Craig Walls

SPRING W AKCJI

Kompendium wiedzy na temat **Spring Framework!**

Wydanie IV

Helion

Spis treści

Przedmowa

To, co najlepsze, ulega nieustannej poprawie. Ponad dwanaście lat temu Spring wkroczył w świat aplikacji Java, stawiając sobie ambitny cel uproszczenia modelu programowania aplikacji klasy enterprise. Postawił wyzwanie panującemu w tym czasie ciężkiemu modelowi programowania, dając jako alternatywę prostszy i lżejszy model programowania oparty na zwykłych obiektach Javy.

Teraz, kilka lat i wiele wydań później, widzimy, że Spring miał olbrzymi wpływ na tworzenie aplikacji klasy enterprise. Stał się standardem dla niezliczonych projektów Java i wpłynął na ewolucję licznych specyfikacji i frameworków, które miał w założeniu zastąpić. Trudno zaprzeczyć, że gdyby nie pojawienie się Springa i postawione przez niego wyzwanie dla wczesnych wersji EJB (*Enterprise JavaBeans*) obecna specyfikacja EJB mogłaby wyglądać zupełnie inaczej.

Ale Spring również podlega ciągłej ewolucji i rozwojowi, dążąc do tego, aby trudne zadania stały się prostsze, a programiści Javy otrzymywali innowacyjne rozwiązania. Od chwili, kiedy Spring po raz pierwszy rzucił wyzwanie dotychczasowemu modelowi pracy, poczynił duży skok do przodu i przeciera nowe szlaki w rozwoju aplikacji Java.

Nadszedł więc czas na zaktualizowaną wersję książki, aby zaprezentować bieżący status Springa. Od momentu ukazania się poprzedniej edycji książki zmieniło się tak dużo, że niemożliwe byłoby opisanie tego wszystkiego w jednym wydaniu. Mimo to próbowałem w tym czwartym już wydaniu książki *Spring w akcji* zawrzeć tak wiele informacji, jak to tylko było możliwe. Oto zaledwie kilka z wielu nowych ekscytujących funkcjonalności, które zostały opisane w tej edycji:

- nacisk na konfigurację opartą na klasach Javy, możliwą do zastosowania w niemal wszystkich obszarach programowania w Springu;
- warunkowa konfiguracja i profile, umożliwiające podjęcie decyzji o wyborze konfiguracji po uruchomieniu aplikacji;
- kilka rozszerzeń i ulepszeń w projekcie Spring MVC, w szczególności związanych z tworzeniem usług REST-owych;
- wykorzystanie szablonów Thymeleaf w aplikacjach internetowych Springa jako alternatywy dla JSP;
- włączenie Spring Security za pomocą konfiguracji w plikach Java;
- wykorzystanie Spring Data do automatycznego generowania implementacji repozytoriów dla JPA, MongoDB i Neo4j w trakcie działania aplikacji;
- wsparcie dla nowego deklaratywnego cachowania w Springu;
- asynchroniczna komunikacja internetowa z użyciem protokołów WebSocket i STOMP;
- Spring Boot — nowe, rewolucyjne podejście do pracy ze Springiem.

Jeśli jesteś doświadczonym weteranem Springa, przekonasz się, że te nowe elementy stanowią wartościowy dodatek do tego frameworka. Z drugiej strony, jeżeli jesteś nowicjuszem i dopiero poznajesz Springa, jest to doskonały czas na rozpoczęcie nauki i ta książka w dużym stopniu Ci w tym pomoże.

Jest to rzeczywiście ekscytująca pora na pracę ze Springiem. Ja już od dwunastu lat korzystam z tego frameworka i piszę na jego temat, wciąż czerpiąc z tego prawdziwą przyjemność. Nie mogę się doczekać, co Spring przyniesie w kolejnych wydaniach.

Podziękowania

Zanim ta książka trafiła do druku, została zszyta, oprawiona i wysłana, a zanim trafiła w Twoje ręce, musiała przejść przez ręce wielu innych osób. Nawet jeśli posiadasz egzemplarz elektroniczny, który nie przechodził przez cały ten proces, przy bitach i bajtach pobranej przez Ciebie książki też pracowało wiele rąk — uczestniczących w jej edycji, recenzowaniu, przygotowywaniu i weryfikowaniu. Gdyby nie one, książka ta nigdy by nie powstała.

W pierwszej kolejności chciałbym podziękować wszystkim w wydawnictwie Manning za cierpliwość, gdy tempo jej powstawania nie było wystarczająco szybkie, i za naciskanie mnie, abym doprowadził pracę do końca. Byli to: Marjan Bace, Michael Stephens, Cynthia Kane, Andy Carroll, Benjamin Berg, Alyson Brener, Dottie Marisco, Mary Piergies, Janet Vail oraz wiele innych osób.

Częste uwagi od recenzentów już na wczesnym etapie tworzenia książki są równie cenne jak uwagi przy tworzeniu oprogramowania. Kiedy książka dopiero zaczynała nabierać kształtu, znalazło się kilku wspaniałych recenzentów, którzy poświęcili swój czas na przeczytanie jej wersji roboczej i opisanie swoich wrażeń, co wpłynęło na ostateczną postać tej publikacji. Wśród osób, do których kieruję swoje podziękowania, są: Bob Casazza, Chaoho Hsieh, Christophe Martini, Gregor Zurowski, James Wright, Jeelani Basha, Jens Richter, Jonathan Thoms, Josh Hart, Karen Christenson, Mario Arias, Michael Roberts, Paul Balogh oraz Ricardo da Silva Lima. Specjalne podziękowania należą się Johnowi Ryanowi za szczegółową recenzję techniczną rękopisu na krótko przed oddaniem książki do druku.

Chciałbym też oczywiście podziękować swojej pięknej żonie — za to, że przetrwała jeszcze jeden mój projekt pisarski, i za jej wsparcie przez cały okres jego trwania. Kocham Cię bardziej, niż jesteś w stanie to sobie wyobrazić.

Maisy i Madi, najwspanialsze małe dziewczynki na świecie — dziękuję Wam ponownie za wszystkie uściski, uśmiechy i oryginalne pomysły na to, co powinno znaleźć się w książce.

Moi koledzy z zespołu Spring — cóż mogę powiedzieć? Dajecie czadu! Czuję pokorę i wdzięczność za to, że mogę być członkiem organizacji, która popycha Springa do przodu. Nigdy nie przestaną mnie zadziwiać niesamowite rzeczy, które wytwarzacie.

I specjalne podziękowania dla każdego, kogo spotykam, podróżując po kraju, głosząc prelekcje na spotkaniach grup użytkowników i konferencjach No Fluff/Just Stuff.

Na koniec chciałbym podziękować Fenicjanom. Wy (i fani Epcota) wiecie, za co.

O książce

Framework Spring powstał z myślą o bardzo konkretnym celu — by ułatwić tworzenie aplikacji w Java EE. Idąc tym samym tropem, książka *Spring w akcji. Wydanie IV* zostało napisane, by ułatwić naukę korzystania ze Springa. Nie było moim celem przedstawienie Ci, Czytelniku, szczegółowego listingu Spring API. Zamiast tego mam nadzieję przedstawić framework Spring w sposób, który jest najbardziej odpowiedni dla programujących w Java EE, dzięki użyciu praktycznych przykładów kodu opisujących zjawiska z rzeczywistego świata. Ponieważ Spring jest środowiskiem modułowym, modułowy jest także układ treści w tej książce. Zdaję sobie sprawę, że nie wszyscy programiści mają takie same potrzeby. Niektórzy będą chcieli nauczyć się frameworka Spring od podstaw, podczas gdy inni będą woleli wybrać na początek inne tematy i zapoznawać się z nimi w zmienionej kolejności. W ten sposób książka może służyć jako narzędzie do nauki Springa dla początkujących, a także jako przewodnik i punkt odniesienia dla tych, którzy zechcą zagłębić się bardziej w jakąś specyficzną funkcjonalność.

Książka *Spring w akcji. Wydanie IV* jest adresowana do wszystkich programujących w języku Java, lecz szczególnie użyteczna będzie dla twórców oprogramowania klasy enterprise w Javie. Będę Cię, Czytelniku, prowadził za pomocą przykładów kodu, których złożoność będzie łagodnie wzrastać. Jednak prawdziwa moc Springa polega na możliwości jego użycia do ułatwienia pracy programistom tworzącym aplikacje biznesowe. Zatem twórcy oprogramowania klasy enterprise w największym stopniu zdołają skorzystać z przykładów przedstawionych w tej książce. Ponieważ znaczna część Springa jest poświęcona realizacji usług klasy enterpise, istnieje wiele podobieństw między Springiem i EJB.

Plan działania

Książka *Spring w akcji. Wydanie IV* jest podzielona na cztery części. Część I wprowadza w podstawy frameworka Spring. W części II uczynimy kolejny krok, ucząc się, jak budować aplikacje internetowe w Springu. W części III wyjdziemy poza obszar widoczny dla użytkownika i zobaczymy, jak pracować ze Springiem po stronie serwerowej aplikacji. Ostatnia część demonstruje możliwość użycia Springa podczas integracji z innymi aplikacjami lub usługami.

W części I poznamy kontener Springa, wstrzykiwanie zależności (ang. *Dependency Injection* — DI) oraz programowanie aspektowe (ang. *Aspect Oriented Programming* — AOP), podstawowe mechanizmy frameworka Spring. To pozwoli nam na dobre zrozumienie zupełnych podstaw Springa, z których będziemy nieustannie korzystać w całej książce.

- Rozdział 1. jest wprowadzeniem do Springa i przedstawieniem kilku podstawowych przykładów DI oraz AOP. Powiem w nim też ogólnie o całym ekosystemie Springa.
- W rozdziale 2. w bardziej szczegółowy sposób przyglądamy się DI, pokazuję też różne sposoby wiązania komponentów ze sobą. Wykorzystamy w tym celu wiązanie za pomocą plików XML i klas Javy oraz mechanizm autowiązania.
- Gdy już opanujemy podstawy wiązania komponentów, rozdział 3. pozwoli nam zapoznać się z kilkoma zaawansowanymi technikami wiązania. Nie będą nam one często potrzebne, ale kiedy już zajdzie taka potrzeba, będziesz wiedział, jak w maksymalnym stopniu wykorzystać możliwości kontenera Springa.
- Rozdział 4. pozwoli nam zgłębić zastosowania programowania aspektowego do izolowania zagadnień przecinających od obiektów, na które wpływają. Rozdział ten stanowi także przygotowanie do lektury dalszych rozdziałów, w których posłużymy się programowaniem aspektowym do realizacji w sposób deklaratywny takich usług jak transakcje, bezpieczeństwo i pamięć podręczna.

W części II dowiesz się, jak budować aplikacje internetowe.

- W rozdziale 5. zostały omówione podstawy pracy ze Spring MVC, głównym frameworkiem Springa do tworzenia aplikacji internetowych. Dowiesz się, jak tworzyć kontrolery do obsługi żądań i odpowiadać na żądania, zwracając dane zapisane w modelu.
- Po zakończeniu pracy kontrolera przychodzi czas na wygenerowanie danych z modelu za pomocą widoków. W rozdziale 6. zostaną opisane różne technologie tworzenia widoków, z których można korzystać w Springu, wliczając w to JSP, Apache Tiles oraz Thymeleaf.
- Rozdział 7. wykracza poza podstawy Spring MVC. W tym rozdziale dowiesz się, jak dostosować konfigurację Spring MVC, obsłużyć żądanie przesyłania plików, poradzić sobie z wyjątkami, które mogą wystąpić w kontrolerach, a także jak przekazywać dane pomiędzy żądaniami za pośrednictwem atrybutów jednorazowych.
- Rozdział 8. zawiera omówienie Spring Web Flow, rozszerzenia do Spring MVC, które pozwala na konstruowanie konwersacyjnych aplikacji internetowych. W rozdziale tym nauczymy się tworzenia aplikacji internetowych, które prowadzą użytkownika podczas określonej wymiany danych.
- W rozdziale 9. nauczymy się, jak zastosować mechanizmy bezpieczeństwa w warstwie internetowej aplikacji za pomocą Spring Security.

W części III opuścimy warstwę widoku aplikacji, by poznać mechanizmy przetwarzania i utrwalania danych.

- Z utrwalaniem danych spotkamy się po raz pierwszy w rozdziale 10., gdy będziemy korzystać z warstwy abstrakcji Springa na JDBC do pracy z danymi zapisanymi w relacyjnej bazie danych.

- W rozdziale 11. przyjrzymy się tematowi utrwalania danych z innej perspektywy, wykorzystując JPA (*Java Persistent API*) do zapisania danych w relacyjnej bazie danych.
- W rozdziale 12. dowiesz się, jak wygląda współpraca Springa z bazami nierelacyjnymi, takimi jak MongoDB czy Neo4j. Niezależnie od wyboru bazy cachowanie pomaga zwiększyć wydajność aplikacji, bo unikamy dzięki niemu niepotrzebnych zapytań do bazy.
- Rozdział 13. opisuje wsparcie Springa dla cachowania deklaratywnego.
- W rozdziale 14. wrócimy do tematu Spring Security, aby dowiedzieć się, jak wykorzystać programowanie aspektowe do wprowadzenia zabezpieczeń na poziomie metod.

W ostatniej części poznasz sposoby integracji Springa z innymi systemami.

