
    .....
}
```

W powyższym listingu atrybut `value` adnotacji `@InitBinder` ma przypisaną wartość `newFixedDepositDetails`. Oznacza to, że egzemplarz `WebDataBinder` zainicjowany przez metodę `initBinder_New()` będzie miał zastosowanie *tylko* dla atrybutu modelu `newFixedDepositDetails`. Metoda oznaczona adnotacją `@InitBinder` może akceptować ten sam zestaw argumentów (np. `HttpServletRequest`, `SessionStatus` itd.), który można przekazać metodzie oznaczonej adnotacją `@RequestMapping`. Jednak metoda oznaczona adnotacją `@InitBinder` nie może być zdefiniowana w sposób pozwalający na akceptację atrybutów modelu i obiektów `BindingResult` (lub `Errors`) jako argumentów. Najczęściej metodzie oznaczonej adnotacją `@InitBinder` jest przekazywany egzemplarz `WebDataBinder` wraz z oferowanym przez Spring egzemplarzem `WebRequest` lub `java.util.Locale`. Zwróć uwagę na fakt, że typem wartości zwrotnej metody oznaczonej adnotacją `@InitBinder` musi być `void`.

Metoda `registerCustomEditor()` klasy `WebDataBinder` jest używana do rejestracji implementacji `PropertyEditor` w egzemplarzu `WebDataBinder`. W listingu 11.16 metoda `initBinder_New()` rejestruje `CustomDateEditor` (implementacja `PropertyEditor`) w egzemplarzu `WebDataBinder`.

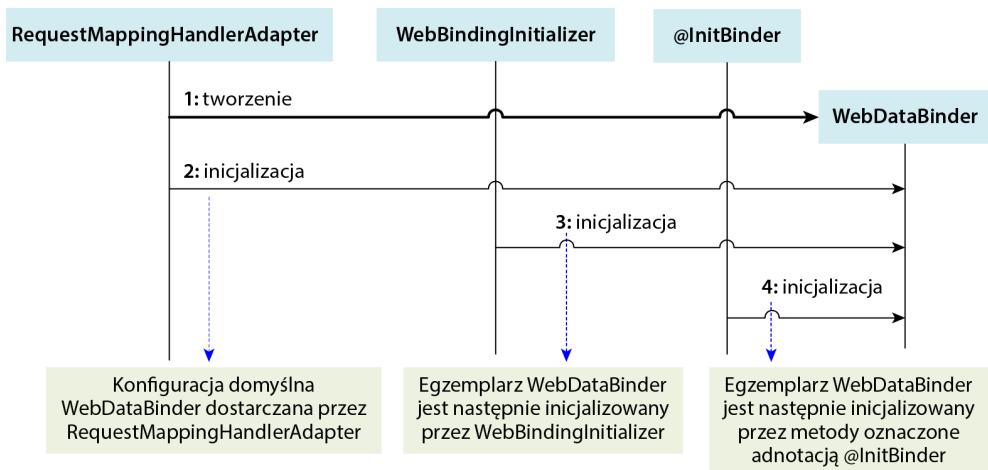
Istnieje możliwość zdefiniowania metody oznaczonej adnotacją `@InitBinder` dla każdego atrybutu modelu kontrolera lub też zdefiniowanie pojedynczej metody, która będzie miała zastosowanie dla wszystkich atrybutów modelu kontrolera. Jeżeli nie określisz atrybutu `value` adnotacji `@InitBinder`, egzemplarz `WebDataBinder` inicjalizowany przez metodę będzie stosowany we wszystkich atrybutach modelu kontrolera.

Konfiguracja implementacji interfejsu `WebBindingInitializer`

Egzemplarz `WebDataBinder` jest najpierw inicjalizowany przez `RequestMappingHandlerAdapter`, a następnie dalsza inicjalizacja jest przeprowadzana przez `WebBindingInitializer` oraz metody oznaczone adnotacjami `@InitBinder`.

Element `<annotation-driven>` schematu mvc oferowanego przez Spring tworzy egzemplarz `RequestMappingHandlerAdapter`, który inicjalizuje `WebDataBinder`. Istnieje możliwość przekazania implementacji interfejsu `WebBindingInitializer` do `RequestMappingHandlerAdapter` w celu dalszej inicjalizacji egzemplarza `WebDataBinder`. Ponadto metody oznaczone adnotacją `@InitBinder` w klasie kontrolera można wykorzystać do dalszej inicjalizacji egzemplarza `WebDataBinder`.

Na rysunku 11.4 pokazujemy sekwencję, w jakiej `RequestMappingHandlerAdapter`, `WebBindingInitializer` oraz metody oznaczone adnotacją `@InitBinder` przeprowadzają inicjalizację egzemplarza `WebDataBinder`.



Rysunek 11.4. Sekwencja, w jakiej egzemplarz `WebDataBinder` jest inicjalizowany przez `RequestMappingHandlerAdapter`, `WebBindingInitializer` i metody klasy kontrolera oznaczone adnotacją `@InitBinder`

Inicjalizacja `WebDataBinder` przeprowadzana za pomocą oznaczonej adnotacją `@InitBinder` metody klasy kontrolera jest dozwolona jedynie dla atrybutów modelu danego kontrolera. Na przykład jeśli używasz oznaczonej adnotacją `@InitBinder` metody w kontrolerze X do ustawienia właściwości

edytora `CustomDateEditor` w egzemplarzu `WebDataBinder`, to podczas operacji dołączania danych właściwość edytora `CustomDateEditor` będzie dostępna tylko dla atrybutów modelu kontrolera `X`. W aplikacji `MyBank` właściwość `CustomDateEditor` jest wymagana jedynie przez atrybuty modelu `FixedDepositController`. Dlatego oznaczonych adnotacją `@InitBinder` metod klasy `FixedDepositController` użyto do rejestracji implementacji `CustomDateEditor` w egzemplarzu `WebDataBinder`.

Oferowany przez Spring interfejs `WebBindingInitializer` jest interfejsem wywołań zwrotnych, którego implementacja jest odpowiedzialna za inicjalizację egzemplarza `WebDataBinder` wraz z konfiguracją mającą zastosowanie we *wszystkich* kontrolerach (a tym samym we wszystkich atrybutach modelu) w aplikacji. Zobaczmy, jak można skonfigurować własną wersję interfejsu `WebBindingInitializer` za pomocą elementu `<annotation-driven>` schematu mvc.

Element `<annotation-driven>` schematu mvc tworzy i rejestruje w kontenerze Spring obiekty `RequestMappingHandlerAdapter` i `RequestMappingHandlerMapping`. Pozostałe obiekty konfigurowane przez element `<annotation-driven>` to `LocalValidatorFactoryBean` (omówimy go dokładnie w podrozdziale 11.5) i `FormattingConversionServiceFactoryBean` (omówimy go dokładnie w podrozdziale 13.5). Element `<annotation-driven>` dostarcza wiele atrybutów pomagających w dostosowaniu do własnych potrzeb obiektów `RequestMappingHandlerAdapter` i `RequestMappingHandlerMapping`. Jeżeli zmiany, które chcesz wprowadzić w obiekcie `RequestMappingHandlerAdapter` lub `RequestMappingHandlerMapping`, nie są obsługiwane przez element `<annotation-driven>`, jedynym rozwiązaniem pozostaje usunięcie elementu `<annotation-driven>` oraz wyraźne skonfigurowanie obiektów `RequestMappingHandlerAdapter` i `RequestMappingHandlerMapping` w pliku XML kontekstu aplikacji sieciowej. Ponieważ element `<annotation-driven>` nie oferuje żadnej możliwości dostarczenia własnego egzemplarza `WebBindingInitializer` obiektowi `RequestMappingHandlerAdapter`, konieczne jest wyraźne skonfigurowanie obiektów `RequestMappingHandlerAdapter` i `RequestMappingHandlerMapping` w pliku XML kontekstu aplikacji sieciowej.

W listingu 11.7 pokazujemy sposób użycia oferowanego przez Spring egzemplarza `ConfigurableWebBindingInitializer` (implementacja `WebBindingInitializer`) w celu udostępnienia edytora właściwości `CustomDateEditor` wszystkim kontrolerom znajdującym się w aplikacji `MyBank`.

Listing 11.17. Konfiguracja `WebBindingInitializer`

```
<bean id="handlerAdapter"
    class="org.springframework.web.servlet.mvc.method
        .annotation.RequestMappingHandlerAdapter">
    <property name="webBindingInitializer" ref="myInitializer" />
</bean>

<bean id="handlerMapping"
    class="org.springframework.web.servlet.mvc.method
        .annotation.RequestMappingHandlerMapping" />

<bean id="myInitializer"
    class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer">
    <property name="propertyEditorRegistrars">
        <list>
            <bean class="mypackage.MyPropertyEditorRegistrar" />
        </list>
    </property>
</bean>
```

W powyższym listingu widać, że komponenty `RequestMappingHandlerAdapter` i `RequestMappingHandlerMapping` zostały *wyraźnie* zdefiniowane w pliku XML kontekstu aplikacji sieciowej. Właściwość `webBindingInitializer` egzemplarza `RequestMappingHandlerAdapter` odwołuje się do komponentu `ConfigurableWebBindingInitializer` implementującego interfejs `WebBindingInitializer`. Z kolei właściwość `propertyEditorRegistrars` egzemplarza `ConfigurableWebBindingInitializer` określa klasy rejestrujące w egzemplarzu `WebDataBinder` jedną lub więcej implementacji `PropertyEditors`. Listing 11.18 pokazuje, że klasa `MyPropertyEditorRegistrar` rejestruje w egzemplarzu `WebDataBinder` edytor właściwości `CustomDateEditor`.

Listing 11.18. Klasa `MyPropertyEditorRegistrar`

```
import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;
import org.springframework.beans.propertyeditors.CustomDateEditor;

public class MyPropertyEditorRegistrar implements PropertyEditorRegistrar {

    @Override
    public void registerCustomEditors(PropertyEditorRegistry registry) {
        registry.registerCustomEditor(Date.class, new CustomDateEditor(
            new SimpleDateFormat("MM-dd-yyyy"), false));
    }
}
```

W powyższym listingu widać, że klasa `MyPropertyEditorRegistrar` implementuje oferowany przez Spring interfejs `PropertyEditorRegistrar`, a także dostarcza implementację dla metody `registerCustomEditors()` zdefiniowanej w interfejsie `PropertyEditorRegistrar`. Egzemplarz `PropertyEditorRegistry` przekazywany metodzie `registerCustomEditors()` jest używany do rejestracji edytorów właściwości. Metoda `registerCustomEditor()` klasy `PropertyEditorRegistry` jest wykorzystywana do rejestracji implementacji `PropertyEditor` w egzemplarzu `WebDataBinder`. W omawianym listingu metodę `registerCustomEditor()` klasy `PropertyEditorRegistry` zastosowano do zarejestrowania w egzemplarzu `WebDataBinder` edytora właściwości `CustomDateEditor`.

Jak zauważyłeś, użycie interfejsu `WebBindingInitializer` do inicjalizacji egzemplarza `WebDataBinder` wymaga wykonania dość dużej pracy. Znacznie prostszą alternatywą dla użycia interfejsu `WebBindingInitializer` będzie zdefiniowanie metod oznaczonych adnotacją `@InitBinder` w klasie oznaczonej adnotacją `@ControllerAdvice`.

Zdefiniowanie metody oznaczonej adnotacją `@InitBinder` w klasie oznaczonej adnotacją `@ControllerAdvice`

Podobnie jak adnotacje `@Service`, `@Controller` i `@Repository`, także `@ControllerAdvice` jest specjalną formą adnotacji `@Component`. Adnotacja `@ControllerAdvice` w klasie wskazuje, że dana klasa zapewnia obsługę kontrolerów. W klasie oznaczonej adnotacją `@ControllerAdvice` można zdefiniować metody oznaczone adnotacjami `@InitBinder`, `@ModelAttribute` i `@ExceptionHandler`, a będą one stosowane dla *wszystkich* kontrolerów oznaczonych adnotacjami w aplikacji. Podobnie jak w przypadku adnotacji `@Service`, `@Controller` i `@Repository`, element `<classpath-scanning>` schematu context powoduje automatyczne wykrycie i zarejestrowanie w kontenerze Springa klas oznaczonych adnotacją `@ControllerAdvice`.

Jeżeli zauważysz powielanie oznaczonych adnotacjami `@InitBinder`, `@ModelAttribute` i `@ExceptionHandler` ↪ `Handler` metod w wielu kontrolerach, rozważ zdefiniowanie tego rodzaju metod w klasie oznaczonej adnotacją `@ControllerAdvice`. Na przykład jeśli chcesz inicjalizować egzemplarz `WebDataBinder` wraz z konfiguracją odpowiednią dla wielu kontrolerów w aplikacji, metodę oznaczoną adnotacją `@InitBinder` zdefiniuj w klasie oznaczonej adnotacją `@ControllerAdvice` zamiast definiować ją w wielu klasach kontrolerów.

W tabeli 11.3 przedstawiamy podsumowanie trzech omówionych tutaj podejść w zakresie inicjalizacji egzemplarza `WebDataBinder`.

Tabela 11.3. *Możliwe do zastosowania podejścia podczas inicjalizacji egzemplarza `WebDataBinder`*

Metoda oznaczona adnotacją <code>@InitBinder</code> w klasie kontrolera	Egzemplarz <code>WebBindingInitializer</code>	Metoda oznaczona adnotacją <code>@InitBinder</code> w klasie oznaczonej adnotacją <code>@ControllerAdvice</code>
Wymaga zdefiniowania w kontrolerze metody oznaczonej adnotacją <code>@InitBinder</code> .	Wymaga jawnego skonfigurowania egzemplarza. <code>RequestMappingHandlerAdapter</code> w pliku XML kontekstu aplikacji sieciowej.	Wymaga określenia metody oznaczonej adnotacją <code>@InitBinder</code> w klasie oznaczonej adnotacją <code>@ControllerAdvice</code> .
Inicjalizacja egzemplarza <code>WebDataBinder</code> jest stosowana tylko dla kontrolera zawierającego metodę oznaczoną adnotacją <code>@InitBinder</code> .	Inicjalizacja egzemplarza <code>WebDataBinder</code> jest stosowana we wszystkich kontrolerach oznaczonych adnotacjami w aplikacji.	Inicjalizacja egzemplarza <code>WebDataBinder</code> jest stosowana we wszystkich kontrolerach oznaczonych adnotacjami w aplikacji.

Przechodzimy teraz do tematu włączenia lub wyłączenia udziału poszczególnych właściwości atrybutu modelu w procesie dołączania danych.

Włączenie lub wyłączenie poszczególnych właściwości w procesie dołączania danych

Egzemplarz `WebDataBinder` pozwala na wskazanie właściwości atrybutu modelu, które mają być włączone lub wyłączone w procesie dołączania danych. Zaleca się włączenie lub wyłączenie właściwości, ponieważ w przeciwnym razie może powstać luka w *zabezpieczeniach* aplikacji. Przechodzimy teraz do przykładu, w którym będziemy włączać i wyłączać właściwości w procesie dołączania danych.

Kiedy w aplikacji `MyBank` użytkownik wybierze lokatę do edycji, informacje szczegółowe o tej lokacie są pobierane z magazynu danych i tymczasowo buforowane w `HttpSession`. Użytkownik wprowadza żądane zmiany w szczegółach lokaty i je zapisuje. Listing 11.19 przedstawia metody oznaczone adnotacją `@RequestMapping` odpowiedzialne za wczytywanie informacji o lokatach oraz zapis uaktualnionych danych o lokatach.

W aplikacji `MyBank` lokata jest unikalnie identyfikowana za pomocą właściwości `id` obiektu `FixedDepositDetails` (patrz: klasa `FixedDepositDetails` w projekcie *ch11-data-binding*). Kiedy

Listing 11.19. Kontroler *FixedDepositController*Projekt: *ch11-data-binding*Położenie pliku: *src/main/java/sample/spring/chapter11/web*

```

package sample.spring.chapter11.web;
.....
@SessionAttributes(value = { "newFixedDepositDetails", "editableFixedDepositDetails" })
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=view", method = RequestMethod.GET)
    public ModelAndView viewFixedDepositDetails(
        @RequestParam(value = "fixedDepositId") int fixedDepositId) {
        FixedDepositDetails fixedDepositDetails = fixedDepositService
            .getFixedDeposit(fixedDepositId);
        Map<String, Object> modelMap = new HashMap<String, Object>();
        modelMap.put("editableFixedDepositDetails", fixedDepositDetails);
        .....
        return new ModelAndView("editFixedDepositForm", modelMap);
    }
    .....
    @RequestMapping(params = "fdAction=edit", method = RequestMethod.POST)
    public String editFixedDeposit(
        @ModelAttribute("editableFixedDepositDetails") FixedDepositDetails
            fixedDepositDetails,) {
        .....
    }
}

```

użytkownik wybierze lokatę do edycji, wartość właściwości *id* zostanie przekazana metodzie *viewFixedDepositDetails()* za pomocą parametru żądania *fixedDepositId*. Następnie metoda *viewFixedDepositDetails()* używa wartości parametru żądania *fixedDepositId* w celu wczytania informacji szczegółowych o lokacie i wyświetla je w formularzu pokazanym na rysunku 11.5.

Rysunek 11.5.

Formularz

umożliwiający

edycję szczegółów

istniejącej lokaty

Edycja lokaty

Id:	4
Wartość (w PLN):	50000
Czas trwania:	05-22-2013
E-mail:	a4email@nazwa-domeny
Zapisz Wróć	

Ponieważ wartość właściwości *id* (odpowiada atrybutowi *id* obiektu *FixedDepositDetails*) unikalnie identyfikuje lokatę w systemie, formularz pokazany na rysunku 11.5 nie ma żadnego mechanizmu pozwalającego na zmianę tej wartości. Po kliknięciu przycisku *Zapisz* przez użytkownika następuje wywołanie metody *editFixedDeposit()* klasy *FixedDepositController*. Metoda ta zapisuje zmiany wprowadzone w informacjach szczegółowych o lokacie.

Po wywołaniu metody `editFixedDeposit()` klasy `FixedDepositController` egzemplarz `WebDataBinder` łączy wartości parametrów żądania do właściwości atrybutu modelu `editableFixedDepositDetails`, czyli obiektu `FixedDepositDetails` wczytanego przez metodę `viewFixedDepositDetails()` i tymczasowo przechowywanego w `HttpSession` (patrz: adnotacja `@SessionAttributes` w listingu 11.19). Jeżeli złośliwy użytkownik wyśle parametr żądania o nazwie `id` i wartości `10`, wówczas egzemplarz `WebDataBinder` po prostu ustawi atrybut `id` obiektu `FixedDepositDetails` i przypisze mu wartość `10` podczas operacji dołączania danych. To *nie* jest pożądane, ponieważ zmiana atrybutu `id` obiektu `FixedDepositDetails` spowoduje uszkodzenie danych aplikacji.

Egzemplarz `WebDataBinder` oferuje metody `setAllowedFields()` i `setDisallowedFields()` pozwalające na wskazanie właściwości atrybutu modelu, które mają lub nie mają być uwzględniane w trakcie procesu dołączania danych. Listing 11.20 przedstawia metodę `initBinder_Edit()` klasy `FixedDepositController` określającą, że właściwość `id` atrybutu modelu `editableFixedDepositDetails` *nie* może być uwzględniana w trakcie procesu dołączania danych.

Listing 11.20. Kontroler `FixedDepositController` przedstawiający metodę `setDisallowedFields()` klasy `WebDataBinder`

Projekt: `ch11-data-binding`

Położenie pliku: `src/main/java/sample/spring/chapter11/web`

```
package sample.spring.chapter11.web;
.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=edit", method = RequestMethod.POST)
    public String editFixedDeposit(
        @ModelAttribute("editableFixedDepositDetails") FixedDepositDetails
        fixedDepositDetails, ..... ) {
        .....
    }
    .....
    @InitBinder(value = "editableFixedDepositDetails")
    public void initBinder_Edit(WebDataBinder webDataBinder) {
        webDataBinder.registerCustomEditor(Date.class, new CustomDateEditor(
            new SimpleDateFormat("MM-dd-yyyy"), false));
        webDataBinder.setDisallowedFields("id");
    }
}
```

W powyższym listingu metoda `initBinder_Edit()` inicjalizuje egzemplarz `WebDataBinder` dla atrybutu modelu `editableFixedDepositDetails`. Metoda `setDisallowedFields()` określa, że właściwość `id` atrybutu modelu `editableFixedDepositDetails` jest wyłączona z procesu dołączania danych, dlatego właściwość `id` *nie* jest ustawiana, nawet jeśli w żądaniu znajduje się parametr o nazwie `id`.

Przejdźmy teraz do analizy udostępnianego przez Spring obiektu `BindingResult`, który przechowuje błędy, jakie wystąpiły w trakcie operacji dołączania danych i weryfikacji.

Analiza błędów dołączania danych i weryfikacji przechowywanych przez obiekt BindingResult

Obiekt `BindingResult` dostarcza metodę kontrolera wraz z wynikami operacji dołączania parametrów żądania do właściwości atrybutów modelu. Na przykład jeśli podczas dołączania danych wystąpi błąd konwersji typu, zostanie on zgłoszony przez obiekt `BindingResult`.

W listingu 11.21 została przedstawiona metoda `openFixedDeposit()` klasy `FixedDepositController`, która tworzy lokatę tylko wtedy, gdy obiekt `BindingResult` nie zgłosił wystąpienia żadnych błędów.

Listing 11.21. Kontroler `FixedDepositController` przedstawia sprawdzanie pod kątem błędów operacji dołączania danych i weryfikacji, używając obiektu `BindingResult`

Projekt: `ch11-data-binding`

Położenie pliku: `src/main/java/sample/spring/chapter11/web`

```
package sample.spring.chapter11.web;

import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
    public String openFixedDeposit(
        @ModelAttribute(value = "newFixedDepositDetails") FixedDepositDetails
        fixedDepositDetails, BindingResult bindingResult, SessionStatus
        sessionStatus) {
        .....
        if (bindingResult.hasErrors()) {
            return "createFixedDepositForm";
        } else {
            fixedDepositService.saveFixedDeposit(fixedDepositDetails);
            sessionStatus.setComplete();
            return "redirect:/fixedDeposit/list";
        }
    }
    .....
}
```

W powyższym listingu wartością zwrótną metody `hasErrors()` klasy `BindingResult` jest `true`, jeśli obiekt `BindingResult` przechowuje choć jeden błąd wykryty podczas operacji dołączania danych lub weryfikacji. W podrozdziale 11.5 zobaczysz, jak błędy wykryte w trakcie weryfikacji są przechowywane w obiekcie `BindingResult`. Jeżeli obiekt `BindingResult` przechowuje jakiegokolwiek błędy, wówczas metoda `openFixedDeposit()` wygeneruje formularz zakładania nowej lokaty wraz z odpowiednimi komunikatami błędów. Z kolei brak błędów powoduje zapis w magazynie danych informacji szczegółowych o lokacie.

Trzeba zwrócić uwagę na fakt, że argument `BindingResult` musi znajdować się tuż po argumencie atrybutu modelu, do którego obiekt `BindingResult` ma uzyskać dostęp w metodzie kontrolera. Na przykład w listingu 11.21 argument `BindingResult` znajduje się tuż po atrybucie modelu `newFixedDepositDetails`. W listingu 11.22 pokazujemy nieprawidłową kolejność atrybutu modelu i obiektu `BindingResult` w metodzie `openFixedDeposit()`.

Listing 11.22. *Niewłaściwa kolejność atrybutu modelu i obiektu BindingResult*

```

.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
    public String openFixedDeposit(
        @ModelAttribute(value = "newFixedDepositDetails") FixedDepositDetails
        fixedDepositDetails, SessionStatus sessionStatus, BindingResult
        bindingResult) {
        .....
    }
    .....
}

```

W powyższym listingu kolejność atrybutu modelu `newFixedDepositDetails` i obiektu `BindingResult` jest nieprawidłowa, ponieważ między nimi został zdefiniowany argument `SessionStatus`.

Jeżeli metoda kontrolera akceptuje wiele atrybutów modelu, wówczas obiekty `BindingResult` odpowiadające poszczególnym atrybutom modelu są podawane tuż po argumentach atrybutu modelu, jak przedstawia listing 11.23.

Listing 11.23. *Wiele atrybutów modelu i odpowiadające im obiekty BindingResult*

```

@RequestMapping
public String doSomething(
    @ModelAttribute(value = "a") AObject aObj, BindingResult bindingResultA,
    @ModelAttribute(value = "b") BObject bObj, BindingResult bindingResultB,) {
    .....
}

```

W powyższym listingu widać, że tuż po atrybutach modelu `a` i `b` znajdują się odpowiadające im obiekty `BindingResult`.

Skoro poznaliśmy proces dołączania danych, możemy teraz przejść do sposobów przeprowadzania weryfikacji w aplikacjach Spring Web MVC.

11.5. Weryfikacja w Springu

W poprzednim podrozdziale zobaczyłeś, że egzemplarz `WebDataBinder` dołącza parametry żądania do atrybutów modelu. Kolejnym krokiem w procesie przetwarzania żądania jest więc weryfikacja atrybutów modelu. W aplikacjach Spring Web MVC weryfikację atrybutów modelu można przeprowadzić za pomocą API Validation oferowanego przez Spring (patrz: podrozdział 6.9) lub przez zdefiniowanie ograniczeń JSR 303 (patrz: podrozdział 6.10) dla właściwości atrybutów modelu.



W tym rozdziale API Validation udostępnianego przez Spring i specyfikacji JSR 303 (API Bean Validation) używamy do weryfikacji obiektów obsługujących formularz (będących atrybutami modeli) na warstwie sieciowej aplikacji. Warto zwrócić uwagę, że zarówno JSR 303, jak i API Validation można wykorzystać do weryfikacji obiektów na dowolnej warstwie aplikacji.

Najpierw zobaczysz, jak przeprowadzić weryfikację atrybutów modelu za pomocą interfejsu `Validator` oferowanego przez API Validation.

Weryfikacja atrybutów modelu za pomocą oferowanego przez Spring interfejsu `Validator`

W listingu 11.24 została przedstawiona klasa `FixedDepositDetailsValidator` aplikacji `MyBank` odpowiedzialna za weryfikację obiektu `FixedDepositDetails`.

Listing 11.24. Klasa `FixedDepositDetailsValidator` przedstawiająca użycie oferowanego przez Spring interfejsu `Validator`

Projekt: `ch11-data-binding`

Położenie pliku: `src/main/java/sample/spring/chapter11/web`

```
package sample.spring.chapter11.web;

import org.springframework.validation.*;
import sample.spring.chapter11.domain.FixedDepositDetails;

public class FixedDepositDetailsValidator implements Validator {

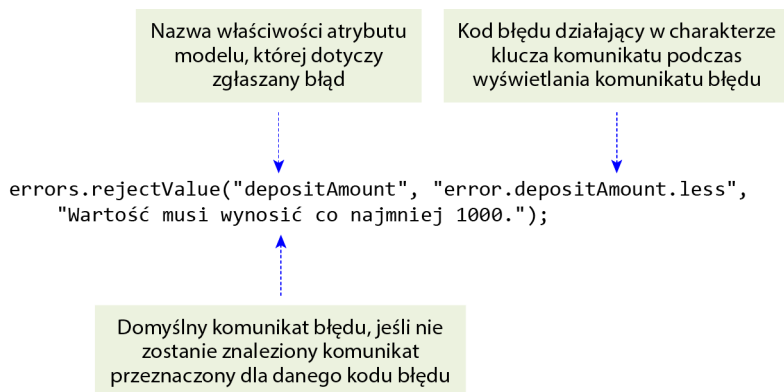
    public boolean supports(Class<?> clazz) {
        return FixedDepositDetails.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        FixedDepositDetails fixedDepositDetails = (FixedDepositDetails) target;
        long depositAmount = fixedDepositDetails.getDepositAmount();
        ....
        if (depositAmount < 1000) {
            errors.rejectValue("depositAmount", "error.depositAmount.less",
                "Wartość musi wynosić co najmniej 1000.");
        }
        if (email == null || "".equalsIgnoreCase(email)) {
            ValidationUtils.rejectIfEmptyOrWhitespace(errors, "email", "error.email.blank",
                "Podanie adresu e-mail jest konieczne.");
        }
        ....
    }
}
```

Istniejący w Springu interfejs `Validator` definiuje metody `supports()` i `validate()`. Metoda `supports()` sprawdza, czy dostarczony egzemplarz obiektu (przedstawiany przez atrybut `clazz`) może być zweryfikowany. Jeżeli wartością zwrótną metody `supports()` będzie `true`, wówczas metoda `validate()` zostanie użyta do przeprowadzenia weryfikacji obiektu. W powyższym listingu metoda `supports()` klasy `FixedDepositDetailsValidator` sprawdza, czy dostarczony egzemplarz obiektu jest typu `FixedDepositDetails`. Jeżeli wartością zwrótną metody `supports()` będzie `true`, metoda `validate()` klasy `FixedDepositDetailsValidator` zweryfikuje obiekt. Metoda `validate()` akceptuje egzemplarz obiektu przeznaczonego do weryfikacji i egzemplarz `Errors`. W egzemplarzu `Errors` są przechowywane błędy wykryte podczas weryfikacji. Egzemplarz `Errors` dostarcza wiele metod `reject()` i `rejectValue()` przeznaczonych do rejestrowania błędów w egzemplarzu `Errors`. Metody

`rejectValue()` są używane do zgłaszania błędów na poziomie wartości, natomiast metody `reject()` — do zgłaszania błędów dotyczących weryfikowanego obiektu. Udostępniana przez Spring klasa `ValidationUtils` to klasa narzędziowa dostarczająca wygodne metody przeznaczone do wywołania interfejsu `Validator` i odrzucania pustych właściwości.

Na rysunku 11.6 zostały pokazane parametry przekazywane metodzie `rejectValue()` przedstawionej w listingu 11.24 odpowiedzialnej za zgłaszanie błędów dotyczących właściwości `depositAmount` egzemplarza `FixedDepositDetails`.



Rysunek 11.6. Opis parametrów przekazywanych metodzie `rejectValue` egzemplarza `Errors` w celu zgłaszania błędów weryfikacji dotyczących właściwości `depositAmount` egzemplarza `FixedDepositDetails`

Na rysunku 11.6 widać, że metodzie `rejectValue()` są przekazywane dane w postaci nazwy właściwości, kodu błędu (to praktycznie klucz komunikatu) i domyślnego komunikatu błędu. Z rozdziału 13. dowiesz się, jak klucze komunikatów są używane przez strony JSP do wyświetlania komunikatów pobieranych z paczek zasobów.

Atrybuty modelu można zweryfikować na następujące sposoby:

- wyrażne wywołanie metody `validate()` implementacji interfejsu `Validator`;
- ustawienie w egzemplarzu `WebDataBinder` implementacji interfejsu `Validator` oraz oznaczenie adnotacją `JSR 303 @Valid` argumentu atrybutu modelu w metodzie oznaczonej adnotacją `@RequestMapping`.

Przejdźmy do dokładnego omówienia wymienionych powyżej podejść.

Weryfikacja atrybutów modelu przez wyrażne wywołanie metody `validate()`

W listingu 11.25 została przedstawiona metoda `openFixedDeposit()` klasy `FixedDepositController` używająca implementacji `FixedDepositDetailsValidator` (patrz: listing 11.24) do przeprowadzenia weryfikacji atrybutu modelu `FixedDepositDetails`.

W powyższym listingu widać, że metoda `openFixedDeposit()` tworzy egzemplarz `FixedDepositDetailsValidator` i wywołuje jego metodę `validate()`. Ponieważ `BindingResult` jest subinterejsem

Listing 11.25. Kontroler *FixedDepositController* przedstawiający weryfikację przez jawne wywołanie metody *validate()* klasy *FixedDepositDetailsValidator*

Projekt: ch11-data-binding

Położenie pliku: src/main/java/sample/spring/chapter11/web

```
package sample.spring.chapter11.web;
.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
    public String openFixedDeposit(
        @ModelAttribute(value = "newFixedDepositDetails") FixedDepositDetails
        fixedDepositDetails, BindingResult bindingResult, SessionStatus
        sessionStatus) {

        new FixedDepositDetailsValidator().validate(fixedDepositDetails,bindingResult);
        if (bindingResult.hasErrors()) {
            logger.info("Metoda openFixedDeposit(): błędy w trakcie weryfikacji
                - ponowne wyświetlenie formularza zakładania nowej lokaty.");
            return "createFixedDepositForm";
        }
        .....
    }
}
```

Errors, istnieje możliwość przekazania obiektu *BindingResult* tam, gdzie oczekiwany jest obiekt *Errors*. Metoda *openFixedDeposit()* przekazuje metodzie *validate()* atrybut modelu *fixedDepositDetails* i obiekt *BindingResult*. Ponieważ *BindingResult* zawiera już błędy wykryte podczas dołączania danych, przekazanie obiektu *BindingResult* metodzie *validate()* powoduje dodanie do obiektu *BindingResult* błędów wykrytych w trakcie weryfikacji.

Wywołanie weryfikacji atrybutów modelu za pomocą adnotacji JSR 303 *@Valid*

Istnieje możliwość nakazania Springowi automatycznego weryfikowania argumentu atrybutu modelu przekazanego metodzie oznaczonej adnotacją *@RequestMapping*. To wymaga dodania adnotacji JSR 303 *@Valid* do argumentu atrybutu modelu oraz wskazania dla niego weryfikatora w egzemplarzu *WebDataBinder*.

W listingu 11.26 pokazujemy, jak metoda *openFixedDeposit()* klasy *FixedDepositController* może użyć adnotacji *@Valid* do przeprowadzenia weryfikacji atrybutu modelu *FixedDepositDetails*.

Listing 11.26. Kontroler *FixedDepositController* przedstawiający wywołanie weryfikacji za pomocą adnotacji *@Valid*

```
import javax.validation.Valid;
.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
    public String openFixedDeposit(
        @Valid @ModelAttribute(value = "newFixedDepositDetails") FixedDepositDetails
        fixedDepositDetails, BindingResult bindingResult, SessionStatus sessionStatus) {

        if (bindingResult.hasErrors()) {
```

```

        logger.info("Metoda openFixedDeposit(): błędy w trakcie weryfikacji
        - ponowne wyświetlenie formularza zakładania nowej lokaty.");
        return "createFixedDepositForm";
    }
    .....
}
.....
@InitBinder(value = "newFixedDepositDetails")
public void initBinder_New(WebDataBinder webDataBinder) {
    webDataBinder.registerCustomEditor(Date.class, new CustomDateEditor(
        new SimpleDateFormat("MM-dd-yyyy"), false));
    webDataBinder.setValidator(new FixedDepositDetailsValidator());
}
.....
}

```

W powyższym listingu metoda `initBinder_New()` wywołuje metodę `setValidator()` egzemplarza `WebDataBinder` w celu wskazania `FixedDepositDetailsValidator` jako weryfikatora dla atrybutu modelu `newFixedDepositDetails`. W metodzie `openFixedDeposit()` atrybut modelu `newFixedDepositDetails` został oznaczony adnotacją JSR 303 `@Valid`. Po wywołaniu metody `openFixedDeposit()` dla atrybutu modelu `newFixedDepositDetails` zostaną przeprowadzone operacje dołączania danych i weryfikacji. Wynik tych operacji jest dostępny za pomocą argumentu `BindingResult`.

Trzeba koniecznie zwrócić uwagę na fakt, że adnotacja `@InitBinder` określa nazwę atrybutu modelu, a weryfikator wskazany w egzemplarzu `WebDataBinder` jest stosowany tylko dla tego konkretnego atrybutu modelu. Na przykład w listingu 11.26 weryfikator `FixedDepositDetailsValidator` jest stosowany tylko dla atrybutu modelu `newFixedDepositDetails`. Jeżeli weryfikator ma zastosowanie dla wielu kontrolerów w aplikacji, rozważ zdefiniowanie oznaczonej adnotacją `@InitBinder` metody wewnątrz klasy oznaczonej adnotacją `@ControllerAdvice` (lub użyj interfejsu `WebBindingInitializer`), aby wskazać weryfikator w egzemplarzu `WebDataBinder`.

Teraz zobaczysz, jak dla właściwości komponentu `JavaBean` można zdefiniować ograniczenia, używając do tego adnotacji JSR 303.

Zdefiniowanie ograniczeń za pomocą adnotacji JSR 303

JSR 303 (API Bean Validation) definiuje adnotacje, których można użyć do określenia ograniczeń dla właściwości komponentów `JavaBeans`.

IMPORT **Rozdział11/ch11-jsr303-validation.** (Ten projekt pokazuje zmodyfikowaną wersję projektu `ch11-data-binding` używającą adnotacji JSR 303 do zdefiniowania ograniczeń dla obiektu `FixedDepositDetails`. Po wdrożeniu aplikacji w serwerze Tomcat przejdź pod adres URL <http://localhost:8080/ch11-jsr303-validation>, a zobaczysz wyświetloną listę lokat w systemie).

W listingu 11.27 została przedstawiona klasa `FixedDepositDetails` używająca adnotacji JSR 303 do zdefiniowania ograniczeń dla znajdujących się w niej właściwości.

`@Min`, `@Max`, `@Email`, `@Size` i `@NotNull` to tylko niektóre z adnotacji zdefiniowanych przez JSR 303. W powyższym listingu widać, że dzięki użyciu adnotacji JSR 303 klasa `FixedDepositDetails` wyraźnie

Listing 11.27. Klasa *FixedDepositDetails* przedstawiająca przykład zdefiniowania ograniczeń JSR 303Projekt: *ch11-jsr303-validation*Położenie pliku: *src/main/java/sample/spring/chapter11/domain*

```
package sample.spring.chapter11.domain;

import javax.validation.constraints.*;

public class FixedDepositDetails {
    private long id;

    @Min(1000)
    @Max(500000)
    private long depositAmount;

    @Email
    @Size(min=10, max=25)
    private String email;

    @NotNull
    private Date maturityDate;
    .....
}
```

wskazuje ograniczenia stosowane względem jej właściwości. Natomiast jeśli do przeprowadzenia weryfikacji obiektu używasz oferowanej przez Spring implementacji interfejsu *Validator*, ograniczenia będą się znajdowały w implementacji interfejsu *Validator* (patrz: listing 11.24).

W tabeli 11.4 wymieniamy ograniczenia narzucane przez JSR 303 w obiekcie *FixedDepositDetails* stworzonym na podstawie klasy przedstawionej w listingu 11.27.

W celu użycia adnotacji JSR 303 projekt *ch11-jsr303-validation* definiuje zależność od pliku JAR API JSR 303 (*validation-api-1.0.0.GA*) oraz frameworku Hibernate Validator (*hibernate-validation-4.3.0.Final*). Hibernate Validator zawiera referencyjną implementację JSR 303. Ponadto dostarcza dodatkowe adnotacje ograniczeń, które można wykorzystywać obok oferowanych przez JSR 303. Na przykład udostępnianej przez framework Hibernate Validator adnotacji *@NotBlank* można użyć do określenia, że wartość właściwości nie może być *null* lub pusta.

W tym miejscu trzeba koniecznie dodać, że JSR 303 pozwala także na tworzenie własnych ograniczeń, a następnie ich użycie w aplikacji. Na przykład możesz utworzyć ograniczenie o nazwie *@MyConstraint* i odpowiadający jej weryfikator, aby wymusić na obiektach stosowanie tego ograniczenia.

Po zdefiniowaniu ograniczeń JSR 303 w klasie *FixedDepositDetails* przekonajmy się, jak można zweryfikować obiekt *FixedDepositDetails*.

Weryfikacja obiektu używającego adnotacji JSR 303

Jeżeli w ścieżce dostępu zdefiniowanej przez zmienną środowiskową *CLASSPATH* znajdzie się dostawca JSR 303 (taki jak Hibernate Validator) oraz w pliku XML kontekstu aplikacji sieciowej zdefiniowano element *<annotation-driven>* oferowanego przez Spring schematu *mvc*, wówczas Spring

Tabela 11.4. Ograniczenia narzucane przez JSR 303 egzemplarzowi klasy *FixedDepositDetails*

Adnotacja JSR 303	Opis ograniczenia
@NotNull	Wartość właściwości oznaczonej tą adnotacją nie może wynosić null. Na przykład wartość właściwości <code>maturityDate</code> musi być inna niż null.
@Min	Wartość właściwości oznaczonej tą adnotacją musi być większa lub równa określonej wartości minimalnej. Na przykład adnotacja <code>@Min(1000)</code> dla właściwości <code>depositAmount</code> obiektu <code>FixedDepositDetails</code> oznacza, że wartość właściwości <code>depositAmount</code> musi być większa lub równa 1000.
@Max	Wartość właściwości oznaczonej tą adnotacją musi być mniejsza lub równa określonej wartości. Na przykład adnotacja <code>@Max(500000)</code> dla właściwości <code>depositAmount</code> obiektu <code>FixedDepositDetails</code> oznacza, że wartość właściwości <code>depositAmount</code> musi być mniejsza lub równa 500000.
@Size	Wielkość pola oznaczonego tą adnotacją musi zawierać się w przedziale zdefiniowanym przez atrybuty <code>min</code> i <code>max</code> . Na przykład adnotacja <code>@Size(min=5, max=100)</code> dla właściwości <code>email</code> obiektu <code>FixedDepositDetails</code> oznacza, że wielkość właściwości <code>email</code> musi wynosić co najmniej 5, ale nie więcej niż 100.
@Email	Wartość właściwości oznaczonej tą adnotacją musi zawierać poprawny adres e-mail. Na przykład adnotacja <code>@Email</code> dla właściwości <code>email</code> obiektu <code>FixedDepositDetails</code> oznacza, że wartość właściwości <code>email</code> musi być poprawnym adresem e-mail.

automatycznie włącza obsługę dla JSR 303. W tle element `<annotation-driven>` przeprowadza konfigurację egzemplarza klasy `LocalValidatorFactoryBean` odpowiedzialnej za wykrywanie obecności dostawcy JSR 303 (takiego jak Hibernate Validator) w ścieżce dostępu klas aplikacji i jego inicjalizację.

Klasa `LocalValidatorFactoryBean` implementuje oferowane przez JSR 303 interfejsy `Validator` i `ValidatorFactory` oraz udostępniany przez Spring interfejs `Validator`. Z tego powodu weryfikację obiektu można przeprowadzić poprzez wywołanie metody `validate()` interfejsu `Validator` oferowanego przez Spring lub metody `validate()` udostępnianej przez `Validator` JSR 303. Jak wspomnieliśmy na początku rozdziału, można także nakazać Springowi automatyczną weryfikację argumentu atrybutu modelu przekazanego metodzie oznaczonej adnotacją `@RequestMapping`. Takie rozwiązanie wymaga po prostu dodania adnotacji `@Valid` do argumentu atrybutu modelu.

Weryfikacja atrybutów modelu przez wyraźne wywołanie metody `validate()`

W listingu 11.28 została przedstawiona klasa `FixedDepositController` używająca oferowanego przez Spring interfejsu `Validator` do przeprowadzenia weryfikacji obiektu `FixedDepositDetails` (patrz: listing 11.27) zawierającego zdefiniowane ograniczenia JSR 303.

Listing 11.28. Klasa *FixedDepositController* przedstawiająca przykład weryfikacji obiektu *FixedDepositDetails* przy użyciu API *Validation* udostępnianego przez Spring

Projekt: *ch11-jsr303-validation*

Położenie pliku: *src/main/java/sample/spring/chapter11/web*