- W rozdziale 15. przyjrzymy się sposobom tworzenia i pobierania danych ze zdalnych usług, wliczając w to serwisy oparte na RMI, Hessian, Burlap oraz SOAP.
- W rozdziale 16. wrócimy do Spring MVC, żeby się dowiedzieć, jak tworzyć usługi REST-owe, korzystając z tego samego modelu programistycznego, który opisany został w rozdziale 5.
- Rozdział 17. jest przeglądem możliwości Springa w zakresie komunikacji asynchronicznej. W rozdziale tym opisano pracę z protokołami JMS (*Java Message Service*) oraz AMQP (*Advanced Message Queuing Protocol*).
- W rozdziale 18. przyjrzymy się komunikacji asynchronicznej z innej strony, by dowiedzieć się, jak wykorzystać Springa do komunikacji asynchronicznej pomiędzy klientem a serwerem z użyciem protokołów WebSocket oraz STOMP.
- Rozdział 19. opisuje sposoby wysyłania wiadomości e-mail w Springu.
- Rozdział 20. przedstawia obsługę zarządzania dla JMX (*Java Management Extensions*) w Springu, co umożliwia monitorowanie i modyfikację ustawień działania w aplikacji Springa.
- Na koniec, w rozdziale 21., zaprezentowane zostanie nowe, rewolucyjne podejście do pracy ze Springiem — Spring Boot. Zobaczysz, jak Spring Boot pozwala zminimalizować konfigurację potrzebną do działania aplikacji opartej na Springu, co umożliwia skoncentrowanie się na rozwoju właściwych funkcjonalności biznesowych.

Konwencje zapisu i pobieranie kodu

W książce tej znajduje się wiele przykładów kodu. Przykłady te zostały zapisane czcionką o stałej szerokości, jak `tutaj`. Występujące w tekście książki nazwy klas, metod oraz fragmenty w języku XML także zostały wyróżnione zapisem za pomocą czcionki o stałej szerokości.

Wiele spośród klas i pakietów Springa posiada wyjątkowo długie (choć ułatwiające zrozumienie) nazwy. Z tego powodu w niektórych miejscach pojawiła się potrzeba użycia znaku kontynuacji wiersza (➥).

Nie wszystkie przykłady kodu w tej książce będą kompletne. Często pokazano tylko metodę lub dwie z klasy, by skupić się na określonym temacie. Kompletne kody źródłowe aplikacji omawianych w tej książce można znaleźć na stronie internetowej, pod adresem *www.helion.pl/ksiazki/sprwa4.htm*.

O autorze

Craig Walls jest starszym inżynierem w firmie Pivotal, gdzie pełni funkcję lidera w projektach Spring Social i Spring Sync. Jest też autorem książki *Spring w akcji*, zaktualizowanej teraz w czwartym wydaniu. Jest gorliwym promotorem frameworka Spring, częstym prelegentem na spotkaniach lokalnych grup użytkowników i konferencjach oraz twórcą publikacji na temat Springa. Kiedy tylko nie koduje, stara się jak najwięcej czasu spędzić ze swoją żoną, dwiema córkami, dwoma ptakami oraz dwoma psami.

Część I

Podstawy frameworka Spring

Framework Spring wykonuje wiele czynności. Lecz gdybyśmy poszukali fundamentów tych wszystkich fantastycznych funkcjonalności, które wprowadza do świata aplikacji klasy enterprise, to najważniejszymi funkcjami Springa są: wstrzykiwanie zależności (ang. *dependency injection* — DI) i programowanie aspektowe (ang. *aspect-oriented programming* — AOP).

Rozpoczynając od rozdziału 1., „Zrywamy się do działania", podamy krótki przegląd Spring Frameworka oraz użycia DI oraz AOP w Springu, a także zademonstrujemy ich użyteczność podczas izolowania komponentów w aplikacji.

W rozdziale 2., „Tworzymy powiązania między komponentami", poznamy dokładniej sposób wiązania ze sobą komponentów aplikacji. Spojrzymy na opcje konfiguracji udostępniane przez Springa, czyli konfigurację automatyczną, konfigurację opartą na klasach Javy oraz konfigurację bazującą na języku XML.

W rozdziale 3., „Zaawansowane autowiązania", wyjdziemy poza podstawy i zobaczymy różne przydatne techniki i sztuczki, które ułatwiają wykorzystanie potencjału Springa, wliczając w to konfigurację warunkową, radzenie sobie z niejednoznacznością przy autowiązaniach, zagadnienia związane z zasięgiem komponentów oraz język wyrażeń Springa (SpEL).

W rozdziale 4., „Aspektowy Spring", poznamy sposób użycia programowania aspektowego w środowisku Spring w celu odizolowania usług systemowych (takich jak bezpieczeństwo czy audyt) od obiektów, które z nich korzystają. Ten rozdział będzie punktem wyjścia dla późniejszych rozdziałów, takich jak 9., 13. oraz 14., w których nauczymy się sposobów deklarowania cachowania i zapewniania bezpieczeństwa.

1 Zrywamy się do działania

W tym rozdziale omówimy:

- Kontener dla komponentów we frameworku Spring
- Zapoznanie się z podstawowymi modułami frameworka Spring
- Spojrzenie na ekosystem frameworka Spring
- Co nowego w Springu

To dobry czas dla programistów Javy.

W ciągu niemal 20 lat swojego istnienia Java przeżywała zarówno dobre, jak i złe chwile. Pomimo kilku niezbyt udanych projektów, takich jak aplety, wczesne wersje Enterprise JavaBeans (EJB), Java Data Objects (JDO) i niezliczone frameworki służące do logowania zdarzeń, Java jest platformą o bogatej i zróżnicowanej historii, na którą powstało wiele projektów klasy enterprise. Framework Spring ma duży udział w tworzeniu tej historii.

Spring pojawił się początkowo jako alternatywa dla „ciężkich" technologii javowych, w szczególności EJB. W porównaniu do EJB Spring zaoferował lekki, „odchudzony" model programistyczny. Wykorzystywał zwykłe obiekty Javy (POJO), ale wzbogacił je o możliwości dostępne wcześniej jedynie w EJB i innych specyfikacjach enterprise Javy.

Na przestrzeni czasu zarówno EJB, jak i Java 2 Enterprise Edition rozwinęły się, a w EJB wprowadzono model programistyczny oparty na obiektach POJO. W chwili obecnej w specyfikacji EJB pojawiły się takie idee jak wstrzykiwanie zależności (*dependency injection* — DI) oraz programowanie aspektowe (*aspect oriented programming* — AOP). Inspiracją dla tych zmian był bez wątpienia sukces odniesiony przez Springa.

Chociaż J2EE (nazywana teraz JEE) dogoniła już Springa w tym zakresie, ten również nie przestał się rozwijać. Wprowadzane są do niego usprawnienia w obszarach, których specyfikacja JEE jeszcze nie obejmuje lub w odniesieniu do których dopiero prowadzone są testy związane z ich włączeniem. Aplikacje mobilne, integracja z API serwisów społecznościowych, bazy danych NoSQL, chmury obliczeniowe oraz zagadnienia dotyczące tematyki „big data" (wielkie bazy danych) — to tylko kilka obszarów, w których Spring był, i cały czas jest, głównym motorem innowacyjności.

Tak jak powiedziałem wcześniej, to dobry czas dla programistów Javy.

W książce tej odbędziemy podróż po świecie Springa. Ten rozdział poświęcimy przyjrzeniu się frameworkowi Spring na wysokim poziomie, dając Ci, Czytelniku, pogląd na to, do czego w ogóle służy Spring. Rozdział ten da niezłą orientację w typach problemów, jakie pozwala rozwiązać Spring, i będzie stanowił podstawę dla reszty książki.

1.1. Upraszczamy programowanie w Javie

Spring jest frameworkiem open source, zaproponowanym przez Roda Johnsona i opisanym w jego książce *Expert One-on-One: J2EE Design and Development*. Spring został utworzony, by uporać się ze złożonością programowania aplikacji klasy enterprise i umożliwić osiągnięcie za pomocą zwykłych, prostych komponentów JavaBean efektów wcześniej zarezerwowanych wyłącznie dla EJB. Lecz Spring jest użyteczny nie tylko podczas programowania aplikacji działających po stronie serwera. Pisząc dowolną aplikację w Javie, możemy czerpać korzyści z użycia frameworka Spring — zyskując na prostocie, łatwości testowania i luźnych powiązaniach między obiektami.

Bean pod inną nazwą… Chociaż użytkownicy Springa, odwołując się do komponentów aplikacji, posługują się zwrotami „komponent *bean*" oraz „komponent *JavaBean*" w znaczeniu dosłownym, nie oznacza to, że komponenty w Springu muszą ściśle wypełniać specyfikację JavaBeans. Komponenty te mogą być dowolnym typem POJO. W tej książce przyjmiemy dość swobodną definicję pojęcia JavaBean jako synonimu *POJO*.

Jak się przekonamy podczas lektury tej książki, framework Spring wykonuje wiele czynności. Jednak u podstaw prawie wszystkiego, co umożliwia Spring, leży kilka fundamentalnych idei, a wszystkie skupione na podstawowym zadaniu tego środowiska: *Spring upraszcza programowanie w Javie*.

To śmiałe stwierdzenie! Wiele frameworków obiecuje uproszczenie tego czy tamtego. Lecz Spring ma za zadanie uprościć samo zadanie programowania w Javie. To wymaga dalszych wyjaśnień. Przygotowując atak na złożoność programowania w Javie, Spring korzysta z czterech kluczowych strategii:

- programowanie lekkie i niezbyt „inwazyjne" dzięki użyciu obiektów POJO;
- luźne wiązanie dzięki wstrzykiwaniu zależności i zorientowaniu na interfejs;
- programowanie deklaratywne z użyciem aspektów i wspólnych konwencji;
- ograniczenie konieczności wielokrotnego pisania tego samego kodu dzięki aspektom i wzorcom.

Prawie wszystko, co można osiągnąć we frameworku Spring, da się sprowadzić do zastosowania jednej lub kilku z tych strategii. W dalszej części tego rozdziału rozwiniemy

każdą z powyższych koncepcji, pokazując na konkretnych przykładach, jak dobrze Spring radzi sobie ze spełnianiem obietnicy uproszczenia programowania w Javie. Rozpocznijmy od przekonania się, w jaki sposób Spring zachował minimalną inwazyjność dzięki wykorzystaniu programowania bazującego na POJO.

1.1.1. Uwalniamy moc zawartą w POJO

Jeśli programujesz w Javie od długiego czasu, prawdopodobnie widziałeś frameworki (a może nawet pracowałeś w którymś z nich), które ograniczały Cię, wymuszając rozszerzanie którejś z wbudowanych w nie klas albo implementację jakiegoś interfejsu. Prostym przykładem takiego inwazyjnego modelu programistycznego są bezstanowe komponenty sesyjne z czasów EJB 2. Choć wczesne wersje EJB stanowią świetny przykład, podobny model pojawił się we wczesnych wersjach innych frameworków, takich jak Struts, WebWork, Tapistry, i w niezliczonych innych javowych specyfikacjach i frameworkach.

Spring unika (tak bardzo, jak to tylko możliwe) zaśmiecania kodu naszej aplikacji swoim API. Prawie nigdy nie będziemy zmuszani do implementowania charakterystycznego dla Springa interfejsu lub rozszerzania zawartej w nim klasy. Przeciwnie, klasy w aplikacji bazującej na Springu często w żaden sposób nie wskazują na fakt korzystania z tego frameworka. W najgorszym razie mogą zostać opatrzone jedną z adnotacji Springa, lecz poza tym pozostają zwykłymi POJO.

Ilustruje to przykład klasy `HelloWorldBean` pokazany na listingu 1.1.

Listing 1.1. Spring nie żąda umieszczania niczego w HelloWorldBean bez powodu

```
package com.habuma.spring;
public class HelloWorldBean{
   public String sayHello(){ <-- To wszystko, czego potrzebujemy
      return "Witaj Świecie";
   }
}
```

Jak widać, jest to prosta, typowa klasa Javy — zwykłe POJO. Nic nie wskazuje na to, że jest to komponent Springa. Nieinwazyjny model programistyczny Springa oznacza, że klasa ta może w równie dobrze funkcjonować w aplikacji springowej, jak i w aplikacji niewykorzystującej tego frameworka.

Mimo swojej prostoty POJO mogą być potężne. Jednym ze sposobów, w jaki Spring zwiększa możliwości POJO, jest łączenie ich za pomocą wstrzykiwania zależności. Przekonajmy się, w jaki sposób można dzięki tej technice uzyskać luźne wiązanie między obiektami.

1.1.2. Wstrzykujemy zależności

Zwrot **wstrzykiwanie zależności** może brzmieć onieśmielająco, budząc skojarzenia z jakąś skomplikowaną techniką programistyczną albo wzorcem projektowym. Lecz okazuje się, że DI nie jest ani trochę tak skomplikowane, jak brzmi jego nazwa. Stosując DI w swoich projektach, zauważymy, że nasz kod staje się znacząco prostszy, łatwiejszy do zrozumienia i do testowania.

JAK DZIAŁA WSTRZYKIWANIE ZALEŻNOŚCI

Każda niebanalna aplikacja (prawie każda bardziej skomplikowana niż przykład *Witaj świecie*) jest skonstruowana z dwóch lub więcej klas, które współpracują między sobą, by realizować jakąś logikę biznesową. Tradycyjnie każdy obiekt jest odpowiedzialny za przechowywanie referencji do obiektów, z którymi współpracuje (swoich zależności). Może to prowadzić do bardzo silnie powiązanego i trudnego do testowania kodu.

Jako przykład rozważmy klasę `Knight` (rycerz) pokazaną na listingu 1.2.

Listing 1.2. Rycerz należący do klasy DamselRescuingKnight może się podjąć tylko misji RescueDamselQuest

```
package com.springinaction.knights;

public class DamselRescuingKnight implements Knight{
   private RescueDamselQuest quest;

   public DamselRescuingKnight(){
      this.quest = New RescueDamselQuest(); <-- Ścisłe powiązanie z misją RescueDamselQuest
   }

   public void embarkOnQuest() {
      quest.embark();
   }
}
```

Jak widzimy, `DamselRescuingKnight` (rycerz ratujący niewiasty) tworzy sobie quest (misję), `RescueDamselQuest` (misja ratowania niewiasty), wewnątrz konstruktora. W ten sposób klasa `DamselRescuingKnight` jest ściśle powiązana z misją `RescueDamselQuest` i ma bardzo ograniczony repertuar misji, jakich może się podjąć. Jeśli jest jakaś niewiasta, którą trzeba uratować, rycerz jest gotów do działania. Lecz jeśli pojawi się smok, którego trzeba pokonać, czy okrągły stół, który trzeba… powiedzmy… zaokrąglić, wówczas nasz rycerz będzie siedział bezczynnie.

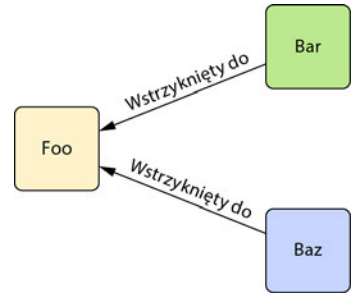
Co więcej, będzie strasznie trudno napisać test jednostkowy dla klasy `DamselRes`
`↪cuingKnight`. Podczas takiego testu chcielibyśmy mieć możliwość stwierdzenia, że faktycznie wywołanie dla rycerza metody `embarkOnQuest()` skutkuje wywołaniem metody `embark()` dla misji. Lecz nie istnieje naturalny sposób, by w tym miejscu osiągnąć taki efekt. Niestety, klasa `DamselRescuingKnight` pozostanie nieprzetestowana.

Wiązanie obiektów jest dwugłową bestią. Z jednej strony, ściśle powiązany kod jest trudny do testowania, trudny do ponownego użycia, trudny do zrozumienia, a usuwanie błędów przypomina polowanie na mole z użyciem packi (poprawienie jednego błędu skutkuje pojawieniem się jednego lub kilku błędów gdzie indziej). Z drugiej strony, pewna liczba powiązań jest konieczna — zupełnie niepowiązany kod niczego nie robi. Aby móc wykonywać jakąkolwiek użyteczną czynność, klasy muszą w jakiś sposób wiedzieć o sobie nawzajem. Wiązanie obiektów jest konieczne, lecz należy nim zarządzać z dużą ostrożnością.

Gdy skorzystamy z techniki DI, obiekty otrzymają swoje zależności w momencie utworzenia od swego rodzaju „osoby trzeciej", która koordynuje każdy z obiektów w systemie. Nie wymagamy od obiektów, by tworzyły swoje zależności lub uzyskiwały

informacje o nich. Zamiast tego zależności są do obiektów wstrzykiwane, kiedy okazują się potrzebne, co pokazano na rysunku 1.1.

By zilustrować ten punkt, spójrzmy na klasę `BraveKnight` (dzielny rycerz) na listingu 1.3. Rycerz nie tylko jest dzielny, lecz także zdolny do podjęcia się każdej misji, jaką otrzyma.



Rysunek 1.1. Wstrzykiwanie zależności polega na podawaniu obiektom ich zależności zamiast zmuszania ich, by same te zależności uzyskiwały

Listing 1.3. Dzielny rycerz jest wystarczająco elastyczny, by zająć się każdą otrzymaną misją.

```
package com.springinaction.knights;

public class BraveKnight implements Knight{
   private Quest quest;

   public BraveKnight(Quest quest){
      this.quest = quest;
   }

   public void embarkOnQuest(){
      quest.embark();
   }
}
```

Wstrzyknięcie misji

Jak widzimy, w odróżnieniu od klasy `DamselRescuingKnight` klasa `BraveKnight` nie tworzy sobie misji. Zamiast tego otrzymuje misję, gdy jest tworzony obiekt tej klasy — jako argument jej konstruktora. Jest to typ wstrzykiwania zależności, znany jako **wstrzykiwanie przez konstruktor**.

Co więcej, misja, którą otrzymuje rycerz, jest typu `Quest` — interfejsu, który implementują wszystkie misje. Zatem nasz dzielny rycerz może podjąć się misji typu `RescueDamselQuest` (ratowanie niewiasty), `SlayDragonQuest` (zgładzenie smoka), `MakeRoundTableRounder` (zaokrąglanie okrągłego stołu) lub dowolnej innej, będącej implementacją interfejsu `Quest`, jaką otrzyma.

Chodzi o to, że klasa `BraveKnight` nie jest powiązana z żadną konkretną implementacją interfejsu `Quest`. Dla rycerza nie ma znaczenia, jakiego rodzaju misji przyszło mu się podjąć, dopóki implementuje ona interfejs `Quest`. Jest to główna zaleta techniki DI — luźne wiązanie. Jeśli obiekt zna swoje zależności tylko z interfejsu (nie zaś z implementacji czy instancji), wówczas zależność może zostać zastąpiona inną implementacją, a zależący od niej obiekt nawet nie zauważy różnicy.

Jednym z najpopularniejszych sposobów podmiany zależności jest pozorna implementacja na czas testów. Nie byliśmy w stanie odpowiednio przetestować klasy `DamselRescuingKight` z powodu jej ścisłego wiązania. Lecz możemy łatwo przetestować klasę `BraveKnight`, podając jej pozorną implementację interfejsu `Quest`, jak poniżej na listingu 1.4.

Korzystamy tu z frameworka pozornych obiektów zwanego `Mockito`, by utworzyć pozorną implementację interfejsu `Quest`. Gdy mamy już do dyspozycji pozorny obiekt, tworzymy nową instancję klasy `BraveKnight`, wstrzykując jej pozorną implementację

Listing 1.4. Aby przetestować klasę BraveKnight, wstrzykujemy do niej pozorną implementację interfejsu Quest

```
package com.springinaction.knights;

import static org.mockito.Mockito.*;

import org.junit.Test;

public class BraveKnightTest{
   @Test
   public void knightShouldEmbarkOnQuest(){
      Quest mockQuest = mock(Quest.class);

      BraveKnight knight = new BraveKnight(mockQuest);
      knight.embarkOnQuest();

      verify(mockQuest,times(1)).embark();
   }
}
```

Tworzymy pozorną implementację interfejsu Quest

Wstrzykujemy pozorną implementację interfejsu Quest

interfejsu `Quest` przez konstruktor. Po wywołaniu metody `embarkOnQuest()` możemy użyć frameworka `Mockito` do weryfikacji, czy należąca do pozornej implementacji interfejsu `Quest` metoda `embark()` została wywołana dokładnie raz.

WSTRZYKUJEMY RYCERZOWI MISJĘ

Gdy klasa `BraveKnight` została napisana w taki sposób, że możemy jej zlecić dowolną misję, w jaki wówczas sposób wskażemy, jaką misję zlecimy rycerzowi? Przypuśćmy, że zlecamy dzielnemu rycerzowi misję zabicia smoka. Odpowiednia wydaje się misja `SlayDragonQuest`, pokazana na listingu 1 5.

Listing 1.5. Misja SlayDragonQuest ma zostać wstrzyknięta do obiektu klasy BraveKnight

```
package com.springinaction.knights;
import java.io.PrintStream;

public class SlayDragonQuest implements Quest {
   private PrintStream stream;
   public SlayDragonQuest(PrintStream stream) {
      this.stream = stream;
   }
   public void embark() {
      stream.println("Embarking on quest to slay the dragon!");
   }
}
```

Jak widzimy, klasa `SlayDragonQuest` implementuje interfejs `Quest`, co czyni ją odpowiednią zależnością dla klasy `BraveKnight`. Warto też zauważyć, że nasz przykład, w przeciwieństwie do większości innych prostych przykładów, nie wykorzystuje metody `System.out.println()`. Klasa `SlayDragonQuest` przyjmuje poprzez konstruktor obiekt bardziej ogólnej klasy `PrintStream`. Musimy sobie odpowiedzieć na dwa pytania: jak przekazać misję `SlayDragonQuest` rycerzowi `BraveKnight` i jak przekazać obiekt klasy `PrintStream` do instancji klasy `SlayDragonQuest`?

Czynność kojarzenia ze sobą komponentów często jest określana nazwą **wiązanie**. Spring daje nam wiele sposobów na wiązanie komponentów ze sobą, lecz zawsze popularne było zastosowanie w tym celu języka XML. Na listingu 1.6 pokazano prosty plik konfiguracyjny Springa, *knights.xml*, za pomocą którego zlecamy dzielnemu rycerzowi misję `SlayDragonQuest`, a do misji przekazujemy obiekt klasy `PrintStream`.

Listing 1.6. Wstrzykujemy w Springu misję SlayDragonQuest do obiektu klasy BraveKnight

```
<?xml version="1.0"encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="knight"class="com.springinaction.knights.BraveKnight">
      <constructor-argref="quest"/> <-- Wstrzykujemy komponent o nazwie quest
   </bean>

   <bean id="quest"
      class="com.springinaction.knights.SlayDragonQuest"> <-- Tworzymy misję SlayDragonQuest
      <constructor-arg value="#{T(System).out}" />
   </bean>
</beans>
```

Zarówno dzielnego rycerza, jak i misję `SlayDragonQuest` zadeklarowaliśmy w postaci komponentów w Springu. Komponent `BraveKnight` powstaje przez przekazanie jako argumentu konstruktora referencji do komponentu `SlayDragonQuest`. Deklaracja komponentu `SlayDragonQuest` wykorzystuje język wyrażeń Springa (*Spring Expression Language*), aby przekazać do konstruktora obiekt `System.out` (instancję klasy `PrintStream`).

Jeśli nie odpowiada Ci konfiguracja oparta na plikach XML, ucieszyć Cię może fakt, że Spring wspiera też konfigurację opartą na klasach Javy. Poniżej (listing 1.7) przedstawiony został kod Javy odpowiadający listingowi 1.6.

Listing 1.7. Spring umożliwia konfigurację za pomocą kodu Javy jako alternatywę dla plików XML

```
package com.springinaction.knights.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.springinaction.knights.BraveKnight;
import com.springinaction.knights.Knight;
import com.springinaction.knights.Quest;
import com.springinaction.knights.SlayDragonQuest;

@Configuration
public class KnightConfig {
   @Bean
   public Knight knight() {
      return new BraveKnight(quest());
   }
   @Bean
   public Quest quest() {
```

```
        return new SlayDragonQuest(System.out);
    }
}
```

Niezależnie od tego, czy wybierzemy konfigurację opartą na plikach XML, czy na plikach klasy Javy, otrzymujemy te same korzyści, wynikające ze wstrzykiwania zależności. Chociaż byt naszego dzielnego rycerza uzależniony jest od istnienia misji, rycerz nie wie, jaką konkretną misję otrzyma i kto mu ją zleci. Podobnie, choć utworzenie instancji klasy `SlayDragonQuest` uzależnione jest od obiektu klasy `PrintStream`, jej implementacja nie zawiera żadnego kodu związanego z przekazywaniem tej zależności. Jedynie Spring, dzięki swej konfiguracji, wie, jak złożyć elementy tej układanki. Umożliwia to zmianę zależności bez ingerencji w kod wiązanych klas.

Przykład ten wskazał proste podejście do wiązania komponentów w Springu. Nie skupiajmy się w tej chwili za bardzo na szczegółach. Zakopiemy się głębiej w konfigurację Springa i przekonamy się, co się dokładnie dzieje, gdy dojdziemy do rozdziału 2. Przyjrzymy się też innym sposobom wiązania komponentów w Springu. Dowiesz się między innymi, jak umożliwić Springowi automatyczne wyszukiwanie komponentów i tworzenie relacji pomiędzy nimi.

Gdy już zdeklarowaliśmy relację między dzielnym rycerzem a misją, pora załadować plik XML z konfiguracją i „odpalić" aplikację.

OGLĄDAMY APLIKACJĘ W DZIAŁANIU

Podczas uruchamiania aplikacji w Springu **kontekst aplikacji** ładuje definicje komponentów i realizuje ich powiązania. W Springu kontekst aplikacji jest w pełni odpowiedzialny za tworzenie i wiązanie obiektów tworzących aplikację. Spring posiada kilka implementacji swojego kontekstu aplikacji, głównie różniących się sposobem ładowania przez nie konfiguracji.

Ponieważ plik *knights.xml*, w którym zdeklarowaliśmy komponenty, jest plikiem w języku XML, odpowiednim wyborem kontekstu może być `ClassPathXmlApplication`
`↪Context`[1]. Ta implementacja kontekstu ładuje konfigurację z jednego lub więcej plików XML znajdujących się w ścieżce klas aplikacji. Metoda `main()` na listingu 1.8 używa implementacji `ClassPathXmlApplicationContext` do załadowania pliku *knights.xml* i uzyskania referencji do obiektu `Knight`.