```
package sample.spring.chapter11.web;

import javax.validation.Valid;
.....
public class FixedDepositController {
    .....
    @Autowired
    private Validator validator;
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
    public String openFixedDeposit(
        @ModelAttribute(value = "newFixedDepositDetails") FixedDepositDetails
        fixedDepositDetails, BindingResult bindingResult, SessionStatus
        sessionStatus) {
        validator.validate(fixedDepositDetails, bindingResult);

        if (bindingResult.hasErrors()) { ..... } .....
    }
    .....
}
```

W powyższym listingu klasa *LocalValidatorFactoryBean* (implementująca oferowany przez Spring interfejs *Validator*) jest automatycznie wiązana w zmiennej egzemplarza *validator* obiektu *FixedDepositController*. W metodzie *openFixedDeposit()* wywołanie metody *validate()* interfejsu *Validator* skutkuje wywołaniem metody *validate(Object, Errors)* klasy *LocalValidatorFactoryBean* w celu weryfikacji egzemplarza *FixedDepositDetails*. Metodzie *validate()* jest przekazywany obiekt *BindingResult* przeznaczony do przechowywania błędów wykrytych podczas weryfikacji. W omawianym listingu warto zwrócić uwagę na fakt, że kontroler *FixedDepositController* nie używa bezpośrednio API JSR 303 w celu przeprowadzenia weryfikacji obiektu *FixedDepositDetails*. Zamiast tego użyte zostało API *Validation* oferowane przez Spring.

W listingu 11.29 przedstawiamy alternatywną wersję klasy *FixedDepositController* używającą API JSR 303 do przeprowadzenia weryfikacji obiektu *FixedDepositDetails*.

Listing 11.29. Klasa *FixedDepositController* przedstawiająca weryfikację obiektu *FixedDepositDetails* za pomocą API JSR 303

```
import javax.validation.ConstraintViolation;
import javax.validation.Validator;
import java.util.Set;
.....
public class FixedDepositController {
    .....
    @Autowired
    private Validator validator;
    .....
    @RequestMapping(params = "fdAction=create", method = RequestMethod.POST)
```

```

public String openFixedDeposit(
    @ModelAttribute(value = "newFixedDepositDetails") FixedDepositDetails
    fixedDepositDetails, BindingResult bindingResult, SessionStatus
    sessionStatus) {
    Set<ConstraintViolation<FixedDepositDetails>> violations =
        validator.validate(fixedDepositDetails);
    Iterator<ConstraintViolation<FixedDepositDetails>> itr = violations.iterator();

    if(itr.hasNext()) { ..... } .....
}
.....
}

```

W powyższym listingu klasa `LocalValidatorFactoryBean` (implementująca interfejs `JSR 303 Validator`) jest automatycznie wiązana w zmiennej egzemplarza `validator` obiektu `FixedDepositController`. W metodzie `openFixedDeposit()` wywołanie metody `validate()` interfejsu `Validator` skutkuje wywołaniem metody `validate(T)` klasy `LocalValidatorFactoryBean` w celu przeprowadzenia weryfikacji egzemplarza `FixedDepositDetails`. Wartością zwrótną metody `validate()` jest obiekt `java.util.Set` zawierający zgłoszone przez dostawcę `JSR 303` informacje o złamanych ograniczeniach. Sprawdzając obiekt `java.util.Set` zwrócony przez metodę `validate()`, możesz się dowiedzieć, czy zgłoszono złamanie jakichkolwiek ograniczeń.

Wywołanie weryfikacji atrybutów modelu za pomocą adnotacji `JSR 303 @Valid`

Istnieje możliwość nakazania Springowi, by przeprowadził automatyczną weryfikację argumentu atrybutu modelu przekazanego metodzie oznaczonej adnotacją `@RequestMapping`. To wymaga dodania do argumentu atrybutu modelu adnotacji `JSR 303 @Valid`. W listingu 11.30 została przedstawiona metoda `editFixedDeposit()` klasy `FixedDepositController`, która używa adnotacji `@Valid` do przeprowadzenia weryfikacji atrybutu modelu `editableFixedDepositDetails`.

Listing 11.30. Klasa `FixedDepositController` przedstawiająca wywoływanie weryfikacji przy użyciu adnotacji `@Valid`

Projekt: `ch11-jsr303-validation`

Położenie pliku: `src/main/java/sample/spring/chapter11/web`

```

package sample.spring.chapter11.web;

import javax.validation.Valid;
.....
public class FixedDepositController {
    .....
    @RequestMapping(params = "fdAction=edit", method = RequestMethod.POST)
    public String editFixedDeposit(
        @Valid @ModelAttribute("editableFixedDepositDetails") FixedDepositDetails
        fixedDepositDetails, BindingResult bindingResult, SessionStatus sessionStatus) {

        if (bindingResult.hasErrors()) { ..... } .....
    }
    .....
}

```

W powyższym listingu adnotacja `@Valid` w atrybucie modelu `editableFixedDepositDetails` powoduje automatyczne przeprowadzanie weryfikacji przez Spring. Informacje o złamanych ograniczeniach zgłaszane podczas weryfikacji są dodawane do obiektu `BindingResult`, który zawiera już informacje o błędach wykrytych w trakcie operacji dołączania danych.

Teraz zobaczysz, jak oferowana przez Spring biblioteka znaczników form może ułatwić tworzenie formularzy sieciowych na stronach JSP.

11.6. Biblioteka znaczników form

Istniejąca w Springu biblioteka znaczników o nazwie `form` oferuje znaczniki ułatwiające tworzenie stron JSP w aplikacjach Spring Web MVC. Biblioteka ta potrafi wygenerować różne elementy danych wejściowych formularza i dołączać dane formularza do obsługujących je obiektów.

W listingu 11.31 przedstawiamy stronę `createFixedDepositForm.jsp` projektu `ch11-jsr303-validation` używającą znaczników pochodzących z omawianej tutaj biblioteki `form`.

Listing 11.31. Zawartość pliku `createFixedDepositForm.jsp` przedstawiającego użycie biblioteki `form` oferowanej przez Spring

Projekt: `ch11-jsr303-validation`

Położenie pliku: `src/main/webapp/WEB-INF/jsp`

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<html>
.....
    <form:form commandName="newFixedDepositDetails"
        name="createFixedDepositForm" method="POST"
        action="${pageContext.request.contextPath}/fixedDeposit?fdAction=create">
        .....
        <tr>
            <td class="td"><b>Wartość (w PLN):</b></td>
            <td class="td"><form:input path="depositAmount" />
                <font style="color: #C11B17;"><form:errors path="depositAmount"/></font>
            </td>
        </tr>
        <tr>
            <td class="td"><b>Czas trwania:</b></td>
            <td class="td"><form:input path="maturityDate" />
                <font style="color: #C11B17;"><form:errors path="maturityDate"/></font></td>
            </tr>
        .....
        <td class="td"><input type="submit" value="Zapisz" />
        .....
    </form:form>
</html>
```

W omawianym listingu znajduje się przedstawiona poniżej dyrektywa `taglib` udostępniająca stronie JSP znaczniki z oferowanej przez Spring biblioteki `form`:

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
```

Znaczniki `<form>` z oferowanej przez Spring biblioteki generują formularz HTML, którego wartości pól są dołączane do właściwości atrybutu modelu wskazanego przez `commandName`. Znacznik `<form>` zawiera znaczniki `<input>` odpowiadające właściwościom atrybutu modelu wskazanego przez `commandName`. Podczas generowania formularza właściwości są odczytywane z atrybutu modelu i wyświetlane przez znaczniki `<input>`. Ponadto w trakcie wysyłania formularza wartości wprowadzone w jego polach są dołączane do odpowiadających im właściwości atrybutu modelu.

W listingu 11.31 znacznik `<form>` generuje formularz HTML przeznaczony do zakładania nowych lokat. Wartością atrybutu `commandName` jest `newFixedDepositDetails`, co oznacza mapowanie pól formularza na właściwości atrybutu modelu `newFixedDepositDetails`. Atrybut `name` określa nazwę formularza HTML generowanego przez znacznik `<form>`. Atrybut `method` wskazuje metodę HTTP, która powinna zostać użyta do wysyłania formularza po kliknięciu przycisku *Zapisz*. Z kolei atrybut `action` określa adres URL, do którego zostaną wysłane dane formularza. Adres URL wskazywany przez atrybut `action` musi być mapowany na unikalną metodę oznaczoną adnotacją `@RequestMapping` w aplikacji Spring Web MVC. W omawianym listingu adres URL w postaci `/${pageContext.request.contextPath}/fixedDeposit?fdAction=create` jest mapowany na metodę `openFixedDeposit()` klasy `FixedDepositController` (patrz: plik `FixedDepositController.java` w projekcie `ch11-jsr303-validation`). Warto zwrócić uwagę na fakt, że wartością zwrotną wyrażenia `/${pageContext.request.contextPath}` jest *ścieżka dostępu kontekstu* aplikacji sieciowej.

Znacznik `<input>` oferowanej przez Spring biblioteki znaczników form powoduje wygenerowanie elementu HTML `<input>`, którego atrybut `type` ma przypisaną wartość `text`. Atrybut `path` określa właściwość atrybutu modelu, na którą będzie mapowane dane pole. Kiedy formularz jest generowany, wartość tej właściwości zostanie wyświetlona przez element `<input>`. Ponadto po wysłaniu formularza wartością właściwości będzie wartość wprowadzona przez użytkownika w elemencie `<input>`.

Znacznik `<errors>` oferowanej przez Spring biblioteki znaczników form wyświetla komunikaty błędów wykryte podczas operacji weryfikacji i dołączania danych. Odpowiednie komunikaty błędów są dodawane do obiektu `BindingResult` w trakcie tych operacji. Jeżeli chcesz wyświetlić komunikaty błędów odpowiadające konkretnej właściwości, jej nazwę należy podać jako wartość atrybutu `path`. Jeżeli chcesz wyświetlić wszystkie komunikaty błędów przechowywane w obiekcie `BindingResult`, jako wartość atrybutu `path` podaj gwiazdkę (*).

Na stronie `createFixedDepositForm.jsp` została użyta jedynie niewielka liczba znaczników spośród oferowanych przez bibliotekę form. W tabeli 11.5 wymieniamy inne znaczniki, które znajdziesz w bibliotece.

Zobaczmy teraz, jak przedstawia się obsługa HTML5 oferowana przez bibliotekę form.

Obsługa HTML5 przez bibliotekę form

Począwszy od wydania Spring 3.0, biblioteka form pozwala na użycie w znacznikach atrybutów zdefiniowanych w standardzie HTML5. Na przykład przedstawiony poniżej znacznik `<textarea>` używa atrybutu `required` wprowadzonego w HTML5:

```
<form:textarea path="myProperty" required="required"/>
```

Tabela 11.5. Znaczniki oferowane przez bibliotekę form

Znacznik	Opis
<checkbox>	Generuje pole wyboru HTML (czyli element <code><input type="checkbox" /></code>). Gdy pole wyboru nie jest zaznaczone, to jego wartość <i>nie</i> jest wysyłana do serwera, dlatego znacznik <checkbox> dodatkowo generuje ukryte pole dla każdego pola wyboru, co pozwala na wysłanie serwerowi informacji o stanie danego pola. Przykład: <code><form:checkbox path="myProperty" /></code>. Atrybut <code>path</code> określa nazwę właściwości, do której zostanie dołączona wartość pola wyboru.
<checkboxes>	Generuje wiele pól wyboru HTML. Przykład: <code><form:checkboxes path="myPropertyList" items="{someList}" /></code>. Atrybut <code>path</code> określa nazwę właściwości, do której zostaną dołączone wartości zaznaczonych pól wyboru. Z kolei atrybut <code>items</code> wskazuje nazwę atrybutu modelu zawierającego listę opcji do wyświetlenia w charakterze pól wyboru.
<radiobutton>	Generuje przycisk opcji HTML (czyli element <code><input type="radio" /></code>). Przykład: <code><form:radiobutton path="myProperty" value="myValue" /></code>. Atrybut <code>path</code> określa nazwę właściwości, z którą jest połączony przycisk opcji, natomiast atrybut <code>value</code> wskazuje wartość przypisaną przyciskowi opcji.
<radiobuttons>	Generuje wiele przycisków opcji HTML. Przykład: <code><form:radiobuttons path="myProperty" items="{myValues}" /></code>. Atrybut <code>items</code> określa listę opcji do wyświetlenia w charakterze przycisków opcji, natomiast atrybut <code>path</code> wskazuje nazwę właściwości, do której zostaną dołączone wartości zaznaczonych przycisków opcji.
<password>	Generuje pole pozwalające na podanie hasła (czyli element <code><input type="password" /></code>).
<select>	Generuje element HTML <code><select></code>. Przykład: <code><form:select path="book" items="{books}" /></code>. Atrybut <code>items</code> określa właściwość atrybutu modelu zawierającą listę opcji do wyświetlenia w elemencie HTML <code><select></code>, natomiast atrybut <code>path</code> wskazuje nazwę właściwości, do której zostaną dołączone wartości zaznaczonych opcji.
<option>	Generuje element HTML <code><option></code>. Przykład: <pre><form:select path="book"> <form:option value="Rozpoczęcie pracy ze Spring Framework"/> <form:option value="Rozpoczęcie pracy ze Spring Web MVC"/> </form:select></pre>
<options>	Generuje elementy HTML <code><option></code>.
<textarea>	Generuje element HTML <code><textarea></code>.
<hidden>	Generuje ukryte pole formularza HTML (czyli element <code><input type="hidden" /></code>).

Atrybut `required="required"` wskazuje, że użytkownik musi wprowadzić informacje w danym znaczniku `<textarea>`. Użycie atrybutu `required` pozwala uniknąć konieczności tworzenia kodu JavaScriptu przeznaczonego do przeprowadzania weryfikacji po stronie klienta dotyczącej wymaganych pól formularza. Jeżeli użytkownik nie wprowadzi żadnych danych w znaczniku `<textarea>` i spróbuje wysłać formularz, wówczas przeglądarka internetowa wyświetli komunikat informujący, że wypełnienie danego znacznika `<textarea>` jest obowiązkowe i pole nie może zostać puste.

W HTML5 istnieje możliwość przypisania atrybutowi `type` wartości `email`, `datetime`, `date`, `month`, `week`, `time`, `range`, `color`, `reset` itd. Począwszy od wydania Spring 3.1, znacznik `<input>` pozwala na przypisanie atrybutowi `type` wartości innej niż `text`. Na przykład w poniższym znaczniku `<input>` wartość atrybutu `type` została określona jako `email`:

```
<form:input path="myProperty" type="email"/>
```

Kiedy użytkownik spróbuje wysłać formularz sieciowy zawierający pole typu `email`, przeglądarka internetowa sprawdzi, czy to pole zawiera poprawny adres e-mail. Jeżeli adres e-mail nie jest poprawny, przeglądarka wyświetli odpowiedni komunikat. Ponieważ weryfikacja wartości pola jest przeprowadzana przez przeglądarkę internetową, nie musisz tworzyć żadnego kodu JavaScriptu przeznaczonego do sprawdzenia, czy dane pole zawiera poprawny adres e-mail.

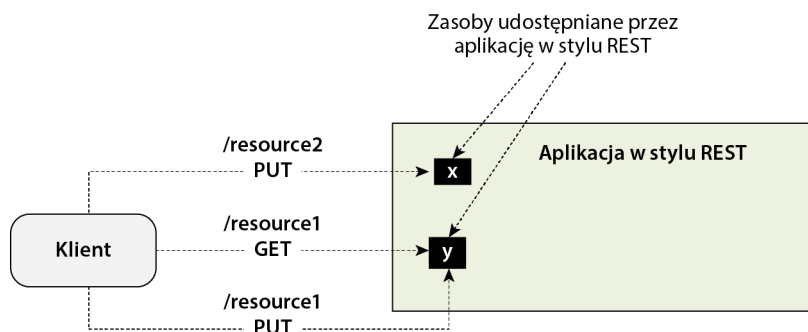
11.7. Podsumowanie

W tym rozdziale poznałeś wiele podstawowych funkcji Spring Web MVC. Przeanalizowaliśmy adnotacje `@ModelAttribute` i `@SessionAttributes`, które są najczęściej stosowane podczas tworzenia kontrolerów oznaczonych adnotacjami. Ponadto dokładnie omówiliśmy temat dołączania danych i weryfikacji danych w Springu. Z następnego rozdziału dowiesz się, jak opracować usługi sieciowe typu RESTful, używając frameworku Spring Web MVC.

Rozdział 12. Tworzenie usług sieciowych typu RESTful za pomocą Spring Web MVC

12.1. Wprowadzenie

REST (ang. *Representational State Transfer*) to styl architektoniczny, w którym aplikacja definiuje zasoby unikalnie identyfikowane przez adresy URI (ang. *Uniform Resource Identifier*). Klienci aplikacji w stylu REST pracują z zasobem, wysyłając za pomocą metody HTTP GET, POST, PUT lub DELETE żądania do adresu URI, na który jest mapowany dany zasób. Na rysunku 12.1 pokazujemy aplikację w stylu REST używaną przez klienty.



Rysunek 12.1. Aplikacja w stylu REST definiuje zasoby *x* i *y*, które są unikalnie identyfikowane przez adresy URI odpowiednio */resource2* i */resource1*

Na rysunku 12.1 przedstawiamy aplikację w stylu REST oferującą dwa zasoby — *x* i *y*. Zasób *x* jest mapowany na adres URI */resource2*, natomiast zasób *y* jest mapowany na adres URI */resource1*. Klient może wykorzystywać zasób *x* przez wykonywanie żądań HTTP do adresu URL */resource2*. Użycie zasobu *y* wymaga wykonywania żądań HTTP do adresu URI */resource1*.

Jeżeli usługa sieciowa stosuje styl REST, wówczas jest określana mianem *usługi sieciowej w stylu RESTful*. W kontekście usług sieciowych RESTful zasób można traktować jako *dane* udostępniane przez tę usługę sieciową. Na udostępnionych danych klient ma możliwość wykonywania operacji CRUD (CREATE, READ, UPDATE i DELETE) za pomocą żądań HTTP wykonywanych do usługi sieciowej RESTful. Klient i usługa sieciowa wymieniają się danymi, które mogą być w formacie XML lub JSON (ang. *JavaScript Object Notation*), w postaci zwykłego ciągu tekstowego lub w innym dowolnym typie MIME obsługiwany przez protokół HTTP.

Usługi sieciowe RESTful są znacznie prostsze w implementacji i znacznie bardziej skalowalne w porównaniu do usług sieciowych typu SOAP. W usługach sieciowych typu SOAP żądania i odpowiedzi zawsze są w formacie XML. Z kolei w usługach sieciowych typu RESTful dla żądań i odpowiedzi na żądania można użyć formatu JSON, XML, zwykłego tekstu itd. Z tego rozdziału dowiesz się, jak Spring Web MVC ułatwia tworzenie i uzyskiwanie dostępu do usług sieciowych typu RESTful.

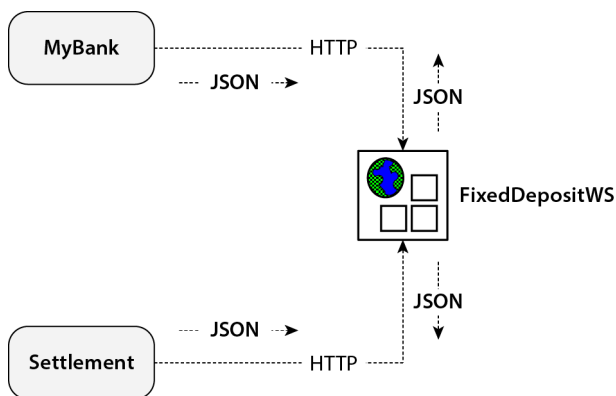
Rozpocniemy od przeanalizowania wymagań usługi sieciowej typu RESTful, która będzie implementowana za pomocą Spring Web MVC.

12.2. Usługa sieciowa do obsługi lokat

We wcześniejszej części książki zobaczyłeś, że aplikacja sieciowa MyBank oferuje funkcje wyświetlania listy lokat, zakładania nowych lokat oraz modyfikowania i zamykania istniejących lokat. Ponieważ funkcje związane z lokatami mogą być dostępne dla innych aplikacji, trzeba je wyodrębnić z aplikacji sieciowej MyBank i wdrożyć w postaci usługi sieciowej w stylu RESTful. Usłudze tej nadajemy nazwę FixedDepositWS.

Na rysunku 12.2 widać, że usługa sieciowa FixedDepositWS jest używana przez aplikacje MyBank i Settlement.

Rysunek 12.2.
Aplikacje MyBank
i Settlement uzyskują
dostęp do usługi
sieciowej
FixedDepositWS



Z rysunku 12.2 wynika, że aplikacje sieciowe MyBank i Settlement wykorzystują usługę sieciową FixedDepositWS, wymieniając dane w formacie JSON. Wkrótce przekonasz się, że JSON to znacznie prostsza alternatywa dla XML w zakresie wymiany danych między aplikacjami.

Zobaczmy, jak za pomocą Spring Web MVC zaimplementować FixedDepositWS jako usługę sieciową w stylu RESTful.

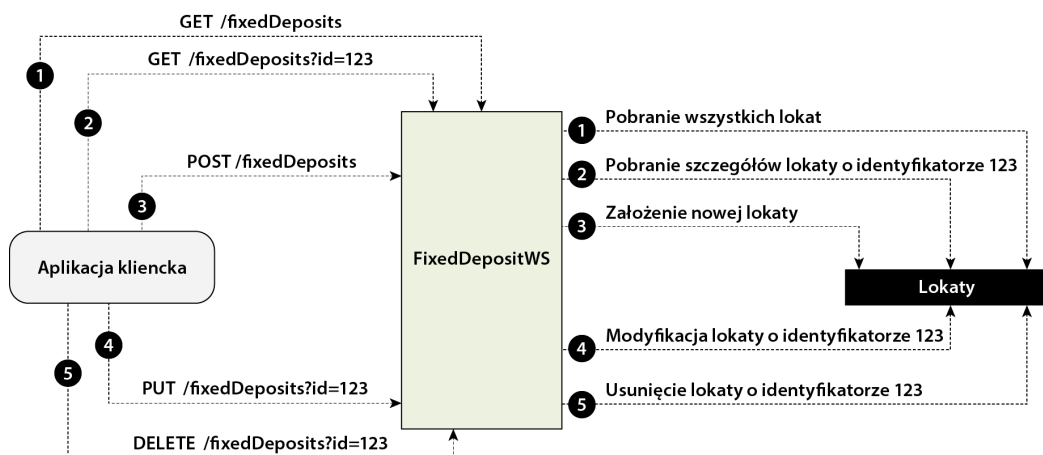
12.3. Implementacja usługi sieciowej w stylu RESTful za pomocą Spring Web MVC

W celu opracowania usługi sieciowej w stylu RESTful konieczne jest wykonanie następujących kroków:

- określenie zasobów udostępnianych przez usługę sieciową;
- określenie adresów URI odpowiadających wskazanym zasobom;
- określenie operacji, które mogą być przeprowadzane na zasobach;
- mapowanie metod HTTP na wskazane operacje.

W przypadku usługi sieciowej FixedDepositWS dane lokat przedstawiają zasoby udostępniane przez usługę sieciową. Jeżeli lokaty dostępne w systemie są przez usługę sieciową FixedDepositWS mapowane na adres URI */fixedDeposits*, wówczas klient usługi sieciowej może wykonywać operacje na lokatach przez wykonywanie żądań HTTP do adresu URI */fixedDeposits*.

W usłudze sieciowej stylu RESTful metoda HTTP używana przez klienty do pracy z zasobami wskazuje operację, która ma zostać wykonana na zasobie. Żądanie HTTP GET pobiera stan zasobu, POST tworzy nowy zasób, PUT modyfikuje stan zasobu, natomiast DELETE usuwa dany zasób. Rysunek 12.3 przedstawia operacje przeprowadzane przez usługę sieciową FixedDepositWS, gdy klient wykonuje żądania HTTP GET, POST, PUT i DELETE do adresu URI */fixedDeposits*.



Rysunek 12.3. Żądania HTTP wykonywane przez klienta FixedDepositWS do adresu URI */fixedDeposits*, aby pracować z danymi lokat

Na rysunku 12.3 widać, że klient usługi sieciowej FixedDepositWS wykonuje żądania HTTP GET, POST, PUT i DELETE na adres URI */fixedDeposits* i tym samym pracuje z danymi lokat. Parametr id ciągu tekstowego zapytania unikalnie identyfikuje lokatę w systemie. W tabeli 12.1 przedstawiamy przeznaczenie poszczególnych żądań pokazanych na rysunku 12.3.

Z tabeli 12.1 wynika, że usługa sieciowa FixedDepositWS i korzystające z niej klienty wymieniają informacje w formacie JSON. Zanim przejdziemy do szczegółów implementacji usługi sieciowej FixedDepositWS, warto zobaczyć, jak przedstawiają się dane w formacie JSON.

Format JSON

JSON to tekstowy format danych używany przez aplikacje w celu wymiany danych *strukturalnych*. Dane przedstawione w formacie JSON są znacznie zwięźlejsze niż te same dane w formacie XML, więc JSON można uznać za prostszą alternatywę dla XML. Aby ułatwić konwersję obiektów Javy na JSON i na odwrót, można skorzystać z biblioteki JSON, takiej jak FlexJson (<http://flexjson.sourceforge.net/>) lub Jackson (<https://github.com/FasterXML/jackson>).

Tabela 12.1. *Przeznaczenie żądań HTTP w przypadku usługi sieciowej FixedDepositWS*

Metoda HTTP	Adres URI	Opis
GET	<i>/fixedDeposits</i>	Pobranie informacji szczegółowych o wszystkich lokatach znajdujących się w systemie. Usługa sieciowa FixedDepositWS udziela odpowiedzi w formacie JSON.
GET	<i>/fixedDeposits?id=123</i>	Pobranie informacji szczegółowych o lokacie, której identyfikator wynosi 123. Usługa sieciowa FixedDepositWS udziela odpowiedzi w formacie JSON.
POST	<i>/fixedDeposits</i>	Założenie nowej lokaty w systemie. Informacje szczegółowe o zakładanej lokacie klient usługi sieciowej przekazuje w formacie JSON.
PUT	<i>/fixedDeposits?id=123</i>	Modyfikowanie lokaty, której identyfikator wynosi 123. Informacje szczegółowe o modyfikowanej lokacie klient usługi sieciowej przekazuje w formacie JSON.
DELETE	<i>/fixedDeposits?id=123</i>	Usunięcie lokaty, której identyfikator wynosi 123.

Przyjmujemy założenie, że klasa `Person` definiuje atrybuty `firstName` i `lastName`. Jeżeli utworzysz egzemplarz obiektu `Person` i przypiszesz atrybutowi `firstName` wartość `Moje_imię`, natomiast atrybutowi `lastName` wartość `Moje_nazwisko`, wówczas obiekt `Person` w formacie JSON będzie się przedstawiać jak w listingu 12.1.

Listing 12.1. *Obiekt `Person` przedstawiony w formacie JSON*

```
{
  "firstName" : "Moje_imię",
  "lastName" : "Moje_nazwisko"
}
```

W powyższym listingu widać, że każdy atrybut obiektu `Person` jest w formacie JSON przedstawiany jako `<nazwa-atrybutu> : <wartość-atrybutu>`.

W formacie JSON można wyrazić także kolekcję obiektów Javy. W listingu 12.2 pokazujemy, jak w formacie JSON przedstawić kolekcję obiektów `Person`.

Listing 12.2. *Kolekcja obiektów `Person` przedstawionych w formacie JSON*

```
[
  {
    "firstName" : "Moje_imię",
    "lastName" : "Moje_nazwisko"
  },
  {
    "firstName" : "Twoje_imię",
    "lastName" : "Twoje_nazwisko"
  }
]
```

Nie musisz tworzyć kodu przeprowadzającego konwersję obiektu na format JSON i na odwrot. Zamiast tego usługa sieciowa i jej klienci wykorzystują bibliotekę FlexJson lub Jackson do przeprowadzenia konwersji. Jak się wkrótce przekonasz, Spring Web MVC używa biblioteki Jackson do konwersji formatu JSON na postać obiektów Javy i na odwrot.

Przechodzimy teraz do implementacji usługi sieciowej FixedDepositWS za pomocą Spring Web MVC.

IMPORT **Rozdział12/ch12-webservice.** (Ten projekt pokazuje implementację za pomocą Spring Web MVC usługi sieciowej w stylu RESTful o nazwie FixedDepositWS. W dalszej części rozdziału zobaczysz, w jaki sposób usługa sieciowa FixedDepositWS jest dostępna dla klientów).

Implementacja usługi sieciowej FixedDepositWS

Oferowane przez Spring Web MVC adnotacje takie jak @Controller, @RequestMapping, @Request ↗Param, @PathVariable, @ResponseBody, @RequestBody itd. obsługują tworzenie usług sieciowych w stylu RESTful. W tym podrozdziale zobaczysz przykłady użycia niektórych z wymienionych powyżej adnotacji podczas tworzenia usługi sieciowej FixedDepositWS.

W usłudze sieciowej FixedDepositWS klasa FixedDepositController (kontroler Spring Web MVC) jest odpowiedzialna za obsługę żądań usługi sieciowej. Klasa FixedDepositController jest jak każdy inny kontroler Spring Web MVC, z wyjątkiem faktu, że metody oznaczone adnotacjami @Request ↗Mapping *nie* generują widoków.

W listingu 12.3 pokazano, że adnotacje @Controller i @RequestMapping są używane do mapowania żądań sieciowych na odpowiednie metody klasy FixedDepositController.

Listing 12.3. Klasa FixedDepositController jest odpowiedzialna za obsługę usługi sieciowej

Projekt: ch12-webservice

Położenie pliku: src/main/java/sample/spring/chapter12/web

```
package sample.spring.chapter12.web;

import org.springframework.http.ResponseEntity;
.....
@Controller
@RequestMapping(value = "/fixedDeposits")
public class FixedDepositController {
    .....
    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<List<FixedDepositDetails>> getFixedDepositList() { ..... }

    @RequestMapping(method = RequestMethod.GET, params = "id")
    public ResponseEntity<FixedDepositDetails> getFixedDeposit(@RequestParam("id")
        int id) { ..... }
    .....
}
```

Metoda getFixedDepositList() zwraca listę lokat znajdujących się w systemie, natomiast metoda getFixedDeposit() zwraca informacje szczegółowe o lokacie wskazanej przez argument id. W omawianym listingu widać, że adnotacja @RequestMapping została użyta na poziomach klasy i metod

do mapowania żądań na metody `getFixedDepositList()` i `getFixedDeposit()`. Metoda `getFixedDepositList()` jest wywoływana, gdy klient wykonuje żądanie HTTP GET na adres URI `/fixedDeposits`, natomiast wywołanie metody `getFixedDeposit()` następuje po wykonaniu przez klienta na adres URI `/fixedDeposits` żądania HTTP GET zawierającego parametr żądania o nazwie `id`. Dlatego jeśli adres URI żądania będzie miał postać `/fixedDeposits?id=123`, nastąpi wywołanie metody `getFixedDeposit()`.

W tabeli 12.2 przedstawiamy podsumowanie mapowania adresów URI żądań i metod HTTP na metody zdefiniowane w klasie `FixedDepositController`.

Tabela 12.2. Mapowanie metod HTTP i adresów URL na metody klasy `FixedDepositController`

Metoda HTTP	URI	Metoda <code>FixedDepositController</code>
GET	<code>/fixedDeposits</code>	<code>getFixedDepositList()</code>
GET	<code>/fixedDeposits?id=123</code>	<code>getFixedDeposit()</code>
POST	<code>/fixedDeposits</code>	<code>openFixedDeposit()</code>
PUT	<code>/fixedDeposits?id=123</code>	<code>editFixedDeposit()</code>
DELETE	<code>/fixedDeposits?id=123</code>	<code>closeFixedDeposit()</code>

W rozdziałach 10. i 11. pokazaliśmy, że metody oznaczone adnotacją `@RequestMapping` zwracają informacje o widoku, które są następnie używane przez egzemplarz `DispatcherServlet` do wygenerowania widoku (np. strony JSP lub serwlet). W usłudze sieciowej w stylu RESTful metody oznaczone adnotacją `@RequestMapping` zwracają klientom dane, a nie informacje o widoku. Dlatego metody `getFixedDepositList()` i `getFixedDeposit()` zostały zdefiniowane w taki sposób, aby zwracały obiekty typu `ResponseEntity`. Zobaczmy teraz, jak można użyć obiektu `ResponseEntity` w klasie `FixedDepositController`.

Określenie odpowiedzi HTTP za pomocą `ResponseEntity`

Egzemplarz `ResponseEntity` przedstawia odpowiedź HTTP zawierającą nagłówki, część główną odpowiedzi oraz kod stanu. Obiekt definiowany jako *część główna* w egzemplarzu `ResponseEntity` jest umieszczany w odpowiedzi HTTP przez Spring Web MVC.

W listingu 12.4 pokazujemy, jak obiekt `ResponseEntity` jest tworzony przez metodę `getFixedDepositList()` klasy `FixedDepositController`.

Listing 12.4. Klasa `FixedDepositController` przedstawiająca przykład tworzenia egzemplarza `ResponseEntity`

Projekt: `ch12-webservice`

Położenie pliku: `src/main/java/sample/spring/chapter12/web`

```
package sample.spring.chapter12.web;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
.....
public class FixedDepositController {
    .....
    @RequestMapping(method = RequestMethod.GET)
```

```

public ResponseEntity<List<FixedDepositDetails>> getFixedDepositList() {
    .....
    return new ResponseEntity<List<FixedDepositDetails>>(
        fixedDepositService.getFixedDeposits(), HttpStatus.OK);
}
    .....
}

```

W powyższym listingu lokata przekazana konstruktorowi `ResponseEntity` zostaje umieszczona w części głównej odpowiedzi HTTP. Komponent `HttpStatus` to typ enum definiujący kody stanu HTTP. Stałą `OK` odwołuje się do kodu stanu HTTP 200. Zwróć uwagę, że typem wartości zwrotnej metody `getFixedDepositList()` jest `ResponseEntity<List<FixedDepositDetails>>`. Oznacza to, że w części głównej odpowiedzi zostaje umieszczony obiekt typu `List<FixedDepositDetails>`. Spring Web MVC używa odpowiedniego egzemplarza `HttpMessageConverter` (omówiony szczegółowo w podrozdziale 12.5) do przeprowadzenia konwersji obiektu `List<FixedDepositDetails>` na format oczekiwany przez klienta.



W dalszej części rozdziału zobaczysz, że klienci mogą używać oferowanego przez Spring `RestTemplate` do wywoływania metod zdefiniowanych w klasie `FixedDepositController` i do pobierania obiektów znajdujących się w części głównej odpowiedzi.

Wszystkie metody oznaczone adnotacją `@RequestMapping` w klasie `FixedDepositController` wskazują `ResponseEntity` jako ich typ wartości zwrotnej. Jeżeli *nie* ma konieczności wysyłania kodu stanu HTTP w odpowiedzi, wówczas można użyć oferowanej przez Spring klasy `HttpEntity` zamiast `ResponseEntity`. Klasa `HttpEntity` przedstawia żądanie HTTP lub odpowiedź na żądanie, a `ResponseEntity` jest podklasą klasy `HttpEntity`, która dodaje do odpowiedzi kod stanu HTTP.

W listingu 12.5 przedstawiamy zmodyfikowaną wersję metody `getFixedDepositList()`, która tworzy i zwraca egzemplarz `HttpEntity`.

Listing 12.5. Klasa `FixedDepositController` przedstawiająca przykład użycia `HttpEntity` zamiast `ResponseEntity`

```

import org.springframework.http.HttpStatus;
import org.springframework.http.HttpEntity;
    .....
public class FixedDepositController {
    .....
    @RequestMapping(method = RequestMethod.GET)
    public HttpEntity<List<FixedDepositDetails>> getFixedDepositList() {
        .....
        return new HttpEntity<List<FixedDepositDetails>>
            (fixedDepositService.getFixedDeposits());
    }
    .....
}

```

W powyższym listingu pokazaliśmy, że lokaty znalezione w systemie są przekazywane konstruktorowi `HttpEntity`. Podobnie jak w przypadku `ResponseEntity` (patrz: listing 12.4), przekazane do `HttpEntity` informacje o lokatach są umieszczane w części głównej odpowiedzi HTTP.

Obiekty `HttpEntity` i `ResponseEntity` pozwalają na ustawienie nagłówków HTTP odpowiedzi. W listingu 12.6 przedstawiamy sytuację, gdy w odpowiedzi zostaje ustawiony nagłówek `some-header`.

Listing 12.6. Przykład użycia obiektu `HttpHeaders`

```
import org.springframework.http.HttpHeaders;
.....
@RequestMapping(method = RequestMethod.GET)
public HttpEntity<String> doSomething() {
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("some-header", "dowolna-wartość");

    return new HttpEntity<String>("Witaj, świecie!", responseHeaders);
}
.....
```

Oferowany przez Spring obiekt `HttpHeaders` zawiera nagłówki ustawiane w odpowiedzi HTTP. W omawianym listingu metoda `set()` obiektu `HTTPHeader` powoduje ustawienie w odpowiedzi nagłówka `some-header` (wraz z wartością `dowolna-wartość`). Po wywołaniu metody `doSomething()` w części głównej odpowiedzi zostaje umieszczony ciąg tekstowy `Witaj, świecie!`, a ponadto w odpowiedzi HTTP jest wstawiany nagłówek `some-header`.

Ponieważ metody oznaczone adnotacją `@RequestMapping` mogą być zdefiniowane w sposób pozwalający na akceptację argumentu w postaci obiektu `HttpServletResponse`, przechodzimy teraz do tematu bezpośredniego ustawiania nagłówków i części głównej odpowiedzi za pomocą obiektu `HttpServletResponse`.

Przygotowanie odpowiedzi za pomocą `HttpServletResponse`

W listingu 12.7 przedstawiamy przykład oznaczonej adnotacją `@RequestMapping` metody, która bezpośrednio zapisuje dane w obiekcie `HttpServletResponse`.

Listing 12.7. Przygotowanie odpowiedzi za pomocą obiektu `HttpServletResponse`

```
import javax.servlet.http.HttpServletResponse;
.....
@RequestMapping(method = RequestMethod.GET)
public void doSomething(HttpServletResponse response) throws IOException {
    response.setHeader("some-header", "dowolna-wartość");
    response.setStatus(200);
    response.getWriter().write("Witaj, świecie!");
}
.....
```

Zamiast bezpośrednio zapisywać odpowiedź w obiekcie `HttpServletResponse` należy użyć obiektu `ResponseEntity` (lub `HttpEntity`) w celu zapewnienia lepszej testowalności kontrolerów.

Zobaczmy teraz, jak oferowana przez Spring adnotacja `@ResponseBody` na poziome metody zapisuje w części głównej odpowiedzi HTTP wartość zwrótną metody.



W wydaniu Spring 4.0 adnotacja `@ResponseBody` może być użyta również na poziomie klasy. Jeżeli adnotacja `@ResponseBody` zostanie podana na poziomie klasy, wówczas jest dziedziczona przez metody kontrolera oznaczone adnotacją `@RequestMapping`.

Użycie adnotacji `@ResponseBody` w celu dołączenia wartości zwrotnej metody do części głównej odpowiedzi HTTP

W listingu 12.8 przedstawiamy przykład użycia adnotacji `@ResponseBody`.

Listing 12.8. Przykład użycia adnotacji `@ResponseBody`