Listing 1.8. Plik KnightMain.java ładuje kontekst Springa zawierający opis rycerza

```
package com.springinaction.knights;

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

public class KnightMain{
   public static void main(String[]args) throws Exception {
      ClassPathXmlApplicationContext context =
         new ClassPathXmlApplicationContext("knights.xml");
```

Ładujemy kontekst Springa

[1] W konfiguracji opartej na klasach Javy Spring udostępnia kontekst `AnnotationConfigApplication`
`↪Context`.

```
    Knight knight = context.getBean("knight"); ←— Uzyskujemy komponent knight
    knight.embarkOnQuest(); ←— Korzystamy z komponentu knight
      context.close();
  }
}
```

Metoda `main()` tworzy tu kontekst aplikacji w Springu na podstawie pliku *knights.xml*. Następnie używa kontekstu aplikacji jako fabryki, by uzyskać komponent, który będzie miał nazwę `knight`. Mając referencję do obiektu typu `Knight`, wywołuje metodę `embark ↪OnQuest()`, aby rycerz podjął się misji, którą otrzymał. Zauważmy, że ta klasa nie posiada żadnych informacji, jaki rodzaj misji otrzymał nasz bohater. Jeśli idzie o ścisłość, pozostaje także w błogiej nieświadomości, że ma do czynienia z obiektem klasy `BraveKnight`. Jedynie plik *knights.xml* posiada pewną informację, z jakimi implementacjami mamy do czynienia.

W tym miejscu kończy się krótkie wprowadzenie do wstrzykiwania zależności. Podczas lektury tej książki dowiemy się o technice DI znacznie więcej. Lecz gdybyś, Czytelniku, potrzebował jeszcze więcej informacji o wstrzykiwaniu zależności, polecam książkę autorstwa Dhanji R. Prasanna pod tytułem *Dependency Injection*, która opisuje technikę DI aż do najdrobniejszego szczegółu.

Lecz teraz przyjrzyjmy się kolejnej z upraszczających programowanie w Javie strategii Springa: programowaniu deklaracyjnemu z użyciem aspektów.

1.1.3. Stosujemy aspekty

DI pozwala na luźne wiązanie ze sobą komponentów oprogramowania. Natomiast programowanie aspektowe (AOP) umożliwia zgromadzenie w komponentach możliwych do ponownego wykorzystania rozwiązań, których użycie jest rozproszone po całej aplikacji.

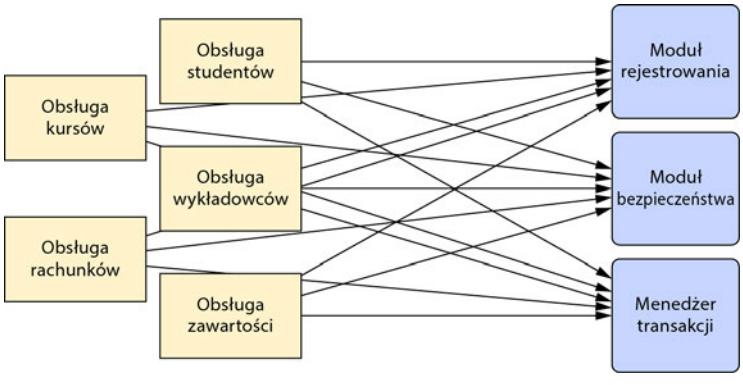
Programowanie aspektowe jest często definiowane jako technika wspierająca izolowanie dziedzin działania w systemie oprogramowania. Systemy są zbudowane z pewnej liczby komponentów; każdy jest odpowiedzialny za określony obszar działania. Często się jednak zdarza, że komponenty odpowiadają dodatkowo za działania, które nie należą do ich głównej funkcji. Usługi systemowe, takie jak logowanie, zarządzanie transakcjami i bezpieczeństwo, często znajdują sposób, by pojawić się w komponentach, które przede wszystkim odpowiadają za zupełnie inne działania. Takie usługi systemowe często bywają określane jako **zagadnienia przekrojowe**, ponieważ mają tendencję do pojawiania się w wielu komponentach w przekroju systemu.

Rozproszenie takich zagadnień po wielu komponentach wprowadza do naszego kodu dwa poziomy złożoności:

- Kod implementujący zagadnienia ogólnosystemowe jest powielany w wielu komponentach. To znaczy, że chcąc zmienić obsługę tych zagadnień, musimy modyfikować wiele komponentów. Nawet gdybyśmy wydzielili problem do osobnego modułu, tak że wpływ na pozostałe komponenty ograniczałby się do użycia pojedynczego wywołania metody, to wywołanie jest powielone w wielu miejscach.

- Nasze komponenty są zaśmiecone kodem, który nie jest związany z ich główną funkcją. Metoda dodająca wpis do książki adresowej powinna zajmować się wyłącznie dodawaniem adresu, a nie tym, czy odbywa się to w sposób bezpieczny lub transakcyjny.

Na rysunku 1.2 przedstawiono tę złożoność. Obiekty biznesowe po lewej są zbyt głęboko zaangażowane w działanie usług systemowych. Nie tylko każdy obiekt „wie", że jego działanie jest rejestrowane, zabezpieczane i realizowane w sposób transakcyjny, lecz także każdy obiekt sam odpowiada za realizowanie tych usług.



Rysunek 1.2. Wywołania systemowych usług, dotyczących dziedzin takich jak logowanie i bezpieczeństwo, często są rozproszone po modułach, w których usługi te nie stanowią głównej funkcji

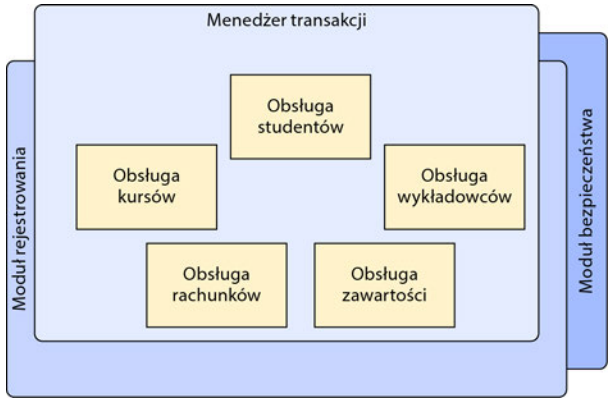
AOP umożliwia umieszczenie takich usług w osobnych modułach i stosowanie ich w sposób deklaratywny dla komponentów, które tego potrzebują. Skutkuje to tworzeniem komponentów o bardziej zwartej budowie i działaniu skoncentrowanym na właściwych sobie zadaniach, z zupełnym lekceważeniem potrzebnych usług systemowych. W skrócie, aspekty pozwalają, by POJO pozostały proste.

Może pomóc myślenie o aspektach jako o kocach okrywających wiele komponentów w aplikacji, jak pokazano na rysunku 1.3. Rdzeń aplikacji składa się z modułów implementujących funkcje biznesowe. Dzięki AOP możemy następnie okryć nasz rdzeń aplikacji warstwami realizującymi usługi systemowe. Warstwy te można nałożyć na naszą aplikację za pomocą deklaracji w sposób elastyczny i w dodatku niewidoczny dla rdzenia aplikacji. Jest to potężna koncepcja. Pozwala ochronić przed zaśmieceniem rdzenia aplikacji, realizującego logikę biznesową, sprawami bezpieczeństwa, obsługi transakcji czy też logowania.

By zademonstrować sposób stosowania aspektów w Springu, wróćmy do przykładu z rycerzem, dodając do mieszanki prosty aspekt.

AOP W DZIAŁANIU

Wszystkie informacje, jakie ktokolwiek posiada na temat rycerzy, dotarły do nas dzięki sławieniu ich czynów w pieśniach przez uzdolnionych muzycznie gawędziarzy, zwanym minstrelami. Przyjmijmy, że wyjścia na misje i powroty z nich naszego dzielnego rycerza chcemy rejestrować z użyciem usług świadczonych przez minstrela. Poniżej na listingu 1.9 pokazano klasę `Minstrel`, której moglibyśmy użyć.



Rysunek 1.3. Dzięki użyciu AOP możemy okryć warstwami usług systemowych komponenty, które tych usług potrzebują. To pozwala w komponentach aplikacji skupić się na specyficznej dla nich logice biznesowej

Listing 1.9. Minstrel jest muzycznie uzdolnionym systemem logowania z czasów średniowiecza

```
package com.springinaction.knights;

import java.io.PrintStream;

public class Minstrel {
   private PrintStream stream;

   public Minstrel(PrintStream stream) {
      this.stream = stream;
   }

   public void singBeforeQuest() {
      stream.println("Tra la la; Jakiż rycerz jest dzielny!");
   }

   public void singAfterQuest() {
      stream.println("Hip hip hura; Dzielny rycerz wypełnił misję!");
   }
}
```

Wywołanie przed wyruszeniem na misję

Wywołanie po powrocie z misji

Jak widzimy, `Minstrel` jest prostą klasą z dwiema metodami. Metoda `singBeforeQuest()` jest przeznaczona do wywoływania przed podjęciem się misji przez rycerza. Z kolei metoda `singAfterQuest()` powinna być wywołana po tym, jak rycerz wypełni misję. W obu przypadkach minstrel śpiewa o bohaterskich czynach rycerza, korzystając ze wstrzykniętego przez konstruktor obiektu klasy `PrintStream`.

Powinno być łatwo dodać tę klasę do naszego kodu, wystarczy ją tylko wstrzyknąć do klasy `BraveKnight`, nieprawdaż? Poczyńmy zatem niezbędne przystosowania w klasie `BraveKnight`, umożliwiające korzystanie z klasy `Minstrel`. Pierwsze podejście pokazano na listingu 1.10.

Listing 1.10. Klasa BraveKnight, która musi wywoływać metody klasy Minstrel

```
package com.springinaction.knights;
public class BraveKnight implements Knight {
  private Quest quest;
```

```
  private Minstrel minstrel;

  public BraveKnight(Quest quest, Minstrel minstrel) {
    this.quest = quest;
    this.minstrel = minstrel;
  }

  public void embarkOnQuest() throws QuestException {
    minstrel.singBeforeQuest();
    quest.embark();
    minstrel.singAfterQuest();
  }
}
```

Czy rycerz powinien zajmować się zarządzaniem swoim minstrelem?

To powinno działać. Musimy tylko powrócić do konfiguracji Springa, zadeklarować komponent `Minstrel` i wstrzyknąć go do konstruktora komponentu `BraveKnight`. Ale chwila… Coś tu wygląda na nieprawidłowe. Czy rzeczywiście do zakresu zainteresowań rycerza należy zarządzanie jego minstrelem? Wydaje mi się, że minstrele powinni zajmować się swoją robotą, nie czekając, aż rycerze ich o to poproszą. W końcu taką minstrel ma pracę — śpiewać o dokonaniach rycerza. Czemu rycerz miałby mu o tym ciągle przypominać?

Co więcej, ponieważ rycerz nie wie nic o minstrelu, jesteśmy zmuszeni do wstrzyknięcia kodu Minstrela do klasy `BraveKnight`. To nie tylko komplikuje kod klasy `BraveKnight`, lecz także wywołuje we mnie refleksję, czy kiedykolwiek będziemy chcieli rycerza, który nie ma minstrela. Co się stanie, jeśli `Minstrel` będzie miał wartość `null`? Czy powinniśmy wprowadzić jakąś logikę sprawdzającą wystąpienie wartości `null`, by obsłużyć taki przypadek?

Nasza prosta klasa `BraveKnight` zaczyna się stawać coraz bardziej skomplikowana, a będzie jeszcze gorzej, gdy pojawi się potrzeba obsłużenia scenariusza z wartością `minstrel` równą `null`. Lecz korzystając z AOP, możemy zdeklarować, że minstrel powinien śpiewać o misjach rycerza i zwolnić rycerza z obowiązku bezpośredniej obsługi metod klasy `Minstrel`.

Aby uczynić klasę `Minstrel` **aspektem**, musimy tylko zadeklarować ten fakt w jednym z plików konfiguracyjnych Springa. Na listingu 1.11 znajduje się uaktualniony plik *knights.xml*, uzupełniony o deklarację klasy `Minstrel` jako aspektu.

Listing 1.11. Deklarujemy komponent Minstrel jako aspekt

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xsi:schemaLocation="http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="knight" class="com.springinaction.knights.BraveKnight">
      <constructor-arg ref="quest"/>
   </bean>
```

```
  <bean id="quest" class="com.springinaction.knights.SlayDragonQuest">
    <constructor-arg value="#{T(System).out}" />
  </bean>

  <bean id="minstrel"
    class="com.springinaction.knights.Minstrel">
    <constructor-arg value="#{T(System).out}" />
  </bean>

 <aop:config>
  <aop:aspect ref="minstrel">

    <aop:pointcut id="embark"
      expression="execution(* *.embarkOnQuest(..))"/>

    <aop:before pointcut-ref="embark"
      method="singBeforeQuest"/>

    <aop:after pointcut-ref="embark"
      method="singAfterQuest"/>

  </aop:aspect>
 </aop:config>
</beans>
```

Deklarujemy komponent Minstrel

Definiujemy punkt przecięcia

Deklarujemy poradę before

Deklarujemy poradę after

Korzystamy tu z konfiguracyjnej przestrzeni nazw `aop` w Springu, by zadeklarować, że komponent `Minstrel` jest aspektem. Najpierw musimy zadeklarować, że `Minstrel` jest komponentem. Następnie odwołujemy się do tego komponentu w elemencie `<aop:aspect>`. Dalej definiując aspekt, deklarujemy (korzystając z elementu `<aop:before>`), że przed wykonaniem metody `embarkOnQuest()` powinna zostać wywołana metoda `singBeforeQuest()` klasy `Minstrel`. Jest to porada typu **`before`**. Deklarujemy także (za pomocą elementu `<aop:after>`), że po wykonaniu metody `embarkOnQuest()` należy wywołać metodę `singAfterQuest()`. To z kolei jest porada typu **`after`**.

W obydwu przypadkach atrybut `pointcut-ref` odwołuje się do punktu przecięcia nazwanego `embark`. Ten punkt przecięcia jest zdefiniowany we wcześniejszym elemencie `<pointcut>`, z atrybutem `expression` o wartości wskazującej, w którym miejscu należy zastosować się do porady. Składnia wyrażeń pochodzi z wyrażeń punktów przecięcia w języku AspectJ.

Nie powinniśmy się teraz martwić brakiem znajomości języka AspectJ, w tym szczegółami zapisu wyrażeń punktów przecięcia. Później, w rozdziale 4., powiemy więcej na temat AOP w Springu. Na chwilę obecną wystarczy wiedzieć, że poleciliśmy Springowi, by wywoływał metody klasy `Minstrel`, `singBeforeQuest()` oraz `singAfterQuest()`, odpowiednio przed podjęciem się i po podjęciu się przez dzielnego rycerza wykonania misji.

I to już wszystko, czego potrzeba! Za pomocą ociupinki XML przekształciliśmy klasę `Minstrel` do aspektu w Springu. Nie martw się, Czytelniku, jeśli nie widzisz w tym jeszcze wiele sensu. Wiele powinny wyjaśnić przykłady AOP w Springu z rozdziału 4. (a będzie ich sporo). Na razie warto zapamiętać z tego przykładu dwie ważne sprawy.

Po pierwsze, `Minstrel` nadal jest POJO — nic w nim nie wskazuje, że będzie używany jako aspekt. By stał się aspektem, wystarczyło zadeklarować w kontekście Springa, że nim będzie.

Po drugie i najważniejsze, klasa `BraveKnight` może korzystać z usług klasy `Minstrel`, nie musząc ich wprost wywoływać. Właściwie `BraveKnight` pozostaje zupełnie nieświadomy, że `Minstrel` istnieje.

Powinniśmy wskazać także, że pomimo użycia odrobiny magii Springa, by przekształcić klasę `Minstrel` w aspekt, najpierw została ona zadeklarowana jako `<bean>`. Chodzi o to, że możemy zrobić z aspektami w Springu to wszystko, co z każdym innym rodzajem komponentów, na przykład użyć ich do wstrzykiwania zależności.

Użycie aspektów do śpiewania o rycerzach może być zabawne. Lecz AOP w Springu nadaje się do znacznie bardziej praktycznych zastosowań. Jak się przekonamy później, programowanie aspektowe możemy wykorzystać do zapewnienia usług takich jak transakcje deklaracyjne i bezpieczeństwo (rozdziały 9. i 14.).

Lecz w tej chwili spójrzmy na jeszcze jeden ze sposobów, w jaki Spring upraszcza programowanie w Javie.

1.1.4. Ograniczamy powtórzenia kodu dzięki szablonom

Czy zdarzyło się kiedyś, że po napisaniu fragmentu kodu czułeś się, Czytelniku, jakbyś już wcześniej pisał taki sam kod? To nie déjà vu, mój przyjacielu. Zdarzają się takie fragmenty kodu (zwane kodem szablonowym), które musimy często pisać raz za razem w celu realizacji popularnych i skądinąd prostych zadań.

Niestety, wiele jest miejsc, w których API Javy wymaga nieco takiego ciągle powtarzanego kodu. Częstym przykładem kodu szablonowego jest użycie JDBC do pobrania informacji z bazy danych. Jeśli kiedykolwiek pracowałeś z JDBC, to prawdopodobnie pisałeś coś podobnego do poniższego przykładu zaprezentowanego na listingu 1.12.

Listing 1.12. Wiele elementów Java API, na przykład JDBC, wymaga użycia kodu szablonowego

```
public Employee getEmployeeById(long id) {
   Connection conn = null;
   PreparedStatement stmt = null;
   ResultSetrs = null;
   try {
      conn = dataSource.getConnection();
      stmt = conn.prepareStatement( "select id, firstname, lastname, salaryfrom " +
         "employee where id=?"); <-- Wybieramy pracownika
      stmt.setLong(1,id);
      rs = stmt.executeQuery();
      Employe eemployee = null;
      if (rs.next()) {
         employee = newEmployee(); <-- Tworzymy obiekt z danych
         employee.setId(rs.getLong("id"));
         employee.setFirstName(rs.getString("firstname"));
         employee.setLastName(rs.getString("lastname"));
         employee.setSalary(rs.getBigDecimal("salary"));
      }
      return employee;
```

```
    } catch (SQLException e) {
    } finally {
       if (rs != null) {
          try {
             rs.close();
          }
          catch(SQLException e) {}
       }
       if (stmt != null) {
          try {
             stmt.close();
          } catch(SQLExceptione) {}
       }
       if (conn != null) {
          try {
             conn.close();
          } catch(SQLExceptione) {}
       }
    }
    return null;
}
```

Jaką czynność należy wykonać w tym miejscu?

Sprzątamy bałagan

Jak widzimy, ten kod obsługujący JDBC wysyła do bazy danych zapytanie o nazwisko i wynagrodzenie pracownika. Lecz założę się, że trzeba było bardzo dokładnie się przyjrzeć, by to dostrzec. To dlatego, że mały fragment kodu, który odpowiada za zapytanie o pracownika, jest zagrzebany pod stosem operacji związanych z „ceremoniałem" JDBC. Najpierw musimy utworzyć połączenie, następnie wyrażenie i w końcu możemy wykonać samo zapytanie, by uzyskać wyniki. Dodatkowo, by uniknąć gniewu JDBC, musimy obsłużyć wyjątek `SQLException`, mimo że nie jesteśmy w stanie wiele zdziałać w razie zgłoszenia wyjątku.

Na koniec, po wykonaniu wszystkich czynności musimy posprzątać, zamykając połączenie, wyrażenie oraz zbiór wynikowy. To także może wywołać gniew JDBC. Zatem tutaj także musimy obsługiwać wyjątek `SQLException`.

Najbardziej charakterystyczny w kodzie na listingu 1.12 jest fakt, że większość tego kodu musimy powtarzać dla prawie każdej operacji w JDBC. Tylko niewielki fragment ma cokolwiek wspólnego z zapytaniem o dane pracownika, a większość jest kodem szablonowym JDBC.

A przecież JDBC nie ma wyłączności na stosowanie kodów szablonowych. Wiele czynności wymaga podobnego kodu szablonowego. JMS, JNDI czy też korzystanie z usług REST często wymaga sporej ilości często powtarzającego się kodu.

Spring stara się ograniczać użycie kodu szablonowego, przez przeniesienie go do zewnętrznych szablonów. Szablon `JdbcTemplate` w Springu umożliwia wykonywanie operacji na bazie danych bez całego ceremoniału tradycyjnie wymaganego przez JDBC.

Na przykład należący do Springa szablon `SimpleJdbcTemplate` (specjalizacja szablonu `JdbcTemplate`, która korzysta z możliwości Javy 5) umożliwia przepisanie metody `get` `↪EmployeeById()` w sposób pozwalający skoncentrować się na zadaniu pobierania danych pracownika, zamiast obsługiwać żądania API JDBC. Poniżej na listingu 1.13 pokazano, jak taka poprawiona metoda `getEmployeeById()` mogłaby wyglądać.

Listing 1.13. Szablony pozwalają skoncentrować się na głównym zadaniu podczas pisania kodu

```
public Employee getEmployeeById(long id) {
   return jdbcTemplate.queryForObject(
      "select id, firstname, lastname, salary " +
         "from employee where id=?",
      new RowMapper<Employee>() {
         public Employee mapRow(ResultSet rs,
            int rowNum) throws SQLException {
            Employee employee = newEmployee();
            employee.setId(rs.getLong("id"));
            employee.setFirstName(rs.getString("firstname"));
            employee.setLastName(rs.getString("lastname"));
            employee.setSalary(rs.getBigDecimal("salary"));
            return employee;
         }
      },
      id);
}
```

Zapytanie SQL

Odwzorowujemy wyniki na obiekt

Przekazujemy parametr zapytania

Jak widzimy, nowa wersja metody `getEmployeById()` jest znacznie prostsza i skoncentrowana na zadaniu pobrania z bazy danych informacji o pracowniku. Należąca do szablonu metoda `queryForObject()` otrzymuje zapytanie w języku SQL, obiekt `RowMapper` (na potrzeby odwzorowania danych zbioru do obiektu domeny) oraz zero lub więcej parametrów zapytania. Tym, czego już nie widzimy w metodzie `getEmployeById()`, będzie cały kod szablonowy JDBC, który znajdował się tam wcześniej. Kod ten jest w całości realizowany wewnątrz szablonu.

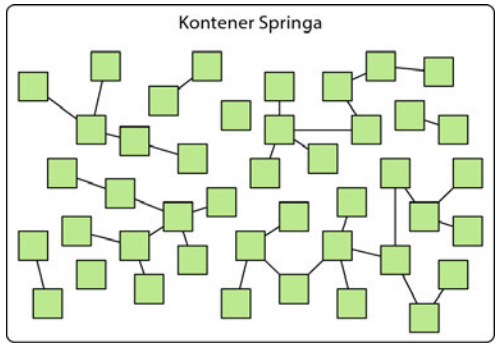
Pokazaliśmy, w jaki sposób Spring radzi sobie ze zmniejszaniem złożoności programowania w Javie dzięki programowaniu opartemu na POJO, wstrzykiwaniu zależności, aspektom oraz szablonom. Przy okazji pokazaliśmy także sposób konfiguracji komponentów i aspektów za pomocą plików XML. Lecz w jaki sposób odbywa się ładowanie tych plików? I do czego są one ładowane? Przyjrzyjmy się kontenerowi Springa, miejscu, gdzie znajdują się komponenty wchodzące w skład naszej aplikacji.

1.2. Kontener dla naszych komponentów

Obiekty aplikacji napisanej z użyciem Springa znajdować się będą wewnątrz **kontenera** Springa. Jak pokazano na rysunku 1.4, kontener tworzy obiekty, powiązania między nimi, konfiguruje je i zarządza całym ich cyklem życia „od kołyski aż po grób" (czy w tym przypadku raczej od operatora `new` aż do metody `finalize()`).

W następnym rozdziale zobaczymy, jak skonfigurować Springa, by wiedział, jakie obiekty utworzyć, skonfigurować i powiązać. W pierwszej kolejności ważne jest jednak, byśmy poznali kontener, w jakim znajdują się nasze obiekty. Zrozumienie działania kontenera pozwoli nam uchwycić sposób, w jaki będą zarządzane nasze obiekty.

Kontener jest podstawą frameworka Spring. Kontener Springa korzysta ze wstrzykiwania zależności do zarządzania komponentami tworzącymi aplikację. Między innymi tworzy powiązania między współpracującymi komponentami. Jako takie, obiekty te są bardziej czytelne i łatwiejsze do zrozumienia, nadają się do wielokrotnego użycia i ułatwiają wykonywanie testów jednostkowych.



Rysunek 1.4. Aplikacja w Springu tworzy obiekty i powiązania między nimi oraz przechowuje obiekty wewnątrz kontenera Springa

Nie jesteśmy w Springu ograniczeni do tylko jednego typu kontenera. Spring posiada kilka implementacji kontenerów, które można podzielić na dwie grupy. **Fabryki komponentów** (zdefiniowane przez interfejs `org.springframework.beans.factory.BeanFactory`) są najprostszymi kontenerami, zapewniającymi podstawowe wsparcie dla techniki DI. **Konteksty aplikacji** (zdefiniowane przez interfejs `org.springframework.context.Appli`
`➥cationContext`) są rozwinięciem koncepcji fabryki komponentów, zapewniając aplikacji dodatkowo usługi środowiska, takie jak możliwość analizowania komunikatów tekstowych z plików właściwości czy też przekazywania zdarzeń w aplikacji do zainteresowanych procesów nasłuchujących.

Choć można pracować w Springu, korzystając albo z fabryk komponentów, albo z kontekstów aplikacji, fabryki okazują się często rozwiązaniem zbyt niskopoziomowym dla większości zastosowań. Dlatego konteksty aplikacji są rozwiązaniem preferowanym. Skupimy się na pracy z nimi, nie marnując więcej czasu na rozmowę o fabrykach komponentów.

1.2.1. Pracujemy z kontekstem aplikacji

Spring posiada kilka odmian kontekstu aplikacji. Te, które prawdopodobnie najczęściej napotkasz, to:

- `AnnotationConfigApplicationContext` — ładuje kontekst aplikacji Springa z jednej lub kilku klas konfiguracji Javy.
- `AnnotationConfigWebApplicationContext` — ładuje kontekst aplikacji internetowej Springa z jednej lub kilku klas konfiguracji Javy.
- `ClassPathXmlApplicationContext` — ładuje definicję kontekstu z jednego lub kilku plików XML, znajdujących się w ścieżce klas; traktuje pliki z definicjami kontekstu jako zasoby w ścieżce do klas.
- `FileSystemXmlApplicationContext` — ładuje definicję kontekstu z jednego lub kilku plików XML, znajdujących się w systemie plików.
- `XmlWebApplicationContext` — ładuje definicję kontekstu z jednego lub kilku plików XML, znajdujących się wewnątrz aplikacji internetowej.

Więcej o kontekstach `AnnotationConfigWebApplicationContext` i `XmlWebApplicationContext` powiemy w rozdziale 8., podczas dyskusji o aplikacjach internetowych w Springu.

Na razie załadujmy kontekst aplikacji z systemu plików, korzystając z kontekstu `FileSystemXmlApplicationContext`, lub ze ścieżki do klas, korzystając z kontekstu `ClassPathXmlApplicationContext`.

Ładowanie kontekstu aplikacji z systemu plików lub ścieżki do klas odbywa się w podobny sposób do tego, jak ładowaliśmy komponenty do fabryki. Na przykład, w następujący sposób ładujemy kontekst `FileSystemXmlApplicationContext`:

```
ApplicationContext context = new
       FileSystemXmlApplicationContext("c:/knight.xml");
```

Podobnie ładujemy kontekst z użyciem `ClassPathXmlApplicationContext`:

```
ApplicationContext context = new
       ClassPathXmlApplicationContext("knight.xml");
```

Różnica między tymi dwoma sposobami polega na tym, że kontekst `FileSystemXmlApplicationContext` oczekuje pliku *knight.xml* w określonej lokalizacji, podczas gdy kontekst `ClassPathXmlApplicationContext` szuka go gdziekolwiek w ścieżce do klas (włącznie z plikami JAR).

Możemy również użyć klasy `AnnotationConfigApplicationContext`, aby załadować kontekst aplikacji z konfiguracji zawartej w klasie Javy:

```
ApplicationContext context = new AnnotationConfigApplicationContext(
   com.springinaction.knights.config.KnightConfig.class);
```

W tym wypadku, gdy ładujemy kontekst, nie wskazujemy żadnego pliku XML. Komponenty wczytywane są z klasy konfiguracyjnej przekazanej jako argument konstruktora klasy `AnnotationConfigApplicationContext`.

Mając już kontekst aplikacji, możemy pobrać komponenty z kontenera Springa, wywołując należącą do kontenera metodę `getBean()`.

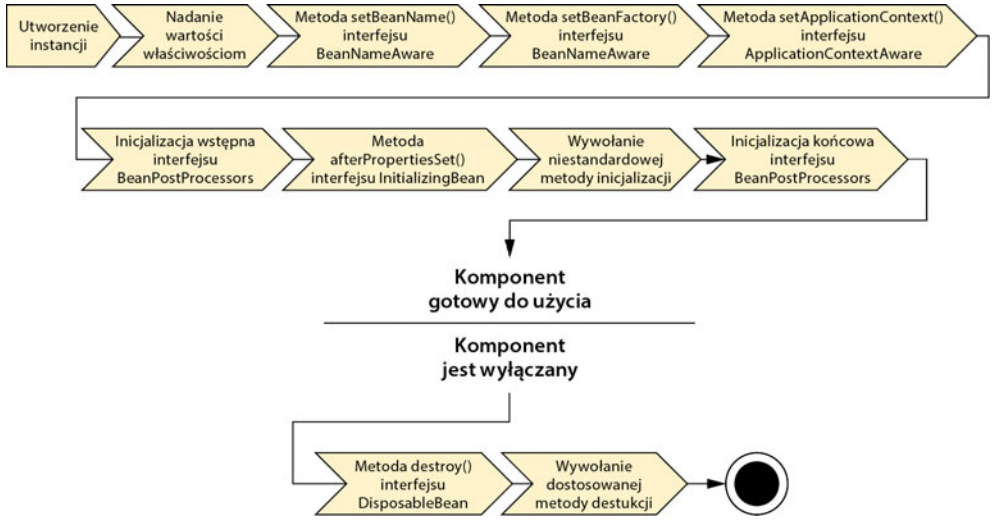
Kiedy znamy już podstawy tworzenia kontenera w Springu, przyjrzyjmy się bliżej cyklowi życia komponentu w kontenerze.

1.2.2. Cykl życia komponentu

W przypadku tradycyjnej aplikacji w Javie, cykl życia komponentu jest prosty. Słowo kluczowe `new` powoduje utworzenie instancji komponentu i od tego momentu komponent jest gotów do działania. Odkąd komponent nie jest już w użyciu, jest on przekazywany do procesu oczyszczania pamięci i w pewnym momencie trafia do „wielkiego kubła bitów na niebie".

Z kolei cykl życia komponentu w kontenerze Springa jest bardziej skomplikowany. Ważne jest zrozumienie cyklu życia komponentu w Springu, ponieważ możemy chcieć skorzystać z niektórych oferowanych przez Springa możliwości dostosowania sposobu tworzenia komponentu. Na rysunku 1 5 pokazano początek cyklu życia dla typowego komponentu, ładowanego do kontekstu aplikacji w Springu.

Jak widzimy, fabryka komponentów wykonuje wiele kroków konfiguracji, zanim komponent będzie gotowy do użycia. Ujmując zawartość rysunku 1 5 bardziej szczegółowo:



Rysunek 1.5. Komponent w kontenerze Springa przechodzi pewną liczbę etapów pomiędzy utworzeniem i likwidacją. Każdy etap daje możliwość modyfikacji sposobu zarządzania komponentem w Springu

1. Spring tworzy instancję komponentu.
2. Spring wstrzykuje do właściwości komponentu wartości i referencje do innych komponentów.
3. Jeśli komponent implementuje interfejs `BeanNameAware`, Spring przekazuje metodzie `setBeanName()` jako parametr nazwę komponentu.
4. Jeśli komponent implementuje interfejs `BeanFactoryAware`, Spring wywołuje metodę `setBeanFactory()`, z bieżącą fabryką komponentów jako parametrem.
5. Jeśli implementuje interfejs `ApplicationContextAware`, Spring wywołuje metodę `setApplicationContext()`, przekazując jej jako parametr referencję do zawierającego ten komponent kontekstu aplikacji.
6. Jeśli komponent implementuje interfejs `BeanPostProcesor`, Spring wywołuje jego metodę `postProcessBeforeInitialization()`.
7. Jeśli komponent implementuje interfejs `InitializingBean`, Spring wywołuje jego metodę `afterPropertiesSet()`. Podobnie, jeśli komponent posiada deklarację `init-method`, wówczas wywoływana jest wskazana metoda inicjująca.
8. Jeśli komponent implementuje interfejs `BeanPostProcessor`, Spring wywołuje jego metodę `postProcessAfterInitialization()`.
9. Dochodząc do tego punktu, komponent jest gotów do użycia przez aplikację i pozostanie w kontekście aplikacji, dopóki ten nie zostanie zlikwidowany.
10. Jeśli komponent implementuje interfejs `DisposableBean`, Spring wywołuje jego metodę `destroy()`. Podobnie, jeśli komponent posiada deklarację `destroy-method`, wówczas wywoływana jest wskazana metoda.

Teraz już wiemy, jak utworzyć i załadować kontener w Springu. Lecz pusty kontener sam z siebie nie stanowi wielkiej wartości, a nie będzie niczego zawierał, zanim sami

tam czegoś nie umieścimy. Aby osiągnąć korzyść z techniki DI w Springu, musimy dokonać wiązania obiektów naszej aplikacji wewnątrz kontenera. Bardziej szczegółowo do kwestii wiązania komponentów podejdziemy w rozdziale 2.

Obejrzyjmy najpierw krajobraz nowoczesnego Springa — przekonując się, w jaki sposób jest skonstruowane środowisko i co mają do zaoferowania jego najnowsze wersje.

1.3. Podziwiamy krajobraz Springa

Jak widzieliśmy, działanie frameworka Spring koncentruje się na upraszczaniu tworzenia oprogramowania klasy enterprise w Javie dzięki wstrzykiwaniu zależności, programowaniu aspektowemu i eliminacji kodu szablonowego. Nawet gdyby na tym kończyła się korzyść z użycia Springa, byłoby warto. Lecz Spring ma do zaoferowania więcej, niż widać na pierwszy rzut oka.

Sam Spring Framework oferuje kilka sposobów na ułatwienie programowania, lecz za frameworkiem stoi jeszcze potężny ekosystem projektów bazujących na nim jako podłożu, rozciągając możliwości Springa na takie obszary, jak usługi sieciowe, REST, aplikacje mobilne, a nawet bazy NoSQL.

Rozłóżmy frameworka Spring na elementy, by przekonać się, co ma do zaoferowania. Następnie rozszerzymy horyzonty, dokonując przeglądu wielkiej rodziny projektów związanych ze Springiem.

1.3.1. Moduły Springa

Po pobraniu i rozpakowaniu dystrybucji Springa w głębi struktury plików w folderze *libs* znajdziemy kilka plików JAR. Dystrubucja frameworka Spring w wersji 4.0 zbudowana jest z 20 oddzielnych modułów. Każdy z nich składa się z trzech plików JAR (binarnej biblioteki klas, pliku JAR ze źródłami oraz pliku JAR z dokumentacją JavaDoc). Pełną listę plików bibliotek JAR pokazano na rysunku 1.6.

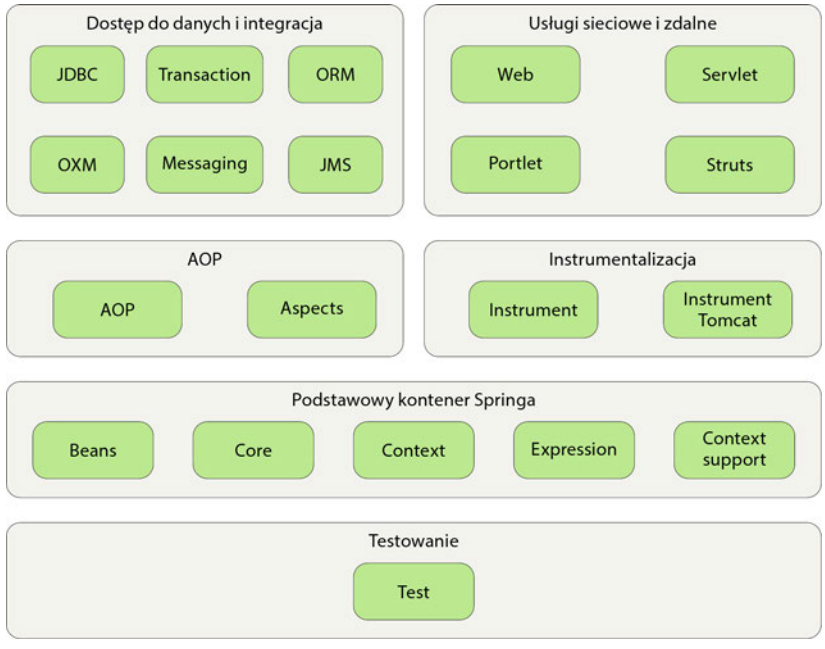
Te moduły można podzielić na sześć kategorii, jak pokazano na rysunku 1.7.

Jako całość moduły te zaspokajają wszystkie potrzeby programisty piszącego aplikacje klasy enterprise. Lecz nasze aplikacje nie muszą w całości bazować na frameworku Spring. Możemy swobodnie dobierać moduły odpowiednie dla naszej aplikacji, korzystając także

Nazwa	Rozmiar	Typ	Czas modyfikacji
spring-aop-4.0.9.RELEASE.jar	352,8 kB	Archiwum Java	30.12.2014
spring-aspects-4.0.9.RELEASE.jar	52,3 kB	Archiwum Java	30.12.2014
spring-beans-4.0.9.RELEASE.jar	672,3 kB	Archiwum Java	30.12.2014
spring-context-4.0.9.RELEASE.jar	978,8 kB	Archiwum Java	30.12.2014
spring-context-support-4.0.9.RELEASE.jar	135,8 kB	Archiwum Java	30.12.2014
spring-core-4.0.9.RELEASE.jar	978,4 kB	Archiwum Java	30.12.2014
spring-expression-4.0.9.RELEASE.jar	205,5 kB	Archiwum Java	30.12.2014
spring-instrument-4.0.9.RELEASE.jar	7,2 kB	Archiwum Java	30.12.2014
spring-instrument-tomcat-4.0.9.RELEASE.jar	10,5 kB	Archiwum Java	30.12.2014
spring-jdbc-4.0.9.RELEASE.jar	424,7 kB	Archiwum Java	30.12.2014
spring-jms-4.0.9.RELEASE.jar	211,1 kB	Archiwum Java	30.12.2014
spring-messaging-4.0.9.RELEASE.jar	266,7 kB	Archiwum Java	30.12.2014
spring-orm-4.0.9.RELEASE.jar	368,0 kB	Archiwum Java	30.12.2014
spring-oxm-4.0.9.RELEASE.jar	80,8 kB	Archiwum Java	30.12.2014
spring-test-4.0.9.RELEASE.jar	449,0 kB	Archiwum Java	30.12.2014
spring-tx-4.0.9.RELEASE.jar	248,4 kB	Archiwum Java	30.12.2014
spring-web-4.0.9.RELEASE.jar	671,3 kB	Archiwum Java	30.12.2014
spring-webmvc-4.0.9.RELEASE.jar	665,0 kB	Archiwum Java	30.12.2014
spring-webmvc-portlet-4.0.9.RELEASE.jar	175,6 kB	Archiwum Java	30.12.2014
spring-websocket-4.0.9.RELEASE.jar	280,1 kB	Archiwum Java	30.12.2014

20 elementów (7,2 MB), wolna przestrzeń: 234,6 GB

Rysunek 1.6. Pliki JAR należące do dystrybucji frameworka Spring



Rysunek 1.7. Framework Spring składa się z sześciu dobrze definiowanych modułów

z innych źródeł, gdy Spring okaże się niewystarczający. W Springu mamy nawet do dyspozycji punkty integracji z kilkoma innymi frameworkami i bibliotekami, więc nie musimy sami ich pisać.

Przyjrzyjmy się kolejno każdemu z modułów, tworząc sobie stopniowo całościowy obraz frameworka Spring.

PODSTAWOWY KONTENER SPRINGA

Centralnym elementem frameworka Spring jest kontener, który kontroluje sposób tworzenia i konfiguracji komponentów w aplikacji stworzonej na bazie tego frameworka oraz zarządzania tymi komponentami. W tym module znajdziemy fabrykę komponentów, czyli element frameworka realizujący wstrzykiwanie zależności. Mamy też kilka implementacji kontekstu aplikacji w Springu, będących rozszerzeniem fabryki komponentów, z których każda umożliwia inny sposób konfiguracji Springa.

Poza fabryką komponentów i kontekstami aplikacji moduł ten zawiera wiele usług dla aplikacji biznesowych, jak poczta elektroniczna, dostęp przez JNDI, integracja EJB i planowanie zadań.

Wszystkie moduły Springa bazują na podstawowym kontenerze. Klas tych oczywiście użyjemy, konfigurując naszą aplikację. Podstawowy moduł będziemy omawiać w całej tej książce, rozpoczynając od rozdziału 2., w którym zagłębimy się we wstrzykiwanie zależności w Springu.

MODUŁ AOP W SPRINGU

Spring posiada w module AOP szerokie wsparcie dla programowania aspektowego. Moduł ten stanowi punkt wyjścia do tworzenia własnych aspektów dla naszej aplikacji w Springu. Podobnie jak DI, AOP pozwala na luźne wiązanie obiektów aplikacji. Lecz dzięki AOP zagadnienia rozproszone po całej aplikacji (jak obsługa transakcji i bezpieczeństwa) zostają oddzielone od obiektów, które z nich korzystają.

W obsługę AOP w Springu zagłębimy się w rozdziale 4.

DOSTĘP DO DANYCH I INTEGRACJA

Praca z JDBC często skutkuje potrzebą użycia sporej ilości kodu szablonowego do nawiązania połączenia, utworzenia wyrażenia, obsługi zbioru wynikowego i na koniec — zamknięcia połączenia. Moduł Springa obsługujący JDBC oraz **obiekty dostępu do danych** (ang. *Data Access Objects* — DAO) wydobywa z aplikacji kod szablonowy, pozwalając na tworzenie czystego i prostego kodu współpracującego z bazami danych, jednocześnie chroniąc przed problemami skutkującymi niepoprawnym zamknięciem lub nawet uszkodzeniem zasobów bazodanowych. Moduł ten obudowuje także komunikaty o błędach pochodzące od wielu serwerów baz danych warstwą wyjątków posiadających znacznie bardziej zrozumiałe komunikaty. Już nigdy więcej rozszyfrowywania tajnych lub zastrzeżonych komunikatów o błędach SQL!