```
import org.springframework.web.bind.annotation.ResponseBody;
....
@RequestMapping(method = RequestMethod.GET)
@ResponseBody
public String doSomething() {
    return "Witaj, świecie!";
}
....
```

W powyższym listingu ciąg tekstowy `Witaj, świecie!` zwrócony przez metodę `doSomething()` zostaje zapisany w części głównej odpowiedzi HTTP. Z podrozdziału 10.7 dowiedziałeś się, że typem wartości zwrotnej metody oznaczonej adnotacją `@RequestMapping` jest `String`, a sama wartość zwrotna jest traktowana jako nazwa *widoku* przeznaczonego do wygenerowania. W omawianym listingu adnotacja `@ResponseBody` w metodzie `doSomething()` nakazuje frameworkowi Spring Web MVC umieszczenie zwróconego ciągu tekstowego w części głównej odpowiedzi HTTP zamiast potraktować go jako nazwę widoku. Warto zwrócić uwagę na fakt, że Spring używa odpowiedniej implementacji `HttpMessageConverter` (zgodnie z wyjaśnieniem przedstawionym w podrozdziale 12.5) do umieszczenia w odpowiedzi HTTP wartości zwróconej przez metodę oznaczoną adnotacją `@ResponseBody`.

Skoro poznaliśmy różne sposoby, na jakie metoda oznaczona adnotacją `@RequestMapping` może umieszczać dane w odpowiedzi HTTP, możemy przejść do sposobów, na jakie metoda oznaczona adnotacją `@RequestMapping` może odczytać informacje z żądania HTTP za pomocą adnotacji `@RequestBody`.

Dołączanie części głównej żądania HTTP do metody parametru z użyciem adnotacji `@RequestBody`

Metoda oznaczona adnotacją `@RequestMapping` może użyć adnotacji na poziomie parametru metody oznaczonej adnotacją `@RequestBody` w celu dołączenia części głównej żądania HTTP do parametru metody. Spring Web MVC używa odpowiedniej implementacji `HttpMessageConverter` (zgodnie z wyjaśnieniem przedstawionym w podrozdziale 12.5) do przeprowadzenia konwersji części głównej żądania HTTP na typ parametru metody. W listingu 12.9 przedstawiamy przykład użycia adnotacji `@RequestBody` w klasie `FixedDepositController` aplikacji `MyBank`.

W powyższym listingu argument metody typu `FixedDepositDetails` jest oznaczony adnotacją `@RequestBody`. Moduł Spring Web MVC jest odpowiedzialny za konwersję części głównej żądania

Listing 12.9. Przykład użycia adnotacji `@RequestBody`Projekt: *ch12-webservice*Położenie pliku: *src/main/java/sample/spring/chapter12/web*

```

package sample.spring.chapter12.web;
.....
import org.springframework.web.bind.annotation.RequestBody;
.....
@Controller
@RequestMapping(value = "/fixedDeposits")
public class FixedDepositController {
    .....
    @RequestMapping(method = RequestMethod.POST)
    public ResponseEntity<FixedDepositDetails> openFixedDeposit(
        @RequestBody FixedDepositDetails fixedDepositDetails,
        BindingResult bindingResult) {
        new FixedDepositDetailsValidator().validate(fixedDepositDetails, bindingResult);
        .....
    }
    .....
}

```

HTTP na obiekt typu `FixedDepositDetails`. W omawianym listingu klasa `FixedDepositDetails` ➔ `Validator` jest implementacją oferowanego przez Spring interfejsu `Validator`, który przeprowadza weryfikację obiektu `FixedDepositDetails` przed próbą założenia nowej lokaty.

Rozwiązaniem alternatywnym dla użycia adnotacji `@RequestBody` jest bezpośredni odczyt części głównej żądania HTTP za pomocą obiektu `HttpServletRequest` i konwersja odczytanych danych na typ danych żądany przez metodę. Adnotacja `@RequestBody` ułatwia konwersję, ponieważ korzysta z odpowiedniej implementacji `HttpMessageConverter` do przeprowadzenia konwersji części głównej żądania HTTP na typ obiektu oczekiwany przez metodę oznaczoną adnotacją `@RequestMapping`.

Zobaczmy teraz, jak adnotacja `@ResponseStatus` pozwala na ustawienie stanu odpowiedzi HTTP.

Ustawienie stanu odpowiedzi HTTP za pomocą adnotacji `@ResponseStatus`

Adnotacji `@ResponseStatus` można użyć do określenia stanu odpowiedzi HTTP zwracanej przez metodę oznaczoną adnotacją `@RequestMapping`. W listingu 12.10 przedstawiamy przykład zastosowania adnotacji `@ResponseStatus`.

Listing 12.10. Przykład użycia adnotacji `@ResponseStatus`

```

import org.springframework.web.bind.annotation.ResponseStatus;

public class SomeController {

    @RequestMapping(method = RequestMethod.GET)
    @ResponseStatus(value = HttpStatus.OK)
    @ResponseBody
    public SomeObject doSomething() {
        .....
    }
}

```


Ponieważ metoda `doSomething()` jest oznaczona adnotacją `@ResponseBody`, obiekt `SomeObject` zwracany przez metodę `doSomething` jest umieszczany w części głównej odpowiedzi HTTP. Ponadto adnotacja `@ResponseStatus` określa kod stanu odpowiedzi HTTP na 200 (przedstawiony za pomocą stałej `HttpStatus.OK`).

Zobaczmy teraz, jak adnotacja `@ExceptionHandler` jest używana w usłudze `FixedDepositWS` do obsługi wyjątków.

Obsługa wyjątków za pomocą adnotacji `@ExceptionHandler`

W podrozdziale 10.9 zobaczyłeś, że adnotacja `@ExceptionHandler` wskazuje metody kontrolera odpowiedzialne za obsługę wyjątków. Podobnie jak metody oznaczone adnotacją `@RequestMapping`, także metody oznaczone adnotacją `@ExceptionHandler` są w usłudze sieciowej w stylu RESTful oznaczone adnotacją `@ResponseBody`, a typ ich wartości zwrotnej zdefiniowany jest jako `ResponseEntity` (lub `HttpEntity`).

W listingu 12.11 przedstawiamy przykład użycia adnotacji `@ExceptionHandler` w klasie `FixedDepositController` projektu `ch12-webservice`.

Listing 12.11. Przykład użycia adnotacji `@ExceptionHandler`

Projekt: `ch12-webservice`

Położenie pliku: `src/main/java/sample/spring/chapter12/web`

```
package sample.spring.chapter12.web;

import sample.spring.chapter12.exception.ValidationException;
.....
public class FixedDepositController {
    .....
    @ExceptionHandler(ValidationException.class)
    @ResponseBody
    @ResponseStatus(value = HttpStatus.BAD_REQUEST)
    public String handleException(Exception ex) {
        logger.info("Obsługa ValidationException " + ex.getMessage());
        return ex.getMessage();
    }
}
```

Adnotacja `@ExceptionHandler` w metodzie `handleException()` wskazuje, że metoda `handleException()` będzie wywoływana po zgłoszeniu wyjątku `ValidationException` przez klasę `FixedDepositController` podczas przetwarzania żądania. Ponieważ metoda `handleException()` jest oznaczona także adnotacją `@ResponseBody`, wyjątek zwrócony przez tę metodę zostanie umieszczony w części głównej odpowiedzi HTTP. Adnotacja `@ResponseStatus` w metodzie `handleException()` skutkuje określeniem kodu stanu odpowiedzi HTTP na 400 (reprezentowany przez stałą `HttpStatus.BAD_REQUEST`).

W tym podrozdziale zobaczyłeś, jak zaimplementować usługę sieciową `FixedDepositWS` za pomocą frameworku Spring Web MVC. Zobaczmy teraz, jak można uzyskać do niej dostęp za pomocą klasy `RestTemplate`.

12.4. Uzyskanie dostępu do usługi sieciowej w stylu RESTful za pomocą klasy RestTemplate

Oferowana przez Spring klasa `RestTemplate` ułatwia uzyskanie dostępu do usług sieciowych RESTful, ponieważ zajmuje się zarządzaniem połączeniami HTTP i ewentualnymi błędami HTTP.

IMPORT **Rozdział12/ch12-webservice-client.** (Ten projekt przedstawia samodzielną aplikację Javy, która uzyskuje dostęp do usługi sieciowej `FixedDepositWS` za pomocą klas `RestTemplate` (*synchroniczny* dostęp do usługi sieciowej) i `AsyncRestTemplate` (*asynchroniczny* dostęp do usługi sieciowej). W projekcie `ch12-webservice-client` przyjęto założenie, że projekt `ch12-webservice` przedstawiający usługę sieciową `FixedDepositWS` został wdrożony pod adresem URL `http://localhost:8080/ch12-webservice`).

Konfiguracja klasy RestTemplate

W listingu 12.12 możesz zobaczyć, jak wygląda konfiguracja klasy `RestTemplate` w pliku XML kontekstu aplikacji znajdującym się w projekcie `ch12-webservice-client`.

Listing 12.12. Plik `applicationContext.xml` przedstawiający konfigurację klasy `RestTemplate`

Projekt: `ch12-webservice-client`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans .....>
  <bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
    <property name="errorHandler" ref="errorHandler" />
  </bean>

  <bean id="errorHandler" class="sample.spring.chapter12.MyErrorHandler" />
  .....
</beans>
```

Właściwość `errorHandler` klasy `RestTemplate` odwołuje się do oferowanej przez Spring implementacji interfejsu `ResponseErrorHandler`, który sprawdza odpowiedź HTTP pod kątem błędów i zajmuje się obsługą odpowiedzi, jeśli wystąpiły jakiegokolwiek błędy. `DefaultResponseErrorHandler` to domyślna implementacja `ResponseErrorHandler` standardowo udostępniana przez Spring. Jeżeli nie określisz właściwości `errorHandler`, Spring użyje implementacji `DefaultResponseErrorHandler`. W omawianym listingu pokazano, że klasa `RestTemplate` stosuje własną procedurę obsługi błędów o nazwie `MyErrorHandler`.

W listingu 12.13 przedstawiamy implementację klasy `MyErrorHandler`.

Listing 12.13. Klasa `MyErrorHandler`, czyli procedura obsługi błędów w odpowiedzi HTTP

Projekt: `ch12-webservice-client`

Położenie pliku: `src/main/java/sample/spring/chapter12`

```
package sample.spring.chapter12;

import org.apache.commons.io.IOUtils;
import org.springframework.http.client.ClientHttpResponse;
import org.springframework.web.client.DefaultResponseErrorHandler;
```

```

public class MyErrorHandler extends DefaultResponseErrorHandler {
    private static Logger logger = Logger.getLogger(MyErrorHandler.class);

    @Override
    public void handleError(ClientHttpResponse response) throws IOException {
        logger.info("Kod stanu przekazany przez usługę sieciową: "
            + response.getStatusCode());
        String body = IOUtils.toString(response.getBody());
        logger.info("Odpowiedź: " + body);
        super.handleError(response);
    }
}

```

W powyższym listingu widać, że klasa `MyErrorHandler` rozszerza klasę `DefaultResponseErrorHandler` i nadpisuje metodę `handleError()`. Jeżeli kod stanu odpowiedzi HTTP wskazuje na istnienie błędu, wówczas metoda `handleError()` stanie się odpowiedzialna za obsługę tej odpowiedzi. Argument `ClientHttpResponse` metody `handleError()` przedstawia odpowiedź HTTP otrzymaną z wywołanej usługi sieciowej w stylu RESTful. Wartością zwrótną wywołania metody `getBody()` klasy `ClientHttpResponse` jest część główna odpowiedzi HTTP otrzymana w postaci obiektu `InputStream`. Metoda `handleError()` klasy `MyErrorHandler` rejestruje informacje o kodzie stanu oraz część główną odpowiedzi HTTP, a także przekazuje metodzie `handleError()` klasy `DefaultResponseErrorHandler` zadanie obsługi błędu. W omawianym listingu pokazano, że klasa `MyErrorHandler` używa klasy `IOUtils` (Apache Commons IO) w celu pobrania zawartości odpowiedzi HTTP jako ciągu tekstowego (`String`).

Skoro poznaliśmy sposób konfiguracji klasy `RestTemplate`, możemy przejść do analizy sposobów jej wykorzystania przez klienty w celu uzyskania dostępu do usługi sieciowej typu RESTful.

Uzyskanie dostępu do usługi sieciowej `FixedDepositWS` za pomocą `RestTemplate`

W listingu 12.14 pokazujemy, jak klasa `FixedDepositWSClient` używa klasy `RestTemplate` w celu uzyskania dostępu do usługi sieciowej `FixedDepositWS`.

Listing 12.14. Klasa `FixedDepositWSClient` przedstawiająca przykład użycia klasy `RestTemplate`

Projekt: *ch12-webservice-client*

Położenie pliku: *src/main/java/sample/spring/chapter12*

```

package sample.spring.chapter12;
.....
import org.springframework.web.client.RestTemplate;

public class FixedDepositWSClient {
    private static ApplicationContext context;

    public static void main(String args[]) {
        context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");
        getFixedDepositList(context.getBean(RestTemplate.class));
        getFixedDeposit(context.getBean(RestTemplate.class));
        .....
    }
}

```

```
private static void getFixedDepositList(RestTemplate restTemplate) { ..... }
.....
}
```

W powyższym listingu widać, że metoda `main()` klasy `FixedDepositWSClient` wykonuje następujące zadania:

- przygotowanie kontenera Springa (reprezentowanego przez obiekt `ApplicationContext`);
- wywołanie metod takich jak `getFixedDepositList()`, `getFixedDeposit()` itd., które akceptują egzemplarz `RestTemplate` i są odpowiedzialne za wywołanie usługi sieciowej `FixedDepositWS`.

W listingu 12.15 przedstawiamy implementację metody `getFixedDepositList()` klasy `FixedDepositWSClient`, która wywołuje usługę sieciową `FixedDepositWS` wdrożoną pod adresem `http://localhost:8080/ch12-webservice` w celu pobrania listy lokat założonych w systemie.

Listing 12.15. *Implementacja metody `getFixedDepositList()` klasy `FixedDepositWSClient`*

Projekt: `ch12-webservice-client`

Położenie pliku: `src/main/java/sample/spring/chapter12`

```
package sample.spring.chapter12;
.....
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.*;
import org.springframework.web.client.RestTemplate;

public class FixedDepositWSClient {
    .....
    private static void getFixedDepositList(RestTemplate restTemplate) {
        HttpHeaders headers = new HttpHeaders();
        headers.add("Accept", "application/json");

        HttpEntity<String> requestEntity = new HttpEntity<String>(headers);

        ParameterizedTypeReference<List<FixedDepositDetails>> typeRef =
            new ParameterizedTypeReference<List<FixedDepositDetails>>() {
        };

        ResponseEntity<List<FixedDepositDetails>> responseEntity = restTemplate
            .exchange("http://localhost:8080/ch12-webservice/fixedDeposits",
                HttpMethod.GET, requestEntity, typeRef);

        List<FixedDepositDetails> fixedDepositDetails = responseEntity.getBody();
        logger.info("Lista lokat: \n" + fixedDepositDetails);
    }
    .....
}
```

W powyższym listingu metoda `exchange()` klasy `RestTemplate` została użyta w celu wykonania żądania HTTP GET do adresu URL `http://localhost:8080/ch12-webservice/fixedDeposits`. Ponieważ usługa sieciowa `FixedDepositWS` została wdrożona pod adresem URL `http://localhost:8080/ch12-webservice`, wykonanie żądania HTTP GET do `http://localhost:8080/ch12-webservice/fixedDeposits` skutkuje wywołaniem metody `getFixedDepositList()` klasy `FixedDepositController`. Wynika to

z faktu, że metoda `getFixedDepositList()` klasy `FixedDepositController` jest mapowana na adres URI `/fixedDeposits` (patrz: listing 12.3 lub klasa `FixedDepositController` w projekcie *ch12-webservice*).

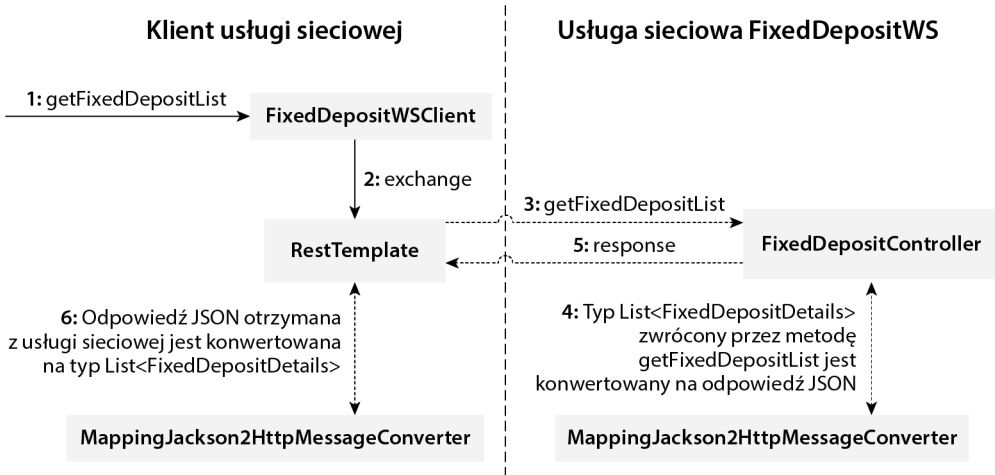
W listingu 12.15 obiekt `HttpEntity` przedstawia żądanie wykonane do usługi sieciowej, obiekt `HttpHeaders` przedstawia nagłówki w tym żądaniu, a obiekt `ParameterizedTypeReference` przedstawia *ogólny* typ odpowiedzi otrzymanej z usługi sieciowej. Wartość nagłówka żądania `Accept` to `application/json` i wskazuje, że odpowiedź z usługi sieciowej `FixedDepositWS` jest oczekiwana w formacie JSON. Po stronie usługi sieciowej wartość nagłówka `Accept` jest używana przez Spring Web MVC do wyboru odpowiedniej implementacji `HttpMessageConverter` przeznaczonej do konwersji wartości zwróconej przez metodę oznaczoną adnotacją `@ResponseBody` na format wskazywany przez nagłówek `Accept`. Na przykład jeśli wartością nagłówka `Accept` jest `application/json`, wówczas Spring Web MVC używa implementacji `MappingJackson2HttpMessageConverter` (implementacja `HttpMessageConverter`) do przeprowadzenia konwersji wartości zwróconej przez metodę oznaczoną adnotacją `@ResponseBody` na format JSON. Klasa `FixedDepositWSClient` określa wartość nagłówka `Accept` na `application/json`. Dlatego wartość zwrotna metody `getFixedDepositList()` klasy `FixedDepositController` jest konwertowana na format JSON.

Metoda `exchange()` klasy `RestTemplate` zwraca egzemplarz `ResponseEntity`, który przedstawia odpowiedź otrzymaną z usługi sieciowej. Ponieważ ogólny typ odpowiedzi otrzymany na skutek wywołania metody `getFixedDepositList()` klasy `FixedDepositController` to `List<FixedDepositDetails>`, następuje utworzenie egzemplarza `ParameterizedTypeReference<List<FixedDepositDetails>>` i jego przekazanie metodzie `exchange()`. Istnieje możliwość wywołania metody `getBody()` klasy `ResponseEntity` w celu pobrania odpowiedzi zwróconej przez usługę sieciową. W listingu 12.15 wartością zwrotną metody `getBody()` klasy `ResponseEntity` jest obiekt typu `List<FixedDepositDetails>` przedstawiający listę lokat przekazaną przez usługę sieciową `FixedDepositWS`.

Na rysunku 12.4 pokazujemy rolę, jaką odgrywa implementacja `MappingJackson2HttpMessageConverter`, gdy klasa `FixedDepositWSClient` wywołuje metodę `getFixedDepositList()` klasy `FixedDepositController`.

Na rysunku 12.4 widać, że implementacja `MappingJackson2HttpMessageConverter` jest używana do konwersji wartości zwrotnej metody `getFixedDepositList()` klasy `FixedDepositController` na format JSON. Ponadto implementacja `MappingJackson2HttpMessageConverter` została wykorzystana przez klasę `RestTemplate` do konwersji odpowiedzi JSON otrzymanej z `FixedDepositController` na obiekt Javy typu `List<FixedDepositDetails>`.

W listingu 12.15 metoda `exchange()` klasy `RestTemplate` została użyta w celu wykonania żądania HTTP GET do usługi sieciowej `FixedDepositWS`. Metoda `exchange()` jest zwykle stosowana, jeśli odpowiedź HTTP udzielona przez usługę sieciową musi być skonwertowana na ogólny typ Javy, a także w celu wysłania nagłówków żądania HTTP. Klasa `RestTemplate` definiuje również metody charakterystyczne dla HTTP, które ułatwiają tworzenie klientów RESTful. Na przykład metodę `getForEntity()` można wykorzystać do wykonywania żądań HTTP GET, metodę `postForEntity()` do wykonywania żądań HTTP POST, metodę `delete()` do wykonywania żądań HTTP DELETE itd.



Rysunek 12.4. Metoda `getFixedDepositList()` klasy `FixedDepositWSClient` używa klasy `RestTemplate` w celu wykonywania żądań do usługi sieciowej `FixedDepositWS`

W listingu 12.16 została przedstawiona metoda `openFixedDeposit()` klasy `FixedDepositWSClient`, która wykonuje żądania HTTP POST do usługi sieciowej `FixedDepositWS` w celu założenia nowej lokaty.

Listing 12.16. Przykład użycia metody `openFixedDeposit()` klasy `FixedDepositWSClient`

Projekt: `ch12-webservice-client`

Położenie pliku: `src/main/java/sample/spring/chapter12`

```

package sample.spring.chapter12;

import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
.....
public class FixedDepositWSClient {
    .....
    private static void openFixedDeposit(RestTemplate restTemplate) {
        FixedDepositDetails fdd = new FixedDepositDetails();
        fdd.setDepositAmount("9999");
        .....
        ResponseEntity<FixedDepositDetails> responseEntity = restTemplate
            .postForEntity("http://localhost:8080/ch12-webservice/fixedDeposits",
                fdd, FixedDepositDetails.class);

        FixedDepositDetails fixedDepositDetails = responseEntity.getBody();
        .....
    }
}
  
```

Metoda `openFixedDeposit()` klasy `FixedDepositWSClient` wysyła do usługi sieciowej `FixedDepositWS` informacje szczegółowe o lokacie, która ma zostać założona. Jeżeli założenie lokaty zakończy się pomyślnie, usługa sieciowa `FixedDepositWS` zwraca nowo utworzony obiekt `FixedDepositDetails` zawierający przypisany do niego unikalny identyfikator. W powyższym listingu widać, że metoda

`postForEntity()` klasy `RestTemplate` akceptuje adres URL usługi sieciowej, obiekt przeznaczony do przekazania (w omawianym przypadku jest to obiekt `FixedDepositDetails`) oraz typ odpowiedzi HTTP (tutaj jest to `FixedDepositDetails.class`). Wykonanie żądania HTTP POST do adresu URL `http://localhost:8080/ch12-webservice/fixedDeposits` powoduje wywołanie metody `openFixedDeposit()` klasy `FixedDepositController` (patrz: listing 12.9 lub klasa `FixedDepositController` w projekcie `ch12-webservice`).

Metoda `openFixedDeposit()` klasy `FixedDepositController` przeprowadza weryfikację informacji szczegółowych o lokacie, zanim podejmie próbę jej założenia. Egzemplarz `FixedDepositDetails` ➔ `Validator` jest odpowiedzialny za weryfikację szczegółów lokaty. Jeżeli wartość lokaty jest mniejsza niż 1000 złotych, czas jej trwania krótszy niż 12 miesięcy lub podano niepoprawny adres e-mail, wtedy metoda `openFixedDeposit()` zgłasza wyjątek. Listing 12.17 przedstawia metody `openFixedDeposit()` i `handleException()` kontrolera `FixedDepositController`.

Listing 12.17. Metody `openFixedDeposit()` i `handleException()` kontrolera `FixedDepositController`

Projekt: `ch12-webservice`

Położenie pliku: `src/main/java/sample/spring/chapter12/web`

```
package sample.spring.chapter12.web;

import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ExceptionHandler;
import sample.spring.chapter12.exception.ValidationException;
.....
@Controller
@RequestMapping(value = "/fixedDeposits")
public class FixedDepositController {
    .....
    @RequestMapping(method = RequestMethod.POST)
    public ResponseEntity<FixedDepositDetails> openFixedDeposit(
        @RequestBody FixedDepositDetails fixedDepositDetails, BindingResult
        bindingResult) {

        new FixedDepositDetailsValidator().validate(fixedDepositDetails, bindingResult);

        if (bindingResult.hasErrors()) {
            throw new ValidationException("Validation errors occurred");
        } else {
            fixedDepositService.saveFixedDeposit(fixedDepositDetails);
            .....
        }

        @ExceptionHandler(ValidationException.class)
        @ResponseBody
        @ResponseStatus(value = HttpStatus.BAD_REQUEST)
        public String handleException(Exception ex) {
            return ex.getMessage();
        }
    }
    .....
}
```

W powyższym listingu widać, że metoda `openFixedDeposit()` zgłasza wyjątek `ValidationException`, jeżeli weryfikacja informacji szczegółowych o lokacie zakończy się niepowodzeniem. Ponieważ metoda `handleException()` jest oznaczona adnotacją `@ExceptionHandler(ValidationException.class)`, wyjątek `ValidationException` zgłoszony przez metodę `openFixedDeposit()` zostaje obsługowany przez metodę `handleException()`. Adnotacje `@ResponseBody` i `@ResponseStatus(value=HttpStatus.BAD_REQUEST)` określają, że komunikat wyjątku przekazany przez metodę `handleException()` będzie umieszczony w części głównej odpowiedzi, a kod stanu HTTP będzie miał przypisaną stałą `HttpStatus.BAD_REQUEST` (odpowiada kodowi stanu HTTP 400).

Metoda `openInvalidFixedDeposit()` klasy `FixedDepositWSClient` podejmuje próbę założenia nowej lokaty o wartości 100, jak przedstawiono w listingu 12.18.

Listing 12.18. Metoda `openInvalidFixedDeposit()` klasy `FixedDepositWSClient`

Projekt: *ch12-webservice-client*

Położenie pliku: *src/main/java/sample/spring/chapter12*

```
private static void openInvalidFixedDeposit(RestTemplate restTemplate) {
    FixedDepositDetails fdd = new FixedDepositDetails();
    fdd.setDepositAmount("100");
    fdd.setEmail("99@nazwa-domeny.pl");
    fdd.setTenure("12");

    ResponseEntity<FixedDepositDetails> responseEntity = restTemplate
        .postForEntity( "http://localhost:8080/ch12-webservice/fixedDeposits",
            fdd, FixedDepositDetails.class);

    FixedDepositDetails fixedDepositDetails = responseEntity.getBody();
    logger.info("Informacje szczegółowe o nowo utworzonej lokacie: "
        + fixedDepositDetails);
}
```

Metoda `openInvalidFixedDeposit()` używa klasy `RestTemplate` w celu wysłania żądania do metody `openFixedDeposit()` klasy `FixedDepositController`. Ponieważ jako wartość lokaty podano 100, metoda `openFixedDeposit()` klasy `FixedDepositController` zgłosi wyjątek `ValidationException` (patrz: listing 12.17). Metoda `handleException()` klasy `FixedDepositController` (patrz: listing 12.17) jest odpowiedzialna za obsługę wyjątku `ValidationException` i ustawienie stanu odpowiedzi HTTP na 400. Skoro kod stanu odpowiedzi otrzymany przez egzemplarz `RestTemplate` wynosi 400, obsługa odpowiedzi jest delegowana do implementacji `MyErrorHandler` (patrz: listingi 12.12 i 12.13) skonfigurowanej dla `RestTemplate`.

Klasa `RestTemplate` pozwala klientom na uzyskanie synchronicznego dostępu do usług sieciowych w stylu RESTful. Zobaczmy teraz, jak w sposób asynchroniczny uzyskać dostęp do usług sieciowych w stylu RESTful, używając do tego oferowanej przez Spring klasy `AsyncRestTemplate`.

Asynchroniczny dostęp do usług sieciowych w stylu RESTful za pomocą `AsyncRestTemplate`

Aby umożliwić klientom uzyskanie asynchronicznego dostępu do usług sieciowych w stylu RESTful, Spring oferuje klasę `AsyncRestTemplate`. Listing 12.19 pokazuje, jak klasa `AsyncRestTemplate` została skonfigurowana w pliku XML kontekstu aplikacji w projekcie *ch12-webservice-client*.

Listing 12.19. Plik *applicationContext.xml* przedstawiający konfigurację klasy *AsyncRestTemplate*

```

<beans .....>
    .....
    <bean id="errorHandler" class="sample.spring.chapter12.MyErrorHandler" />

    <bean id="asyncRestTemplate"
        class="org.springframework.web.client.AsyncRestTemplate">
        <property name="errorHandler" ref="errorHandler" />
    </bean>
</beans>

```

Jeżeli powyższy listing porównasz z przykładem przedstawionym w listingu 12.12, zauważysz, że obie klasy (*AsyncRestTemplate* i *RestTemplate*) zostały skonfigurowane w ten sam sposób i używają tego samego egzemplarza *MyErrorHandler* do obsługi błędów HTTP.

Klasa *AsyncRestTemplate* definiuje metody podobne do oferowanych przez klasę *RestTemplate*. W listingu 12.20 została przedstawiona klasa *FixedDepositWSAsyncClient*, która używa klasy *AsyncRestTemplate* w celu uzyskania dostępu do usługi sieciowej *FixedDepositWS*.

Listing 12.20. Metoda *openFixedDeposit()* klasy *FixedDepositWSAsyncClient*

Projekt: *ch12-webservice-client*

Położenie pliku: *src/main/java/sample/spring/chapter12*

```

package sample.spring.chapter12;

import org.springframework.http.HttpEntity;
import org.springframework.util.concurrent.ListenableFuture;
import org.springframework.util.concurrent.ListenableFutureCallback;
import org.springframework.web.client.AsyncRestTemplate;

public class FixedDepositWSAsyncClient {
    private static ApplicationContext context;

    public static void main(String args[]) {
        context = new ClassPathXmlApplicationContext(
            "classpath:META-INF/spring/applicationContext.xml");
        ....
        openFixedDeposit(context.getBean(AsyncRestTemplate.class));
    }

    private static void openFixedDeposit(AsyncRestTemplate restTemplate) {
        FixedDepositDetails fdd = new FixedDepositDetails();
        fdd.setDepositAmount("9999");
        ....
        HttpEntity<FixedDepositDetails> requestEntity = new
            HttpEntity<FixedDepositDetails>(fdd);

        ListenableFuture<ResponseEntity<FixedDepositDetails>> futureResponseEntity =
            restTemplate.postForEntity(
                "http://localhost:8080/ch12-webservice/fixedDeposits",
                requestEntity, FixedDepositDetails.class);

        futureResponseEntity.addCallback(new
            ListenableFutureCallback<ResponseEntity<FixedDepositDetails>>() {
                @Override

```

```

        public void onSuccess(ResponseEntity<FixedDepositDetails> entity) {
            FixedDepositDetails fixedDepositDetails = entity.getBody();
        }

        @Override
        public void onFailure(Throwable t) { }
    });
}
}

```

W powyższym listingu widać, że metoda `openFixedDeposit()` korzysta z klasy `AsyncRestTemplate` w celu wykonywania żądań do usługi sieciowej `FixedDepositWS`. Metoda `postForEntity()` klasy `AsyncRestTemplate` wykonuje żądanie HTTP POST do usługi sieciowej `FixedDepositWS` wywołując metodę `openFixedDeposit()` klasy `FixedDepositController`. Jeżeli porównasz przedstawioną w omawianym listingu metodę `postForEntity()` klasy `AsyncRestTemplate` z metodą `postForEntity()` klasy `RestTemplate` (patrz: listing 12.16), zauważysz, że metoda `postForEntity()` klasy `AsyncRestTemplate` zwraca obiekt typu `ListenableFuture` (rozszerzający interfejs `java.util.concurrent.Future`). Metoda `addCallback()` klasy `ListenableFuture` jest używana do zarejestrowania wywołania zwrotnego, które będzie wywoływane po zakończeniu zadania `ListenableFuture`. Metoda `addCallback()` klasy `ListenableFuture` akceptuje argument typu `ListenableFutureCallback` definiujący metody `onSuccess()` i `onFailure()`. Metoda `onSuccess()` jest wywoływana, gdy wykonanie zadania `ListenableFuture` zakończy się pomyślnie, natomiast metoda `onFailure()` jest wywoływana, gdy wykonanie zadania `ListenableFuture` zakończy się niepowodzeniem.

Warto zwrócić uwagę, że domyślnie `AsyncRestTemplate` używa wykonawcy `SimpleAsyncTaskExecutor` w celu asynchronicznego wykonania *każdego* żądania w nowym wątku. Istnieje możliwość przekazania `ThreadPoolTaskExecutor` do konstruktora `AsyncRestTemplate`, aby asynchronicznie wykonywać zadania za pomocą wątku pobranego z puli wątków. W podrozdziale 8.6 znajdziesz więcej informacji na temat wykonawców `SimpleAsyncTaskExecutor` i `ThreadPoolTaskExecutor`.

Zobaczmy teraz do jakich celów służy implementacja `HttpMessageConverters` w Spring Web MVC.

12.5. Użycie `HttpMessageConverter` do konwersji obiektów Javy na żądania lub odpowiedzi HTTP i na odwrót

W Springu implementacja `HttpMessageConverters` jest używana w następujących sytuacjach do przeprowadzenia konwersji:

- jeżeli argument metody został oznaczony adnotacją `@RequestBody`, Spring konwertuje część główną żądania HTTP na typ Javy argumentu metody;
- jeżeli metoda jest oznaczona adnotacją `@ResponseBody`, Spring konwertuje obiekt Javy zwrócony przez tę metodę na część główną odpowiedzi HTTP;
- jeżeli typem zwrotnym metody jest `HttpEntity` lub `ResponseEntity`, Spring konwertuje obiekt zwrócony przez tę metodę na część główną odpowiedzi HTTP;
- obiekty przekazane lub zwrócone przez metody klas `RestTemplate` i `AsyncRestTemplate` takie jak `getForEntity()`, `postForEntity()`, `exchange()` itd. są przez Spring konwertowane na żądania HTTP i odpowiedzi HTTP.

W tabeli 12.3 wymieniamy niektóre implementacje `HttpMessageConverter` standardowo udostępniane przez Spring Web MVC.

Tabela 12.3. Wybrane implementacje `HttpMessageConverter` oferowane przez Spring Web MVC

Nazwa	Opis
<code>StringHttpMessageConverter</code>	Przeprowadza konwersje na ciąg tekstowy i z ciągu tekstowego.
<code>FormHttpMessageConverter</code>	Konwertuje dane formularza na typ <code>MultiValueMap<String, String></code> i z tego typu. Ta implementacja <code>HttpMessageConverter</code> jest używana przez Spring podczas pracy z danymi formularza i w trakcie przekazywania plików.
<code>MappingJackson2HttpMessageConverter</code>	Przeprowadza konwersje na dane w formacie JSON i z tychże danych.
<code>MarshallingHttpMessageConverter</code>	Przeprowadza konwersje na dane w formacie XML i z tychże danych.

Implementacje `HttpMessageConverters` wymienione w tabeli 12.3 są automatycznie rejestrowane w kontenerze Springa za pomocą elementu `<annotation-driven>` schematu `mvc`. Aby wyświetlić pełną listę implementacji `HttpMessageConverters` domyślnie rejestrowanych przez element `<annotation-driven>`, zajrzyj do dokumentacji Springa.

Przechodzimy teraz do omówienia adnotacji `@PathVariable` i `@MatrixVariable`, które jeszcze bardziej ułatwiają tworzenie usług sieciowych w stylu RESTful za pomocą frameworku Spring Web MVC.

12.6. Adnotacje `@PathVariable` i `@MatrixVariable`

Zamiast określać rzeczywisty adres URI adnotacja `@RequestMapping` może wskazywać *szablon adresu URI* pozwalający na uzyskanie dostępu do konkretnych fragmentów adresu URI żądania. Szablon adresu URI zawiera *nazwy zmiennych* (podane w nawiasach klamrowych), których wartości są pobierane z rzeczywistych adresów URI żądania. Na przykład szablon URI w postaci `http://www.somebank.com/fd/{fixeddeposit}` zawiera zmienną o nazwie `fixeddeposit`. Jeżeli adresem URI rzeczywistego żądania jest `http://www.somebank.com/fd/123`, to wartością `{fixeddeposit}` szablonu adresu URI jest 123.

`@PathVariable` to adnotacja na poziomie argumentu metody używana przez metody oznaczone adnotacją `@RequestMapping` w celu przypisania wartości *zmiennej szablonu adresu URI* do argumentu metody.

IMPORT Rozdział12/ch12-webservice-uritemplates i rozdział 12/ch12-webservice-client-uritemplates (Projekt `ch12-webservice-uritemplates` jest wariantem projektu `ch12-webservice` i przedstawia implementację usługi sieciowej `FixedDepositWS` używającą adnotacji `@PathVariable`. Z kolei projekt `ch12-webservice-client-uritemplates` to wariant projektu `ch12-webservice-client` uzyskującego dostęp do usługi sieciowej `FixedDepositWS` przedstawianej przez projekt `ch12-webservice-uritemplates`.)

W listingu 12.21 przedstawiamy przykład użycia adnotacji `@PathVariable` w klasie `FixedDepositController` projektu `ch12-webservice-uritemplates`.

Listing 12.21. Przykład użycia adnotacji `@PathVariable` w `FixedDepositController`

Projekt: `ch12-webservice-uritemplates`

Położenie pliku: `src/main/java/sample/spring/chapter12/web`

```
package sample.spring.chapter12.web;

import org.springframework.web.bind.annotation.PathVariable;
.....
@Controller
public class FixedDepositController {
    .....
    @RequestMapping(value="/fixedDeposits/{fixedDepositId}", method = RequestMethod.GET)
    public ResponseEntity<FixedDepositDetails> getFixedDeposit(
        @PathVariable("fixedDepositId") int id) {
        return new ResponseEntity<FixedDepositDetails>(
            fixedDepositService.getFixedDeposit(id), HttpStatus.OK);
    }
    .....
}
```

Zamiast podawać rzeczywisty adres URI adnotacja `@RequestMapping` w powyższym listingu określa szablon adresu URI w postaci `/fixedDeposits/{fixedDepositId}`. Jeżeli teraz adres URI przychodzącego żądania ma postać `/fixedDeposits/1`, to wartość zmiennej `fixedDepositId` szablonu adresu URI wynosi 1. Ponieważ adnotacja `@PathVariable` wskazuje `fixedDepositId` jako nazwę zmiennej szablonu adresu URI, wartość 1 jest przypisywana argumentowi `id` metody `getFixedDeposit()`.

Jeżeli w szablonie adresu URI zdefiniowano więcej zmiennych, wówczas metoda oznaczona adnotacją `@RequestMapping` może zawierać więcej argumentów oznaczonych adnotacją `@PathVariable`, jak przedstawia listing 12.22.

Listing 12.22. Przykład użycia wielu zmiennych w szablonie adresu URI

```
@Controller
public class SomeController {
    .....
    @RequestMapping(value="/users/{userId}/bankstatements/{statementId}", ....)
    public void getBankStatementForUser(
        @PathVariable("userId") String user,
        @PathVariable("statementId") String statement) {
        .....
    }
}
```

W powyższym listingu szablon adresu URI definiuje zmienne `userId` i `statementId`. Jeżeli adres URI żądania przychodzącego ma postać `/users/me/bankstatements/123`, wówczas wartość `me` zostanie przypisana argumentowi `user`, natomiast wartość `123` będzie przypisana argumentowi `statement`.

Jeżeli argumentowi metody chcesz przypisać wszystkie zmienne szablonu adresu URI i ich wartości, możesz użyć adnotacji `@PathVariable` w typie argumentu `Map<String, String>`, jak przedstawia listing 12.23.

Listing 12.23. Uzyskanie dostępu do wszystkich zmiennych szablonu adresu URI i ich wartości

```
@Controller
public class SomeController {
    ....
    @RequestMapping(value="/users/{userId}/bankstatements/{statementId}", .....)
    public void getBankStatementForUser(
        @PathVariable Map<String, String> allVariables) {
        ....
    }
}
```

W powyższym listingu zmienne szablonu adresu URI (`userId` i `statementId`) oraz ich wartości (`me` i `123`) są przypisywane argumentowi metody `allVariables`.

Warto pamiętać także o możliwości określenia szablonu adresu URI za pomocą adnotacji `@RequestMapping` na poziomie klasy, jak przedstawia listing 12.24.

Listing 12.24. Szablon adresu URI wskazany na poziomach klasy i metody za pomocą adnotacji `@RequestMapping`

```
@RequestMapping annotations

@Controller
@RequestMapping(value="/service/{serviceId}", .....)
public class SomeController {
    ....
    @RequestMapping(value="/users/{userId}/bankstatements/{statementId}", .....)
    public void getBankStatementForUser(@PathVariable Map<String, String> allVariables)
    {
        ....
    }
}
```

W powyższym listingu szablon metody URI `/service/{serviceId}` jest określony za pomocą adnotacji `@RequestMapping` na poziomie klasy, natomiast `/users/{userId}/bankstatements/{statementId}` — za pomocą adnotacji `@RequestMapping` na poziomie metody. Jeżeli adres URI żądania będzie miał postać `/service/bankingService/users/me/bankstatements/123`, wówczas argument `allVariables` będzie zawierać informacje szczegółowe o zmiennych `serviceId`, `userId` i `statementId` szablonu adresu URI.

Zdarzają się sytuacje, w których potrzebna jest dokładniejsza kontrola nad danymi wyodrębnianymi z adresu URI żądania. W takich przypadkach można zastosować wyrażenia regularne w szablonach adresów URI. W listingu 12.25 przedstawiamy przykład użycia wyrażeń regularnych w celu wyodrębnienia wartości `123.json` z adresu URI żądania w postaci `/statements/123.json`.

Listing 12.25. Przykład użycia wyrażeń regularnych w szablonie adresu URI

```
@Controller
public class SomeController {
    ....
    @RequestMapping(value="/bankstatement/{statementId:[\\d\\d\\d\\d]}"
        .{responseType:[a-z]}", ..)
    public void getBankStatementForUser(@PathVariable ("statementId") String statement,
```

```

    @PathVariable("responseType") String responseTypeExtension) {
        ....
    }
}

```

Wyrażenie regularne w szablonie adresu URI jest określane w następującym formacie: {nazwa-zmiennej:wyrażenie-regularne}. Jeżeli adres URI żądania ma postać `/statements/123.json`, wówczas zmiennej `statementId` zostaje przypisana wartość 123, a zmiennej `responseType` wartość `json`.



Istnieje także możliwość użycia w szablonach adresów URI wzorców w stylu Ant. Na przykład można zastosować wzorce takie jak `/myUrl/*/myId` i `/myUrl/**/myId`.

Dotychczas w tym podrozdziale przedstawiliśmy przykłady użycia adnotacji `@PathVariable` do selektywnego wyodrębniania informacji z adresu URI żądania. Przechodzimy więc do adnotacji `@MatrixVariable`, która jest używana do wyodrębniania *par nazwa – wartość* z segmentów adresu URI.