Dla tych, którzy wolą używać narzędzi **odwzorowywania obiektowo-relacyjnego** (ang. *object-relational mapping* — ORM) zamiast bezpośredniego JDBC, Spring udostępnia moduł ORM. Obsługa ORM w Springu jest oparta na obsłudze DAO, zapewniając poręczny sposób na zbudowanie DAO dla wielu rozwiązań ORM. Spring nie próbuje implementować jakiegoś własnego rozwiązania ORM, lecz posiada przyłącza dla wielu popularnych środowisk ORM, w tym Hibernate, Java Persistance API, Java Data Objects czy iBATIS SQL Maps. Zarządzanie transakcjami w Springu obsługuje każde z tych środowisk ORM, w podobny sposób jak JDBC.

Przekonamy się, w jaki sposób oparta na szablonach abstrakcja JDBC w Springu pozwala znacznie uprościć kod korzystający z JDBC, analizując dostęp do danych w Springu, opisany w rozdziale 10.

Moduł ten zawiera także abstrakcję usługi wiadomości Javy (ang. *Java Message Service* — JMS), pozwalającej na asynchroniczną integrację z innymi aplikacjami za pomocą wiadomości. Począwszy od wersji 3.0 Springa, w module tym znajdziemy także funkcje odwzorowywania obiektów do XML, które pierwotnie były częścią projektu Spring Web Services.

Dodatkowo, moduł ten korzysta z modułu AOP, by udostępnić obiektom w aplikacji usługi zarządzania transakcjami.

USŁUGI SIECIOWE I ZDALNE

Paradygmat **model-widok-kontroler** (ang. *Model-View-Controller* — MVC) jest powszechnie przyjętym podejściem do konstruowania aplikacji internetowych, w których interfejs użytkownika jest rozdzielony od logiki aplikacji. W Javie nie brakuje frameworków MVC, wśród których mamy Apache Struts, JSF, Web Work czy Tapestry, by wymienić jedynie najbardziej popularne rozwiązania.

Mimo że Spring integruje się z wieloma popularnymi frameworkami MVC, jego moduł usług sieciowych i zdalnych zawiera niezłe środowisko MVC, które wspiera Springową technikę luźnego wiązania w webowej warstwie aplikacji. Frameworkowi MVC Springa przyjrzymy się bliżej w rozdziałach 5 – 7.

Poza aplikacjami internetowymi, z którymi będzie miał bezpośrednio do czynienia użytkownik, moduł ten posiada kilka opcji usług zdalnych, na potrzeby konstruowania aplikacji, które będą współpracować z innymi aplikacjami. Obsługa usług zdalnych w Springu obejmuje rozwiązanie takie jak **zdalne wywoływanie metod** (ang. *Remote Method Invocation* — RMI), Hessian, Burlap, JAX-WS, a także należący do Springa mechanizm HTTP invoker. Spring oferuje też wysoki poziom wsparcia dla wystawiania i pobierania danych z REST-owych API.

W rozdziale 15. poznamy szczegóły usług zdalnych w Springu. W rozdziale 16. dowiesz się z kolei, jak tworzyć API REST-owe i konsumować pobrane z niego dane.

INSTRUMENTACJA

Springowy moduł instrumentacji zawiera wsparcie dla dodawania agentów na platformie JVM. W szczególności dostarcza agenta splatania dla serwera Tomcat, którego zadaniem jest przekształcanie plików klas po załadowaniu przez mechanizm classloadera.

Jeśli czytając ten opis, czujesz się nieco przytłoczony, nie martw się. Funkcjonalność instrumentacji dostarczana przez ten moduł ma wąski zakres zastosowań i w tej książce nie będziemy się nią wcale zajmować.

TESTOWANIE

Dostrzegając ważność testów pisanych przez programistę, w Springu zawarto moduł dedykowany do testowania aplikacji utworzonych w tym środowisku.

W tym module znajdziemy kolekcję imitacji implementowanych obiektów na potrzeby tworzenia przez programistów testów jednostkowych, obsługujących JNDI, serwlety i portlety. Na potrzeby testowania etapu integracji moduł ten zawiera obsługę ładowania kolekcji komponentów do kontekstu aplikacji w Springu i pracy z komponentami w tym kontekście.

Wiele przykładów, które pojawią się w tej książce, powstanie na bazie testów z wykorzystaniem możliwości oferowanych przez Springa w tym zakresie.

1.3.2. Rodzina projektów wokół Springa

Spring to więcej, niż widać na pierwszy rzut oka. Właściwie to znacznie więcej niż dostępny do pobrania z internetu pakiet instalacyjny frameworka Spring. Gdybyśmy zatrzymali się na samych tylko podstawach frameworka Spring, przegapilibyśmy bogactwo potencjału, jaki posiada wielka rodzina otaczających go projektów. Rodzina ta zawiera wiele środowisk uruchomieniowych i bibliotek, które bazują na rdzeniu frameworka Spring lub na sobie nawzajem. Wzięte razem, projekty te obejmują modelem programowania Springa prawie każdą płaszczyznę programowania w Javie.

Zajęłoby wiele tomów, by opisać wszystko, co ma do zaoferowania cała rodzina projektów otaczających Springa, i większość z tego wykracza poza zakres tej książki.

Lecz zerkniemy na niektórych przedstawicieli tej wielkiej rodziny. Oto przedsmak tego wszystkiego, co leży poza rdzeniem frameworka Spring.

SPRING WEB FLOW

Spring Web Flow bazuje na należącym do frameworka Spring module MVC, by umożliwić konstruowanie konwersacyjnych, bazujących na przepływie informacji aplikacji internetowych prowadzących użytkownika do celu (pomyślmy o kreatorach lub koszykach sklepowych). Powiemy więcej o Spring Web Flow w rozdziale 8., a jeszcze więcej o tym projekcie można się dowiedzieć z jego strony domowej pod adresem *http://projects.spring.io/spring-webflow/*.

SPRING WEB SERVICES

Choć rdzeń frameworka Spring pozwala na deklaracyjną obsługę publikowania komponentów Springa jako usług sieciowych, usługi te oparte są na niewątpliwie gorszym pod względem architektury modelu „kontrakt na końcu". Kontrakt na usługę jest ustalany na podstawie interfejsu implementowanego przez komponent. Spring Web Services korzysta z modelu „najpierw kontrakt", w którym implementacje usług pisane są, by spełnić kontrakt na usługi.

Nie będziemy omawiać w tej książce projektu Spring-WS, zainteresowany nim czytelnik może sięgnąć po więcej informacji na stronę domową projektu pod adresem *http://docs.spring.io/spring-ws/site/*.

SPRING SECURITY

Bezpieczeństwo jest najważniejszym aspektem działania wielu aplikacji. Projekt Spring Security, bazujący w swej implementacji na module AOP Springa, umożliwia deklarowanie mechanizmów bezpieczeństwa w aplikacjach opartych na Springu. Jak dodać do warstwy internetowej aplikacji rozwiązania projektu Spring Security, dowiemy się w rozdziale 9. Na potrzeby dalszych poszukiwań, pod adresem *http://projects.spring.io/spring-security/* znajduje się strona domowa projektu Spring Security.

SPRING INTEGRATION

Wiele aplikacji biznesowych musi współpracować z innymi aplikacjami biznesowymi. Projekt Spring Integration udostępnia implementacje wielu popularnych wzorców integracji realizowanych w deklaratywnym stylu Springa.

Nie opisujemy w tej książce projektu Spring Integration. Osobom zainteresowanym szczegółami tego projektu polecamy książkę *Spring Integration in Action*, którą napisali Mark Fisher, Jonas Partner, Marius Bogoevici i Iwein Fuld. Albo też wizytę na stronie domowej projektu, pod adresem *http://projects.spring.io/spring-integration/*.

SPRING BATCH

Kiedy zachodzi potrzeba wykonania hurtowo pewnych operacji na danych, nie do pobicia jest przetwarzanie wsadowe. Jeśli aplikacja, którą zamierzasz pisać, będzie przetwarzała dane wsadowo, możesz oprzeć ją na solidnym, korzystającym z POJO modelu programowania w Springu, korzystając z projektu Spring Batch.

Projekt ten wykracza poza zakres niniejszej książki. Jednak mogą Cię, Czytelniku, oświecić Arnaud Cogoluègnes, Thierry Templier, Gary Gregory i Olivier Bazoud w książce ich autorstwa, pod tytułem *Spring Batch in Action*. Możesz także nauczyć się korzystania ze Spring Batch na stronie domowej projektu, pod adresem *http://projects.spring.io/spring-batch/*.

SPRING DATA

Spring Data ułatwia pracę w Springu z wszelkiego rodzaju bazami danych. Chociaż bazy relacyjne są już od wielu lat obecne w aplikacjach klasy enterprise, w nowoczesnych aplikacjach pojawił się pomysł, że w przypadku niektórych danych kolumny i wiersze w tabeli nie są może najlepszą możliwą reprezentacją. Bazy danych nowego typu, zwane często bazami NoSQL[2], pozwalają na nowe metody pracy z danymi, bardziej odpowiednie niż oferowane do tej pory przez tradycyjne bazy relacyjne.

Niezależnie od tego, czy korzystamy z dokumentowej bazy danych, takiej jak MongoDB, bazy grafowej, takiej jak Neo4j, czy nawet relacyjnej bazy danych, Spring Data udostępnia nam uproszczony model programistyczny persystencji danych. Model ten obejmuje mechanizm automatycznego tworzenia implementacji repozytorium dla wielu typów baz danych.

W rozdziale 11. przyjrzymy się bliżej temu, jak Spring Data ułatwia prace programistyczne związane z JPA (Java Persistence API), a w rozdziale 12. rozszerzymy obserwację o kilka baz NoSQL.

SPRING SOCIAL

Sieci społecznościowe są coraz popularniejszym zagadnieniem w internecie i coraz więcej aplikacji zostaje wyposażonych w możliwość integracji z serwisami sieci społecznościowych, takich jak Facebook lub Twitter. Jeśli jest to tematyka leżąca w kręgu Twoich zainteresowań, na pewno będziesz chciał zapoznać się z projektem Spring Social, czyli rozszerzeniem do Springa obsługującym sieci społecznościowe.

Spring Social to nie tylko wiadomości tekstowe i znajomi. Pomimo nazwy projektu w Spring Social większy jest nacisk na integrację niż na same sieci społecznościowe. Projekt ułatwia integrację aplikacji springowych z REST-owymi API, także tymi, które nie mają nic wspólnego z sieciami społecznościowymi.

Z powodu ograniczonej objętości książki nie będę więcej mówić o Spring Social. Jeśli chcesz dowiedzieć się, w jaki sposób Spring ułatwia integrację z Facebookiem i Twitterem, zajrzyj do sekcji Getting Started („Zaczynamy") oficjalnego podręcznika, znajdującego się pod adresami *https://spring.io/guides/gs/accessing-facebook/* oraz *https://spring.io/guides/gs/accessing-twitter/*.

---

[2] Osobiście od słowa *NoSQL* wolę termin *bazy nierelacyjne* lub *bezschematowe*. Określanie tych baz pojęciem *NoSQL* wskazuje na problem z językiem zapytań, a nie modelem bazy danych.

SPRING MOBILE

Aplikacje mobilne są kolejnym ważnym kierunkiem rozwoju oprogramowania. Smartfony i tablety przejmują u wielu użytkowników pozycję preferowanych rozwiązań klienckich. Spring Mobile jest nowym rozszerzeniem dla Spring MVC, wspierającym programowanie mobilnych aplikacji internetowych.

SPRING FOR ANDROID