Zmienne w postaci macierzy występują w adresie URI żądania w postaci par nazwa – wartość, a ich wartości można przypisywać argumentom metody. Na przykład w adresie URI żądania `/bankstatement/123;responseType=json` zmienna `responseType` przedstawia zmienną w postaci macierzy, której wartość wynosi `json`.



Zwróć uwagę, że domyślnie Spring usuwa zmienne macierzy z adresu URL. Aby mieć gwarancję, że zmienne macierzy nie zostaną usunięte, atrybutowi `enable-matrix-variables` elementu `<annotation-driven>` schematu `mvc` przypisz wartość `true`. Podczas użycia zmiennych macierzy segmenty ścieżki dostępu zawierające zmienne macierzy muszą być przedstawiane przez zmienne szablonu adresu URI.

W listingu 12.26 przedstawiamy przykład użycia adnotacji `@MatrixVariable`.

Listing 12.26. Adnotacja `@MatrixVariable`

```

@Controller
public class SomeController {
    ....
    @RequestMapping(value="/bankstatement/{statementId}", ..)
    public void getBankStatementForUser(@PathVariable("statementId") String statement,
        @MatrixVariable("responseType") String responseTypeExtension) {
        ....
    }
}

```

Jeżeli w powyższym listingu adres URI żądania będzie miał postać `/bankstatement/123;responseType=json`, to wartość `json` zostanie przypisana argumentowi `responseTypeExtension`. Omawiany listing pokazuje również sytuację, w której adnotacje `@PathVariable` i `@MatrixVariable` są używane do pobrania informacji z adresu URI żądania.

Zmienne macierzy mogą występować w dowolnym segmencie adresu URI żądania, dlatego należy wskazać segment, z którego powinna być pobrana zmienna macierzy. W listingu 12.27 przedstawiamy sytuację, w której dwie zmienne macierzy o tej samej nazwie występują w różnych segmentach adresu URI.

Listing 12.27. Adnotacja `@MatrixVariable` przedstawiająca przykład użycia wielu zmiennych macierzy o takiej samej nazwie

```
@Controller
public class SomeController {
    ....
    @RequestMapping(value="/bankstatement/{statementId}/user/{userId}", ..)
    public void getBankStatementForUser(
        @MatrixVariable(value = "id", pathVar = "statementId") int someId,
        @MatrixVariable(value = "id", pathVar = "userId") int someOtherId) {
        ....
    }
}
```

Atrybut `pathVar` adnotacji `@MatrixVariable` określa nazwę zmiennej szablonu adresu URI zawierającej zmienną macierzy. Dlatego jeśli adres URI żądania ma postać `/bankstatement/123;id=555/user/me;id=777`, wówczas wartość 555 zostanie przypisana argumentowi `someId`, natomiast wartość 777 będzie przypisana argumentowi `someOtherId`.

Podobnie jak w przypadku adnotacji `@PathVariable`, istnieje możliwość oznaczenia argumentu metody typu `Map<String, String>` adnotacją `@MatrixVariable` w celu przypisania wszystkich zmiennych macierzy argumentowi metody. Jednak w przeciwieństwie do adnotacji `@PathVariable` adnotacja `@MatrixVariable` pozwala na określenie wartości domyślnej dla zmiennej macierzy, używając do tego atrybutu `defaultValue`. Ponadto atrybutowi `required` adnotacji `@MatrixVariable` można przypisać wartość `false` wskazującą, że zmienna macierzy jest *opcjonalna*. Domyślnie wartością atrybutu `required` jest `true`. Jeżeli atrybut `required` będzie miał przypisaną wartość `true`, a zmienna macierzy nie zostanie znaleziona w żądaniu, nastąpi zgłoszenie wyjątku.

12.7. Podsumowanie

Z tego rozdziału dowiedziałeś się, jak opracować usługę sieciową w stylu RESTful i uzyskać do niej dostęp. Przeanalizowaliśmy użycie szablonów URI wraz z adnotacjami `@PathVariable` i `@MatrixVariable` w celu uzyskania dostępu do informacji z adresu URI żądania. Ponadto zobaczyłeś, jak dostęp do usługi sieciowej w stylu RESTful uzyskać synchronicznie za pomocą klasy `RestTemplate` i asynchronicznie za pomocą klasy `AsyncRestTemplate`.

Rozdział 13. Więcej o Spring Web MVC: internacjonalizacja, przesyłanie plików i przetwarzanie żądań asynchronicznych

13.1. Wprowadzenie

We poprzednich rozdziałach zobaczyłeś, jak Spring Web MVC ułatwia tworzenie aplikacji sieciowych i usług sieciowych w stylu RESTful. W tym rozdziale zajmiemy się bardziej zaawansowanymi funkcjami oferowanymi przez Spring Web MVC, które mogą być wymagane w aplikacjach sieciowych. W szczególności skoncentrujemy się na:

- przetwarzaniu żądań za pomocą *procedur obsługi interceptorów*;
- internacjonalizacji aplikacji Spring Web MVC;
- *asynchronicznym* przetwarzaniu żądań;
- przeprowadzaniu konwersji typu i formatowaniu;
- przekazywaniu plików.

IMPORT **Rozdział13/ch13-bankapp**. (Ten projekt stanowi wariant projektu *ch10-bankapp* i pokazuje, jak zastosować internacjonalizację w aplikacji sieciowej MyBank, a także jak używać procedur obsługi interceptorów).

Rozpoczynamy od przetwarzania żądań za pomocą procedur obsługi interceptorów.

13.2. Przetwarzanie żądań za pomocą procedur obsługi interceptorów

Procedury obsługi interceptorów pozwalają na przetwarzanie żądań przed ich wykonaniem i po. Koncepcja procedur obsługi interceptorów jest podobna do serwletów filtrów. Procedura obsługi interceptorów implementuje oferowany przez Spring interfejs `HandlerInterceptor`, a ponadto zawiera przeznaczoną do przetwarzania żądania przed jego wykonaniem i po wykonaniu logikę wymaganą przez wiele kontrolerów. Procedurę tę można stosować na przykład do rejestrowania danych, sprawdzania zabezpieczeń i zmiany ustawień językowych.

Zobaczmy, jak zaimplementować i skonfigurować procedurę obsługi interceptorów.

Implementacja i konfiguracja procedury obsługi interceptorów

Procedurę obsługi interceptorów można utworzyć przez implementację interfejsu `HandlerInterceptor`, który definiuje następujące metody:

- `preHandle()` — wykonywana, *zanim* kontroler przetworzy żądanie. Jeżeli wartością zwrótną metody `preHandle()` będzie `true`, wówczas Spring wywoła kontroler w celu przetworzenia żądania. Z kolei wartość zwrótna `false` oznacza, że kontroler *nie* będzie wywołany.

- `postHandle()` — wywoływana *po* przetworzeniu żądania przez kontroler, ale jeszcze *przed* wygenerowaniem widoku przez `DispatcherServlet`.
- `afterCompletion()` — wywoływana *po* zakończeniu przetwarzania żądania (to znaczny po wygenerowaniu widoku przez `DispatcherServlet`) i służy do przeprowadzania wszelkich operacji, jeśli są potrzebne.

W listingu 13.1 została przedstawiona klasa `MyRequestHandlerInterceptor` w projekcie *ch13-bankapp* implementująca interfejs `HandlerInterceptor`.

Listing 13.1. Klasa `MyRequestHandlerInterceptor`

Projekt: *ch13-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter13/web*

```
package sample.spring.chapter13.web;

import org.springframework.web.servlet.HandlerInterceptor;
.....
public class MyRequestHandlerInterceptor implements HandlerInterceptor {
    .....
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
        Object handler) throws Exception {
        logger.info("Metoda HTTP --> " + request.getMethod());
        Enumeration<String> requestNames = request.getParameterNames();
        .....
        return true;
    }

    public void postHandle(HttpServletRequest request, HttpServletResponse response,
        Object handler, ModelAndView modelAndView) throws Exception {
        logger.info("Kod stanu --> " + response.getStatus());
    }

    public void afterCompletion(HttpServletRequest request, HttpServletResponse
        response, Object handler, Exception ex) throws Exception {
        logger.info("Zakończono przetwarzanie żądania.");
    }
}
```

W powyższym listingu metoda `preHandle()` analizuje każde żądanie przychodzące i wyświetla informacje o metodzie HTTP powiązanej z tym żądaniem, a także o zawartych w nim parametrach żądania. Wartością zwrótną metody `preHandle()` jest `true`, co oznacza, że żądanie zostanie przetworzone przez kontroler. Następnie metoda `postHandle()` wyświetla kod stanu odpowiedzi HTTP. Na koniec metoda `afterCompletion()` wyświetla komunikat informujący o zakończonym pomyślnie przetworzeniu żądania.



Zamiast bezpośrednio implementować interfejs `HandlerInterceptor` można rozszerzyć klasę *abstrakcyjną* `HandlerInterceptorAdapter`, która dostarcza puste implementacje metod `postHandle()` i `afterCompletion()`, natomiast metoda `preHandle()` jest zdefiniowana do zwrócenia wartości `true`.

W listingu 13.2 przedstawiamy sposób konfiguracji procedury obsługi interceptorów w pliku XML kontekstu aplikacji sieciowej.

Listing 13.2. *Przykład konfiguracji procedury obsługi interceptorów*

Projekt: ch13-bankapp

Położenie pliku: src/main/webapp/WEB-INF/spring/bankapp-config.xml

```
<beans .....xmlns:mvc="http://www.springframework.org/schema/mvc".....>

    <mvc:annotation-driven />
    <mvc:interceptors>
        .....
        <bean class="sample.spring.chapter13.web.MyRequestHandlerInterceptor" />
    </mvc:interceptors>
</beans>
```

W powyższym listingu widać, że element `<interceptors>` schematu `mvc` jest używany do konfiguracji procedur obsługi interceptorów. Element `<interceptors>` może zawierać następujące elementy.

- Element `<bean>` oferowanego przez Spring schematu `beans`. Wskazuje on komponent implementujący interfejs `HandlerInterceptor`. Procedura obsługi interceptorów zdefiniowana za pomocą elementu `<bean>` jest stosowana we wszystkich żądaniach.
- Element `<ref>` oferowanego przez Spring schematu `beans`. Wskazuje on komponent implementujący interfejs `HandlerInterceptor`. Procedura obsługi interceptorów zdefiniowana za pomocą elementu `<ref>` jest stosowana we wszystkich żądaniach.
- Element `<interceptor>` oferowanego przez Spring schematu `mvc`. Wskazuje on komponent implementujący interfejs `HandlerInterceptor` i adres URI, do którego ma zastosowanie interfejs `HandlerInterceptor`.

W listingu 13.3 przedstawiamy sytuację, w której `MyRequestHandlerInterceptor` jest mapowany na adres URI `/audit/**`.

Listing 13.3. *Przykład użycia elementu `<mvc:interceptor>`*

```
<beans .....xmlns:mvc="http://www.springframework.org/schema/mvc".....>

    <mvc:annotation-driven />
    <mvc:interceptors>
        <mvc:interceptor>
            <mvc:mapping path="/audit/**"/>
            <bean class="sample.spring.chapter13.web.MyRequestHandlerInterceptor" />
        </mvc:interceptor>
    </mvc:interceptors>
</beans>
```

W powyższym listingu element `<interceptor>` oferowanego przez Spring schematu `mvc` został użyty do mapowania `MyRequestHandlerInterceptor` na wzorec adresu URI `/audit/**`. Element `<mapping>` schematu `mvc` określa wzorec adresu URI żądania, do którego będzie stosowana procedura obsługi interceptorów wymieniona w elemencie `<bean>`.

Przechodzimy teraz do tematu internacjonalizacji aplikacji Spring Web MVC.

13.3. Internacjonalizacja za pomocą paczek zasobów

Zanim przejdziemy do szczegółów związanych z internacjonalizacją aplikacji Spring Web MVC, zapoznamy się z wymaganiami aplikacji sieciowej MyBank w tym zakresie.

Wymagania aplikacji sieciowej MyBank

Przyjmujemy założenie, że aplikacja sieciowa MyBank ma być dostępna w językach polskim (pl_PL) i angielskim (en_US). Na rysunku 13.1 pokazujemy jedną ze stron aplikacji sieciowej MyBank wyświetloną w języku angielskim.

Rysunek 13.1.

Strona internetowa
wyświetlająca listę lokat
w języku angielskim
(użytkownik może
wybrać dowolną
z dostępnych lokalizacji)

Fixed deposit list

ID	Deposit amount	Tenure	Email	Action
1	10000	24	a1email@nazwa-domeny.pl	Close Edit
2	20000	36	a2email@nazwa-domeny.pl	Close Edit
3	30000	36	a3email@nazwa-domeny.pl	Close Edit
4	50000	36	a4email@nazwa-domeny.pl	Close Edit
5	15000	36	a5email@nazwa-domeny.pl	Close Edit

[Create new Fixed Deposit](#)

Język: [English\(US\)](#) | [Polski](#) | [English\(Canada\)](#)
Locale: en_US

Na rysunku 13.1 widać, że użytkownik może wybrać jeden z następujących języków: English (US), Polski lub English (Canada). Jeżeli użytkownik wybierze opcję *Polski*, strony internetowe będą wyświetlane z użyciem ustawień pl_PL. Wybór opcji English (US) oznacza wyświetlanie stron internetowych z użyciem ustawień en_US. Natomiast po wybraniu opcji English (Canada) strony internetowe są wyświetlane z użyciem ustawień en_CA.

Przechodzimy teraz do sposobów spełniania wymagań aplikacji sieciowej MyBank w zakresie internacjonalizacji i lokalizacji.

Internacjonalizacja i lokalizacja aplikacji sieciowej MyBank

W aplikacji sieciowej Spring Web MVC serwlet `DispatcherServlet` używa klasy `LocaleResolver` w celu automatycznego ustalania komunikatów do wyświetlenia na podstawie ustawień językowych użytkownika. Aby zapewnić obsługę internacjonalizacji, należy w pliku XML kontekstu aplikacji sieciowej przeprowadzić konfigurację następujących komponentów:

- `LocaleResolver` — ustala bieżące ustawienia językowe użytkownika;
- `MessageSource` — na podstawie ustawień językowych użytkownika ustala, które komunikaty z paczek zasobów powinny być wyświetlane;

- `LocaleChangeInterceptor` — umożliwia zmianę bieżących ustawień językowych dla każdego żądania na podstawie konfigurowalnego parametru żądania.

W listingu 13.4 przedstawiamy konfigurację komponentów `LocaleResolver`, `LocaleChangeInterceptor` i `MessageSource` przeprowadzoną w pliku XML kontekstu aplikacji sieciowej znajdującego się w projekcie *ch13-bankapp*.

Listing 13.4. Zawartość pliku *bankapp-config.xml*

Projekt: *ch13-bankapp*

Położenie pliku: *src/main/webapp/WEB-INF/spring*

```
<beans .....>
  <bean class="org.springframework.web.servlet.i18n.CookieLocaleResolver"
    id="localeResolver">
    <property name="cookieName" value="mylocale" />
  </bean>

  <bean
    class="org.springframework.context.support.ReloadableResourceBundleMessageSource"
    id="messageSource">
    <property name="basenames" value="WEB-INF/i18n/messages" />
  </bean>

  <mvc:interceptors>
    .....
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
      <property name="paramName" value="lang" />
    </bean>
  </mvc:interceptors>
  .....
</beans>
```

W powyższym listingu obiekt `CookieLocaleResolver` (implementacja interfejsu `LocaleResolver`) został skonfigurowany do obsługi ustawień językowych. Jeżeli aplikacja sieciowa przechowuje w cookie informacje o ustawieniach językowych, wówczas egzemplarz `CookieLocaleResolver` zostanie użyty do określenia języka. Właściwość `cookieName` obiektu `CookieLocaleResolver` wskazuje nazwę cookie zawierającego informacje o ustawieniach językowych. Jeżeli cookie nie zostanie znalezione w żądaniu, `CookieLocaleResolver` określi ustawienia językowe na podstawie ustawień domyślnych (skonfigurowanych za pomocą właściwości `defaultLocale` obiektu `CookieLocaleResolver`) lub przez analizę wartości nagłówka `Accept-Language` w żądaniu. Ponadto Spring oferuje wbudowane implementacje `LocaleResolver`, które można wykorzystać: `AcceptHeaderLocaleResolver` (zwraca ustawienia językowe wskazywane przez nagłówek żądania `Accept-Language`), `SessionLocaleResolver` (zwraca ustawienia językowe przechowywane w obiekcie `HttpSession` użytkownika) i `FixedLocaleResolver` (zawsze zwraca domyślne ustawienia językowe).

Oprócz ustawień językowych użytkownika warto ustalić także jego strefę czasową, aby dokonać odpowiednich zmian w wyświetlanej dacie i godzinie. Interfejs `LocaleContextResolver` (wprowadzony w Springu 4.0) nie tylko dostarcza informacje o ustawieniach językowych, ale również o jego strefie czasowej. `CookieLocaleResolver`, `SessionLocaleResolver` i `FixedLocaleResolver` implementują interfejs `LocaleContextResolver`. Dlatego jeśli używasz dowolnego z wymienionych obiektów, możesz

w kontrolerze pobrać informacje o strefie czasowej użytkownika — wystarczy wywołanie metody `getTimeZone()` klasy `LocaleContextHolder` (lub klasy `RequestContextUtils`). Jeżeli w kontrolerze chcesz tylko pobrać informacje o ustawieniach językowych, użyj metody `getLocale()` klasy `LocaleContextHolder` (lub klasy `RequestContextUtils`).

Spring dostarcza klasę `LocaleChangeInterceptor` (implementacja `HandlerInterceptor`) używającą konfigurowalnego parametru żądania (określony przez właściwość `paramName`) do zmiany ustawień językowych w każdym żądaniu. W listingu 13.4 właściwość `paramName` ma przypisaną wartość `lang`. Klasa `LocaleResolver` definiuje metodę `setLocale()` używaną przez `LocaleChangeInterceptor` do zmiany bieżących ustawień językowych. Jeżeli nie chcesz korzystać z `LocaleChangeInterceptor`, zmianę ustawień językowych użytkownika możesz przeprowadzić w kontrolerze za pomocą wywołania metody `setLocale()` klasy `LocaleContextHolder` (lub klasy `RequestContextUtils`).

Po ustaleniu ustawień językowych użytkownika Spring używa skonfigurowanej implementacji `MessageSource` do określenia komunikatów, które powinny być wyświetlane. Spring dostarcza następujące wbudowane implementacje interfejsu `MessageSource`:

- `ResourceBundleMessageSource` — implementacja `MessageSource`, która uzyskuje dostęp do paczek zasobów, używając podanych *nazw bazowych*;
- `ReloadableResourceBundleMessageSource` — implementacja podobna do `ResourceBundleMessageSource`, ale obsługuje *odświeżanie* paczek zasobów.

W listingu 13.4 widać, że aplikacja sieciowa `MyBank` używa `ReloadableResourceBundleMessageSource`. Właściwość `basenames` ma przypisaną wartość `WEB-INF/i18n/messages`, co oznacza, że `ReloadableResourceBundleMessageSource` szuka w katalogu `WEB-INF/i18n` paczek z zasobami o nazwie `messages`. Jeżeli ustawienia językowe użytkownika zostaną określone jako `en_US`, wówczas egzemplarz `ReloadableResourceBundleMessageSource` będzie pobierał komunikaty z pliku `messages_en_US.properties`.

Jeżeli spojrzysz na katalog `/src/main/webapp/WEB-INF/i18n` projektu `ch13-bankapp`, zauważysz tam istnienie następujących plików: `messages.properties`, `messages_en_US.properties` i `messages_pl_PL.properties`. Plik `messages_pl_PL.properties` zawiera komunikaty i etykiety dla ustawień językowych `pl_PL`, plik `messages_en_US.properties` zawiera komunikaty i etykiety dla ustawień językowych `en_US`, natomiast plik `messages.properties` zawiera komunikaty i etykiety wyświetlane w przypadku, gdy *nie* zostanie znaleziona paczka z zasobami dla wybranego języka. Ponieważ projekt nie zawiera pliku `messages_en_CA.properties` przeznaczonego dla ustawień językowych `en_CA`, wybór opcji `English(Canada)` pokazanej na rysunku 13.1 powoduje wyświetlanie komunikatów pobranych z pliku `messages.properties`.

Na rysunku 13.1 widziałeś, że zmiana języka aplikacji sieciowej `MyBank` następuje przez wybór opcji `English(US)`, `English(Canada)` i `Polski`. Wcześniej zobaczyłeś, że egzemplarz `LocaleChangeInterceptor` może zmienić ustawienia językowe aplikacji sieciowej `MyBank` na podstawie informacji dostarczanych przez parametr żądania o nazwie `lang`. Dla ułatwienia operacji zmiany ustawień językowych parametr żądania `lang` jest dołączony do łączy wyświetlanych przez opcje `English(US)`, `English(Canada)` i `Polski`, jak przedstawia listing 13.5.

Listing 13.5. Zawartość pliku *fixedDepositList.jsp*Projekt: *ch13-bankapp*Położenie pliku: *src/main/webapp/WEB-INF/jsp*

```

<b>Język:</b>
<a href="{pageContext.request.contextPath}/fixedDeposit/list?lang=en_US">English(US)</a> |
<a href="{pageContext.request.contextPath}/fixedDeposit/list?lang=pl_PL">Polski</a> |
<a href="{pageContext.request.contextPath}/fixedDeposit/list?lang=en_CA">English(Canada)</a>

```

Zobaczmy teraz, jak można asynchronicznie przetwarzać żądania w aplikacjach Spring Web MVC.

13.4. Asynchroniczne przetwarzanie żądań

Metoda oznaczona adnotacją `@RequestMapping` zwracająca obiekt `java.util.concurrent.Callable` lub udostępniany przez Spring obiekt `DeferredResult` *asynchronicznie* przetwarza żądania sieciowe. Jeżeli metoda oznaczona adnotacją `@RequestMapping` zwróci obiekt `Callable`, Spring Web MVC zajmuje się przetwarzaniem tego obiektu w wątku aplikacji (a *nie* w wątku kontenera `Servlet`), aby wygenerować wynik. Gdy metoda oznaczona adnotacją `@RequestMapping` zwraca obiekt `DeferredResult`, wówczas aplikacja jest odpowiedzialna za przetworzenie tego obiektu w wątku aplikacji (a *nie* w wątku kontenera `Servlet`), aby wygenerować wynik. Zanim zagłębimy się w szczegóły związane z przetwarzaniem wartości zwrotnych w postaci obiektów `Callable` i `DeferredResult`, zobaczmy, jak skonfigurować aplikację Spring Web MVC, aby zapewnić jej możliwość asynchronicznego przetwarzania żądań.

IMPORT **Rozdział13/ch13-async-bankapp.** (Ten projekt jest wariantem projektu *ch10-bankapp* i asynchronicznie przetwarza żądania. Oznaczona adnotacją `@RequestMapping` metoda zdefiniowana w klasie `FixedDepositController` zwraca obiekt `Callable`. Aby zobaczyć asynchroniczne przetwarzanie żądań w działaniu, należy wdrożyć i uruchomić projekt *ch13-async-bankapp*).

Konfiguracja asynchronicznego przetwarzania żądań

Ponieważ asynchroniczne przetwarzanie żądań jest w Spring Web MVC oparte na `Servlet`ie 3, plik *web.xml* musi odwoływać się do schematu XML `Servlet` 3. Ponadto element `<async-supported>` musi być dodany do definicji `DispatcherServlet` w pliku *web.xml*, aby wskazać włączoną obsługę asynchronicznego przetwarzania żądań. Listing 13.6 przedstawia zawartość pliku *web.xml* w projekcie *ch13-async-bankapp*.

Listing 13.6. Zawartość pliku *web.xml* przedstawiająca konfigurację asynchronicznego przetwarzania żądańProjekt: *ch13-async-bankapp*Położenie pliku: *src/main/webapp/WEB-INF*

```

<web-app .....
  xsi:schemaLocation="java.sun.com/xml/ns/javaee
    java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
  .....
  <servlet>
    <servlet-name>bankapp</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    .....

```

```

        <async-supported>true</async-supported>
    </servlet>
    ....
</web-app>

```

W powyższym listingu widać, że serwlet bankapp został skonfigurowany do obsługi asynchronicznego przetwarzania żądań. Po dodaniu przedstawionej konfiguracji serwlet bankapp może już asynchronicznie przetwarzać żądania sieciowe.

Zwrot obiektu Callable przez metodę oznaczoną adnotacją @RequestMapping

W listingu 13.7 została przedstawiona klasa FixedDepositController, w której metoda oznaczona adnotacją @RequestMapping zwraca obiekt Callable.

Listing 13.7. Klasa FixedDepositController, w której metoda oznaczona adnotacją @RequestMapping zwraca obiekt Callable

Projekt: ch13-async-bankapp

Położenie pliku: src/main/java/sample/spring/chapter13/web

```

package sample.spring.chapter13.web;

import java.util.concurrent.Callable;
....
public class FixedDepositController {
    ....
    @RequestMapping(value = "/list", method = RequestMethod.GET)
    public Callable<ModelAndView> listFixedDeposits() {
        return new Callable<ModelAndView>() {

            @Override
            public ModelAndView call() throws Exception {
                Thread.sleep(5000);
                Map<String, List<FixedDepositDetails>> modelData =
                    new HashMap<String, List<FixedDepositDetails>>();
                modelData.put("fdList", fixedDepositService.getFixedDeposits());
                return new ModelAndView("fixedDepositList", modelData);
            }
        };
    }
    ....
}

```

W powyższym listingu widać, że wartością zwrótną metody listFixedDeposits() jest obiekt Callable<T>, gdzie T wskazuje typ wyniku, który będzie generowany asynchronicznie. Metoda call() obiektu Callable zawiera logikę niezbędną do asynchronicznego wykonania w celu wygenerowania wyniku. Metoda call() przedstawiona w omawianym listingu wywołuje metodę getFixedDeposits() klasy FixedDepositService i zwraca obiekt ModelAndView zawierający informacje o modelu i widoku. Metoda Thread.sleep() jest wywoływana na początku metody call() i ma na celu symulowanie sytuacji, w której przetworzenie żądania wymaga nieco czasu.

Jeżeli w trakcie działania obiektu Callable zwróconego przez kontroler nastąpi zgłoszenie wyjątku, to metoda oznaczona adnotacją @ExceptionHandler (lub skonfigurowany komponent HandlerExcep

tionResolver) kontrolera będzie odpowiedzialna za obsługę wyjątku. Więcej informacji na temat adnotacji @ExceptionHandler znajdziesz w podrozdziale 10.9.

Przykład przedstawiony w listingu 13.7 pokazuje, że jeśli chcesz przejść z synchronicznego na asynchroniczne przetwarzanie żądania, trzeba przenieść logikę z metody oznaczonej adnotacją @RequestMapping do metody call() obiektu Callable, a także zmienić typ wartości zwrótej metody oznaczonej adnotacją @RequestMapping na Callable<T>.

Zobaczmy teraz, jak żądania są przetwarzane asynchronicznie, gdy wartością zwrótną metody oznaczonej adnotacją @RequestMapping jest obiekt DeferredResult.

IMPORT **Rozdział13/ch13-async-webservice i rozdział13/ch13-async-webservice-client.** (Projekt *ch13-async-webservice* jest wariantem usługi sieciowej FixedDepositWS (patrz: projekt *ch12-webservice* w rozdziale 12.), który asynchronicznie przetwarza żądania sieciowe. Metody oznaczone adnotacją @RequestMapping zdefiniowane w klasie FixedDepositController tego projektu zwracają egzemplarz obiektu DeferredResult. Projekt *ch13-async-webservice-client* jest taki sam jak klient usługi sieciowej FixedDepositWS (patrz: projekt *ch12-webservice-client* w rozdziale 12.), w którym przyjęto założenie, że usługa sieciowa została wdrożona pod adresem URL <http://localhost:8080/ch13-async-webservice>).

Zwrot obiektu DeferredResult przez metodę oznaczoną adnotacją @RequestMapping

Egzemplarz DeferredResult przedstawia wynik, który został wygenerowany asynchronicznie. Wynik w egzemplarzu DeferredResult jest ustawiany za pomocą wywołania metody setResult(). Zwykle metody oznaczone adnotacją @RequestMapping przechowują egzemplarz DeferredResult w obiekcie Queue lub Map bądź w innej strukturze danych. Zupełnie inny wątek jest odpowiedzialny za wygenerowanie wyniku i jego umieszczenie w egzemplarzu DeferredResult.

W pierwszej kolejności zajmiemy się metodami @RequestMapping zwracającymi typ DeferredResult.

Implementacja metody oznaczonej adnotacją @RequestMapping

W listingu 13.8 została przedstawiona klasa FixedDepositController, której metoda oznaczona adnotacją @RequestMapping zwraca obiekt DeferredResult.

Listing 13.8. Klasa FixedDepositController przedstawiająca metodę oznaczoną adnotacją @RequestMapping i zwracającą obiekt DeferredResult

Projekt: *ch13-async-webservice*

Położenie pliku: *src/main/java/sample/spring/chapter13/web*

```
package sample.spring.chapter13.web;

import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import org.springframework.web.context.request.async.DeferredResult;
.....
@Controller
@RequestMapping(value = "/fixedDeposits")
public class FixedDepositController {
    private static final String LIST_METHOD = "getFixedDepositList";
```

```

private static final String GET_FD_METHOD = "getFixedDeposit";
.....
private final Queue<ResultContext> deferredResultQueue =
    new ConcurrentLinkedQueue<ResultContext>();
.....
@RequestMapping(method = RequestMethod.GET)
public DeferredResult<ResponseEntity<List<FixedDepositDetails>>>
    getFixedDepositList() {
    DeferredResult<ResponseEntity<List<FixedDepositDetails>>> dr =
        new DeferredResult<ResponseEntity<List<FixedDepositDetails>>>();

    ResultContext<ResponseEntity<List<FixedDepositDetails>>> resultContext =
        new ResultContext<ResponseEntity<List<FixedDepositDetails>>>();
    resultContext.setDeferredResult(dr);
    resultContext.setMethodToInvoke(LIST_METHOD);
    resultContext.setArgs(new HashMap<String, Object>());

    deferredResultQueue.add(resultContext);
    return dr;
}
.....
}

```

Każda oznaczona adnotacją `@RequestMapping` metoda klasy `FixedDepositController` wykonuje następujące kroki:

Krok 1. Utworzenie egzemplarza obiektu `DeferredResult<T>`, w którym *T* przedstawia *typ* wyniku, jaki ma zostać wygenerowany asynchronicznie. Ponieważ typ wyniku generowanego przez metodę `getFixedDepositList()` to `ResponseEntity<List<FixedDepositDetails>>`, następuje utworzenie egzemplarza `DeferredResult<ResponseEntity<List<FixedDepositDetails>>>`.

Krok 2. Utworzenie egzemplarza obiektu `ResultContext`, który przechowuje egzemplarz `DeferredResult` utworzony w kroku 1. oraz inne szczegóły niezbędne do asynchronicznego wygenerowania wyniku dla obiektu `DeferredResult`. W przypadku metody `getFixedDepositList()` klasy `FixedDepositController` wynikiem jest lista lokat pobrana za pomocą wywołania metody `getFixedDeposits()` klasy `FixedDepositService`.

W listingu 13.9 została przedstawiona klasa `ResultContext`.

Listing 13.9. Klasa `ResultContext` przeznaczona do przechowywania obiektu `DeferredResult` i innych informacji

Projekt: *ch13-async-web-service*

Położenie pliku: *src/main/java/sample/spring/chapter13/web*

```

package sample.spring.chapter13.web;

import java.util.Map;
import org.springframework.web.context.request.async.DeferredResult;

public class ResultContext<T> {
    private String methodToInvoke;
    private DeferredResult<T> deferredResult;
    private Map<String, Object> args;

    public void setDeferredResult(DeferredResult<T> deferredResult) {

```

```
        this.deferredResult = deferredResult;
    }
    ....
}
```

Właściwość `deferredResult` odwołuje się do egzemplarza `DeferredResult`, właściwość `methodToInvoke` określa nazwę metody klasy `FixedDepositService` wywoływanej w celu wygenerowania wyniku dla obiektu `DeferredResult`, a właściwość `args` (typu `java.util.Map`) wskazuje argumenty przekazywane metodzie klasy `FixedDepositService`. Oddzielny wątek (jak to zostanie wyjaśnione w dalszej części podrozdziału) używa właściwości `methodToInvoke` i `args` do wywołania wskazanej metody klasy `FixedDepositService` i przypisania wygenerowanego przez nią wyniku egzemplarzowi `DeferredResult`.

Ponieważ `LIST_METHOD`, `GET_FD_METHOD` itd., czyli stałe w klasie `FixedDepositController`, odwołują się do nazw metod klasy `FixedDepositService` (patrz: listing 13.8), wartością właściwości `methodToInvoke` jest jedna z wymienionych stałych. W listingu 13.8 metoda `getFixedDepositList()` klasy `FixedDepositController` powoduje przypisanie właściwości `methodToInvoke` stałej `LIST_METHOD` (której wartość wynosi `getFixedDeposits()`), ponieważ metoda `getFixedDeposits()` klasy `FixedDepositService` musi być wywołana w celu uzyskania wyniku dla obiektu `DeferredResult` zwróconego przez wywołanie metody `getFixedDepositList()` klasy `FixedDepositController`.

Krok 3. Umieszczenie egzemplarza `ResultContext` utworzonego w kroku 2. w obiekcie `Queue` (patrz: zmienna egzemplarza `deferredResultQueue` w listingu 13.8).

Krok 4. Zwrot obiektu `DeferredResult` utworzonego w kroku 1.

Powyższa sekwencja kroków sugeruje, że dla każdego żądania sieciowego egzemplarz `ResultContext` jest przechowywany w `deferredResultQueue`. Na rysunku 13.2 przedstawiamy podsumowanie działań podejmowanych przez metodę `getFixedDepositList()` klasy `FixedDepositController`.

Zobaczmy teraz, jak jest generowany wynik dla egzemplarza `DeferredResult` przechowywanego w obiekcie `ResultContext`.

Wygenerowanie wyniku dla egzemplarza `DeferredResult`

Metoda `processResults()` klasy `FixedDepositController` jest odpowiedzialna za iterację przez obiekty `ResultContext` przechowywane w `deferredResultQueue` (patrz: listing 13.8), wygenerowanie wyniku dla każdego obiektu `DeferredResult` oraz przypisanie każdemu z nich wygenerowanego wyniku. W listingu 13.10 została przedstawiona metoda `processResults()`.

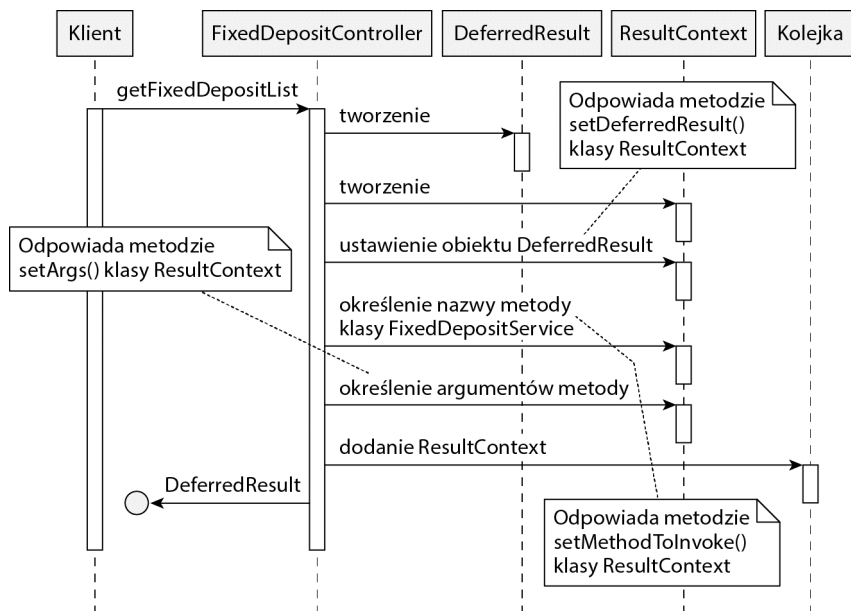
Listing 13.10. Metoda `processResults()` przedstawiająca przykład wygenerowania wyniku i przypisania go obiektowi `DeferredResult`

Projekt: *ch13-async-webservice*

Położenie pliku: *src/main/java/sample/spring/chapter13/web*

```
package sample.spring.chapter13.web;

import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.web.context.request.async.DeferredResult;
```



Rysunek 13.2. Metoda `getFixedDepositList()` klasy `FixedDepositController` dodaje obiekt `ResultContext` do kolejki i zwraca obiekt `DeferredResult`

```

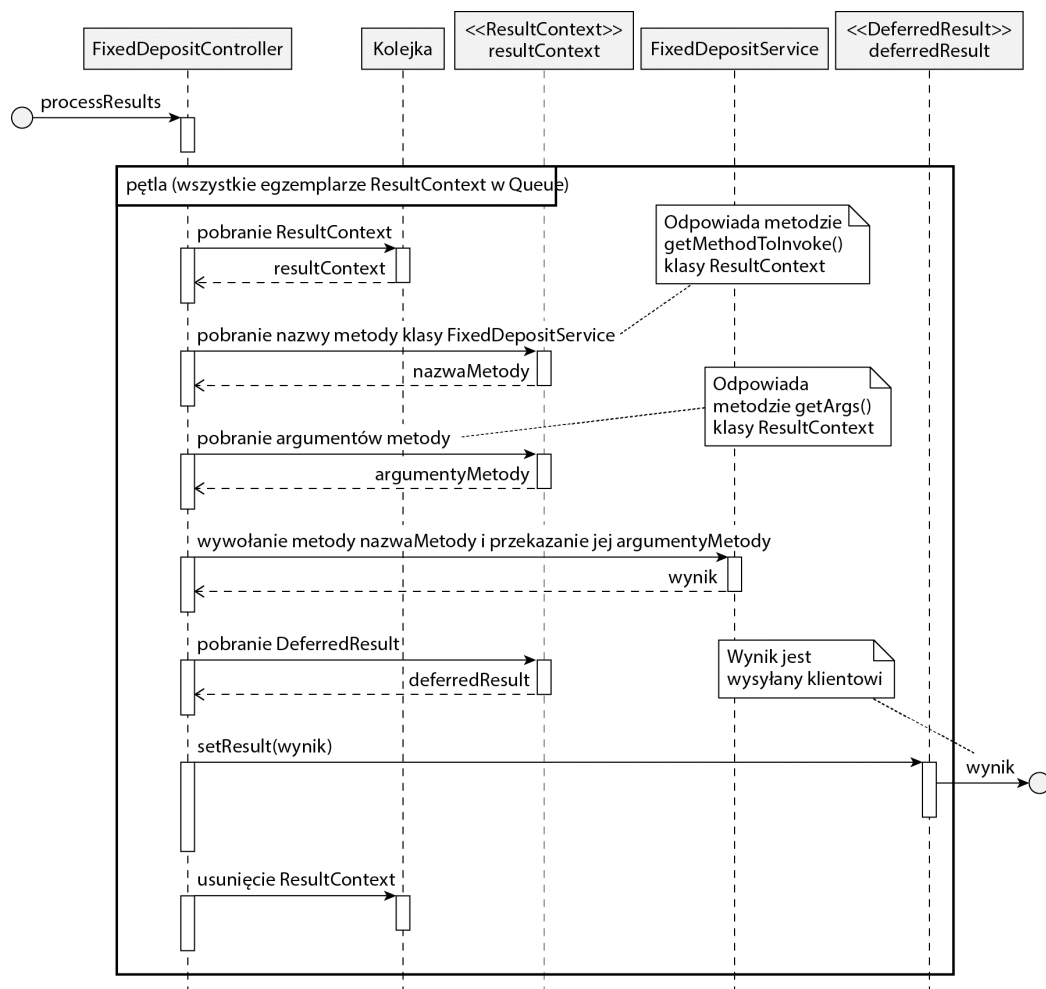
@Controller
@RequestMapping(value = "/fixedDeposits")
public class FixedDepositController {
    private static final String LIST_METHOD = "getFixedDepositList";
    ....
    private final Queue<ResultContext> deferredResultQueue =
        new ConcurrentLinkedQueue<ResultContext>();
    @Autowired
    private FixedDepositService fixedDepositService;
    ....
    @Scheduled(fixedRate = 10000)
    public void processResults() {
        for (ResultContext resultContext : deferredResultQueue) {
            if (resultContext.getMethodToInvoke() == LIST_METHOD) {
                resultContext.getDeferredResult().setResult(
                    new ResponseEntity<List<FixedDepositDetails>>(
                        fixedDepositService.getFixedDeposits(), HttpStatus.OK));
            }
            ....
            deferredResultQueue.remove(resultContext);
        }
    }
}

```

Adnotacja `@Scheduled` (więcej informacji na jej temat znajdziesz w podrozdziale 8.6) w metodzie `processResults()` określa, że co 10 sekund wątek aplikacji jest odpowiedzialny za wykonanie wymienionej metody. Przechowywana w egzemplarzu `ResultContext` nazwa metody i argumenty są przez metodę `processResults()` używane do wywołania odpowiedniej metody klasy `FixedDepositService`. Następnie metoda `processResults()` umieszcza wynik w egzemplarzu `DeferredResult`, wywołując

jego metodę `setResult()`. Na koniec metoda `processResults()` usuwa egzemplarz `ResultContext` z obiektu `Queue`. Po przetworzeniu egzemplarza `ResultContext` metoda `processResults()` usuwa egzemplarz `ResultContext` z obiektu `Queue`, aby nie został ponownie przetworzony przez `processResults()`, gdy ta metoda zostanie wywołana po kolejnych 10 sekundach.

Na rysunku 13.3 przedstawiamy podsumowanie działań podejmowanych przez metodę `processResults()` klasy `FixedDepositController` w celu wygenerowania wyniku i umieszczenia go w egzemplarzu `DeferredResult`.



Rysunek 13.3. Metoda `processResults()` odczytuje z obiektu `ResultContext` nazwę metody (i jej argumenty) przeznaczoną do wygenerowania wyniku dla egzemplarza `DeferredResult`

Zobaczmy teraz, jak przebiega obsługa wyjątków, gdy metoda oznaczona adnotacją `@RequestMapping` zwraca egzemplarz `DeferredResult`.

Obsługa wyjątków

Jeżeli ustawisz obiekt typu `java.lang.Exception`, używając do tego metody `setErrorResult()` klasy `DeferredResult`, wówczas wynik będzie obsługiwany przez oznaczoną adnotacją `@ExceptionHandler` metodę kontrolera (lub przez skonfigurowany komponent `HandlerExceptionResolver`). Więcej informacji na temat adnotacji `@ExceptionHandler` znajdziesz w podrozdziale 10.9.

W listingu 13.11 została przedstawiona metoda `openFixedDeposit()` klasy `FixedDepositController` odpowiedzialna za zakładanie nowych lokat.

Listing 13.11. Metoda `openFixedDeposit()` klasy `FixedDepositController`

Projekt: *ch13-async-webservice*

Położenie pliku: *src/main/java/sample/spring/chapter13/web*

```
package sample.spring.chapter13.web;

@Controller
@RequestMapping(value = "/fixedDeposits")
public class FixedDepositController {
    private static final String OPEN_FD_METHOD = "openFixedDeposit";
    ....
    private final Queue<ResultContext> deferredResultQueue =
        new ConcurrentLinkedQueue<ResultContext>();

    @RequestMapping(method = RequestMethod.POST)
    public DeferredResult<ResponseEntity<FixedDepositDetails>> openFixedDeposit(
        @RequestBody FixedDepositDetails fixedDepositDetails, BindingResult
        bindingResult) {

        DeferredResult<ResponseEntity<FixedDepositDetails>> dr =
            new DeferredResult<ResponseEntity<FixedDepositDetails>>();

        ResultContext<ResponseEntity<FixedDepositDetails>> resultContext =
            new ResultContext<ResponseEntity<FixedDepositDetails>>();
        resultContext.setDeferredResult(dr);
        resultContext.setMethodToInvoke(OPEN_FD_METHOD);

        Map<String, Object> args = new HashMap<String, Object>();
        args.put("fixedDepositDetails", fixedDepositDetails);
        args.put("bindingResult", bindingResult);
        resultContext.setArgs(args);
        deferredResultQueue.add(resultContext);
        return dr;
    }
    ....
}
```

W powyższym listingu widać, że argumenty (`fixedDepositDetails` i `bindingResult`) przekazywane metodzie `openFixedDeposit()` są ustawiane w egzemplarzu `ResultContext`. Tym samym argumenty stają się dostępne, gdy metoda `processResults()` wykonuje logikę przeznaczoną do zakładania nowych lokat. Argument `fixedDepositDetails` zawiera informacje szczegółowe o zakładanej lokacie, natomiast argument `bindingResult` zawiera wynik operacji dołączania danych.

W listingu 13.12 pokazujemy, jak metoda `processResults()` wykonuje logikę odpowiedzialną za zakładanie nowych lokat.

Listing 13.12. Metoda `processResults()` klasy `FixedDepositController`

Projekt: *ch13-async-webservice*

Położenie pliku: *src/main/java/sample/spring/chapter13/web*

```
package sample.spring.chapter13.web;

@Controller
@RequestMapping(value = "/fixedDeposits")
public class FixedDepositController {
    private static final String OPEN_FD_METHOD = "openFixedDeposit";
    .....
    private final Queue<ResultContext> deferredResultQueue =
        new ConcurrentLinkedQueue<ResultContext>();

    @Autowired
    private FixedDepositService fixedDepositService;
    .....
    @ExceptionHandler(ValidationException.class)
    @ResponseBody
    @ResponseStatus(value = HttpStatus.BAD_REQUEST)
    public String handleException(Exception ex) {
        logger.info("Obsługa ValidationException " + ex.getMessage());
        return ex.getMessage();
    }

    @Scheduled(fixedRate = 10000)
    public void processResults() {
        for (ResultContext resultContext : deferredResultQueue) {
            .....
            if (resultContext.getMethodToInvoke() == OPEN_FD_METHOD) {
                FixedDepositDetails fixedDepositDetails = (FixedDepositDetails)
                    resultContext.getArgs().get("fixedDepositDetails");
                BindingResult bindingResult = (BindingResult)
                    resultContext.getArgs().get("bindingResult");

                new FixedDepositDetailsValidator().validate(fixedDepositDetails,
                    bindingResult);

                if (bindingResult.hasErrors()) {
                    logger.info("Metoda openFixedDeposit(): Wystąpiły błędy podczas
                        weryfikacji");
                    resultContext.getDeferredResult().setErrorResult(new
                        ValidationException("Wystąpiły błędy podczas weryfikacji"));
                } else {
                    fixedDepositService.saveFixedDeposit(fixedDepositDetails);
                    resultContext.getDeferredResult().setResult(new
                        ResponseEntity<FixedDepositDetails>(
                            fixedDepositDetails, HttpStatus.CREATED));
                }
            }
            .....
        }
    }
}
```

Powyższy listing przedstawia oznaczoną adnotacją `@ExceptionHandler` metodę `handleException()` obsługującą wyjątki typu `ValidationException`. Zadaniem metody `handleException()` jest rejestracja wyjątku procesu weryfikacji i zwrot komunikatu generowanego przez ten wyjątek.

W celu założenia nowej lokaty metoda `processResults()` pobiera z obiektu `ResultContext` argumenty `fixedDepositDetails` (typu `FixedDepositDetails`) i `bindingResult` (typu `BindingResult`), a następnie przeprowadza weryfikację obiektu `fixedDepositDetails`, wywołując metodę `validate()` klasy `FixedDepositValidator`. Jeżeli zostaną wykryte jakiekolwiek błędy podczas weryfikacji, to `processResults()` wywołuje metodę `setErrorResult()` klasy `DeferredResult`, aby zgłosić wyjątek `ValidationException` (typu `java.lang.Exception`) jako wynik działania metody. Zgłoszenie wyjątku `ValidationException` za pomocą metody `setErrorResult()` klasy `DeferredResult` spowoduje obsługę wyniku przez metodę `handleException()` klasy `FixedDepositController`.

Zalecane jest wdrożenie projektu *ch13-async-webservice* (przedstawia usługę sieciową w stylu RESTful o nazwie `FixedDepositWS`) i uzyskanie dostępu do niego przez wykonanie metody `main()` klasy `FixedDepositWSClient` w projekcie *ch13-async-webservice-client* (przedstawia klienta usługi sieciowej `FixedDepositWS`). Metoda `openInvalidFixedDeposit()` klasy `FixedDepositWSClient` wywołuje metodę `openFixedDeposit()` klasy `FixedDepositController` w taki sposób, że skutkiem jest wyjątek `ValidationException`. Zglądając do dziennika zdarzeń, możesz potwierdzić, że metoda `handleException()` klasy `FixedDepositController` zajmuje się obsługą wyjątku, gdy `processResults()` zgłosi wyjątek `ValidationException` w obiekcie `DeferredResult` przez wywołanie metody `setErrorResult()` klasy `DeferredResult`.

Zobaczmy teraz, jak zdefiniować wartość domyślną limitu czasu dla żądań asynchronicznych.

Ustawienie domyślnego limitu czasu

Istnieje możliwość zdefiniowania domyślnego limitu czasu dla żądań asynchronicznych za pomocą atrybutu `default-timeout` elementu `<async-support>`, jak przedstawia listing 13.13.

Listing 13.13. Ustawienie domyślnego limitu czasu dla żądań asynchronicznych

Projekt: *ch13-async-webservice*

Położenie pliku: *src/main/webapp/WEB-INF/spring/webservice-config.xml*

```
<mvc:annotation-driven>
  <mvc:async-support default-timeout="10000" >
    .....
  </mvc:async-support>
</mvc:annotation-driven>
```

W powyższym listingu domyślny limit czasu dla żądań asynchronicznych wynosi 10 sekund. Jeżeli nie zdefiniujesz domyślnego limitu czasu, wówczas limit czasu dla żądań asynchronicznych będzie zależał od kontenera Servleta, w którym została wdrożona aplikacja sieciowa.

Zobaczmy teraz, w jaki sposób można przechwytywać żądania asynchroniczne za pomocą `CallableProcessingInterceptor` i `DeferredResultProcessingInterceptor`.

Przechwytywanie żądań asynchronicznych

Jeżeli wykorzystasz egzemplarz `Callable` do asynchronicznego przetwarzania żądań, możesz użyć interfejsu wywołań zwrotnych `CallableProcessingInterceptor` do przechwytywania żądań przed wykonaniem zadania `Callable` i po wykonaniu. Na przykład metoda `postProcess()` jest wykonywana po wygenerowaniu wyniku przez `Callable`, natomiast metoda `preProcess()` jest wywoływana przed wykonaniem zadania przez `Callable`. Podobnie jeśli wykorzystasz obiekt `DeferredResult`, możesz użyć interfejsu wywołań zwrotnych `DeferredResultProcessingInterceptor` do przechwytywania operacji przetwarzania żądań asynchronicznych.

Konfigurację `CallableProcessingInterceptor` można przeprowadzić za pomocą elementu `<callable-interceptors>` schematu `mvc` oferowanego przez Spring. Z kolei konfigurację `DeferredResultProcessingInterceptor` przeprowadza się za pomocą elementu `<deferred-result-interceptors>` tego schematu. Listing 13.14 przedstawia konfigurację `MyDeferredResultInterceptor` (implementacja `DeferredResultProcessingInterceptor`).

Listing 13.14. Konfiguracja implementacji `DeferredResultProcessingInterceptor`

Projekt: *ch13-async-webservice*

Położenie pliku: *src/main/webapp/WEB-INF/spring/webservice-config.xml*

```
<mvc:annotation-driven>
  <mvc:async-support default-timeout="30000">
    <mvc:deferred-result-interceptors>
      <bean class="sample.spring.chapter13.web.MyDeferredResultInterceptor"/>
    </mvc:deferred-result-interceptors>
  </mvc:async-support>
</mvc:annotation-driven>
```

Przechodzimy teraz do obsługi przez Spring konwersji typu i formatowania.

13.5. Obsługa konwersji typu i formatowania

Oferowany przez Spring interfejs `Converter` ułatwia konwersję typu obiektu na inny typ. Z kolei udostępniany przez Spring interfejs `Formatter` okazuje się użyteczny podczas konwersji typu obiektu na jego *zlokalizowaną* postać w formacie `String` i na odwrót. Wiele wbudowanych implementacji interfejsu `Converter` znajdziesz w pakiecie `org.springframework.core.convert.support` dostępnym w pliku JAR o nazwie *spring-core*. Spring oferuje również wbudowane implementacje `Formatters` dla typów `java.lang.Number` i `java.util.Date`, znajdziesz je w pakietach odpowiednio `org.springframework.format.number` i `org.springframework.format.datetime`.

IMPORT **Rozdział13/ch13-converter-formatter-bankapp**. (Ten projekt jest wariantem projektu *ch13-bankapp* i pokazuje, jak utworzyć własne implementacje interfejsów `Converter` i `Formatter`).

Najpierw zobaczysz, jak utworzyć własną implementację interfejsu `Converter`.

Utworzenie własnej implementacji interfejsu Converter

Konwerter implementuje oferowany przez Spring interfejs `Converter<S, T>`, gdzie `S` (określany mianem typu *źródłowego*) wskazuje typ obiektu przekazanego konwerterowi, natomiast `T` (określany mianem typu *docelowego*) to typ, na który zostanie skonwertowany przekazany obiekt. Interfejs `Converter` definiuje metodę `convert()` zawierającą logikę odpowiedzialną za przeprowadzenie konwersji.

W listingu 13.15 przedstawiamy implementację `IdToFixedDepositDetailsConverter`, która przeprowadza konwersję obiektu typu `String` (reprezentującego identyfikator lokaty) na obiekt typu `FixedDepositDetails` (reprezentujący lokatę odpowiadającą podanemu identyfikatorowi).

Listing 13.15. Implementacja interfejsu Converter

Projekt: *ch13-converter-formatter-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter13/converter*

```
package sample.spring.chapter13.converter;

import org.springframework.core.convert.converter.Converter;
.....
public class IdToFixedDepositDetailsConverter implements Converter<String,
    FixedDepositDetails> {

    @Autowired
    private FixedDepositService fixedDepositService;
    @Override
    public FixedDepositDetails convert(String source) {
        return fixedDepositService.getFixedDeposit(Integer.parseInt(source));
    }
}
```

Konwerter `IdToFixedDepositDetailsConverter` implementuje interfejs `Converter<String, FixedDepositDetails>`, gdzie `String` to typ obiektu źródłowego, natomiast `FixedDepositDetails` wskazuje typ obiektu docelowego. Metoda `convert()` klasy `IdToFixedDepositDetailsConverter` używa metody `getFixedDeposit()` klasy `FixedDepositService` do pobrania obiektu `FixedDepositDetails` odpowiadającego podanemu identyfikatorowi lokaty.

Teraz przechodzimy do konfiguracji i użycia własnego konwertera.

Konfiguracja i użycie własnej implementacji interfejsu Converter

W celu użycia własnego konwertera trzeba go zarejestrować za pomocą oferowanego przez Spring egzemplarza `ConversionService`. Egzemplarz `ConversionService` działa w charakterze rejestru dla `Converters` i `Formatters`, a Spring deleguje konwersję typu do zarejestrowanego egzemplarza `ConversionService`. Domyślnie element `<annotation-driven>` schematu `mvc` automatycznie rejestruje w kontenerze Springa egzemplarz `FormattingConversionService` (implementacja `ConversionService`). Spring jest domyślnie dostarczany z wieloma wbudowanymi implementacjami interfejsów `Converter` i `Formatter` automatycznie zarejestrowanymi za pomocą `FormattingConversionService`. Jeżeli chcesz użyć innej implementacji w egzemplarzu `ConversionService`, możesz ją wskazać za pomocą atrybutu `conversion-service` elementu `<annotation-driven>`.

Aby zarejestrować własny konwerter w egzemplarzu `FormattingConversionService`, należy skonfigurować oferowany przez Spring komponent `FormattingConversionServiceFactoryBean` (implementacja `FactoryBean` tworząca i konfiguruje `FormattingConversionService`) oraz wskazać własne konwertery jako część tej konfiguracji. Takie rozwiązanie przedstawia listing 13.16.

Listing 13.16. *Rejestracja własnej implementacji Converter za pomocą FormattingConversionService*

Projekt: *ch13-converter-formatter-bankapp*

Położenie pliku: *src/main/webapp/WEB-INF/spring*

```
<mvc:annotation-driven conversion-service="myConversionService" />

<bean id="myConversionService"
      class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
  <property name="converters">
    <set>
      <bean class="sample.spring.chapter13.converter
              .IdToFixedDepositDetailsConverter" />
    </set>
  </property>
</bean>
.....
</bean>
```

Domyślnie `FormattingConversionServiceFactoryBean` rejestruje w egzemplarzu `FormattingConversionService` jedynie wbudowane implementacje `Converter` i `Formatter`. Własne implementacje można zarejestrować za pomocą właściwości `converters` i `formatters` egzemplarza `FormattingConversionServiceFactoryBean`. Ponieważ chcemy, aby nasza aplikacja Springa używała egzemplarza `FormattingConversionService` utworzonego przez `FormattingConversionServiceFactoryBean`, atrybut `conversion-service` elementu `<annotation-driven>` odwołuje się do `FormattingConversionServiceFactoryBean`.

Implementacje `Converter` i `Formatter` zarejestrowane w egzemplarzu `FormattingConversionService` są używane przez kontener Springa do przeprowadzania konwersji typu podczas operacji *dołączania danych*. W listingu 13.17 metoda `viewFixedDepositDetails()` klasy `FixedDepositController` przedstawia sytuację, w której kontener Springa używa `IdToFixedDepositDetailsConverter<String, FixedDepositDetails>` do konwersji identyfikatora lokaty (typ `String`) na egzemplarz `FixedDepositDetails`.

Listing 13.17. *Metoda viewFixedDepositDetails() klasy FixedDepositController*

Projekt: *ch13-converter-formatter-bankapp*

Położenie pliku: *src/main/java/sample/spring/chapter13/web*

```
package sample.spring.chapter13.web;
.....
public class FixedDepositController {
  .....
  @RequestMapping(params = "fdAction=view", method = RequestMethod.GET)
  public ModelAndView viewFixedDepositDetails(
    @RequestParam(value = "fixedDepositId")
    FixedDepositDetails fixedDepositDetails) {
    .....
  }
}
```

Adnotacja `@RequestParam` wskazuje, że wartość parametru żądania `fixedDepositId` będzie przypisana argumentowi metody `fixedDepositDetails`. Parametr żądania unikalnie identyfikuje lokatę. Ponieważ parametr żądania `fixedDepositId` ma wartość typu `String`, a typ metody argumentu to `FixedDepositDetails`, Spring używa `IdToFixedDepositDetailsConverter<String, FixedDepositDetails>` do przeprowadzenia konwersji typu.

Użycie egzemplarza `ConversionService` nie ogranicza się jedynie do warstwy sieciowej. Istnieje możliwość użycia `ConversionService` do programowego przeprowadzenia konwersji typu na dowolnej warstwie aplikacji. Listing 13.18 przedstawia wariant metody `viewFixedDepositDetails()` klasy `FixedDepositController`, która bezpośrednio używa egzemplarza `ConversionService` do przeprowadzenia konwersji typu.

Listing 13.18. Przykład programowego przeprowadzenia konwersji typu

```
import org.springframework.core.convert.ConversionService;
.....
public class FixedDepositController {
    @Autowired
    private ConversionService conversionService;
    .....
    @RequestMapping(params = "fdAction=view", method = RequestMethod.GET)
    public ModelAndView viewFixedDepositDetails(HttpServletRequest request) {
        String fixedDepositId = request.getParameter("fixedDepositId");
        FixedDepositDetails fixedDepositDetails =
            conversionService.convert(fixedDepositId, FixedDepositDetails.class);
        .....
    }
}
```

W powyższym listingu egzemplarz `ConversionService` zarejestrowany w kontenerze Springa jest automatycznie wiązany w kontrolerze `FixedDepositController`. Metoda `viewFixedDepositDetails()` używa metody `convert()` klasy `ConversionService` do przeprowadzenia konwersji `fixedDepositId` (typu `String`) na typ `FixedDepositDetails`. W tle, do przeprowadzenia konwersji typu, egzemplarz `ConversionService` wykorzystuje zarejestrowany w nim konwerter `IdToFixedDepositDetailsConverter<String, FixedDepositDetails>`.

Skoro już wiesz, jak utworzyć własną implementację `Converter` i jej używać, możemy przejść do tematu tworzenia i użycia własnej implementacji `Formatter`.

Tworzenie własnej implementacji `Formatter`

Implementacja `Formatter` konwertuje obiekt typu `T` na wartość `String` w celu jej wyświetlenia, a także przetwarza wartość `String` na obiekt typu `T`. Implementuje oferowany przez Spring interfejs `Formatter<T>`, gdzie `T` oznacza typ obiektu poddawanego formatowaniu. To może przypominać działanie `PropertyEditors` w aplikacjach sieciowych. Jak się przekonasz w tym rozdziale, implementacja `Formatter` stanowi bardziej niezawodną alternatywę dla `PropertyEditors`.



Biblioteka znaczników w Springu używa implementacji `Formatter` zarejestrowanych w egzemplarzu `FormattingConversionService` do przeprowadzania konwersji typu podczas operacji dołączania danych i ich generowania.

W listingu 13.19 przedstawiamy implementację `AmountFormatter` używaną przez aplikację `MyBank` do wyświetlenia wartości lokaty w walucie zgodnej z ustawieniami językowymi użytkownika, a także do przetworzenia wartości lokaty podanej przez użytkownika. Dla zachowania prostoty nie jest przeprowadzana konwersja walut, a jedynie następuje dołączenie do wartości lokaty symbolu waluty zgodnego z ustawieniami językowymi użytkownika.

Listing 13.19. Klasa `AmountFormatter` przedstawiająca implementację `Formatter`

Projekt: `ch13-converter-formatter-bankapp`

Położenie pliku: `src/main/java/sample/spring/chapter13/formatter`

```
package sample.spring.chapter13.formatter;

import java.text.ParseException;
import java.util.Locale;
import org.springframework.format.Formatter;

public class AmountFormatter implements Formatter<Long>{

    @Override
    public String print(Long object, Locale locale) {
        String returnStr = object.toString() + " USD";
        if(locale.getLanguage().equals(new Locale("pl").getLanguage())) {
            returnStr = object.toString() + " PLN";
        }
        return returnStr;
    }

    @Override
    public Long parse(String text, Locale locale) throws ParseException {
        String str[] = text.split(" ");
        return Long.parseLong(str[0]);
    }
}
```

Klasa `AmountFormatter` implementuje interfejs `Formatter<Long>`, co oznacza, że jest stosowana dla obiektów typu `Long`. Metoda `print()` konwertuje obiekt typu `Long` (przedstawiający wartość lokaty) na wartość typu `String`, która jest wyświetlana użytkownikowi. Opierając się na kodzie języka pobranym z bieżących ustawień językowych, metoda `print()` po prostu dołącza do wartości lokaty znaki `USD` (dla kodu `en`) lub `PLN` (dla kodu `pl`). Na przykład jeśli wartość lokaty wynosi 1000, a kod języka to `pl`, wówczas wartością zwrótną metody `print()` jest `1000 PLN`. Metoda `parse()` pobiera wartość lokaty wprowadzoną przez użytkownika (np. `1000 PLN`) i konwertuje ją na obiekt typu `Long`, po prostu wyodrębniając wartość lokaty z danych wprowadzonych przez użytkownika.

Zobaczmy teraz, jak można skonfigurować własną implementację `Formatter`.

Konfiguracja własnej implementacji Formatter

Własną implementację Formatter można zarejestrować w egzemplarzu FormattingConversionService ↗Service za pomocą właściwości formatters obiektu FormattingConversionServiceFactoryBean, jak przedstawia listing 13.20.

Listing 13.20. Rejestracja własnej implementacji Formatter za pomocą FormattingConversionService

```
<beans .....>
    .....
    <mvc:annotation-driven conversion-service="myConversionService" />
    .....
    <bean id="myConversionService"
        class="org.springframework.format.support
        .FormattingConversionServiceFactoryBean">
        <property name="formatters">
            <set>
                <bean class="sample.spring.chapter13.formatter.AmountFormatter" />
            </set>
        </property>
    </bean>
</beans>
```

Implementacja AmountFormatter zarejestrowana w egzemplarzu FormattingConversionService ma zastosowanie dla *wszystkich* właściwości typu Long w trakcie operacji dołączania danych i ich generowania.

Właściwości, dla których będzie stosowana implementacja Formatter, można wskazywać za pomocą oferowanego przez Spring egzemplarza AnnotationFormatterFactory. Egzemplarz ten tworzy implementację Formatter dla właściwości oznaczonych odpowiednią *adnotacją*. Zobaczmy więc, w jaki sposób można użyć AnnotationFormatterFactory do sformatowania jedynie właściwości typu Long oznaczonych adnotacją @AmountFormat.

Utworzenie implementacji AnnotationFormatterFactory do formatowania jedynie właściwości oznaczonych adnotacją @AmountFormat

W listingu 13.21 została przedstawiona definicja adnotacji @AmountFormat.

Listing 13.21. Definicja adnotacji @AmountFormat

Projekt: ch13-converter-formatter-bankapp

Położenie pliku: src/main/java/sample/spring/chapter13/formatter

```
package sample.spring.chapter13.formatter;
.....
@Target(value={ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface AmountFormat { }
```

W powyższym listingu adnotacja @Target wskazuje, że adnotacja @AmountFormat może być stosowana jedynie dla właściwości.

W listingu 13.22 przedstawiamy implementację `AnnotationFormatterFactory` przeznaczoną do formatowania właściwości oznaczonych adnotacją `@AmountFormat`.

Listing 13.22. Klasa `AmountFormatAnnotationFormatterFactory`

Projekt: `ch13-converter-formatter-bankapp`

Położenie pliku: `src/main/java/sample/spring/chapter13/formatter`

```
package sample.spring.chapter13.formatter;

import org.springframework.format.AnnotationFormatterFactory;
import org.springframework.format.Parser;
import org.springframework.format.Printer;

public class AmountFormatAnnotationFormatterFactory implements
    AnnotationFormatterFactory<AmountFormat> {

    public Set<Class<?>> getFieldTypes() {
        Set<Class<?>> fieldTypes = new HashSet<Class<?>>(1, 1);
        fieldTypes.add(Long.class);
        return fieldTypes;
    }

    public Parser<?> getParser(AmountFormat annotation, Class<?> fieldType) {
        return new AmountFormatter();
    }

    public Printer<?> getPrinter(AmountFormat annotation, Class<?> fieldType) {
        return new AmountFormatter();
    }
}
```

W powyższym listingu klasa `AmountFormatAnnotationFormatterFactory` implementuje interfejs `AnnotationFormatterFactory<AmountFormat>`. Oznacza to, że klasa `AmountFormatAnnotationFormatterFactory` tworzy implementacje `Formatter` dla właściwości oznaczonych adnotacją `@AmountFormat`.

Metoda `getFieldTypes()` zwraca typy właściwości, które mogą być oznaczone adnotacją `@AmountFormat`. W omawianym przykładzie metoda `getFieldTypes()` zwraca pojedynczy typ — `Long`. Dlatego jedynie właściwości typu `Long` oznaczone adnotacją `@AmountFormat` będą brane pod uwagę do sformatowania przez implementację utworzoną przez `AmountFormatAnnotationFormatterFactory`. Metody `getParser()` i `getPrinter()` zwracają implementacje `Formatter` dla właściwości oznaczonych adnotacją `@AmountFormat`. Warto zwrócić uwagę na fakt, że `Formatter` jest subinterfejsem interfejsów `Parser` i `Printer`.

Konfiguracja implementacji `AnnotationFormatterFactory`

Podobnie jak w przypadku konfiguracji `Formatter`, także implementacja `AnnotationFormatterFactory` jest rejestrowana w egzemplarzu `FormattingConversionService` za pomocą właściwości `formatters` egzemplarza `FormattingConversionServiceFactoryBean`, jak przedstawia listing 13.23.

Listing 13.23. Konfiguracja implementacji `AmountFormatAnnotationFormatterFactory`

Projekt: `ch13-converter-formatter-bankapp`

Położenie pliku: `src/main/webapp/WEB-INF/spring`

```
<beans .....>
.....
<mvc:annotation-driven conversion-service="myConversionService" />
.....
<bean id="myConversionService"
      class="org.springframework.format.support
        .FormattingConversionServiceFactoryBean">
  <property name="formatters">
    <set>
      <bean
        class="sample.spring.chapter13.formatter
          .AmountFormatAnnotationFormatterFactory" />
    </set>
  </property>
</bean>
</beans>
```

Skoro zobaczyłeś, jak używać implementacji `AnnotationFormatterFactory` w celu umożliwienia sformatowania właściwości oznaczonych określoną adnotacją, to warto przekonać się, jak tę możliwość można wykorzystać w projekcie `ch13-converter-formatter-bankapp`.

Na rysunku 13.4 pokazujemy stronę internetową projektu `ch13-converter-formatter-bankapp` wyświetlającą listę założonych lokat.

Rysunek 13.4.

Kolumna wartość lokaty wyświetla USD lub PLN w zależności od kodu języka pobranego z bieżących ustawień językowych użytkownika

Fixed deposit list

ID	Deposit amount	Tenure	Email	Action
1	10000 USD	24	a1email@nazwa-domeny.pl	Close Edit
2	20000 USD	36	a2email@nazwa-domeny.pl	Close Edit
3	30000 USD	36	a3email@nazwa-domeny.pl	Close Edit
4	50000 USD	36	a4email@nazwa-domeny.pl	Close Edit
5	15000 USD	36	a5email@nazwa-domeny.pl	Close Edit

[Create new Fixed Deposit](#)

Język: [English\(US\)](#) | [Polski](#)
Locale: en_US

Na rysunku 13.4 widać, że symbol USD zostanie dołączony do wartości lokaty, gdy językiem wybranym przez użytkownika jest angielski. Po przełączeniu języka na polski symbol USD zostanie zmieniony na PLN. W listingu 13.19 przedstawionym we wcześniejszej części rozdziału pokazaliśmy, że klasa `AmountFormatter` zawiera logikę odpowiedzialną za wyświetlenie odpowiedniego symbolu waluty na podstawie kodu języka pobranego z bieżących ustawień językowych użytkownika.

Aby mieć pewność, że implementacje `Formatter` skonfigurowane w egzemplarzu `FormattingCon` ↪ `versionService` będą wywoływane podczas generowania strony i wysyłania formularzy sieciowych, oferowana przez Spring biblioteka znaczników (takich jak `<eval>` i `<input>`) została użyta na stronach projektu *ch13-converter-formatter-bankapp*.

Zobaczmy teraz, jak Spring Web MVC ułatwia przekazywanie plików.

13.6. Obsługa przekazywania plików w Spring Web MVC

Żądania wieloczęściowe w aplikacjach Spring Web MVC można obsługiwać przez skonfigurowanie implementacji interfejsu `MultipartResolver`. Standardowo Spring oferuje wymienione poniżej implementacje interfejsu `MultipartResolver`, które można wykorzystać we własnych aplikacjach sieciowych:

- `CommonsMultipartResolver` — oparty na bibliotece Apache Commons FileUpload;
- `StandardServletMultipartResolver` — oparty na API Servlet 3.0 Part.

Po otrzymaniu wieloczęściowego żądania `DispatcherServlet` używa skonfigurowanej implementacji `MultipartResolver` w celu opakowania `HttpServletRequest` egzemplarzem `MultipartHttpServletRequest` ↪ `Request`. W aplikacji Spring Web MVC przekazany plik jest reprezentowany przez obiekt `MultipartFile`. Kontroler odpowiedzialny za obsługę operacji przekazywania pliku uzyskuje do niego dostęp za pomocą metod zdefiniowanych przez `MultipartHttpServletRequest` lub bezpośrednio pracując z obiektem `MultipartFile`.

Najpierw przyjrzymy się aplikacji używającej do przekazywania plików implementacji `CommonsMultipartResolver`.

IMPORT `Rozdział13/ch13-commons-file-upload`. (Ten projekt pokazuje przykład użycia implementacji `CommonsMultipartResolver` w celu przekazywania plików. Ponieważ `CommonsMultipartResolver` używa biblioteki Apache Commons FileUpload, projekt jest zależny od pliku JAR o nazwie *commons-fileupload*).

Przekazywanie plików za pomocą implementacji `CommonsMultipartResolver`

Listing 13.24 przedstawia formularz przeznaczony do przekazywania plików wyświetlany przez projekt *ch13-commons-file-upload*.

Listing 13.24. Zawartość pliku *uploadForm.jsp* wyświetlającego formularz sieciowy

Projekt: *ch13-commons-file-upload*

Położenie pliku: *src/main/webapp/WEB-INF/jsp*

```
.....
<form method="post" action="/ch13-commons-file-upload/uploadFile"
      enctype="multipart/form-data">
  <table style="padding-left: 200px;">
    <tr>
      <td colspan="2"><c:out value="${uploadMessage}" /></td>
    </tr>
  </table>
```

```

        <td><b>Wybierz plik do przekazania: &nbsp;</b></td>
        <td><input type="file" name="myFileField" /></td>
    </tr>
    <tr>
        <td colspan="2" align="center"><input type="button"
            value="Przełącz plik" onclick="document.forms[0].submit();" /></td>
    </tr>
</table>
</form>
.....

```

W powyższym listingu widać, że atrybut `enctype` elementu `<form>` ma przypisaną wartość `multipart/form-data`. Oznacza to, że wysłanie formularza skutkuje wykonaniem wieloczęściowego żądania do serwera. Atrybut `uploadMessage` powoduje wyświetlenie odpowiedniego komunikatu po wybraniu przez użytkownika pliku i kliknięciu przycisku *Przełącz plik*.

W listingu 13.25 przedstawiamy konfigurację implementacji `CommonsMultipartResolver` zajmującej się obsługą wieloczęściowych żądań.

Listing 13.25. Zawartość pliku `fileupload-config.xml` przedstawiająca konfigurację `CommonsMultipartResolver`

Projekt: `ch13-commons-file-upload`

Położenie pliku: `src/main/webapp/WEB-INF/spring`

```

<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="maxUploadSize" value="100000" />
    <property name="resolveLazily" value="true" />
</bean>

```

Trzeba koniecznie zwrócić uwagę na fakt, że implementacja `MultipartResolver` musi być w pliku XML kontekstu aplikacji sieciowej skonfigurowana wraz z atrybutem `id` o wartości `multipartResolver`. Właściwość `maxUploadSize` określa (wyrażoną w bajtach) maksymalną wielkość pliku, jaki można przekazać. Jeżeli w omawianym przykładzie spróbujesz przekazać plik o wielkości przekraczającej 100 KB, wówczas nasza implementacja `CommonsMultipartResolver` spowoduje zgłoszenie wyjątku. Jeżeli egzemplarz `CommonsMultipartResolver` zgłosi wyjątek, kontroler odpowiedzialny za obsługę operacji przekazania pliku nie będzie miał okazji obsłużyć wyjątku. Z tego powodu właściwości `resolveLazily` przypisano wartość `true`. Jeżeli właściwość `resolveLazily` ma wartość `true`, wówczas wieloczęściowe żądanie jest obsługiwane tylko wtedy, gdy przekazywany plik jest dostępny dla kontrolera. W ten sposób kontroler ma możliwość obsługi wyjątków, jakie mogą być zgłaszane w trakcie wieloczęściowego żądania.

W listingu 13.26 została przedstawiona klasa `FileUploadController` odpowiedzialna za obsługę przekazywania plików.

Listing 13.26. Klasa `FileUploadController`

Projekt: `ch13-commons-file-upload`

Położenie pliku: `src/main/java/sample/spring/chapter13/web`

```

package sample.spring.chapter13.web;

import org.springframework.web.multipart.MultipartFile;

```

```

.....
public class FileUploadController {
    .....
    @RequestMapping(value = "/uploadFile", method = RequestMethod.POST)
    public ModelAndView handleFileUpload(
        @RequestParam("myFileField") MultipartFile file) throws IOException {
        ModelMap modelData = new ModelMap();
        if (!file.isEmpty()) {
            // Zapisanie w systemie plików przekazanego pliku.
            String successMessage = "Przekazanie pliku zakończyło się powodzeniem.";
            modelData.put("uploadMessage", successMessage);
            return new ModelAndView("uploadForm", modelData);
        }
        .....
    }

    @ExceptionHandler(value = Exception.class)
    public ModelAndView handleException() {
        .....
    }
}

```

Metoda `handleFileUpload()` klasy `FileUploadController` akceptuje argument typu `MultipartFile` identyfikujący przekazany plik. Zauważ, że adnotacja `@RequestParam` określa nazwę elementu `<input type="file" .....>` na stronie `uploadForm.jsp` (patrz: listing 13.24). Jeżeli przekazanie pliku zakończy się powodzeniem, wówczas metoda `handleFileUpload()` wyświetli użytkownikowi odpowiedni komunikat. Oznaczona adnotacją `@ExceptionHandler` metoda wyświetli komunikat błędu, gdy w trakcie operacji przekazywania pliku nastąpi zgłoszenie wyjątku. Na przykład jeśli spróbujesz przekazać plik o wielkości przekraczającej 100 KB, użytkownikowi zostanie wyświetlony komunikat błędu.

Skoro już wiesz, jak używać implementacji `CommonsMultipartResolver` do przekazywania plików, możemy przejść do wykonania tego samego zadania za pomocą implementacji `StandardServletMultipartResolver`.

IMPORT `Rozdział13/ch13-servlet3-file-upload`. (Ten projekt pokazuje przykład użycia implementacji `StandardServletMultipartResolver` do przekazywania plików).

Przekazywanie plików za pomocą `StandardServletMultipartResolver`

Obsługa wieloczęściowych żądań jest standardowo wbudowana w Servlet 3. Jeżeli chcesz wykorzystać tę możliwość, włącz obsługę wieloczęściowych żądań przez określenie elementu `<multipart-config>` w konfiguracji `DispatcherServlet`, a następnie w pliku XML kontekstu aplikacji sieciowej skonfiguruj implementację `StandardServletMultipartResolver`. W przeciwieństwie do `CommonsMultipartResolver` implementacja `StandardMultipartResolver` nie definiuje żadnych właściwości.

W listingu 13.27 została przedstawiona konfiguracja `DispatcherServlet` przeprowadzona w pliku `web.xml`.

Listing 13.27. Zawartość pliku *web.xml**Projekt: ch13-servlet3-file-upload**Położenie pliku: src/main/webapp*

```
<servlet>
  <servlet-name>fileupload</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  .....
  <multipart-config>
    <max-file-size>10000</max-file-size>
  </multipart-config>
</servlet>
```

Ponieważ element `<multipart-config>` został określony, serwlet `fileupload` może obsłużyć żądania składające się z wielu części. Element `<max-file-size>` wskazuje maksymalną wielkość pliku, który może zostać przekazany. Zwróć uwagę na fakt, że wielkość pliku została teraz podana jako część elementu `<multipart-config>`.

13.7. Podsumowanie

W tym rozdziale przedstawiliśmy pewne najważniejsze funkcje frameworku Spring Web MVC, które ułatwiają tworzenie aplikacji sieciowych. Z kolejnego rozdziału dowiesz się, jak zabezpieczyć aplikację Spring za pomocą frameworku Spring Security.

Rozdział 14. Zabezpieczanie aplikacji za pomocą Spring Security

14.1. Wprowadzenie

Bezpieczeństwo to bardzo ważny aspekt każdej aplikacji. Spring Security został zbudowany na bazie Springa i oferuje potężne możliwości w zakresie zabezpieczania aplikacji. Z tego rozdziału dowiesz się, jak wykorzystać framework Spring Security do następujących celów:

- uwierzytelnianie użytkowników;
- implementacja zabezpieczeń żądania sieciowego;
- implementacja zabezpieczeń na poziomie metody;
- zabezpieczanie obiektów domeny za pomocą list ACL (ang. *access control list*).

Rozpoczynamy od przeanalizowania wymagań aplikacji sieciowej MyBank w zakresie bezpieczeństwa. Następnie wymagania te spełnimy za pomocą frameworku Spring Security.

14.2. Wymagania aplikacji sieciowej MyBank w zakresie bezpieczeństwa

Użytkownikami aplikacji sieciowej MyBank są *klienci* i *administratorzy*, którzy zarządzają lokatami w systemie. Klient może założyć nową lokatę i modyfikować jej parametry, ale *nie* może jej zamknąć. Z kolei administrator *nie* ma możliwości zakładania lub modyfikowania lokat, ale może zamykać lokaty należące do klientów.

Ponieważ tylko uwierzytelniony użytkownik może uzyskać dostęp do aplikacji sieciowej MyBank, nie-uwierzytelnionym użytkownikom jest wyświetlany formularz logowania pokazany na rysunku 14.1.

Rysunek 14.1.
Formularz logowania
wyświetlany
nieuwierzytelnionym
użytkownikom

Zaloguj się podając nazwę użytkownika i hasło

Nazwa użytkownika:

Hasło:

☒ Zapamiętaj mnie w tym komputerze.

Na rysunku 14.1 widać, że formularz logowania jest wyświetlany nieuwierzytelnionym użytkownikom. Jeżeli użytkownik zaznaczy pole wyboru *Zapamiętaj mnie w tym komputerze*, aplikacja sieciowa MyBank zachowa jego dane uwierzytelniające i będzie je automatycznie używać w przyszłości do uwierzytelnienia tego użytkownika.

Po zalogowaniu się *klienta* do systemu zostanie mu wyświetlona strona zawierająca informacje szczegółowe o wszystkich lokatach powiązanych z danym klientem (patrz: rysunek 14.2).

Rysunek 14.2.

Po zalogowaniu klient
ma dostęp do listy
założonych przez siebie
lokat

Lista lokat

ID	Wysokość lokaty	Czas trwania	E-mail	Operacja
0	10000	24	cust1@nazwa-domeny.pl	Edytuj

Załącz nową lokatę

[Wyloguj](#)

Użytkownik: cust1

Na rysunku 14.2 widać łącze *Wyloguj* umożliwiające klientowi wylogowanie się z aplikacji sieciowej MyBank. Modyfikacja informacji umożliwiający założenie nowej lokaty jest wyświetlany po kliknięciu przycisku *Załącz nową lokatę*. Zwróć uwagę, że nazwa uwierzytelnionego użytkownika jest wyświetlana pod łączem *Wyloguj*.

Gdy do systemu zaloguje się administrator, wówczas aplikacja sieciowa MyBank wyświetli informacje o *wszystkich* lokatach założonych w systemie, jak pokazano na rysunku 14.3.

Lista lokat

ID	Klient	Wysokość lokaty	Czas trwania	E-mail	Operacja
0	cust1	10000	24	cust1@nazwa-domeny.pl	Zamknij
1	cust2	10000	24	cust2@nazwa-domeny.pl	Zamknij

Załącz nową lokatę

[Wyloguj](#)

Użytkownik: admin

Rysunek 14.3. Administratorowi ma dostęp do informacji o wszystkich lokatach założonych w systemie

Na rysunku 14.3 widać, że administrator może zamknąć lokatę, klikając łącze *Zamknij* widoczne obok lokaty. Podobnie jak w przypadku klientów, przycisk *Załącz nową lokatę* jest wyświetlany na stronie, ale próba zapisania informacji szczegółowych o lokacie spowoduje zgłoszenie przez aplikację wyjątku bezpieczeństwa.

Zobaczmy teraz, jak za pomocą frameworku Spring Security spełnić wymagania w zakresie bezpieczeństwa stawiane przez aplikację sieciową MyBank.

IMPORT **Rozdział14/ch14-bankapp-simple-security.** (Ten projekt pokazuje aplikację sieciową MyBank używającą frameworku Spring Security w celu spełnienia wymagań aplikacji w zakresie bezpieczeństwa wymienionych w podrozdziale 14.2).

14.3. Zabezpieczanie aplikacji MyBank za pomocą frameworku Spring Security

Spring Security składa się z wielu modułów pozwalających na spełnienie różnych wymagań aplikacji w zakresie zapewnienia bezpieczeństwa. Najważniejsze z modułów tego frameworku wymieniamy w tabeli 14.1.

Tabela 14.1. Wybrane moduły frameworku Spring Security

Nazwa	Opis
spring-security-core	Definiuje podstawowe klasy i interfejsy frameworku Spring Security. Jest wymagany przez każdą aplikację wykorzystującą Spring Security.
spring-security-web	Zapewnia obsługę dla zabezpieczania aplikacji sieciowych.
spring-security-config	Podobnie jak Spring oferuje schematy tx i mvc, Spring Security definiuje schemat security ułatwiający konfigurację funkcji tego frameworku. Moduł spring-security-config jest odpowiedzialny za analizę elementów przestrzeni nazw security.
spring-security-taglibs	Definiuje znaczniki, za pomocą których można uzyskać dostęp do informacji dotyczących bezpieczeństwa i zabezpieczyć zawartość wyświetlaną na stronach JSP.
spring-security-acl	Pozwala na użycie list ACL w celu zabezpieczenia egzemplarzy obiektów domeny w aplikacji.

W tym podrozdziale skoncentrujemy się na użyciu modułów spring-security-core, spring-security-web, spring-security-config i spring-security-taglibs w celu zabezpieczenia aplikacji sieciowej MyBank. Z dalszej części tego rozdziału dowiesz się, jak używać modułu spring-security-acl do zabezpieczania egzemplarzy obiektów domeny.

Rozpoczynamy od analizy konfiguracji zabezpieczeń żądań sieciowych.

Konfiguracja zabezpieczeń dla żądań sieciowych

Zabezpieczenia żądań sieciowych w aplikacji można wprowadzić na następujące sposoby:

- konfiguracja w pliku *web.xml* oferowanego przez Spring filtru DelegatingFilterProxy;
- włączenie oferowanych przez Spring Security zabezpieczeń dla żądań sieciowych.

Najpierw zobaczymy, jak można skonfigurować filtr DelegatingFilterProxy.

Konfiguracja filtru DelegatingFilterProxy

Moduł web oferowany przez Spring (reprezentowany przez plik *spring-web-4.0.0.RELEASE.jar*) definiuje klasę DelegatingFilterProxy implementującą interfejs Filter API Servlet. Listing 14.1 przedstawia konfigurację filtru DelegatingFilterProxy w pliku *web.xml*.

Element <filter-mapping> określa, że filtr DelegatingFilterProxy jest mapowany na wszystkie przychodzące żądania sieciowe. Nazwa filtru podana w elemencie <filter-name> ma szczególne znaczenie w kontekście filtru DelegatingFilterProxy. Zadanie przetworzenia żądania jest przez

Listing 14.1. *Plik web.xml przedstawiający konfigurację filtra DelegatingFilterProxy*

Projekt: ch14-bankapp-simple-security

Położenie pliku: src/main/webapp/WEB-INF

```

<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

```

klasę DelegatingFilterProxy delegowane do komponentu, którego nazwa została dopasowana do wartości zdefiniowanej w elemencie <filter-name>. W omawianym listingu żądania sieciowe otrzymywane przez filtr DelegatingFilterProxy są przekazywane do komponentu Spring o nazwie springSecurityFilterChain w *głównym* kontekście aplikacji. Wkrótce zobaczysz, że komponent springSecurityFilterChain jest tworzony przez Spring Security.

Skoro mamy skonfigurowany filtr DelegatingFilterProxy, zobaczmy, jak skonfigurować zabezpieczenia żądania sieciowego.

Konfiguracja zabezpieczenia żądania sieciowego

Listing 14.2 przedstawia plik kontekstu aplikacji używający elementu <http> schematu security do konfiguracji zabezpieczeń żądania sieciowego.

Listing 14.2. *Plik applicationContext-security.xml przedstawiający konfigurację zabezpieczeń sieciowych*

Projekt: ch14-bankapp-simple-security

Położenie pliku: src/main/resources/META-INF/spring

```

<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xsi:schemaLocation=".....
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.2.xsd">

  <http use-expressions="true">
    <intercept-url pattern="/*" access="hasAnyRole('ROLE_CUSTOMER', 'ROLE_ADMIN')"/>
    <form-login />
    <logout />
    <remember-me />
    <headers>
      <cache-control/>
      <xss-protection/>
    </headers>
  </http>
  .....
</beans:beans>

```

W powyższym listingu widać, że w pliku XML kontekstu aplikacji znajduje się odwołanie do schematu spring-security-3.2.xsd. Schemat ten znajduje się w pakiecie org.springframework.security.config zdefiniowanym w pliku *spring-security-config-3.2.0.RELEASE.jar*.

Element `<http>` zawiera konfigurację zabezpieczeń żądań sieciowych w aplikacji. Spring Security przetwarza element `<http>` i rejestruje w kontenerze Springa komponent o nazwie `springSecurityFilterChain`. Komponent ten staje się odpowiedzialny za obsługę zabezpieczeń żądań sieciowych. Skonfigurowany wcześniej filtr `DelegatingFilterProxy` (patrz: listing 14.1) zleca komponentowi `springSecurityFilterChain` zadanie obsługi żądań sieciowych. Komponent `springSecurityFilterChain` przedstawia egzemplarz komponentu `FilterChainProxy` (więcej informacji znajdziesz w dokumentacji frameworku Spring Security), który zawiera łańcuch filtrów Servlet dodanych przez subelementy zdefiniowane w elemencie `<http>`.

Atrybut `access` elementu `<intercept-url>` określa wyrażenie Spring EL, które zwróci wartość boolowską. Jeżeli wartością wyrażenia Spring EL będzie `true`, to adresy URL dopasowane przez atrybut `pattern` staną się dostępne dla użytkownika. Z kolei wartość `false` wyrażenia Spring EL oznacza brak dostępu do adresów URL dopasowanych przez atrybut `pattern`. Spring Security dostarcza wiele wbudowanych wyrażen, na przykład `hasRole`, `hasAnyRole`, `isAnonymous` itd.

W listingu 14.2 wyrażenie `hasAnyRole('ROLE_CUSTOMER', 'ROLE_ADMIN')` zwraca wartość `true`, jeżeli uwierzytniony użytkownik ma rolę `ROLE_CUSTOMER` lub `ROLE_ADMIN`. W aplikacji sieciowej `MyBank` `ROLE_CUSTOMER` jest przypisywana klientowi, natomiast `ROLE_ADMIN` — administratorowi. Ponieważ wzorzec `/*` powoduje dopasowanie wszystkich adresów URL, element `<intercept-url>` w listingu 14.2 określa, że tylko użytkownik w roli `ROLE_CUSTOMER` lub `ROLE_ADMIN` może uzyskać dostęp do aplikacji sieciowej `MyBank`. Warto zwrócić uwagę na fakt, że użycie wyrażenia Spring EL w atrybucie `access` jest dozwolone tylko wtedy, gdy atrybutowi `use-expressions` elementu `<http>` przypisano wartość `true`.

Element `<form-login>` konfiguruje stronę logowania przeznaczoną do uwierzytniania użytkowników. Można wykorzystać różne atrybuty elementu `<form-login>`, takie jak `login-page`, `default-target-url` itd., i tym samym dostosować stronę logowania do własnych potrzeb. Atrybut `login-page` określa adres URL używany do wygenerowania strony logowania. Jeżeli atrybut `login-page` nie zostanie podany, to strona logowania zostanie automatycznie wygenerowana pod adresem URL `/spring_security_login`.

Element `<logout>` pozwala na konfigurację przetwarzania funkcji wylogowania oferowanej przez Spring Security. Masz możliwość użycia różnych atrybutów elementu `<logout>`, takich jak `logout-url`, `delete-cookies`, `invalidate-session`, aby skonfigurować niezbędną funkcjonalność wylogowania. Na przykład użycie atrybutu `delete-cookies` umożliwia podanie rozdzielonej przecinkami listy nazw cookies, które powinny zostać usunięte po wylogowaniu użytkownika z aplikacji. Z kolei atrybut `logout-url` pozwala na określenie adresu URL odpowiedzialnego za przetwarzanie procesu wylogowania. Jeżeli atrybut `logout-url` nie zostanie określony, to domyślną wartością atrybutu `logout-url` będzie `/j_spring_security_logout`.

Element `<remember-me>` konfiguruje funkcję „zapamiętaj mnie”. W takim przypadku aplikacja sieciowa zachowuje między sesjami tożsamość uwierzytnionego użytkownika. Kiedy uwierzytnienie użytkownika zakończy się powodzeniem, Spring Security generuje unikalny token, który można zachować w trwałym magazynie danych lub wysłać użytkownikowi w postaci cookie. W listingu 14.2 element `<remember-me>` konfiguruje funkcję „zapamiętaj mnie” za pomocą cookies. Gdy użytkownik ponownie przejdzie do aplikacji sieciowej, token zostanie pobrany z cookies, a użytkownik zostanie automatycznie uwierzytniony.

Element `<headers>` określa nagłówki zabezpieczeń dodawane do odpowiedzi HTTP przez Spring Security. Na przykład kod przedstawiony w listingu 14.2 zawiera element `<cache-control>` konfiguruje dodanie do odpowiedzi nagłówków `Cache-Control`, `Pragma` i `Expires`, natomiast element `<xss-protection>` wskazuje na konieczność dodania nagłówka `X-XSS-Protection`.

Kiedy nieuwierzytlniony użytkownik spróbuje uzyskać dostęp do aplikacji sieciowej MyBank, framework Spring Security wyświetli pokazaną na rysunku 14.1 stronę logowania skonfigurowaną za pomocą elementu `<form-login>`. Zobaczmy teraz, jak wygląda faktyczny proces uwierzytelniania, gdy użytkownik wprowadzi dane uwierzytelniające i kliknie przycisk *Zaloguj*.

Konfiguracja uwierzytelniania

Kiedy użytkownik wprowadzi dane uwierzytelniające i wyśle formularz na stronie logowania, wówczas oferowany przez Spring Security obiekt `AuthenticationManager` staje się odpowiedzialny za przetwarzanie żądania uwierzytelnienia. Egzemplarz `AuthenticationManager` jest skonfigurowany z co najmniej jednym dostawcą `AuthenticationProviders`, który przez `AuthenticationManager` będzie używany do uwierzytelniania użytkowników. Na przykład jeśli użytkownicy mają być uwierzytelniani w serwerze LDAP, można skonfigurować dostawcę `LdapAuthenticationProvider` (implementacja `AuthenticationProvider`) przeprowadzającego uwierzytelnienie w serwerze LDAP.

Schemat security ułatwia konfigurację obiektów `AuthenticationManager` i `AuthenticationProvider`, jak przedstawia listing 14.3.

Listing 14.3. Zawartość pliku `applicationContext-security.xml`

Projekt: `ch14-bankapp-simple-security`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="admin" password="admin" authorities="ROLE_ADMIN" />
      <user name="cust1" password="cust1" authorities="ROLE_CUSTOMER" />
      <user name="cust2" password="cust2" authorities="ROLE_CUSTOMER" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

Element `<authentication-manager>` konfiguruje egzemplarz `AuthenticationManager`. Z kolei element `<authentication-provider>` przeprowadza konfigurację egzemplarza `AuthenticationProvider`. Domyślnie element `<authentication-provider>` zapewnia konfigurację dla dostawcy `DaoAuthenticationProvider` (implementacja `AuthenticationProvider`), który używa oferowanego przez Spring egzemplarza `UserDetailsService` jako obiektu DAO w celu wczytania informacji szczegółowych o użytkowniku.

Dostawca `DaoAuthenticationProvider` używa skonfigurowanego egzemplarza `UserDetailsService` w celu wczytania informacji szczegółowych o użytkowniku z repozytorium, opierając się przy tym na nazwie użytkownika. Dostawca `DaoAuthenticationProvider` przeprowadza uwierzytelnienie, porównując dane uwierzytelnione wprowadzone przez użytkownika z danymi wczytanymi przez

egzemplarz `UserService`. Warto pamiętać, że informacje szczegółowe o użytkowniku egzemplarz `UserService` może wczytywać ze źródła danych, pliku jednorodnego lub innego repozytorium użytkowników.

Element `<user-service>` w elemencie `<authentication-provider>` przeprowadza konfigurację *znajdującego się w pamięci* egzemplarza `UserService`, który wczytuje dane użytkowników zdefiniowane przez elementy `<user>`. W listingu 14.3 element `<user-service>` definiuje trzech użytkowników aplikacji: `admin` (rola `ROLE_ADMIN`), `cust1` (rola `ROLE_CUSTOMER`) i `cust2` (rola `ROLE_CUSTOMER`). Atrybut `name` określa nazwę użytkownika (`username`) przypisaną użytkownikowi, atrybut `password` określa hasło użytkownika, a atrybut `authorities` określa role użytkownika.

Jeżeli teraz wdrożysz projekt *ch14-bankapp-simple-security* i uzyskasz do niego dostęp, przechodząc do adresu URL *http://localhost:8080/ch14-bankapp-simple-security*, wówczas wyświetlona zostanie strona logowania (patrz: rysunek 14.1) aplikacji sieciowej. Po uwierzytelnieniu się przez podanie nazwy użytkownika `cust1` i hasła `cust1` aplikacja sieciowa wyświetli lokaty założone przez użytkownika `cust1` (patrz: rysunek 14.2). Podobnie po zalogowaniu się jako użytkownik `cust2` (hasło `cust2`) aplikacja wyświetli lokaty powiązane z użytkownikiem `cust2`. Z kolei podanie nazwy użytkownika `admin` i hasła `admin` powoduje, że aplikacja sieciowa wyświetli lokaty powiązane z użytkownikami `cust1` i `cust2`.

Zobaczmy teraz, jak można użyć oferowaną przez Spring Security bibliotekę znaczników JSP w celu uzyskania dostępu do informacji o zabezpieczeniach i stosowania ograniczeń dla treści wyświetlanej na stronach JSP.

Zabezpieczenie zawartości strony JSP przez użycie oferowanej przez Spring Security biblioteki znaczników

Jedno z wymagań aplikacji sieciowej *MyBank* polega na tym, że opcja edycji lokaty (patrz: rysunek 14.2) musi być dostępna jedynie dla użytkowników w roli `ROLE_CUSTOMER`. Ponadto opcja zamknięcia lokaty (patrz: rysunek 14.3) ma być dostępna tylko dla użytkowników w roli `ROLE_ADMIN`. Ponieważ łączy *Edytuj* i *Zamknij* musimy zabezpieczyć na podstawie roli uwierzytelnionego użytkownika, aplikacja sieciowa *MyBank* używa oferowanej przez Spring Security biblioteki znaczników do zabezpieczenia zawartości na stronach JSP.

W listingu 14.4 przedstawiamy przykład użycia biblioteki znaczników w celu uzyskania dostępu do nazwy uwierzytelnionego użytkownika i zabezpieczenia zawartości strony JSP na podstawie roli zalogowanego użytkownika.

Listing 14.4. Zawartość pliku *fixedDepositList.jsp*

Projekt: *ch14-bankapp-simple-security*

Położenie pliku: *src/main/webapp/WEB-INF/jsp*

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="security"%>
.....
<body>
.....
    <td style="font-family: 'arial'; font-size: 12px; font-weight: bold" align="right">
        <a href="${pageContext.request.contextPath}/j_spring_security_logout">
```

```

        Wyloguj</a>
    <p>
        Nazwa użytkownika: <security:authentication property="principal.username" />
    </p>
</td>
.....
<td class="td">
    <security:authorize access="hasRole('ROLE_CUSTOMER')">
        <a href="{pageContext.request.contextPath}/fixedDeposit?....." >Edytuj</a>
    </security:authorize>
    <security:authorize access="hasRole('ROLE_ADMIN')">
        <a href="{pageContext.request.contextPath}/fixedDeposit.....">Zamknij</a>
    </security:authorize>
</td>
</body>
</html>

```

W powyższym listingu widać, że łącze *Wyloguj* odwołuje się do adresu URL `${pageContext.request.contextPath}/j_spring_security_logout`. Jak wcześniej wspomnieliśmy, brak zdefiniowanego atrybutu `logout-url` elementu `<logout>` oznacza, że domyślną wartością `logout-url` będzie `/j_spring_security_logout`. Dlatego po kliknięciu łącza *Wyloguj* użytkownik zostanie wylogowany z aplikacji sieciowej MyBank.

W omawianym listingu widać także, że na stronie JSP za pomocą dyrektywy taglib została dołączona biblioteka znaczników oferowana przez Spring Security. Udostępniany przez Spring Security obiekt `Authentication` zawiera informacje o uwierzytelnionym użytkowniku. W obiekcie tym znajdziesz na przykład informacje o nazwie użytkownika podanej w trakcie uwierzytelniania oraz roli przypisanej temu użytkownikowi. Element `<authentication>` wyświetla wskazaną właściwość obiektu `Authentication`. W powyższym listingu właściwość `principal.username` odwołuje się do właściwości `username` uwierzytelnionego użytkownika.

Element `<authorize>` zabezpiecza umieszczoną w nim zawartość JSP, opierając się na wyniku obliczenia wyrażenia zabezpieczeń zdefiniowanego przez atrybut `access`. Jeżeli wynikiem wyrażenia będzie `true`, to zawartość JSP zostanie wygenerowana. W przeciwnym razie zawartość nie jest generowana. W omawianym listingu wartością wyrażenia `hasRole('ROLE_CUSTOMER')` jest `true`, jeśli uwierzytelniony użytkownik ma przypisaną rolę `ROLE_CUSTOMER`. Z kolei wyrażenie `hasRole('ROLE_ADMIN')` zwraca wartość `true`, jeśli rola uwierzytelnionego użytkownika to `ROLE_ADMIN`. W powyższym listingu wyrażenie `hasRole` zostało użyte w taki sposób, aby łącze *Edytuj* było wyświetlane jedynie użytkownikowi w roli `ROLE_CUSTOMER`, natomiast łącze *Zamknij* jedynie użytkownikowi w roli `ROLE_ADMIN`.

Zobaczmy teraz, jak za pomocą Spring Security można zastosować zabezpieczenia na poziomie metody.

Zabezpieczanie metod

Jednym z wymagań aplikacji MyBank jest to, że użytkownik w roli `ROLE_ADMIN` widzi wyświetlony przycisk *Załącz nową lokatę* (patrz: rysunek 14.3), ale próba zapisu informacji szczegółowych o lokacie powoduje zgłoszenie wyjątku bezpieczeństwa. To doskonały przykład zabezpieczenia metody

`saveFixedDeposit()` klasy `FixedDepositService` w taki sposób, aby jedynie użytkownik w roli `ROLE_CUSTOMER` mógł ją wywoływać.

Chcemy także zabezpieczyć inne metody klasy `FixedDepositService`, aby nie mogły być wywoływane przez nieupoważnionych do tego użytkowników. Na przykład użytkownik `cust1` zalogowany w roli `ROLE_CUSTOMER` może wywoływać metodę `closeFixedDeposit()` klasy `FixedDepositService` w celu zamknięcia lokaty, wprowadzając w przeglądarce internetowej następujący adres URL:

```
http://localhost:8080/ch14-bankapp-simple-security/  
fixedDeposit?fdAction=close&fixedDepositId=<fixed-fixed-id>
```

Fragment `<fixed-deposit-id>` w powyższym adresie URL to identyfikator lokaty przeznaczonej do zamknięcia, jak pokazuje rysunek 14.4.

Rysunek 14.4.
*Identyfikator lokaty
jest wyświetlany
w kolumnie ID*

[Wyloguj](#)Użytkownik: `cust1`

Lista lokat

ID	Wysokość lokaty	Czas trwania	E-mail	Operacja
0	10000	24	cust1@nazwa-domeny.pl	Edytuj

[Załącz nową lokatę](#)

Identyfikator lokaty

Aby dodać do aplikacji zabezpieczenia na poziomie metody, należy wykonać następujące zadania:

- skonfigurować w aplikacji zabezpieczenia na poziomie metody, używając elementu `<global-method-security>` schematu `security`;
- dodać adnotacje `@Secured` do metod, które mają być chronione przed nieupoważnionym dostępem.

Najpierw przyjrzymy się elementowi `<global-method-security>`.

Konfiguracja zabezpieczeń na poziomie metody z użyciem elementu `<global-method-security>`

W listingu 14.5 przedstawiamy przykład użycia elementu `<global-method-security>`.

Listing 14.5. Zawartość pliku `applicationContext-security.xml`

Projekt: `ch14-bankapp-simple-security`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans:beans xmlns="http://www.springframework.org/schema/security"  
.....>  
  <global-method-security secured-annotations="enabled" />  
</beans:beans>
```

Element `<global-method-security>` konfiguruje zabezpieczenia na poziomie. Element `<global-method-security>` ma zastosowanie jedynie do kontekstu aplikacji, w którym został zdefiniowany. Na przykład jeśli element `<global-method-security>` jest zdefiniowany w pliku XML głównego

kontekstu aplikacji sieciowej, wówczas ma zastosowanie jedynie dla komponentów zarejestrowanych w egzemplarzu `WebApplicationContext`. W projekcie *ch14-bankapp-simple-security* plik *applicationContext-security.xml* (przedstawiony w powyższym listingu) i plik *applicationContext.xml* (definiujący usługi i obiekty DAO) stanowią pliki XML tworzące główny kontekst aplikacji sieciowej (patrz: plik *web.xml* w projekcie *ch14-bankapp-simple-security*). Dlatego element `<global-method-security>` ma zastosowanie tylko dla komponentów zdefiniowanych w wymienionych plikach XML kontekstu aplikacji.

Atrybut `secured-annotations` elementu `<global-method-security>` określa, czy użycie oferowanej przez Spring adnotacji `@Secured` powinno być włączone (lub wyłączone) dla komponentów zarejestrowanych w kontenerze Springa. Wartością tego atrybutu jest `enabled`, dlatego można użyć adnotacji `@Secured` do wskazania metod, które są zabezpieczone.



Jeżeli chcesz zabezpieczyć metody kontrolera, zdefiniuj element `<global-method-security>` w pliku XML kontekstu aplikacji sieciowej zamiast w pliku XML głównego kontekstu aplikacji sieciowej.

Przechodzimy teraz do zabezpieczania metod za pomocą oferowanej przez Spring adnotacji `@Secured`.

Użycie adnotacji `@Secured` do zdefiniowania zabezpieczeń dla metod komponentu

W listingu 14.6 przedstawiamy przykład użycia adnotacji `@Secured` do zdefiniowania zabezpieczeń dla metod komponentu.

Listing 14.6. Interfejs *FixedDepositService*

Projekt: *ch14-bankapp-simple-security*

Położenie pliku: *src/main/java/sample/spring/chapter14/service*

```
package sample.spring.chapter14.service;

import org.springframework.security.access.annotation.Secured;
.....
public interface FixedDepositService {
    .....
    @Secured("ROLE_CUSTOMER")
    void saveFixedDeposit(FixedDepositDetails fixedDepositDetails);
    .....
    @Secured("ROLE_ADMIN")
    void closeFixedDeposit(int fixedDepositId);

    @Secured("ROLE_CUSTOMER")
    void editFixedDeposit(FixedDepositDetails fixedDepositDetails);
}
```

W powyższym listingu został przedstawiony interfejs `FixedDepositService` definiujący metody, które mogą wykonywać operacje na lokatach. Adnotacja `@Secured("ROLE_CUSTOMER")` w metodach `saveFixedDeposit()` i `editFixedDeposit()` określa, że wymienione metody mogą być wywoływane

jedynie przez użytkownika w roli `ROLE_CUSTOMER`. Adnotacja `@Secured("ROLE_ADMIN")` w metodzie `closeFixedDeposit()` wskazuje, że ta metoda może być wywoływana tylko przez użytkownika w roli `ROLE_ADMIN`.



Domyślnie zabezpieczenia na poziomie metody są oparte na Spring AOP. Jeżeli zamiast Spring AOP chcesz użyć AspectJ, atrybutowi mode elementu `<global-method-security>` przypisz wartość `aspectj`. Ponadto dodaj moduł `spring-security-aspects` do projektu i adnotacją `@Secured` określ klasę, a nie interfejs.

Zamiast używać adnotacji `@Secured` możesz wykorzystać oferowaną przez Spring adnotację `@PreAuthorize` i za jej pomocą zdefiniować zabezpieczenia w metodzie. W przeciwieństwie do adnotacji `@Secured` adnotacja `@PreAuthorize` akceptuje wyrażenia zabezpieczeń, takie jak `hasRole`, `hasAnyRole` itd. Aby włączyć użycie adnotacji `@PreAuthorize`, atrybutowi `pre-post-annotations` elementu `<global-method-security>` przypisz wartość `enabled`. W listingu 14.7 przedstawiamy przykład użycia adnotacji `@PreAuthorize`.

Listing 14.7. Przykład użycia adnotacji `@PreAuthorize`

```
import org.springframework.security.access.prepost.PreAuthorize;
.....
public interface SomeService {
    .....
    @PreAuthorize("hasRole('ROLE_XYZ')")
    void doSomething(.....);
    .....
}
```

W powyższym przykładzie adnotacja `@PreAuthorize` określa, że metoda `doSomething()` jest dostępna jedynie dla użytkowników w roli `ROLE_XYZ`.

Spring Security obsługuje także adnotacje zabezpieczeń, takie jak `@RolesAllowed`, `@DenyAll`, `@PermitAll` itd., zdefiniowane przez specyfikację JSR-250 (Common Annotations). Aby włączyć użycie adnotacji zabezpieczeń JSR-250, atrybutowi `jsr250-annotations` elementu `<global-method-security>` przypisz wartość `enabled`. W listingu 14.8 przedstawiamy przykład użycia adnotacji `@RolesAllowed`.

Listing 14.8. Przykład użycia adnotacji `@RolesAllowed`

```
import javax.annotation.security.RolesAllowed;
.....
public interface SomeService {
    .....
    @RolesAllowed("ROLE_XYZ")
    void doSomething(.....);
    .....
}
```

W powyższym listingu adnotacja `@RolesAllowed` określa, że metoda `doSomething()` jest dostępna jedynie dla użytkowników w roli `ROLE_XYZ`.



We wcześniejszej części książki poznałeś adnotacje JSR 250, takie jak `@PreDestroy`, `@PostConstruct` itd. Są one częścią Javy SE 6 i nowszych wersji. Ponieważ adnotacje zabezpieczeń związane z JSR 250 nie stanowią części Javy SE, do projektu trzeba dodać plik JAR o nazwie *jsr250-api*, aby można było używać adnotacji takich jak `@RolesAllowed`, `@PermitAll` itd.

Z tego podrozdziału dowiedziałeś się, jak używać frameworku Spring Security do uwierzytelniania użytkowników, zabezpieczania żądań sieciowych oraz implementacji zabezpieczeń na poziomie metody. Przechodzimy teraz do udostępnianego przez Spring Security modułu ACL do zabezpieczania egzemplarzy obiektów domeny.

IMPORT **Rozdział 14/ch14-bankapp-db-security.** (Ten projekt pokazuje aplikację sieciową MyBank używającą modułu ACL frameworku Spring Security do zabezpieczania egzemplarzy `FixedDepo` `sitDetails`).

14.4. Zabezpieczanie egzemplarzy `FixedDepositDetails` za pomocą modułu ACL

Projekt *ch14-bankapp-db-security* przedstawia wariant aplikacji sieciowej MyBank używający oferowanego przez Spring Security modułu ACL do zabezpieczenia egzemplarzy `FixedDepositDetails`.

Przechodzimy zatem do wdrożenia i użycia projektu *ch14-bankapp-db-security*.

Wdrożenie i użycie projektu *ch14-bankapp-db-security*

Projekt *ch14-bankapp-db-security* używa bazy danych MySQL do przechowywania informacji o użytkownikach aplikacji, lokatach i listach ACL. Przed wdrożeniem projektu *ch14-bankapp-db-security* w MySQL utwórz bazę danych o nazwie `securitydb` oraz wykonaj skrypt *bankapp.sql* umieszczony w katalogu *scripts* projektu *ch14-bankapp-db-security*.

Wykonanie skryptu *bankapp.sql* spowoduje utworzenie następujących tabel: `ACL_CLASS`, `ACL_ENTRY`, `ACL_OBJECT_IDENTITY`, `ACL_SID`, `FIXED_DEPOSIT_DETAILS`, `AUTHORITIES` i `USERS`. Tabele o nazwach rozpoczynających się od `ACL_` przechowują informacje dotyczące ACL (więcej na temat tych tabel dowiesz się w dalszej części rozdziału). Tabela `FIXED_DEPOSIT_DETAILS` zawiera informacje o lokatach. Z kolei tabele `USERS` i `AUTHORITIES` przechowują informacje, odpowiednio, o użytkownikach i rolach. Skrypt *bankapp.sql* wstawia również dane konfiguracyjne do tabel `USERS`, `AUTHORITIES`, `ACL_CLASS` i `ACL_SID`.

Po przygotowaniu bazy danych dla projektu *ch14-bankapp-db-security* możesz go wdrożyć w osadzonym serwerze Tomcat 7, wydając polecenie `tomcat7:run` z poziomu katalogu projektu (w dodatku A znajdziesz więcej informacji dotyczących wdrażania projektów sieciowych w osadzonym serwerze Tomcat 7). Po zakończonej powodzeniem operacji wdrożenia projektu przejdź do adresu URL <http://localhost:8080/ch14-bankapp-db-security>. Powinieneś zobaczyć stronę logowania, jak na rysunku 14.5.

Rysunek 14.5.
Strona logowania
w aplikacji sieciowej
MyBank

Nazwa użytkownika:

Hasło:

Domyślnie w aplikacji sieciowej MyBank zostało skonfigurowanych trzech następujących użytkowników: cust1 (rola ROLE_CUSTOMER), cust2 (rola ROLE_CUSTOMER) i admin (rola ROLE_ADMIN). Kiedy zalogujesz się, podając nazwę użytkownika cust1 i hasło cust1, wówczas zostaną wyświetlone lokaty założone przez użytkownika cust1, jak na rysunku 14.6.

Rysunek 14.6.
Lista lokat założonych
przez klienta cust1

[Wyloguj](#)

Użytkownik: cust1

Lista lokat

ID	Wysokość lokaty	Czas trwania	E-mail	Operacja

Ponieważ klient cust1 nie założył jeszcze żadnej lokaty, lista pokazana na rysunku 14.6 jest pusta. Kliknięcie przycisku *Założ nową lokatę* powoduje wyświetlenie formularza umożliwiającego założenie nowej lokaty. Jeżeli teraz założysz nową lokatę, pojawi się ona na liście, jak na rysunku 14.7.

[Wyloguj](#)

Użytkownik: cust1

Lista lokat

ID	Wysokość lokaty	Czas trwania	E-mail	Operacja
14	1200	12	cust1@nazwa-domeny.pl	Edytuj Udostępnij administratorowi

Rysunek 14.7. Klient może edytować lokaty, a także umożliwiać administratorowi uzyskanie do nich dostępu

Widoczny na rysunku 14.7 przycisk *Edytuj* pozwala klientowi na edycję informacji o lokatach, natomiast opcja *Udostępnij administratorowi* powoduje, że użytkownik admin uzyskuje dostęp do danej lokaty. Użytkownik admin ma jedynie dostęp do tych lokat, które zostały mu udostępnione przez klientów. Kliknij więc łącze *Udostępnij administratorowi* i tym samym udziel użytkownikowi admin dostępu do lokaty.

Teraz wyloguj się z aplikacji sieciowej MyBank i zaloguj ponownie, ale tym razem podaj nazwę użytkownika admin i hasło admin. Jak widać na rysunku 14.8, użytkownik admin ma wyświetloną listę wszystkich lokat, które zostały mu udostępnione.

Lista lokat

ID	Klient	Wysokość lokaty	Czas trwania	E-mail	Operacja
14	cust1	1200	12	cust1@nazwa-domeny.pl	Zamknij

[Założ nową lokatę](#)

Rysunek 14.8. Użytkownik *admin* może zamknąć lokatę, wybierając opcję *Zamknij*

Na rysunku 14.8 widać, że administrator może wybrać opcję *Zamknij* i tym samym zamknąć daną lokatę. Zamknięcie lokaty powoduje usunięcie z tabeli `FIXED_DEPOSIT_DETAILS` informacji o tej lokacie.

Podsumowując: do aplikacji sieciowej *MyBank* możesz się zalogować za pomocą danych uwierzytelniających `cust1/cust1`, `cust2/cust2` i `admin/admin`, aby uzyskać dostęp do następujących jej funkcji:

- Tylko użytkownicy `cust1` (rola `ROLE_CUSTOMER`) i `cust2` (rola `ROLE_CUSTOMER`) mogą zakładać lokaty.
- Użytkownicy `cust1` i `cust2` mogą modyfikować jedynie należące do nich lokaty. Na przykład `cust1` nie może modyfikować lokaty założonej przez `cust2`.
- Użytkownicy `cust1` i `cust2` mogą udostępniać administratorowi jedynie założone przez siebie lokaty. Na przykład `cust1` nie może udostępnić administratorowi lokaty założonej przez `cust2`.
- Użytkownik `admin` (rola `ROLE_ADMIN`) ma wyświetlane tylko te lokaty, które zostały mu udostępnione przez użytkowników `cust1` i `cust2`.
- Tylko administrator może zamykać lokaty.

Zanim zagłębimy się w szczegóły implementacji aplikacji sieciowej *MyBank*, warto przeanalizować standardowe tabele bazy danych wymagane przez framework *Spring Security* do przechowywania informacji o listach `ACL` i użytkownikach.

Tabele bazy danych przeznaczone do przechowywania informacji o listach `ACL` i użytkownikach

Oferowany przez *Spring Security* moduł `ACL` umożliwia zabezpieczanie egzemplarzy obiektów domeny, czyli `FixedDepositDetails`. W tabelach wymaganych przez *Spring Security* (`ACL_CLASS`, `ACL_ENTRY`, `ACL_OBJECT_IDENTITY` i `ACL_SID`) znajdują się informacje o uprawnieniach stosowanych dla lokat, których dane są przechowywane w tabeli `FIXED_DEPOSIT_DETAILS`. Po uzyskaniu dostępu do egzemplarza `FixedDepositDetails` moduł `ACL` sprawdza, czy uwierzytelniony użytkownik ma wystarczające uprawnienia do pracy z danym egzemplarzem `FixedDepositDetails`.

Przeanalizujemy teraz wszystkie tabele, które są używane przez *Spring Security* do przechowywania informacji `ACL`.

Tabela ACL_CLASS

Tabela ACL_CLASS zawiera w pełni kwalifikowaną nazwę klas domeny, której egzemplarze mają być zabezpieczone w aplikacji. W przypadku aplikacji sieciowej MyBank tabela ACL_CLASS zawiera w pełni kwalifikowaną nazwę klasy FixedDepositDetails, jak pokazano na rysunku 14.9.

Rysunek 14.9.
Tabela ACL_CLASS

 id	class
1	sample.spring.chapter14.domain.FixedDepositDetails

Opis kolumn tabeli ACL_CLASS

- id — zawiera klucz podstawowy.
- class — w pełni kwalifikowana nazwa klasy domeny, której egzemplarze mają być zabezpieczone.

Tabela ACL_SID

Tabela ACL_SID (SID oznacza *security identity*) zawiera nazwy użytkowników lub role w systemie. W przypadku aplikacji sieciowej MyBank tabela ACL_SID zawiera nazwy użytkowników admin, cust1 i cust2, jak pokazano na rysunku 14.10.

Rysunek 14.10.
Tabela ACL_SID

 id	principal	sid
1	true	cust1
2	true	cust2
3	true	admin

Opis kolumn tabeli ACL_SID

- id — zawiera klucz podstawowy.
- principal — wskazuje rodzaj informacji przechowywanych przez kolumnę sid: rola lub nazwa użytkownika. Wartość true oznacza, że kolumna sid przechowuje nazwę użytkownika, natomiast wartość false oznacza, że kolumna sid przechowuje rolę.
- sid — zawiera nazwę użytkownika lub rolę.

Tabela ACL_OBJECT_IDENTITY

Tabela ACL_OBJECT_IDENTITY zawiera identyfikatory obiektów domeny przeznaczonych do zabezpieczenia. W przypadku aplikacji sieciowej MyBank tabela ACL_OBJECT_IDENTITY przechowuje identyfikatory lokat, których dane znajdują się w tabeli FIXED_DEPOSIT_DETAILS, jak pokazano na rysunku 14.11.

Odwołuje się do kolumny id w tabeli ACL_CLASS

Odwołuje się do kolumny id w tabeli FIXED_DEPOSIT_DETAILS

Odwołuje się do kolumny id w tabeli ACL_SID

 id	object_id_class	object_id_identity	parent_object	owner_sid	entries_inheriting
12	1	14	(null)	1	true

Rysunek 14.11. Tabela ACL_OBJECT_IDENTITY

Na rysunku 14.11 widać, że kolumna `object_id_identity` zawiera identyfikatory lokat, których dane są przechowywane w tabeli `FIXED_DEPOSIT_DETAILS`.

Opis kolumn tabeli `ACL_OBJECT_IDENTITY`

- `id` — zawiera klucz podstawowy.
- `object_id_class` — odwołuje się do klasy domeny zdefiniowanej w tabeli `ACL_CLASS`.
- `object_id_identity` — odwołuje się do egzemplarza obiektu domeny zdefiniowanego w tabeli `FIXED_DEPOSIT_DETAILS`.
- `parent_object` — jeżeli dla obiektu domeny wskazywanego przez kolumnę `object_id_identity` istnieje obiekt nadrzędny, wówczas w tej kolumnie znajdzie się identyfikator obiektu nadrzędnego.
- `owner_sid` — odwołuje się do użytkownika lub roli użytkownika będącego właścicielem egzemplarza obiektu domeny.
- `entries_inheriting` — zawiera opcję wskazującą, czy obiekt dziedziczy po obiekcie nadrzędnym wpisy dotyczące ACL.

Tabela `ACL_ENTRY`

Tabela `ACL_ENTRY` zawiera uprawnienia (odczyt, zapis, tworzenie itd.) nadawane użytkownikom w obiektach domeny. W przypadku aplikacji sieciowej MyBank tabela `ACL_ENTRY` zawiera uprawnienia przypisywane użytkownikom dla lokat, których dane są przechowywane w tabeli `FIXED_DEPOSIT_DETAILS`, jak pokazano na rysunku 14.12.

Odwołuje się do kolumny `id` w tabeli `ACL_OBJECT_IDENTITY`

Odwołuje się do kolumny `id` w tabeli `ACL_SID`

Określa uprawnienia (odczyt, zapis itd.) nadawane użytkownikowi

 id	acl_object_identity	ace_order	sid	mask	granting	audit_success	audit_failure
768	12	0	1	1	true	false	false
769	12	1	1	2	true	false	false
770	12	2	3	1	true	false	false
771	12	3	3	16	true	false	false
772	12	4	3	8	true	false	false

Rysunek 14.12. Tabela `ACL_ENTRY`

Na rysunku 14.12 widać, że kolumny `acl_object_identity`, `mask` i `sid` określają uprawnienia nadawane użytkownikowi (lub roli) w egzemplarzu obiektu domeny. Warto w tym miejscu dodać, że rekord tabeli `ACL_ENTRY` jest często określany mianem ACE (ang. *Access Control Entry*).

Opis kolumn tabeli `ACL_ENTRY`

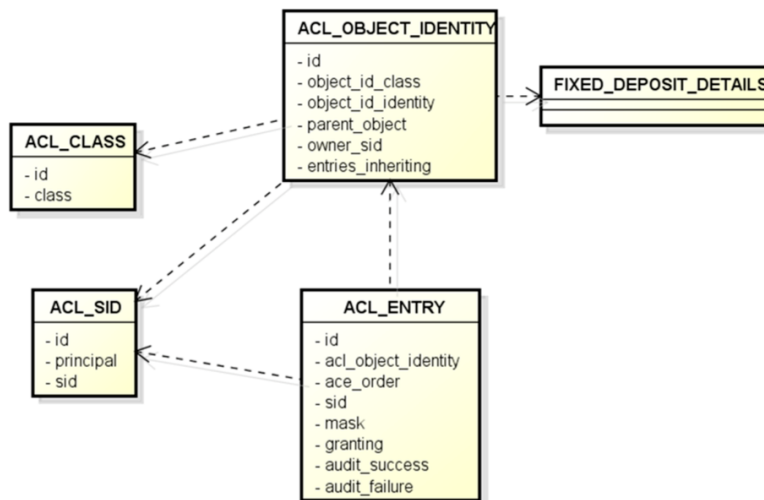
- `id` — zawiera klucz podstawowy.
- `acl_object_identity` — odwołuje się do kolumny `id` tabeli `ACL_OBJECT_IDENTITY`, która z kolei identyfikuje egzemplarz obiektu domeny.
- `ace_order` — określa kolejność wpisów dotyczących kontroli dostępu.
- `sid` — odwołuje się do kolumny `id` tabeli `ACL_SID`, która z kolei identyfikuje użytkownika (lub rolę).

- mask** — określa uprawnienia (odczyt, zapis, tworzenie itd.) nadawane użytkownikowi (lub roli). Wartość 1 wskazuje na odczyt, 2 na zapis, 8 na usunięcie, a 16 na uprawnienie administracyjne.
- granting** — zawiera opcję wskazującą, czy wartość kolumny **mask** określa uprawnienia dostępu, czy raczej brak dostępu. Na przykład jeśli wartością kolumny **mask** będzie 1, a wartością kolumny **granting** będzie **true**, oznacza to, że wskazany SID ma uprawnienie odczytu. Jeżeli jednak wartością kolumny **mask** będzie 1, a wartością kolumny **granting** będzie **false**, oznacza to, że wskazany SID *nie* ma uprawnienia odczytu.
- audit_success** — zawiera opcję wskazującą, czy audyt zakończył się pomyślnie. W dalszej części rozdziału zobaczysz, że oferowana przez Spring Security klasa `ConsoleAuditLogger` może być używana do rejestracji zakończonej pomyślnie operacji nadania uprawnień.
- audit_failure** — zawiera opcję wskazującą, czy audyt zakończył się niepomyślnie. W dalszej części rozdziału zobaczysz, że oferowana przez Spring Security klasa `ConsoleAuditLogger` może być używana do rejestracji zakończonej niepomyślnie operacji nadania uprawnień.

Diagram na rysunku 14.13 pokazuje relacje zachodzące między tabelami ACL. Strzałki przedstawiają odwołania do kluczy zewnętrznych. Na przykład tabela `ACL_OBJECT_IDENTITY` zawiera klucze zewnętrzne odwołujące się to tabel `ACL_CLASS`, `ACL_SID` i `FIXED_DEPOSIT_DETAILS`.

Rysunek 14.13.

Tabele ACL
i zachodzące
między nimi relacje
(strzałki przedstawiają
odwołania kluczy
zewnętrznych z tabel)



Skoro poznałeś tabele ACL wymagane do przechowywania informacji związanych z ACL, możemy przejść do analizy tabel używanych przez Spring Security do przechowywania informacji o użytkownikach i ich rolach.

Tabela USERS

W tabeli USERS znajdują się dane uwierzytelniające użytkowników, jak pokazano na rysunku 14.14.

Rysunek 14.14.

Tabela USERS

username	password	enabled
admin	admin	true
cust1	cust1	true
cust2	cust2	true

Opis kolumn tabeli USERS

username — zawiera nazwę użytkownika.

password — zawiera hasło użytkownika.

enabled — zawiera opcję wskazującą, czy konto danego użytkownika jest włączone czy wyłączone.

Tabela AUTHORITIES

W tabeli AUTHORITIES znajdują się role przypisane poszczególnym użytkownikom zdefiniowanym w tabeli USERS, jak pokazano na rysunku 14.15.

Rysunek 14.15.

Tabela AUTHORITIES

username	authority
admin	ROLE_ADMIN
cust1	ROLE_CUSTOMER
cust2	ROLE_CUSTOMER

Opis kolumn tabeli AUTHORITIES

username — zawiera nazwę użytkownika

authority — zawiera rolę przypisaną użytkownikowi.

Zobaczmy teraz, jak użytkownicy są uwierzytelniani w aplikacji sieciowej MyBank.

Uwierzytelnianie użytkownika

Aplikacja sieciowa MyBank wyraźnie konfiguruje egzemplarz `UserDetailsService` przeznaczony do wczytywania z tabel USERS i AUTHORITIES informacji szczegółowych o użytkowniku, jak przedstawia listing 14.9.

Listing 14.9. Zawartość pliku `applicationContext-security.xml`

Projekt: `ch14-bankapp-db-security`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<authentication-manager>
  <authentication-provider user-service-ref="userDetailsService" />
</authentication-manager>

<beans:bean id="userDetailsService"
class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
  <beans:property name="dataSource" ref="dataSource" />
</beans:bean>
```

W powyższym listingu atrybut `user-service-ref` elementu `<authentication-provider>` odwołuje się do implementacji egzemplarza `UserDetailsService` odpowiedzialnego za wczytywanie danych użytkownika (i jego roli) na podstawie podanej nazwy użytkownika. `JdbcDaoImpl` to implementacja egzemplarza `UserDetailsService` wczytującego dane użytkownika (i jego roli) ze źródła danych (wskazywanego przez właściwość `dataSource`) za pomocą zapytań JDBC. W pliku *application-Context.xml* projektu *ch14-bankapp-db-security* znajdziesz definicję komponentu `dataSource`. Domyślnie implementacja `JdbcDaoImpl` wczytuje dane użytkownika z tabeli `USERS` (patrz: rysunek 14.14), natomiast informacje o jego roli — z tabeli `AUTHORITIES` (patrz: rysunek 14.15). Jeżeli masz własne tabele bazy danych zawierające informacje o użytkownikach i ich rolach oraz uprawnieniach, przypisz właściwościom `usersByUsernameQuery` i `authoritiesByUsernameQuery` egzemplarza `JdbcDaoImpl` odpowiednie wartości pozwalające na pobranie wymaganych informacji.

Właściwość `usersByUsernameQuery` definiuje zapytanie SQL używane do pobierania informacji szczegółowych o użytkowniku na podstawie jego nazwy. Jeżeli dane są przechowywane w tabeli `MY_USERS` zawierającej kolumny `USERNAME` i `PASSWORD`, wówczas poniższe zapytanie SQL można wykorzystać jako wartość właściwości `usersByUsernameQuery`:

```
select USERNAME, PASSWORD, 'true' as ENABLED from MY_USERS where USERNAME = ?
```

Warto pamiętać, że kolumnami zwróconymi przez powyższe zapytanie SQL muszą być `USERNAME`, `PASSWORD` i `ENABLED`. Jeżeli określona kolumna (np. `ENABLED`) nie istnieje w tabeli bazy danych, zwracana jest wartość domyślna (np. `true`) dla tej kolumny.

Właściwość `authoritiesByUsernameQuery` definiuje zapytanie SQL używane do pobierania informacji o roli użytkownika na podstawie jego nazwy. Jeżeli dane są przechowywane w tabeli `MY_AUTHORITIES` zawierającej kolumny `USER` i `ROLE`, poniższe zapytanie SQL można wykorzystać jako wartość właściwości `authoritiesByUsernameQuery`:

```
select USER AS USERNAME, ROLE AS AUTHORITY from MY_AUTHORITIES where USER = ?
```

Warto pamiętać, że kolumnami zwróconymi przez powyższe zapytanie SQL muszą być `USERNAME` i `AUTHORITY`.

Jeżeli aplikacja przechowuje w bazie danych hasła w postaci zaszyfrowanej, można użyć elementu `<password-encoder>` w elemencie `<authentication-provider>` i wskazać egzemplarz (implementację oferowanego przez Spring interfejsu `PasswordEncoder`) stosowany do konwersji wprowadzonego przez użytkownika hasła na jego zaszyfrowaną postać. `BCryptPasswordEncoder` to przykład implementacji `PasswordEncoder` używającej algorytmu szyfrowania `BCrypt` (<http://en.wikipedia.org/wiki/BCrypt>). Dostawca `DaoAuthenticationProvider` wykorzystuje skonfigurowany egzemplarz `PasswordEncoder` do zaszyfrowania wprowadzonego hasła i porównania otrzymanej wartości z pobraną przez `UserDetailsService`.

Przechodzimy teraz do konfiguracji zabezpieczeń żądań sieciowych w aplikacji sieciowej `MyBank`.

Zabezpieczenie żądań sieciowych

W listingu 14.10 przedstawiamy przykład konfiguracji zabezpieczeń żądań sieciowych zastosowanej w aplikacji sieciowej `MyBank`.

Listing 14.10. Zawartość pliku *applicationContext-security.xml* przedstawiająca konfigurację zabezpieczeń sieciowych

Projekt: *ch14-bankapp-db-security*

Położenie pliku: *src/main/resources/META-INF/spring*

```
<http use-expressions="true">
  <access-denied-handler error-page="/access-denied" />
  <intercept-url pattern="/fixedDeposit/*"
    access="hasAnyRole('ROLE_CUSTOMER', 'ROLE_ADMIN')"/>
  <form-login login-page="/login"
    authentication-failure-handler-ref="authFailureHandler" />
  <logout />
  ....
</http>

<beans:bean id="authFailureHandler"
  class="sample.spring.chapter14.security.MyAuthFailureHandler" />
```

Jeżeli konfigurację zabezpieczeń żądań sieciowych przedstawioną w powyższym listingu porównasz z zastosowaną w projekcie *ch14-bankapp-simple-security* (patrz: listing 14.2), zauważysz, że w omawianej konfiguracji dodano kilka informacji.

Atrybut `error-page` elementu `<access-denied-handler>` określa stronę błędu (patrz: plik *scr/main/webapp/WEB-INF/jsp/access-denied.jsp*), na którą zostanie przekierowany uwierzytelniony użytkownik, jeśli spróbuje uzyskać dostęp do nieautoryzowanej dla niego strony internetowej. Atrybut `login-page` elementu `<form-login>` wskazuje adres URL strony logowania. Wartość `/login` tego atrybutu jest mapowana na kontroler `LoginController` (patrz: klasa `LoginController` w projekcie *ch14-bankapp-db-security*) generujący stronę logowania (patrz: plik *scr/main/webapp/WEB-INF/jsp/login.jsp*). Z kolei atrybut `authentication-failure-handler-ref` odwołuje się do komponentu `AuthenticationFailureHandler` odpowiedzialnego za obsługę nieudanych prób uwierzytelniania. Jak przedstawiliśmy w omawianym listingu, egzemplarz `MyAuthFailureHandler` (implementacja `AuthenticationFailureHandler`) jest w aplikacji sieciowej `MyBank` odpowiedzialny za obsługę nieudanych prób uwierzytelniania. W listingu 14.11 przedstawiamy implementację klasy `MyAuthFailureHandler`.

Listing 14.11. Klasa *MyAuthFailureHandler*

Projekt: *ch14-bankapp-db-security*

Położenie pliku: *src/main/java/sample/spring/chatper14/security*

```
package sample.spring.chapter14.security;
.....
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.authentication.AuthenticationFailureHandler;

public class MyAuthFailureHandler implements AuthenticationFailureHandler {

    @Override
    public void onAuthenticationFailure(HttpServletRequest request,
        HttpServletResponse response, AuthenticationException exception)
        throws IOException, ServletException {
```



```
request.setAttribute("exceptionMsg", exception.getMessage());
response.sendRedirect(request.getContextPath() + "/login?exceptionMsg=" +
    exception.getMessage());
}
}
```

Interfejs `AuthenticationFailureHandler` definiuje metodę `onAuthenticationFailure()` wywoływaną w przypadku nieudanego uwierzytelnienia. Metoda `onAuthenticationFailure()` akceptuje egzemplarz `AuthenticationException` przedstawiający nieudane logowanie. W omawianym listingu metoda `onAuthenticationFailure()` przekierowuje użytkownika na stronę logowania, a komunikat wyjątku jest przekazywany za pomocą parametru zapytania. Jeżeli na stronie logowania aplikacji sieciowej `MyBank` wprowadzisz nieprawidłowe dane uwierzytelniające (lub użytkownika, którego konto jest wyłączone w systemie), zauważysz wywołanie metody `onAuthenticationFailure()` klasy `MyAuthFailureHandler`. Na przykład po podaniu nieprawidłowych danych uwierzytelniających zobaczysz komunikat Nieprawidłowe dane uwierzytelniające.

Przechodzimy teraz do konfiguracji ACL w aplikacji sieciowej `MyBank`.

Konfiguracja `JdbcMutableAclService`

Ponieważ uprawnienia ACL są przechowywane w tabelach bazy danych, aplikacja sieciowa `MyBank` używa oferowanego przez Spring egzemplarza `JdbcMutableAclService` do wykonywania operacji CRUD (ang. *create, read, update, delete*) na uprawnieniach ACL w tabelach. Listing 14.12 przedstawia konfigurację `JdbcMutableAclService`.

Listing 14.12. Plik `applicationContext-security.xml` przedstawiający konfigurację `JdbcMutableAclService`

Projekt: `ch14-bankapp-db-security`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans:bean id="aclService"
    class="org.springframework.security.acls.jdbc.JdbcMutableAclService">
    <beans:constructor-arg ref="dataSource" />
    <beans:constructor-arg ref="lookupStrategy" />
    <beans:constructor-arg ref="aclCache" />
</beans:bean>
```

Powyższy listing przedstawia odwołania do komponentów `dataSource`, `lookupStrategy` i `aclCache` przekazywanych konstruktorowi egzemplarza `JdbcMutableAclService`. Zobaczmy teraz, jak skonfigurowane są zależności (`dataSource`, `lookupStrategy` i `aclCache`) egzemplarza `JdbcMutableAclService`.

Komponent `dataSource` wskazuje egzemplarz `javax.sql.DataSource` przechowujący tabele ACL (więcej informacji znajdziesz w definicji komponentu `dataSource` w pliku `applicationContext.xml`).

Komponent `lookupStrategy` reprezentuje implementację oferowanego przez Spring interfejsu `LookupStrategy`, który jest odpowiedzialny za wyszukiwanie informacji ACL. Listing 14.13 przedstawia definicję komponentu `lookupStrategy`.

Listing 14.13. Plik *applicationContext-security.xml* przedstawiający konfigurację *LookupStrategy*Projekt: *ch14-bankapp-db-security*Położenie pliku: *src/main/resources/META-INF/spring*

```

<beans:bean id="lookupStrategy"
  class="org.springframework.security.acs.jdbc.BasicLookupStrategy">

  <beans:constructor-arg ref="dataSource" />
  <beans:constructor-arg ref="aclCache" />

  <beans:constructor-arg>
    <beans:bean
      class="org.springframework.security.acs.domain
        .AclAuthorizationStrategyImpl">
      <beans:constructor-arg>
        <beans:bean
          class="org.springframework.security.core
            .authority.SimpleGrantedAuthority">
            <beans:constructor-arg value="ROLE_ADMIN" />
          </beans:bean>
        </beans:constructor-arg>
      </beans:bean>
    </beans:constructor-arg>

  <beans:constructor-arg>
    <beans:bean
      class="org.springframework.security.acs.domain
        .DefaultPermissionGrantingStrategy">
      <beans:constructor-arg>
        <beans:bean class="org.springframework.security.acs
          .domain.ConsoleAuditLogger" />
        </beans:constructor-arg>
      </beans:bean>
    </beans:constructor-arg>
  </beans:bean>

```

W powyższym listingu egzemplarz *BasicLookupStrategy* (implementacja interfejsu *LookupStrategy*) używa zapytań JDBC do pobrania z tabel ACL (*ACL_CLASS*, *ACL_ENTRY*, *ACL_SID* i *ACL_OBJECT_IDENTITY*) informacji szczegółowych o uprawnieniach ACL. Jeżeli informacje ACL są przechowywane w niestandardowych tabelach, należy zmodyfikować zapytania JDBC za pomocą właściwości *selectClause*, *lookupPrimaryKeysWhereClause*, *lookupObjectIdentitiesWhereClause* i *orderByClause* egzemplarza *BasicLookupStrategy*. Więcej informacji na temat tych właściwości znajdziesz w dokumentacji API Spring Security.

Konstruktor egzemplarza *BasicLookupStrategy* akceptuje argumenty typu *DataSource* (przedstawia bazę danych zawierającą tabele ACL), *AclCache* (przedstawia warstwę buforowania ACL), *AclAuthorizationStrategy* (przedstawia strategię ustalania, czy SID ma uprawnienia pozwalające na wykonanie zadań administracyjnych na rekordach ACL egzemplarza obiektu domeny) i *PermissionGrantingStrategy* (przedstawia strategię udzielania lub odmowy dostępu do zabezpieczonych obiektów na podstawie uprawnień przypisanych SID).

W omawianym listingu klasa *AclAuthorizationStrategyImpl* implementuje *AclAuthorizationStrategy*. Konstruktor obiektu *AclAuthorizationStrategyImpl* akceptuje egzemplarz *GrantedAuthority*

określający rolę, która może wykonywać zadania administracyjne (takie jak zmiana właściciela rekordu ACL) w rekordach ACL (reprezentowanych przez obiekt typu `MutableAcl`) egzemplarza obiektu domeny. W powyższym listingu rola `ROLE_ADMIN` została przekazana klasie `AclAuthorizationStrategyImpl`, co oznacza, że użytkownik w roli `ROLE_ADMIN` może wykonywać zadania administracyjne na rekordach ACL. W dalszej części rozdziału zobaczysz, że `AclAuthorizationStrategy` zabezpiecza egzemplarz `MutableAcl` przed jego nieautoryzowaną modyfikacją.

W omawianym listingu `DefaultPermissionGrantingStrategy` implementuje `PermissionGrantingStrategy`. Konstruktor obiektu `DefaultPermissionGrantingStrategy` akceptuje egzemplarz `AuditLogger` rejestrujący informacje o powodzeniu lub niepowodzeniu podczas nadawania uprawnień dotyczących rekordu ACL w tabeli `ACL_ENTRY`. W powyższym listingu egzemplarz `ConsoleAuditLogger` (implementacja `AuditLogger` wyświetlająca komunikaty w konsoli) informuje o powodzeniu, jeśli wartością kolumny `audit_success` jest `true` (tzn. 1), lub niepowodzeniu, gdy wartością kolumny `audit_failure` jest `true` (tzn. 1). Poniżej przedstawiamy przykład komunikatów wyświetlanych przez `ConsoleAuditLogger` w przypadku zakończonego pomyślnie nadania uprawnień dla rekordu ACL:

```
GRANTED due to ACE: AccessControlEntryImpl[id: 1037; granting: true; sid: PrincipalSid[cust1];
permission: BasePermission[.....R=1]; auditSuccess: true;
auditFailure: true]
```

Obiekt `BasicLookupStrategy` akceptuje egzemplarz `AclCache` (komponent `aclCache` w listingu 14.13) przedstawiający bufor dla ACL. W listingu 14.14 przedstawiamy definicję komponentu `aclCache` używanego przez obiekt `BasicLookupStrategy` do buforowania ACL.

Listing 14.14. Plik `applicationContext-security.xml` przedstawiający konfigurację egzemplarza `Cache`

Projekt: `ch14-bankapp-db-security`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans:bean id="aclCache"
  class="org.springframework.security.acls.domain.EhCacheBasedAclCache">
  <beans:constructor-arg>
    <beans:bean class="org.springframework.cache.ehcache.EhCacheFactoryBean">
      <beans:property name="cacheManager">
        <beans:bean class="org.springframework.cache
          .ehcache.EhCacheManagerFactoryBean" />
      </beans:property>
      <beans:property name="cacheName" value="aclCache" />
    </beans:bean>
  </beans:constructor-arg>
</beans:bean>
```

Obiekt `EhCacheBasedAclCache` to implementacja `AclCache` używająca `EhCache` (<http://ehcache.org/>) do buforowania ACL. Komponent `EhCacheFactoryBean` to implementacja fabryki `FactoryBean` tworząca egzemplarz `net.sf.ehcache.EhCache`. Właściwość `cacheManager` obiektu `EhCacheFactoryBean` wskazuje egzemplarz `net.sf.ehcache.CacheManager` odpowiedzialny za zarządzanie buforem. W omawianym listingu `EhCacheManagerFactoryBean` to egzemplarz oferowanej przez Spring fabryki `FactoryBean` tworzący egzemplarz `net.sf.ehcache.CacheManager`. Właściwość `cacheName` obiektu `EhCacheFactoryBean` odwołuje się do fragmentu bufora utworzonego w `EhCache` i przeznaczonego do przechowywania informacji ACL.

Po skonfigurowaniu obiektu `JdbcMutableAclService` do wykonywania operacji CRUD na listach ACL przechodzimy do konfiguracji zabezpieczeń na poziomie metody używającej informacji ACL wczytanych przez `JdbcMutableAclService`.

Konfiguracja zabezpieczeń na poziomie metody

W listingu 14.15 przedstawiamy przykład konfiguracji zabezpieczeń na poziomie metody zastosowanej w aplikacji sieciowej `MyBank`.

Listing 14.15. Plik `applicationContext-security.xml` przedstawiający konfigurację zabezpieczeń na poziomie metody

Projekt: `ch14-bankapp-db-security`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<global-method-security pre-post-annotations="enabled">
  <expression-handler ref="expressionHandler" />
</global-method-security>
```

Atrybut `pre-post-annotations` elementu `<global-method-security>` ma przypisaną wartość `enabled`, co pozwala na użycie adnotacji `@PreAuthorize` (omówimy ją w dalszej części rozdziału), `@PostAuthorize`, `@PostFilter` i `@PostAuthorize`. W powyższym listingu element `<expression-handler>` odwołuje się do komponentu `expressionHandler` konfigurującego egzemplarz `SecurityExpressionHandler`.

Obiekt `SecurityExpressionHandler` jest przez Spring Security używany do ustalenia wartości wyrażeń zabezpieczeń, takich jak `hasRole`, `hasAnyRole`, `hasPermission` itd. W listingu 14.16 przedstawiamy definicję komponentu `expressionHandler` konfigurującego egzemplarz `DefaultMethodSecurityExpressionHandler` (implementacja `SecurityExpressionHandler`).

Listing 14.16. Plik `applicationContext-security.xml` przedstawiający konfigurację `SecurityExpressionHandler`

Projekt: `ch14-bankapp-db-security`

Położenie pliku: `src/main/resources/META-INF/spring`

```
<beans:bean id="expressionHandler"
  class="org.springframework.security.access.expression.method.
    DefaultMethodSecurityExpressionHandler">
  <beans:property name="permissionEvaluator" ref="permissionEvaluator" />
  <beans:property name="permissionCacheOptimizer">
    <beans:bean class="org.springframework.security.
      .acIs.AclPermissionCacheOptimizer">
      <beans:constructor-arg ref="aclService" />
    </beans:bean>
  </beans:property>
</beans:bean>

<beans:bean id="permissionEvaluator"
  class="org.springframework.security.acIs.AclPermissionEvaluator">
  <beans:constructor-arg ref="aclService" />
</beans:bean>
```

W powyższym listingu `permissionEvaluator` odwołuje się do egzemplarza `ACLPermissionEvaluator` używającego informacji ACL do ustalenia wartości wyrażenia zabezpieczeń. Natomiast właściwość `permissionCacheOptimizer` odwołuje się do egzemplarza `ACLPermissionCacheOptimizer`, który hurtowo wczytuje informacje ACL, aby zoptymalizować wydajność tej operacji.

Zobaczmy teraz, jak w aplikacji sieciowej `MyBank` udało się zapewnić bezpieczeństwo egzemplarzom obiektów domeny.

Bezpieczeństwo egzemplarza obiektu domeny

We wcześniejszej części książki widziałeś, że adnotacja `@PreAuthorize` pozwala na zdefiniowanie ograniczeń dla metody na podstawie roli użytkownika. Jeżeli oznaczona adnotacją `@PreAuthorize` metoda akceptuje argument w postaci egzemplarza obiektu domeny, wówczas adnotacja `@PreAuthorize` umożliwia wskazanie uprawnień ACL, jakie uwierzytelniony użytkownik musi mieć w egzemplarzu obiektu domeny, aby mógł wywołać daną metodę. W listingu 14.17 przedstawiamy przykład adnotacji `@PreAuthorize` określającej uprawnienia ACL.

Listing 14.17. Interfejs `FixedDepositService` przedstawiający adnotację `@PreAuthorize` wraz z uprawnieniami ACL

Projekt: `ch14-bankapp-db-security`

Położenie pliku: `src/main/java/sample/spring/chapter14/service`

```
package sample.spring.chapter14.service;

import org.springframework.security.access.prepost.PreAuthorize;
import sample.spring.chapter14.domain.FixedDepositDetails;
.....
public interface FixedDepositService {
    .....
    @PreAuthorize("hasPermission(#fixedDepositDetails, write)")
    void editFixedDeposit(FixedDepositDetails fixedDepositDetails);
}
```

W powyższym listingu metoda `editFixedDeposit()` klasy `FixedDepositService` akceptuje egzemplarz `FixedDepositDetails`. `#fixedDepositDetails` w wyrażeniu `hasPermission` reprezentuje zmienną wyrażenia odwołującą się do egzemplarza `FixedDepositDetails` przekazanego metodzie `editFixedDeposit()`. Wartością wyrażenia `hasPermission` będzie `true`, jeżeli uwierzytelniony użytkownik ma uprawnienia zapisu w egzemplarzu `FixedDepositDetails` przekazanym metodzie `editFixedDeposit()`. W trakcie działania aplikacji wartość wyrażenia `hasPermission` jest obliczana przez skonfigurowany egzemplarz `ACLPermissionEvaluator` (patrz: listing 14.16). Jeżeli wartością wyrażenia `hasPermission` będzie `true`, to nastąpi wywołanie metody `editFixedDeposit()`.

Jeżeli metoda akceptuje *identyfikator* obiektu domeny (zamiast rzeczywistego egzemplarza obiektu domeny) jako argument, nadal można określić uprawnienia ACL, które będą miały zastosowanie dla egzemplarza obiektu domeny wskazywanego przez identyfikator. Listing 14.18 przedstawia metodę `provideAccessToAdmin()` akceptującą `fixedDepositId` (unikalnie identyfikuje egzemplarz `FixedDepositDetails`) jako argument.

Listing 14.18. *Interfejs FixedDepositService przedstawiający przykład użycia adnotacji @PreAuthorize**Projekt: ch14-bankapp-db-security**Położenie pliku: src/main/java/sample/spring/chatper14/service*

```
package sample.spring.chapter14.service;

import org.springframework.security.access.prepost.PreAuthorize;
.....
public interface FixedDepositService {
    .....
    @PreAuthorize("hasPermission(#fixedDepositId,
        'sample.spring.chapter14.domain.FixedDepositDetails', write)")
    void provideAccessToAdmin(int fixedDepositId);
}
```

W powyższym listingu zmienna wyrażenia `#fixedDepositId` odwołuje się do argumentu `fixedDepositId` przekazanego metodzie `provideAccessToAdmin()`. Ponieważ argument `fixedDepositId` identyfikuje egzemplarz obiektu `FixedDepositDetails`, w pełni kwalifikowana nazwa klasy `FixedDepositDetails` została podana jako drugi argument wyrażenia `hasPermission`. Wartością wyrażenia `hasPermission(#fixedDepositId, 'sample.spring.chapter14.domain.FixedDepositDetails', write)` jest `true`, jeżeli uwierzytelniony użytkownik ma uprawnienia zapisu w egzemplarzu `FixedDepositDetails` wskazywanym przez argument `fixedDepositId` przekazany metodzie `provideAccessToAdmin()`.

Istnieje również możliwość łączenia wyrażeń zabezpieczeń i tworzenia w ten sposób bardziej zaawansowanych wyrażeń, jak przedstawia listing 14.19.

Listing 14.19. *Interfejs FixedDepositService przedstawiający przykład użycia adnotacji @PreAuthorize**Projekt: ch14-bankapp-db-security**Położenie pliku: src/main/java/sample/spring/chatper14/service*

```
package sample.spring.chapter14.service;

import org.springframework.security.access.prepost.PreAuthorize;
.....
public interface FixedDepositService {
    .....
    @PreAuthorize("hasPermission(#fixedDepositId,
        'sample.spring.chapter14.domain.FixedDepositDetails', read) or "
        + "hasPermission(#fixedDepositId,
        'sample.spring.chapter14.domain.FixedDepositDetails', admin)")
    FixedDepositDetails getFixedDeposit(int fixedDepositId);
    .....
}
```

W powyższym listingu dwa wyrażenia `hasPermission` zostały ze sobą połączone za pomocą operatora `or` i tworzą tym samym bardziej skomplikowane wyrażenie zabezpieczeń. Metoda `getFixedDeposit()` będzie wywołana tylko wtedy, gdy uwierzytelniony użytkownik ma uprawnienia `read` lub `admin` w egzemplarzu `FixedDepositDetails` wskazanym przez argument `fixedDepositId`.

Jeżeli wartością zwrótną metody jest lista egzemplarzy obiektu domeny, wynik można filtrować za pomocą adnotacji `@PostFilter`. W listingu 14.20 przedstawiamy przykład użycia adnotacji `@PostFilter`.

Listing 14.20. *Interfejs `FixedDepositService` przedstawiający przykład użycia adnotacji `@PostFilter`*

Projekt: *ch14-bankapp-db-security*

Położenie pliku: *src/main/java/sample/spring/chapter14/service*

```
package sample.spring.chapter14.service;

import org.springframework.security.access.prepost.PostFilter;
.....
public interface FixedDepositService {
    .....
    @PreAuthorize("hasRole('ROLE_ADMIN')")
    @PostFilter("hasPermission(filterObject, read)
               or hasPermission(filterObject, admin)")
    List<FixedDepositDetails> getAllFixedDeposits();
    .....
}
```

Podobnie jak w przypadku adnotacji `@PreAuthorize`, także `@PostFilter` określa wyrażenie zabezpieczeń. Jeżeli metoda jest oznaczona adnotacją `@PostFilter`, Spring Security przeprowadza iterację przez kolekcję zwróconą przez metodę i usuwa elementy, dla których podane wyrażenie zabezpieczeń przyjmuje wartość `false`. W powyższym listingu Spring Security przeprowadza iterację przez kolekcję egzemplarzy `FixedDepositDetails` zwróconych przez metodę `getAllFixedDeposits()` i usuwa te, dla których uwierzytelniony użytkownik nie ma uprawnień `read` lub `admin`. Element `filterObject` w wyrażeniu `hasPermission` adnotacji `@PostFilter` odwołuje się do bieżącego obiektu w kolekcji. Zwróć uwagę na to, że metoda `getAllFixedDeposits()` została oznaczona także adnotacją `@PreAuthorize`. Adnotacja ta oznacza, że metoda `getAllFixedDeposits()` zostanie wywołana tylko wtedy, gdy uwierzytelniony użytkownik ma rolę `ROLE_ADMIN`.

Wcześniej w rozdziale zobaczyłeś, że klient (rola `ROLE_CUSTOMER`) może umożliwić administratorowi (rola `ROLE_ADMIN`) uzyskanie dostępu do lokaty, klikając łącze *Udostępnij administratorowi* (patrz: rysunek 14.7). Po kliknięciu tego łącza aplikacja nadaje administratorowi uprawnienia `read`, `admin` i `delete` dla danej lokaty. W dalszej części rozdziału zobaczysz, jak to zadanie można wykonać programowo. Metoda `getAllFixedDeposits()` klasy `FixedDepositService` jest wywoływana, gdy użytkownik w roli `ROLE_ADMIN` odwiedza stronę internetową wyświetlającą listę lokat w systemie (patrz: rysunek 14.8). Ponieważ użytkownik `admin` powinien mieć możliwość wyświetlania jedynie lokat udostępnionych mu przez klientów, metoda `getAllFixedDeposits()` została oznaczona adnotacją `@PostFilter` usuwającą lokaty, dla których administrator nie ma uprawnień `read` lub `admin`.

Teraz przechodzimy do programowego zarządzania informacjami ACL.

Programowe zarządzanie informacjami ACL

Istnieje możliwość programowego zarządzania informacjami ACL za pomocą egzemplarza `JdbcMutableAclService` skonfigurowanego w pliku XML kontekstu aplikacji (patrz: listing 14.12).

Kiedy klient zakłada nową lokatę, otrzymuje do niej uprawnienia `read` i `write`. Po kliknięciu łącza *Udostępnij administratorowi* wyświetlanego obok lokaty aplikacja sieciowa MyBank nadaje użytkownikowi `admin` uprawnienia `read`, `admin` i `delete` do wskazanej lokaty.

W listingu 14.21 została przedstawiona metoda `provideAccessToAdmin()` klasy `FixedDepositServiceImpl`, która jest wywoływana po kliknięciu łącza *Udostępnij administratorowi*.

Listing 14.21. Klasa `FixedDepositServiceImpl` przedstawiająca przykład dodania list uprawnień ACL

Projekt: *ch14-bankapp-db-security*

Położenie pliku: *src/main/java/sample/spring/chatper14/service*

```
package sample.spring.chapter14.service;

import org.springframework.security.acs.domain.*;
import org.springframework.security.acs.model.*;
.....
@Service
public class FixedDepositServiceImpl implements FixedDepositService {
    .....
    @Autowired
    private MutableAclService mutableAclService;

    @Override
    public void provideAccessToAdmin(int fixedDepositId) {
        addPermission(fixedDepositId, new PrincipalSid("admin"), BasePermission.READ);
        addPermission(fixedDepositId, new PrincipalSid("admin"),
            BasePermission.ADMINISTRATION);
        addPermission(fixedDepositId, new PrincipalSid("admin"), BasePermission.DELETE);
    }

    private void addPermission(long fixedDepositId, Sid recipient, Permission
        permission) { ..... }
}
```

W powyższym listingu metoda `provideAccessToAdmin()` używa metody `addPermission()` do nadania uprawnień `read`, `admin` i `delete` użytkownikowi `admin`. Wymienione poniżej argumenty są przekazywane metodzie `addPermission()`:

- `fixedDepositId` — unikalnie identyfikuje egzemplarz `FixedDepositDetails`, do którego użytkownikowi są nadawane uprawnienia.
- `PrincipalSid` — ten obiekt przedstawia SID (to znaczy nazwę użytkownika lub rolę) otrzymujący uprawnienia. Klasa `PrincipalSid` implementuje oferowany przez Spring Security interfejs `Sid`.
- Nadawane uprawnienia — klasa `BasePermission` definiuje stałe, takie jak `READ`, `ADMINISTRATION`, `DELETE` itd., przedstawiające standardowe uprawnienia, które można nadać egzemplarzowi `PrincipalSid`. Klasa `BasePermission` implementuje oferowany przez Spring Security interfejs `Permission`.

W listingu 14.22 przedstawiamy implementację metody `addPermission()`.

Ponieważ klasa `JdbcMutableAclService` implementuje interfejs `MutableAclService`, egzemplarz `JdbcMutableAclService` jest automatycznie wiązany w klasie `FixedDepositServiceImpl`.

Listing 14.22. Klasa *FixedDepositServiceImpl* przedstawiająca przykład dodawania listy uprawnień ACLProjekt: *ch14-bankapp-db-security*Położenie pliku: *src/main/java/sample/spring/chapter14/service*

```
package sample.spring.chapter14.service;
import org.springframework.security.acs.domain.*;
import org.springframework.security.acs.model.*;
.....
@Service
public class FixedDepositServiceImpl implements FixedDepositService {
    .....
    @Autowired
    private MutableAclService mutableAclService;
    .....
    private void addPermission(long fixedDepositId, Sid recipient, Permission
        permission) {
        MutableAcl acl;
        ObjectIdentity oid = new ObjectIdentityImpl(FixedDepositDetails.class,
            fixedDepositId);

        try {
            acl = (MutableAcl) mutableAclService.readAclById(oid);
        } catch (NotFoundException nfe) {
            acl = mutableAclService.createAcl(oid);
        }
        acl.insertAce(acl.getEntries().size(), permission, recipient, true);
        mutableAclService.updateAcl(acl);
    }
    .....
}
```

W celu nadania uprawnień metoda `addPermission()` wykonuje następujące kroki:

1. Deklaracja obiektu typu `MutableAcl`. Obiekt `MutableAcl` przedstawia rekord uprawnień ACL dla egzemplarza obiektu domeny. Obiekt ten definiuje metody, które można wykorzystać do modyfikacji rekordu uprawnień ACL.
2. Utworzenie egzemplarza `ObjectIdentityImpl` przez przekazanie typu obiektu domeny (tutaj będzie to *FixedDepositDetails.class*) i identyfikatora (tutaj będzie to `fixedDepositId`) jako argumentów konstruktora.
3. Pobranie rekordu uprawnień ACL dla egzemplarza obiektu domeny przez wywołanie metody `readAclById()` klasy `MutableAclService`. Jeżeli rekord uprawnień ACL nie zostanie znaleziony, metoda `readAclById()` zgłosi wyjątek `NotFoundException`.
 - W przypadku zgłoszenia wyjątku `NotFoundException` metoda `createAcl()` klasy `MutableAclService` zostanie użyta do utworzenia pustego egzemplarza `MutableAcl`, który nie zawiera żadnych rekordów uprawnień ACL. Takie działanie jest odpowiednikiem utworzenia rekordu w tabeli `ACL_OBJECT_IDENTITY` (patrz: rysunek 14.11).
4. Dodanie rekordu uprawnień ACL do egzemplarza `MutableAcl` za pomocą metody `insertAce()`. Rekordy ACL dodane do egzemplarza `MutableAcl` ostatecznie znajdują się w tabeli `ACL_ENTRY` (patrz: rysunek 14.12). Argumentami przekazywanymi metodzie `insertAce()` są indeks położenia, w którym ma zostać dodany rekord uprawnień ACL (odpowiada kolumnie `ACE_ORDER`),

dobawane uprawnienia (odpowiada kolumnie MASK), SID, do którego dodawane są uprawnienia (odpowiada kolumnie SID), oraz opcja wskazująca, czy rekord ACL udziela czy odmawia uprawnień (odpowiada kolumnie GRANTING).

5. Trwałe zachowanie zmian wprowadzonych w egzemplarzu MutableAcl, co wymaga wywołania metody `updateAcl()` klasy `MutableAclService`.

W listingu 14.23 została przedstawiona metoda `closeFixedDeposit()` klasy `FixedDepositServiceImpl`, która jest wywoływana, gdy użytkownik admin kliknie łącze *Zamknij*, aby zamknąć lokatę (patrz: rysunek 14.8).

Listing 14.23. Klasa `FixedDepositServiceImpl` przedstawiająca przykład usunięcia uprawnień ACL

Projekt: *ch14-bankapp-db-security*

Położenie pliku: *src/main/java/sample/spring/chatper14/service*

```
package sample.spring.chapter14.service;

import org.springframework.security.acs.domain.ObjectIdentityImpl;
import org.springframework.security.acs.model.MutableAclService;
import org.springframework.security.acs.model.ObjectIdentity;
import ....
@Service
public class FixedDepositServiceImpl implements FixedDepositService {
    ....
    @Autowired
    private MutableAclService mutableAclService;
    ....
    @Override
    public void closeFixedDeposit(int fixedDepositId) {
        fixedDepositDao.closeFixedDeposit(fixedDepositId);
        ObjectIdentity oid = new ObjectIdentityImpl(FixedDepositDetails.class,
            fixedDepositId);
        mutableAclService.deleteAcl(oid, false);
    }
    ....
}
```

W powyższym listingu metoda `deleteAcl()` klasy `MutableAclService` jest używana w celu usunięcia rekordów uprawnień ACL dotyczących lokaty wskazywanej przez egzemplarz `ObjectIdentity`. Na przykład jeśli wartość argumentu `fixedDepositId` wynosi 101, to metoda `deleteAcl()` usunie z tabel `ACL_ENTRY` (patrz: rysunek 14.12) i `ACL_OBJECT_IDENTITY` (patrz: rysunek 14.11) wszystkie rekordy uprawnień ACL dotyczących lokaty o identyfikatorze 101.

Przekonajmy się teraz, jak egzemplarz `MutableAcl` jest zabezpieczony przed wprowadzaniem w nim nieautoryzowanych modyfikacji.

Egzemplarz `MutableAcl` i bezpieczeństwo

Oferowany przez Spring Security interfejs `MutableAcl` definiuje metody przeznaczone do modyfikacji rekordów uprawnień ACL dla egzemplarza obiektu domeny. Wcześniej widziałeś, że aplikacja sieciowa *MyBank* używa metody `insertAce()` klasy `MutableAcl` w celu dodania rekordu uprawnień

ACL dla egzemplarza obiektu domeny (patrz: listing 14.22). Egzemplarz `AcIAuthorizationStrategy` ↪ `Impl` dostarczony `BasicLookupStrategy` (patrz: listing 14.13) jest używany w tle do zagwarantowania, że uwierzytelniony użytkownik ma odpowiednie uprawnienia do modyfikacji rekordów uprawnień ACL.

Uwierzytelniony użytkownik może modyfikować rekordy uprawnień ACL egzemplarza obiektu domeny, jeśli co najmniej jeden z poniższych warunków przyjmuje wartość `true`:

- Jeżeli uwierzytelniony użytkownik jest właścicielem egzemplarza obiektu domeny, może modyfikować dotyczące go rekordy uprawnień ACL.
- Jeżeli uwierzytelniony użytkownik posiada uprawnienia przekazane konstruktorowi klasy `AcIAuthorizationStrategyImpl`. W listingu 14.13 konstruktorowi `AcIAuthorizationStrategyImpl` została przekazana rola `ROLE_ADMIN`, a tym samym użytkownik w tej roli może wprowadzać zmiany w rekordach uprawnień ACL dla egzemplarza obiektu domeny.
- Jeżeli uwierzytelniony użytkownik ma dla egzemplarza obiektu domeny uprawnienie `ADMINISTRATION` klasy `BasePermission`.

14.5. Podsumowanie

W tym rozdziale zobaczyłeś, jak używać frameworku Spring Security do zabezpieczania aplikacji Springa. Przeanalizowaliśmy zagadnienia stosowania zabezpieczeń dla żądań sieciowych, zabezpieczeń na poziomie metody oraz zabezpieczeń egzemplarzy obiektów domeny.

Dodatek A *Import i wdrożenie przykładowych projektów w Eclipse IDE (lub IntelliJ IDEA)*

Z tego dodatku dowiesz się, jak skonfigurować środowisko pracy, zaimportować przykładowy projekt do środowiska Eclipse IDE (lub IntelliJ IDEA) oraz uruchomić projekt jako samodzielną aplikację (jeżeli dany projekt jest samodzielną aplikacją Javy) lub wdrożyć go w serwerze Tomcat 7 (jeżeli projekt to aplikacja sieciowa).

A.1. Przygotowanie środowiska pracy

Przed skonfigurowaniem środowiska pracy musisz wykonać następujące zadania:

- **Pobranie i instalacja Eclipse IDE (lub IntelliJ IDEA).** Oprogramowanie Eclipse IDE dla programistów Javy EE możesz pobrać ze strony <http://www.eclipse.org/downloads/>. Instalacja oprogramowania sprowadza się do rozpakowania pobranego pliku ZIP w wybranym katalogu.
- **Pobranie i instalacja serwera Tomcat 7.** Serwer Tomcat 7 możesz pobrać ze strony <http://tomcat.apache.org/download-70.cgi>. Zaleca się pobranie oprogramowania w postaci archiwum ZIP, a następnie jego rozpakowanie w systemie plików.
- **Pobranie i instalacja narzędzia Maven 3.** Narzędzie Maven 3 możesz pobrać ze strony <http://maven.apache.org/download.cgi>. Instalacja oprogramowania sprowadza się do rozpakowania pobranego pliku ZIP w wybranym katalogu. Maven jest używany do konwersji przykładowych projektów sieciowych dostarczonych w tej książce na postać projektów Eclipse IDE lub IntelliJ IDEA.

Przechodzimy teraz do importu przykładowego projektu do Eclipse IDE.

A.2. Import przykładowego projektu do Eclipse IDE (lub IntelliJ IDEA)

Zaleca się pobranie przykładowych projektów dołączonych do książki z wymienionej poniżej strony projektu Google Code:

<https://code.google.com/p/getting-started-with-spring-framework-2edition/>

W pozostałej części tego dodatku przyjęliśmy założenie, że w lokalnym systemie plików utworzyłeś katalog *spring-samples* zawierający wszystkie przykładowe projekty dołączone do książki.

Aby prawidłowo zaimportować przykładowy projekt, należy wykonać następujące kroki:

- Skonwertować projekt na postać projektu Eclipse IDE lub IntelliJ IDEA.
- Skonfigurować zmienną `M2_REPO` w Eclipse IDE (lub IntelliJ IDEA). Zmienna ta wskazuje lokalne repozytorium narzędzia Maven zawierające pliki JAR będące zależnościami projektu.

Teraz przeanalizujemy dokładnie powyższe kroki.

Import przykładowego projektu

Każdy przykładowy projekt zawiera plik *pom.xml* z konfiguracją wtyczek Mavena dla Eclipse IDE, IntelliJ IDEA oraz Tomcat. Wtyczki są przez Maven używane do konwersji przykładowego projektu na postać projektu Eclipse IDE lub IntelliJ IDEA oraz do wdrożenia projektu w osadzonym egzemplarzu serwera Tomcat 7. Warto zwrócić uwagę na fakt, że wtyczka Tomcat Maven (<http://tomcat.apache.org/maven-plugin.html>) została skonfigurowana jedynie dla przykładowych projektów przedstawiających *aplikacje sieciowe*. Plik *pom.xml* wskazuje także pliki JAR (np. *spring-core*, *spring-beans* itd.) będące zależnościami danego projektu.

Aby utworzyć konkretne pliki konfiguracyjne Eclipse IDE lub IntelliJ IDEA dla przykładowego projektu, należy wykonać następujące kroki:

- Przejść do wiersza poleceń i ustawić zmienną środowiskową `JAVA_HOME` wskazującą katalog instalacyjny Javy SDK:

```
C:\> set JAVA_HOME=C:\Program Files\Java\jdk1.7.0_25
```

- Przejść do katalogu zawierającego przykładowy projekt:

```
C:\> cd spring-samples
C:\spring-samples> cd ch01-bankapp-xml
C:\spring-samples\ch01-bankapp-xml>
```

- Do zmiennej środowiskowej `PATH` dodać ścieżkę dostępu do katalogu *bin* instalacji Mavena:

```
C:\spring-samples\ch01-bankapp-xml> set path=%path%; C:\apache-maven-3.0.4\bin
```

- Jeżeli przykładowy projekt ma zostać zaimportowany do Eclipse IDE, to w *Maven Eclipse Plugin* należy wykonać `eclipse:eclipse` (<http://maven.apache.org/plugins/maven-eclipse-plugin/>):

```
C:\spring-samples\ch01-bankapp-xml> mvn eclipse:eclipse
```

Wykonanie `eclipse:eclipse` powoduje pobranie zależności przykładowego projektu i wygenerowanie plików konfiguracyjnych (takich jak *.classpath* i *.project*) dla Eclipse IDE.

LUB

- Jeżeli przykładowy projekt ma zostać zaimportowany do IntelliJ IDEA, to w *Maven IDEA Plugin* należy wykonać `idea:idea` (<http://maven.apache.org/plugins/maven-idea-plugin/>):

```
C:\spring-web-mvc-samples\ch01-xml-config> mvn idea:idea
```

Wykonanie `idea:idea` powoduje pobranie zależności przykładowego projektu i wygenerowanie plików konfiguracyjnych (takich jak *.ipr*, *.iml* i *.iws*) dla IntelliJ IDEA.



Plik *pom.xml* jest dostarczony również w katalogu głównym zawierającym podkatalogi poszczególnych projektów. Możesz więc przejść do katalogu *spring-samples*, a następnie wydać polecenie `mvn eclipse:eclipse` (lub `mvn idea:idea`) w celu konwersji wszystkich przykładowych projektów na postać projektów Eclipse IDE (lub IntelliJ IDEA).

Teraz można już zaimportować przykładowy projekt do Eclipse IDE, wykonując poniższe kroki:

- Wybierz opcję *File/Import*.
- W wyświetlonym oknie dialogowym wybierz opcję *General/Existing Projects into Workspace*, a następnie kliknij przycisk *Next*.

- Wybierz przykładowy projekt (np. *ch01-bankapp-xml*) i kliknij przycisk *Finish*.

Konfiguracja zmiennej M2_REPO w Eclipse IDE

Po wykonaniu `eclipse:eclipse` lub `idea:idea` zależności projektu zostaną pobrane do katalogu `<katalog_użytkownika>/m2/repository`, gdzie `<katalog_użytkownika>` oznacza katalog domowy danego użytkownika. W systemie Windows będzie to katalog `C:\Documents and Settings\nazwa_użytkownika\.m2/repository`. Domyślnie plik `.classpath` tworzony w trakcie wywołania `eclipse:↵eclipse` odwołuje się do zależności JAR projektu, używając do tego zmiennej `M2_REPO`. Dlatego musisz skonfigurować nową zmienną `M2_REPO` Eclipse IDE odwołującą się do katalogu `<katalog_użytkownika>/m2/repository`.

Aby skonfigurować nową zmienną `M2_REPO`, wykonaj poniższe kroki:

- Wybierz opcję *Windows/Preferences*. Na ekranie zostanie wyświetlone okno dialogowe *Preferences*.
- W oknie *Preferences* wybierz opcję *Java/Build Path/Classpath Variables*, aby wyświetlić skonfigurowane zmienne środowiskowe `CLASSPATH`.
- Teraz kliknij przycisk *New*, aby skonfigurować nową zmienną `M2_REPO`. Trzeba koniecznie zwrócić uwagę na to, aby nowej zmiennej `M2_REPO` przypisać katalog `<home-directory>/m2/repository`.

Na tym etapie mamy pomyślnie zaimportowany przykładowy projekt do Eclipse IDE i ustawioną zmienną `M2_REPO`. Jeżeli projekt jest samodzielną aplikacją, można ją uruchomić, wykonując następujące kroki:

- W karcie Eclipse IDE o nazwie *Project Explorer* prawym przyciskiem myszy kliknij klasę Javy zawierającą metodę `main()` aplikacji. Zobaczysz listę akcji, jakie można wykonać na zaznaczonej klasie.
- Wybierz opcję *Run As/Java Application*, która spowoduje wykonanie metody `main()` klasy Javy.

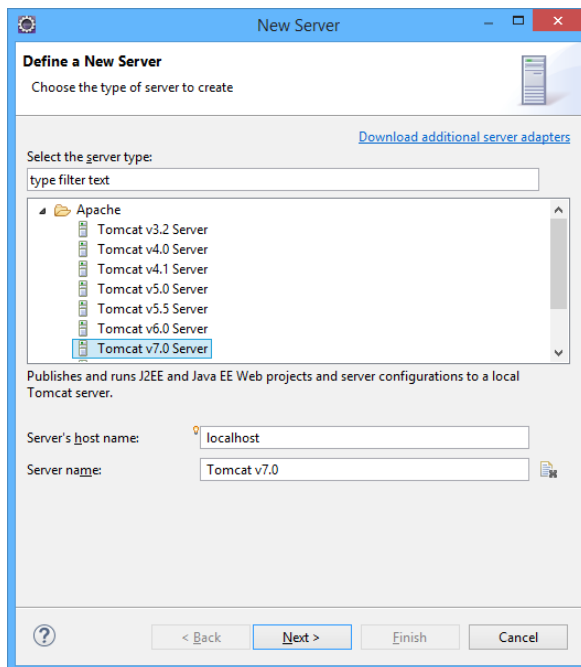
Zobaczmy, jak środowisko Eclipse IDE jest skonfigurowane do pracy z serwerem Tomcat 7.

A.3. Konfiguracja Eclipse IDE do pracy z serwerem Tomcat 7

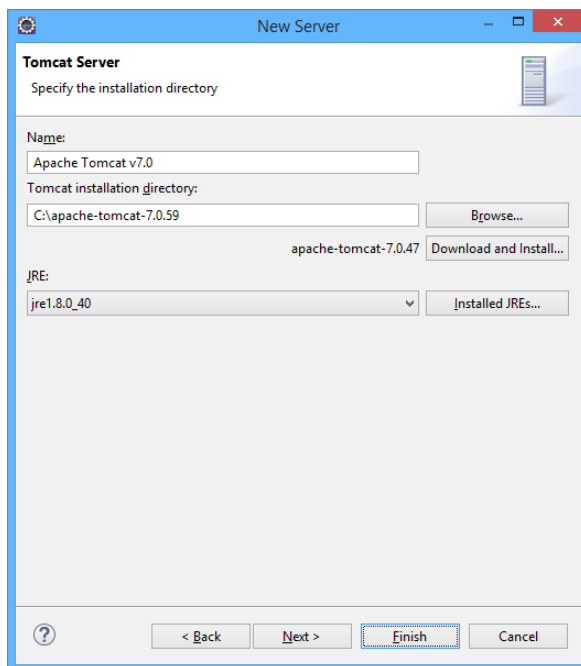
Aby skonfigurować Eclipse IDE do pracy z serwerem Tomcat 7, musisz w Eclipse IDE przejść do widoku *Servers*. W tym celu wybierz opcję *Window/Show View/Servers*. Mając wyświetlony widok *Servers*, kliknij go prawym przyciskiem myszy, a następnie wybierz opcję *New/Server*. Na ekranie zostanie wyświetlony kreator zatytułowany *New Server* i umożliwiający skonfigurowanie krok po kroku serwera do pracy z Eclipse IDE. Pierwszym krokiem jest wybór *typu* i *wersji* serwera, który ma zostać skonfigurowany do pracy z Eclipse IDE. Na rysunku A.1 pokazujemy pierwszy krok kreatora.

Jako serwer wybierz *Apache/Tomcat v7.0 Server* i podaj Tomcat v7.0 jako jego nazwę. Kliknięcie przycisku *Next* powoduje przejście do kolejnego kroku konfiguracji serwera Tomcat 7 do pracy z Eclipse IDE. Teraz musisz wskazać katalog instalacyjny serwera Tomcat 7, jak pokazano na rysunku A.2.

Rysunek A.1.
*Wybór wersji serwera
 Tomcat, która będzie
 używana z Eclipse IDE*



Rysunek A.2.
*Wskazanie katalogu
 instalacyjnego serwera
 Tomcat i wybór
 Javy SDK do użycia
 z serwerem*

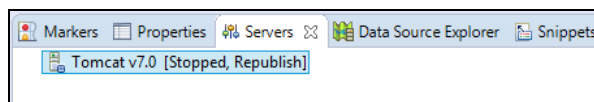


Aby wybrać katalog instalacyjny Tomcat, kliknij przycisk *Browse* (patrz: rysunek A.2), a następnie wskaż katalog, w którym rozpakowałeś archiwum ZIP zawierające serwer Tomcat. Ponadto kliknij przycisk *Installed JREs* i wybierz Javę SDK do użycia przez Eclipse IDE podczas uruchamiania

serwera Tomcat. Kliknięcie przycisku *Finish* kończy przygotowanie serwera Tomcat 7 do pracy z Eclipse IDE. Nowo skonfigurowany serwer Tomcat 7 pojawi się teraz w widoku *Servers*, jak pokazano na rysunku A.3.

Rysunek A.3.

Widok *Servers* pokazuje nowo skonfigurowany serwer Tomcat 7



Mając skonfigurowany serwer Tomcat 7, możemy zobaczyć, jak wdrożyć w nim przykładowy projekt sieciowy.

A.4. Wdrożenie projektu sieciowego w serwerze Tomcat 7

Aby wdrożyć projekt sieciowy (np. *ch10-helloworld*) w serwerze Tomcat 7, należy wykonać następujące kroki:

- Prawym przyciskiem myszy kliknij przykładowy projekt w karcie *Project Explorer*. Powinienesz zobaczyć listę akcji, jakie można wykonać na wybranym projekcie sieciowym.
- Jeżeli chcesz po prostu wdrożyć projekt sieciowy, wybierz opcję *Run As/Run on Server*. W ten sposób projekt sieciowy zostanie wdrożony w serwerze Tomcat 7 skonfigurowanym w podrozdziale A.3.

LUB

- Jeżeli chcesz wdrożyć i *debugować* projekt sieciowy, wybierz opcję *Debug As/Debug on Server*. W ten sposób projekt sieciowy zostanie wdrożony w serwerze Tomcat 7 skonfigurowanym w podrozdziale A.3, a ponadto zyskasz możliwość debugowania przez ustawianie punktów kontrolnych w Eclipse IDE.

Jeżeli serwer Tomcat 7 został prawidłowo skonfigurowany w Eclipse IDE, zauważysz jego uruchomienie i wdrożenie w nim projektu sieciowego. Teraz po przejściu do przeglądarki internetowej i wprowadzeniu adresu *http://localhost:8080/<nazwa-katalogu-projektu>* zostanie wyświetlona strona główna projektu sieciowego. *<nazwa-katalogu-projektu>* odnosi się do nazwy katalogu zawierającej projekt.

Uruchomienie serwera Tomcat 7 w trybie osadzonym

Prostszym sposobem wdrożenia i uruchomienia projektu sieciowego jest użycie *osadzonego* serwera Tomcat 7. We wszystkich przykładowych projektach sieciowych wtyczka Maven Tomcat (<http://tomcat.apache.org/maven-plugin-2.0/>) została skonfigurowana w pliku *pom.xml*. Jeżeli po przejściu do katalogu projektu wykonasz *tomcat7:run*, wtyczka zajmie się pobraniem i uruchomieniem Tomcat 7 w trybie osadzonym. Ponadto automatycznie wdroży projekt sieciowy w osadzonym egzemplarzu Tomcat 7. Aby zatrzymać działanie serwera, należy nacisnąć klawisze *Ctrl+C*.

A

ACL, access control list, 429, 440

adnotacja

- @ Scheduled, 268
- @AmountFormat, 422
- @Aspect, 280
- @Async, 267, 268
- @Autowired, 194–198
- @Bean, 216
- @Before, 280
- @Cacheable, 271, 272
- @CacheEvict, 271, 273
- @CachePut, 271, 274
- @Component, 191, 293
- @Configuration, 216
- @ConstructorProperties, 80–82
- @Controller, 314, 317
- @ControllerAdvice, 357
- @DependsOn, 203
- @ExceptionHandler, 333, 385, 414, 416
- @InitBinder, 354–357
- @Inject, 199
- @Lazy, 202
- @MatrixVariable, 395, 398, 399
- @ModelAttribute, 337–345
- @Named, 199
- @NotBlank, 367
- @PathVariable, 395
- @Pointcut, 286
- @PostConstruct, 161, 162
- @PostFilter, 455
- @PreAuthorize, 439, 453, 454
- @PreDestroy, 161, 162
- @Primary, 203
- @Qualifier, 198
- @Qualifier, 198, 199
- @RequestBody, 383, 384
- @RequestMapping, 314–318, 324, 340, 380–382, 408–413
- @RequestParam, 318, 326, 420
- @Required, 174
- @Resource, 201
- @ResponseBody, 382, 383, 394

- @ResponseStatus, 384
- @RolesAllowed, 439
- @Scheduled, 267, 412
- @Scope, 202
- @Secured, 437–439
- @SessionAttributes, 347, 348
- @Transactional, 238–241
- @Valid, 370
- @Value, 203–207
- JSR 303, 211, 213, 368
- RequestMappingHandlerMapping, 318
- Spring Web MVC, 316

adnotacje buforowania, 271

adres URI, 377

AOP, aspect-oriented programming, 279, 282

API

- Hibernate, 232
- Validation, 362

aplikacja MyBank, 22, 28, 221, 275, 304

aplikacje

- sieciowe, 314
- w stylu REST, 375

argument

- JoinPoint, 281
- typu HttpServletRequest, 324

argumenty konstruktora, 82

aspekt AOP, 299

asynchroniczne

- odbieranie komunikatów JMS, 255
- przetwarzanie żądań, 407
- wykonywanie metod, 245

atrybut

- access, 433
- defaultValue, 330
- depends-on, 123, 128
- destroy-method, 157, 158, 223
- error-page, 448
- expression, 301
- init-method, 157
- method, 320
- name, 34, 79
- proxy-target-class, 283
- ref, 121
- required, 329

atrybut
 scope, 55
 secured-annotations, 438
 transaction-manager, 240
 type, 194, 217
 value, 241, 280, 329
 xmlns, 100
 automatyczne wiązanie zależności, 145, 150, 194, 198

B

baza danych, 221
 bezargumentowy konstruktor klasy, 35
 bezpieczeństwo, 24, 429, 458
 bezpieczeństwo egzemplarza obiektu domeny, 453
 biblioteka
 CGLIB, 141, 144
 form, 372
 znaczników, 421
 znaczników form, 371
 broker ActiveMQ, 247
 buforowanie, 27, 245, 269
 ACL, 451
 atomytów modelu, 347

C

CRUD, 375, 449

D

definicja komponentu PersonalBankingService, 52
 definiowanie
 komponentów, 48, 99
 ograniczeń, 366
 wartości, 92
 właściwości lokalnych, 186
 zależności, 33
 deklaracyjne zarządzanie transakcjami, 23, 238
 deskryptor
 wdrożenia aplikacji, 308
 args, 287, 290
 bean, 291
 execution, 287
 desygatory oparte na adnotacjach, 292
 DI, dependency injection, 17
 dodanie
 atomytów modelu, 337, 340
 nazw komponentów, 90

 odwołania do komponentu, 89
 wartości null, 91
 dołączanie danych, 337, 350–352, 358
 dopasowanie argumentu konstruktora, 73, 74, 79
 dostawca DaoAuthenticationProvider, 434
 dostęp do
 egzemplarza FactoryBean, 118
 obiektów, 36, 313
 parametrów żądania, 328
 usługi sieciowej, 386, 387
 zmiennych szablonu, 397
 dostosowanie
 komponentów, 155
 logiki inicjalizacji, 155
 dynamiczni odbiorcy komunikatów JMS, 253
 dziedziczenie definicji komponentu, 65–68, 80, 128

E

Eclipse IDE, 461
 edytor
 CustomCollectionEditor, 95
 CustomMapEditor, 96
 PropertyEditor, 353
 edytory właściwości, 83, 93, 97
 egzemplarz
 FactoryBean, 118
 JMS Session, 249
 element
 <access-denied-handler>, 448
 <annotation-driven>, 272, 316, 356, 398
 <around>, 301
 <aspect>, 299
 <aspectj-autoproxy>, 283, 299
 <authentication>, 436
 <bean>, 31, 122, 128, 403
 <beans>, 62
 <broker>, 247
 <component-scan>, 192
 <config>, 301
 <constant>, 110
 <constructor-arg>, 53, 73, 76
 <entry>, 106
 <exclude-filter>, 193
 <executor>, 264
 <filter-mapping>, 431
 <filter-name>, 431
 <form>, 372
 <form-login>, 433

<global-method-security>, 437, 439
 <http>, 433
 <include-filter>, 193
 <interceptor>, 403
 <intercept-url>, 433
 <jta-transaction-manager>, 242
 <list>, 87, 104, 122
 <listener-container>, 256
 <logout>, 436
 <lookup-method>, 138, 139
 <map>, 88–90, 105
 <pointcut>, 301
 <prop>, 87
 <properties>, 109
 <property>, 32, 33, 100, 121
 <property-path>, 111
 <property-placeholder>, 186
 <queue>, 249, 251
 <ref>, 403
 <remember-me>, 433
 <replaced-method>, 142
 <scheduled-tasks>, 266
 <scheduler>, 266
 <set>, 88, 107
 <xss-protection>, 434
 FieldRetrievingBean, 111
 ListFactoryBean, 104
 MapFactoryBean, 107
 PropertiesFactoryBean, 110
 PropertyPathFactoryBean, 112
 SetFactoryBean, 108
 elementy schematu util, 103

F

fabryka, 35, 46
 filtr DelegatingFilterProxy, 431
 filtrowanie klas komponentów, 192
 format
 JSON, 377
 XML, 66
 formatowanie, 417
 formularz, 359
 framework
 AOP, 279, 282
 Hibernate, 232
 Hibernate Validator, 367
 Spring Security, 431
 Spring Web MVC, 306

frameworki zbudowane, 37
 funkcje automatycznego wiązania, 150

H

harmonogram zadań, 263
 Hibernate, 231

I

identyfikacja
 komponentów, 191
 metody komponentu, 144
 obiektów, 28
 identyfikator lokaty, 437
 implementacja
 AmountFormatter, 422
 AnnotationFormatterFactory, 423
 AutowiredAnnotationBeanPostProcessor, 195
 BeanFactoryPostProcessor, 176, 177
 BeanFactoryPostProcessors, 181
 BeanPostProcessors, 181
 CacheManager, 270
 CommonsMultipartResolver, 425, 427
 Formatter, 420–422
 HttpMessageConverter, 383
 HttpMessageConverters, 394
 interfejsu Converter, 418
 PlatformTransactionManager, 242
 PropertyEditorRegistrar, 97
 RequiredAnnotationBeanPostProcessor, 174
 usługi sieciowej, 376, 379
 import projektu, 461
 informacja
 o listach, 442
 o lokatach, 441
 informacje ACL, 455
 inicjacja
 egzemplarzy, 114
 komponentu, 61
 inicjalizacja
 egzemplarza WebDataBinder, 358
 WebDataBinder, 355
 IntelliJ IDEA, 461
 interceptor, 403
 interfejs
 ApplicationContextAware, 136, 137
 BeanFactoryPostProcessor, 176
 BeanPostProcessor, 163, 165

interfejs

- Cache, 270
- CacheManager, 269–271
- Converter, 417, 418
- DependencyResolver, 168, 169
- DestructionAwareBeanPostProcessor, 175
- DisposableBean, 161
- FactoryBean, 113, 114, 172
- FixedDepositService, 438, 453–455
- InitializingBean, 161
- JavaMailSender, 261
- LocaleContextResolver, 405
- MailSender, 260
- MessageListener, 257
- MessageSource, 406
- MethodReplacer, 143
- MutableAcl, 458
- PlatformTransactionManager, 242
- ServletConfigAware, 313
- ServletContextAware, 313
- TaskExecutor, 263
- TaskScheduler, 265
- TransactionTemplate, 236
- Validator, 207, 363
- WebBindingInitializer, 355

- internacionalizacja, 401, 404

- IoC, Inversion of Control, 17

J

- JDBC, 23

- JDO, Java Data Objects, 231

- język wyrażeń SpEL, 206

- JMS, Java Message Service, 26, 30

- JMX, Java Management Extensions, 24

- JNDI, 23

- JPA, Java Persistence API, 231

- JSON, JavaScript Object Notation, 375

- JTA, Java Transaction API, 23, 241

K

- catalog, 303

klasa

- AccountStatementServiceImpl, 151, 194
- AclAuthorizationStrategyImpl, 450
- AmountFormatAnnotationFormatterFactory, 423
- AmountFormatter, 421, 424

- ApplicationConfigurer, 177

- ApplicationContext, 58

- AsyncRestTemplate, 392, 393

- BankAccountDaoImpl, 228

- BankAccountServiceImpl, 284, 285

- BankApp, 35, 119, 275

- BankAppConfiguration, 216, 218

- BankAppWithHook, 159

- BankDetails, 83, 84

- BankStatement, 101

- CGLIB, 284

- CollectionTypesExample, 94

- Configuration, 205

- ControllerFactory, 71

- CustomEditorConfigurer, 98

- CustomerRegistrationServiceImpl, 146, 195

- CustomerRequestService, 140, 141

- CustomerRequestServiceContextAwareImpl, 137

- CustomerRequestServiceImpl, 136, 139, 142, 147, 196

- DatabaseOperations, 65

- DataTypesExample, 86

- DependencyResolutionBeanPostProcessor, 169, 170, 171

- EmailMessageListener, 260

- EmailMessageSender, 53

- EventSenderFactoryBean, 115, 172

- EventSenderSelectorServiceImpl, 123, 125

- FileUploadController, 426

- FixedDepositController, 19, 32, 324–331, 351, 369, 370, 379–381, 408, 419

- FixedDepositDao, 39

- FixedDepositDaoFactory, 44, 46, 50

- FixedDepositDaoImpl, 156, 161, 225, 232

- FixedDepositDetails, 208, 211, 332, 350, 367

- FixedDepositDetailsValidator, 363, 365

- FixedDepositJdbcDao, 50

- FixedDepositMessageListener, 257

- FixedDepositProcessorJob, 261, 262

- FixedDepositService, 21, 25–27, 43

- FixedDepositServiceImpl, 114, 124, 149, 156, 191, 209, 236, 250, 456–458

- FixedDepositServiceImplJsr303, 215

- FixedDepositValidator, 208

- FixedDepositWSCClient, 387, 388, 390

- FixedDepositController, 29

- HttpEntity, 381

- InstanceValidationBeanPostProcessor, 167

- IOUtils, 387

- JavaMailSenderImpl, 258
- JdbcTemplate, 224, 225
- JmsTemplate, 248, 249, 253–255
- LocaleChangeInterceptor, 406
- LocaleContextHolder, 406
- LocalSessionFactoryBean, 231
- LocalValidatorFactoryBean, 214, 368
- LoggingAspect, 280, 286, 291
- MailSender, 261
- MyAuthFailureHandler, 448
- MyErrorHandler, 386
- MyMethodReplacer, 142
- MyPropertyEditorRegistrar, 357
- MyRequestHandlerInterceptor, 402
- NamedParameterJdbcTemplate, 226, 227
- PersonalBankingService, 48, 51, 53
- PersonalBankingServiceImpl, 70
- PropertyOverrideConfigurer, 188
- PropertySourcesPlaceholderConfigurer, 185, 188
- ResponseEntity, 380
- RestTemplate, 386, 392
- ResultContext, 410
- SampleAspect, 295, 296
- SampleBean, 76
- ServiceTemplate, 75, 81
- SimpleCacheManager, 271
- SimpleJdbcInsert, 228, 229
- SimpleMailMessage, 259
- SingletonTest, 58
- SingletonTest z JUnit, 56
- TransactionTemplate, 234, 252
- TransferFundsServiceImpl, 77
- UserRequestControllerImpl, 73
- WebDataBinder, 360
- klasy
 - aplikacji MyBank, 114
 - DAO, 66
 - JdbcTemplate, 225
 - POJO, 29
 - zależne, 40
- kolejność
 - attributu modelu, 362
 - inicjalizacji procesu, 123
 - wykonywania metod, 349
 - wywołania, 344
- komponent
 - configuration, 205
 - customerRegistrationService, 134, 135
 - dataSource, 449
 - LocalValidatorFactoryBean, 213
 - lookupStrategy, 449
 - PropertyOverrideConfigurer, 187
 - PropertySourcesPlaceholderConfigurer, 182, 186
 - SimpleUrlHandlerMapping, 307, 308
- komponenty
 - o zasięgu prototypu, 135, 160
 - o zasięgu singleton 58, 61, 62
 - wewnętrzne, 121
 - zarządzane, 25
- komunikat, 134, 245, 345, 350
 - błędu, 427
 - JMS, 246
- komunikaty JMS, 246
 - asynchroniczne odbieranie, 255
 - dynamiczni odbiorcy, 253
 - synchroniczne odbieranie, 255
 - wysyłanie, 246–249
- konfiguracja
 - ACL, 449
 - aspektu AOP, 299
 - asynchronicznego przetwarzania żądań, 407
 - brokera ActiveMQ, 247
 - CallableProcessingInterceptor, 417
 - CustomEditorConfigurer, 98
 - Eclipse IDE, 463
 - egzemplarza
 - ContextLoaderListener, 335
 - ThreadPoolTaskExecutor, 264
 - WebDataBinder, 353
 - filtru DelegatingFilterProxy, 431
 - implementacji
 - AnnotationFormatterFactory, 423
 - DeferredResultProcessingInterceptor, 417
 - Formatter, 422
 - interfejsu CacheManager, 271
 - interfejsu WebBindingInitializer, 355
 - JdbcMutableAclService, 449
 - JdbcTemplate, 225
 - JmsTemplate, 248, 253
 - JMS ConnectionFactory, 247
 - klasy, 231
 - komponentów, 65, 217
 - menedżera transakcji JTA, 242
 - radę after returning, 300
 - radę after throwing, 300
 - RestTemplate, 386

konfiguracja

- SessionFactory, 231
- SimpleCacheManager, 271
- SimpleMailMessage, 259
- ThreadPoolTaskScheduler, 266
- TransactionTemplate, 252, 235
- uwierzytelniania, 434
- WebBindingInitializer, 356
- właściwości, 85
- wykonawcy ThreadPoolTaskExecutor, 264
- zabezpieczenia żądania sieciowego, 432
- zabezpieczeń, 431, 437, 452
- zmiennej M2_REPO, 463
- źródła danych, 223, 224

kontekst

- aplikacji, 307
- aplikacji sieciowej, 335

kontener IoC, 18**kontener podstawowy, 17****kontroler FixedDepositController, 344, 359–361, 365****konwersja**

- komunikatu, 254
- typu, 417, 420

L**liczba wątków, 264****limit czasu dla żądań, 416****lista lokat, 430, 441****logika inicjalizacji, 155, 156****logowanie, 429, 441****lokalizacja aplikacji sieciowej, 404****M****mapowanie**

- metod HTTP, 380
- obiektoowo relacyjne, 221
- wrażenia execution, 287
- żądań
 - akceptowane typy MIME odpowiedzi, 323
 - metody HTTP, 320
 - parametry żądania, 321
 - ścieżka dostępu żądania, 319
 - wartość nagłówka żądania, 323

MBean, managed bean, 25**menedżery transakcji JTA, 242****metadane konfiguracyjne, 20, 30****metoda**

- addCallback(), 394
- addPermission(), 456, 457
- afterCompletion(), 402
- afterReturningAdvice(), 300
- closeFixedDeposit(), 326
- convert(), 418
- convertAndSend(), 254
- createBankAccount(), 276, 282, 284
- createFixedDeposit(), 24, 157, 237, 240, 282
- createMessage(), 251
- currentProxy(), 283
- doInTransaction(), 237
- editFixedDeposit(), 348
- exchange(), 388, 389
- findFixedDepositsByBankAccount(), 246, 276
- getBeanDefinition(), 178
- getBeanDefinitionNames(), 178
- getCurrentSession(), 233
- getErrorCount(), 210
- getFieldTypes(), 423
- getFixedDeposit(), 277
- getFixedDepositDao(), 47, 51
- getFixedDepositDetails(), 27
- getFixedDepositFromCache(), 277
- getFixedDepositList(), 379–390, 412
- getForEntity(), 389
- getObject(), 115–117
- getObjectType(), 115
- getValue(), 179
- handleError(), 387
- handleException(), 385, 391, 416
- handleFileUpload(), 427
- handleRequest(), 305
- hasPrototypeDependency(), 178–180
- init(), 56
- isPrototype(), 180
- isSingleton(), 115, 178
- listFixedDeposits(), 340, 343, 408
- log(), 281
- myExceptionHandler(), 334
- obtainCustomerRegistrationDetails(), 196
- onMessage(), 257
- openFixedDeposit(), 328, 348, 390–394, 414
- openInvalidFixedDeposit(), 392
- perform(), 323
- perform(), 321, 341
- postForEntity(), 389

postHandle(), 402
 postProcessBeforeInitialization(), 170
 preHandle(), 401
 prepare(), 263
 processResults(), 411–415
 queryForObject(), 228
 reimplement(), 143
 reject(), 209
 send(), 251, 261, 262
 sendEmail(), 262
 setDisallowedFields(), 360
 submitRequest(), 27, 140, 141
 supports(), 363
 testInstances(), 63
 testReference, 57
 testSingletonScope(), 58
 testSingletonScopePerBeanDef(), 60
 update(), 226
 validate(), 210, 364, 365, 368
 viewFixedDepositDetails(), 419
 withTableName(), 229

metody

czyszczące, 158, 160
 docelowe, 280
 fabryki, 81
 HTTP, 320, 375
 inicjalizacji, 160
 komponentu MyBeanPostProcessor, 164
 przeciążone, 145
 typu setter, 47, 50

moduł, 17

ACL, 440, 442
 JDBC, 222, 224
 Spring Web MVC, 303

moduły frameworku Spring Security, 431

modyfikacja definicji komponentu, 176

MyBatis, 231

N

nagłówek żądania, 323

narzędzie Maven, 304

nazwa

argumentu, 79
 argumentu konstruktora, 79
 komponentu, 90, 152
 parametru żądania, 329

niejawne zależności, 124, 128

O

obiekt

BindingResult, 361
 Callable, 408
 DeferredResult, 409–412
 EhCacheBasedAclCache, 451
 HttpHeaders, 382
 HttpServletResponse, 382
 HttpSession, 349
 JDBC Connection, 22
 proxy, 282
 Queue, 411, 413
 RequestToViewNameTranslator, 346
 SecurityExpressionHandler, 452
 ServletConfig, 313
 ServletContext, 313
 weryfikatora, 303

obiekty DAO, 224

obsługa

błędów w odpowiedzi HTTP, 386
 HTML5, 372
 interceptorów, 401, 403
 JMS, 26
 JMX, 24
 JTA, 241
 konwersji typu, 417
 lokat, 376
 przekazywania plików, 425
 wyjątków, 333
 zadań, 264

odbieranie komunikatów JMS, 255

odpowiedź HTTP, 380

odwołanie do komponentu, 89

ograniczenia, 211, 213

ograniczenia automatycznego wiązania, 152

określanie ograniczeń, 211

opcja debug, 82

operacje CRUD, 375, 449

operacje użytkownika, 345

opracowanie aplikacji sieciowej, 314

ORM, object relational mapping, 221

P

parametry żądania, 321, 326

plik

appConfig.properties, 127

applicationContext.xml, 31, 55, 72, 183–188,
 248, 386, 393

plik

applicationContext-security.xml, 434, 446, 450, 452
 bankapp-config.xml, 326, 405
 createFixedDepositForm.jsp, 330, 332, 371
 database.properties, 183, 185, 189
 email.properties, 259
 emailtemplate.properties, 260
 fileupload-config.xml, 426
 FixedDepositController.java, 372
 fixedDepositList.jsp, 407, 435
 helloworld.js, 306
 list.jsp, 340
 myapp-config.xml, 307
 pom.xml, 465
 uploadForm.jsp, 425
 web.xml, 308, 309
 webservice.properties, 183–186, 189

pliki

kodu źródłowego, 305
 konfiguracyjne, 305
 XML, 335

pobieranie, 135

pobieranie atrybutów modelu, 337, 342

POJO, plain old Java objects, 20

programowa konfiguracja komponentów, 216

programowanie

interfejsów, 39, 42
 klas, 41
 oparte na adnotacjach, 191
 zorientowane aspektowo, AOP, 279

programowe zarządzanie transakcjami, 234, 252

projekt ch01-bankapp.xml, 37

prosta aplikacja, 28

protokół SMTP, 259

prototyp, 63

proxy, 283

przechowywanie zdarzeń, 113

przechwytywanie żądań asynchronicznych, 417

przeciążanie metod, 145

przekazywanie

odwołań do komponentów, 73
 parametrów żądania, 327
 plików, 425, 427
 wartości, 73

przestrzeń nazw

c, 100

p, 99

przesyłanie plików, 401

przetwarzanie żądań, 310, 343

przetwarzanie żądań asynchronicznych, 401, 407

punkt

przecięcia, pointcut, 279, 286
 przecięcia z radą, 301
 złączenia, join points, 279

R

rada, advice, 279

after, 296
 after returning, 294, 300
 after throwing, 295, 300
 argumentów, 290
 around, 297
 before, 293

rejestracja edytorów właściwości, 97

REST, Representational State Transfer, 375

rola

ROLE_ADMIN, 436
 ROLE_CUSTOMER, 436
 ROLE_XYZ, 439

rozwiązywanie zależności komponentu, 168

S

schemat

mvc, 398
 security, 434
 spring-security-3.2.xsd, 432
 util, 103, 110

sekwencja

działań, 234
 zdarzeń, 131, 135

serwer Tomcat, 463, 464

serwlet DispatcherServlet, 310, 311

singleton, 55, 58

SMTP, Simple Mail Transfer Protocol, 259

specyfikacja

JSR 250, 201
 JSR 303, 214

SpEL, Spring Expression Language, 191, 204

Spring Security, 429

stała

HttpStatus.BAD_REQUEST, 385
 HttpStatus.OK, 385

statyczna metoda fabryki, 44

struktura katalogów, 303, 304

styl

- adnotacji, 280
- schematu XML, 298
- szablon adresu URI, 396, 397

Ś

- ścieżka dostępu żądania, 319
- środowisko

IDE

- Eclipse, 15
- IntelliJ IDEA, 15
- pracy, 461

T

tabela

- ACL_CLASS, 443
- ACL_ENTRY, 444
- ACL_OBJECT_IDENTITY, 443
- ACL_SID, 443
- AUTHORITIES, 446
- USERS, 446

tabele bazy danych, 221

tablica, 92

technologia JMX, 25

testowanie klasy zależnej, 41

transakcje

- globalne, 23
- JTA, 242

tryb osadzony, 465

tworzenie

- bazy danych, 222
- egzemplarza kontenera, 35
- EventSenderFactoryBean, 173
- implementacji PropertyEditorRegistrar, 97
- komponentów, 44
- komponentu, 46
- kontenera, 135
- metadanych konfiguracyjnych, 30
- obiektów DAO, 224
- proxy, 283
- usług sieciowych, 375

typy

- kolekcji, 86
- MIME odpowiedzi, 323
- rad, 293
- wartości zwrotnych, 325

U

- uniedostępnienie komponentu, 150
- uprawnienie ADMINISTRATION, 459
- URI, Uniform Resource Identifier, 375
- uruchomienie serwera Tomcat 7, 465
- usługa sieciowa
 - FixedDepositWS, 376, 379, 386–390
 - RESTful, 375
- usuwanie
 - atributów modelu, 349
 - komponentu, 155
 - obiektu, 160
- uwierzelnianie użytkownika, 429, 434, 446, 459

W

wartości argumentów konstruktora, 102

wartość

- atributu type, 194, 217
- byName, 148
- byType, 146
- constructor, 147
- default, 150
- no, 150
- null, 91

wątek, 264

wbudowane

- edytory właściwości, 93
- implementacje PropertyEditor, 85

wdrażanie projektów, 461

wdrożenie projektu sieciowego, 465

weryfikacja, 330

- atributów modelu, 363–370
- danych, 337
- egzemplarzy komponentu, 165
- obiektów, 207
- obiektu, 215, 367, 369
- w Springu, 362

wiadomości e-mail, 245, 258

wiązania zależności, 145, 194

- według nazwy, 201

widok, 308

widok Servers, 465

właściwości

- komponentu, 82
- lokalne, 186

właściwość

- annotatedClasses, 232
- basenames, 406
- cacheManager, 451
- corePoolSize, 264
- deferredResult, 411
- depositAmount, 332
- eventSenderClass, 127
- FixedDepositDao, 175
- localOverride, 184
- locations, 183
- propagationBehaviorName, 235
- rejectedExecutionHandler, 264
- tenure, 332
- url, 184
- webServiceUrl, 184

włączenie

- adnotacji buforowania, 272
- adnotacji Spring Web MVC, 316
- opcji debug, 82

wstrzykiwanie zależności, 35, 47, 50–53, 121

wstrzyknięcie egzemplarza, 46, 119

wybór

- Javy SDK, 464
- zasięgu komponentu, 64

wyjątek, 333

- java.lang.NullPointerException, 67
- ValidationException, 416

wykonawca

- ThreadPoolExecutor, 263
- ThreadPoolTaskExecutor, 265

wykonywanie

- asynchroniczne, 263
- zadań, 265

wymagania aplikacji

- MyBank, 221, 245
- sieciowej, 317, 404

wymagany parametr żądania, 329

wyrażenia

- punktu przecięcia, 286
- regularne, 397

wyrażenie

- execution, 288
- określające wzorzec metody, 289
- punktu przecięcia, 280, 288, 290
- SpEL, 204

wysyłanie

- komunikatów
 - JMS, 246, 248
 - JMS w transakcji, 249
- wiadomości e-mail, 258, 260

wyświetlanie komunikatu, 306

wywołanie weryfikacji atrybutów, 370

wzorzec nazwy komponentu, 152

Z

zabezpieczanie

- aplikacji, 429, 431
- egzemplarzy FixedDepositDetails, 440
- metod, 436
- strony JSP, 435
- zadań sieciowych, 432, 447

zabezpieczenia

- dla metod komponentu, 438
- domyślne, 439
- na poziomie metody, 24

zagadnienia przekrojowe, cross-cutting concern, 279

zalety frameworka, 21

zależności

- komponentu, 129
 - o zasięgu prototypu, 132
 - o zasięgu singleton, 132
- między komponentami, 123
- między modułami, 18
- niejawne, 127
- o zasięgu prototypu, 131
- obiektów, 28

zapisywanie danych, 225

zapytanie SQL, 447

zarządzanie

- buforem, 451
- informacjami ACL, 455
- transakcjami, 233, 234
 - deklaracyjne, 238
 - programowe, 234
- transakcjami JMS, 252

zasięg komponentu, 54, 64

- globalSession, 312
- request, 312
- session, 312
- singleton, 129

zasób transakcyjny, 22

zmienna

 M2_REPO, 463

 środowiskowa CLASSPATH, 116

znacznik, *Patrz* element

znaczniki biblioteki form, 373

zwrot obiektu DeferredResult, 409, 411

Ż

żądania

 asynchroniczne, 416

 HTTP, 377

żądanie, 310

 Atrybut pathVar, 399

 DELETE, 378

 GET, 378

 POST, 378

 PUT, 378

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>