Spokrewniony ze Spring Mobile jest projekt Spring Android. Celem tego projektu jest przeniesienie prostoty osiągniętej przez środowisko Spring do programowania natywnych aplikacji dla urządzeń z systemem Android. Na początkowym etapie rozwoju projekt ten udostępnia jedynie wersję biblioteki `RestTemplate` działającej w aplikacjach dla Androida. Współpracuje też z projektem Spring Social, aby umożliwić integrację natywnej aplikacji na platformę Android z REST-owymi API.

Nie będę omawiać projektu Spring For Android w tej książce, lecz można dowiedzieć się o nim więcej pod adresem *http://projects.spring.io/spring-android/*.

SPRING BOOT

Spring bardzo upraszcza wiele zadań programistycznych, redukuje, a nawet eliminuje dużą część kodu szablonowego, który w przeciwnym wypadku trzeba by samodzielnie napisać. Spring Boot to nowy, ekscytujący projekt i świeże podejście, polegające na programowaniu z użyciem Springa w jeszcze bardziej przystępny sposób.

Spring Boot w dużym stopniu wykorzystuje techniki automatycznej konfiguracji, co pozwala wyeliminować większość ustawień konfiguracji Springa (a często całkowicie je usunąć). Udostępnia też kilka projektów startowych, dzięki czemu zmniejsza się rozmiar plików budowania projektu Springa, niezależnie od tego, czy korzystamy z Mavena czy Gradle'a.

Projektowi Spring Boot przyjrzymy się w jednym z ostatnich rozdziałów tej książki, w rozdziale 21.

1.4. Co nowego w Springu

W chwili pojawienia się trzeciej edycji tej książki najnowszą wersją Springa była wersja 3.0.5. Miało to miejsce około trzech lat temu i od tego czasu wiele się wydarzyło. Ukazały się trzy kolejne wydania frameworka Spring — 3.1, 3.2 oraz 4.0 — a każde z nich przynosiło nowe mechanizmy i ulepszenia służące łatwiejszemu tworzeniu aplikacji. Także wiele z rodziny projektów otaczających framework Spring uległo znaczącym zmianom.

Niniejsza edycja książki została zaktualizowana i opisuje wiele ekscytujących i użytecznych funkcji dołączonych w tych wydaniach. Na razie jednak skrótowo zarysujemy nowości we frameworku Spring.

1.4.1. Co nowego w Springu 3.1?

Spring 3.1 zawiera kilka nowych, użytecznych funkcji i usprawnień. Wiele z nich ma za zadanie uproszczenie i polepszenie konfiguracji. Dodatkowo w Springu 3.1 wprowadzono obsługę deklaratywnego cache'a, jak również usprawnienia w Spring MVC. Poniższa lista zawiera spis najważniejszych zmian wprowadzonych w Springu 3.1:

- W odpowiedzi na częstą potrzebę oddzielnej konfiguracji dla różnych środowisk (na przykład dla środowiska deweloperskiego, testowego i produkcyjnego) wprowadzono obsługę profili środowiskowych. Profile umożliwiają na przykład wybór źródła danych w zależności od środowiska, na które została wdrożona aplikacja.
- Rozszerzono obsługę konfiguracji opartej na klasach Javy, wprowadzonej w wersji 3.0, o możliwość włączenia niektórych funkcji Springa za pomocą pojedynczej adnotacji.
- Pojawiła się obsługa deklaratywnego cache'a, co umożliwiło deklarowanie granic i reguł cachowania z użyciem prostych adnotacji w sposób podobny jak w przypadku deklaracji granic transakcji.
- Nowa przestrzeń nazw c umożliwiła wstrzykiwanie zależności przez konstruktor w taki sam zwięzły sposób jak wprowadzona w Springu 2.0 przestrzeń nazw p, pozwalająca na wstrzykiwanie zależności przez właściwość.
- Wprowadzono obsługę Servlet 3.0, włączając w to możliwość deklarowania sewletów i filtrów w konfiguracji opartej na plikach Java w miejscu pliku *web.xml*.
- Usprawniono obsługę JPA w Springu, dzięki czemu możliwa stała się konfiguracja JPA całkowicie w Springu bez konieczności istnienia pliku *persistence.xml*.

W Springu 3.1 wprowadzono kilka usprawnień Spring MVC:

- automatyczne wiązanie zmiennych ścieżek do atrybutów modeli;
- atrybuty `produces` oraz `consumes` adnotacji `@RequestMatching` na potrzeby dopasowywania nagłówków `Accept` i `Content-Type` w żądaniach HTTP;
- adnotację `@RequestPart`, umożliwiającą wiązanie części żądania wieloczęściowego do parametrów metody obsługującej żądanie;
- obsługę atrybutów `flash` (atrybutów dostępnych po wykonaniu przekierowania) oraz typ `RedirectAttributes` pozwalający na przenoszenie tych atrybutów pomiędzy żądaniami.

Równie istotne jak nowości wprowadzone w Springu 3.1 jest to, co przestało być dostępne. W szczególności klasy `JpaTemplate` i `JpaDaoSupport` zostały uznane za przestarzałe i zrezygnowano z nich na rzecz natywnej klasy `EntityManager`. Pomimo statusu „przestarzałe" obie klasy były jeszcze dostępne w wersji 3.2 Springa. Korzystanie z nich nie było jednak zalecane, bo nie otrzymały aktualizacji do obsługi JPA 2.0, a w wersji 4. Springa ostatecznie je usunięto.

Zobaczmy teraz, jakie nowości przyniosło wprowadzenie Springa 3.2.

1.4.2. Co nowego w Springu 3.2?

Zmiany wprowadzone w Springu 3.1 polegały głównie na ulepszeniach w konfiguracji i tylko w mniejszym stopniu rozszerzały inne funkcjonalności, takie jak Spring MVC. W wydaniu 3.2 Springa skoncentrowano się na zmianach związanych ze Spring MVC. Wprowadzono w nim następujące ulepszenia:

- Kontrolery w Spring 3.2 umożliwiają wykorzystywanie żądań asynchronicznych, które pojawiły się w serwletach w wersji 3. i które pozwalają przekazać przetwarzanie żądań do osobnych wątków, zwalniając tym samym główny wątek serwletu, co z kolei umożliwia przetworzenie większej liczby żądań.
- Testowanie kontrolerów jako obiektów POJO było proste już w Spring MVC 2.5. W Springu 3.2 dołączono framework do testów, który pozwala na przeprowadzenie rozbudowanych testów kontrolerów i weryfikację ich zachowania bez udziału kontenera serwletów.
- Oprócz ulepszeń w zakresie testowania kontrolerów w Springu 3.2 dodano możliwość testowania klientów bazujących na klasie `RestTemplate` bez potrzeby wysyłania żądań do prawdziwych endpointów REST-owych.
- Adnotacja `@ControllerAdvice` umożliwia zebranie metod `@ExceptionHandler`, `@InitBinder` oraz `@ModelAttributes` w ramach jednej klasy i ich zastosowanie do wszystkich kontrolerów.
- Przed wprowadzeniem Springa 3.2 obsługa pełnej negocjacji zawartości była możliwa jedynie za pośrednictwem klasy `ContentNegotiatingViewResolver`. W Springu 3.2 jest już ona również możliwa za pośrednictwem Spring MVC, nawet na metodach kontrolera podlegających działaniu konwertera komunikatów przy odbieraniu żądania i wysyłaniu odpowiedzi.
- W Springu 3.2 dołączono nową adnotację, `@MatrixVariable`, umożliwiającą powiązanie zmiennych żądania w postaci macierzy z parametrami metody obsługującej żądanie.
- Umożliwiono wygodną konfigurację klasy `DispatcherServlet` bez użycia pliku *web.xml* za pomocą abstrakcyjnej klasy bazowej `AbstractDispatcherServletInitializer`.
- Pojawiła się klasa `ResponseEntityExceptionHandler`, która stanowi alternatywę dla klasy `DefaultHandlerExceptionResolver`. Metody klasy `ResponseEntityExceptionHandler` zwracają obiekty typu `ResponseEntity<Object>`, w odróżnieniu od metod swej „konkurentki", która zwraca obiekty typu `ModelAndView`.
- Klasa `RestTemplate` i argumenty adnotacji `@RequestBody` obsługują typy generyczne.
- Klasa `RestTemplate` i adnotacja `@RequestMapping` obsługują metodę HTTP PATCH.
- Mapowane interceptory umożliwiają pominięcie przechwytywania żądań na podstawie ustalonych wzorców URL.

Chociaż główny nacisk odnośnie do zmian wprowadzonych w Springu 3.2 położono na Spring MVC, pojawiło się też kilka ulepszeń w innych obszarach Springa. Najciekawsze nowe funkcjonalności to:

- Adnotacje `@Autowired`, `@Value` oraz `@Bean` można wykorzystać jako metaadnotacje do stworzenia własnych adnotacji wstrzykiwania zależności oraz deklaracji komponentów.
- Adnotacja `@DateTimeFormat` nie jest już w pełni zależna od biblioteki `JodaTime`. Jeśli biblioteka ta jest dostępna, zostanie wykorzystana, ale w przeciwnym wypadku używana jest klasa `SimpleDateFormat`.
- Obsługa deklaratywnego cache'a w Springu otrzymała wsparcie dla standardu JCache 0 5.
- Powstała możliwość deklaracji globalnego formatu parsowania i renderowania daty oraz czasu.
- W testach integracyjnych możliwa jest konfiguracja i wczytanie kontekstu `WebApplicationContext`.
- Testy integracyjne pozwalają na przeprowadzenie testów z wykorzystaniem komponentów o zasięgu żądania i sesji.

W tej książce poświęcimy trochę uwagi funkcjonalnościom wprowadzonym w wersji 3.2 Springa, zwłaszcza w rozdziałach dotyczących aplikacji internetowych i REST-a.

1.4.3. Co nowego w Springu 4.0?

Najnowszą dostępną wersją Springa w chwili powstawania tej książki jest Spring 4.0. Wersja ta przyniosła wiele ekscytujących nowych funkcjonalności, w tym:

- Obsługę programowania z wykorzystaniem WebSocketów, włącznie z obsługą standardu JSR-356: API Java dla WebSocketów.
- WebSockety udostępniają API niskiego poziomu, co aż się prosi o wprowadzenie warstwy abstrakcji wyższego poziomu. Spring 4.0 nałożył więc na nie wysokopoziomowy model programowania zorientowany na komunikaty i oparty na obsłudze biblioteki SockJS oraz subprotokołu STOMP.
- Nowy moduł obsługi komunikatów z możliwością wykorzystania wielu typów zaczerpnięty z projektu Spring Integration. Moduł ten współgra z obsługą SockJS/STOMP w Springu i zawiera obsługę publikacji komunikatów w oparciu o szablony.
- Spring 4.0 jest jednym z pierwszych (jak nie pierwszym) frameworków Java obsługującym funkcje Javy 8, takie jak na przykład lambdy. Zapewnia to szereg korzyści, upraszcza pracę i zwiększa czytelność kodu przy pracy z niektórymi interfejsami wywołań zwrotnych (takimi jak `RowMapper` z `JdbcTemplate`).
- Wraz ze wsparciem Javy 8 wprowadzono obsługę standardu JSR-310: API daty i czasu, pozwalając programistom na pracę z datą i czasem za pośrednictwem bardziej rozbudowanego API, niż oferują klasy `java.util.Data` i `java.util.`
  ↪`Calendar`.
- Tworzenie aplikacji z użyciem języka Groovy jest teraz naturalniejsze i przyjemniejsze. Udostępniono możliwość utworzenia aplikacji springowej w całości w tym języku. Wraz z językiem Groovy doszła możliwość konfiguracji aplikacji z użyciem klasy `BeanBuilder`, pochodzącej z frameworka Grails.

- Wprowadzona została uogólniona obsługa warunkowego tworzenia komponentów, która umożliwia ich deklarowanie jedynie po spełnieniu warunków ustalonych przez dewelopera.
- Pojawiła się nowa asynchroniczna implementacja klasy `RestTemplate`, która nie blokuje aplikacji, ale udostępnia obsługę wywołań zwrotnych po zakończeniu operacji.
- Dodano obsługę wielu specyfikacji JEE, wliczając w to JMS 2.0, JTA 1.2, JPA 2.1 oraz Bean Validation 1.1.

Jak widać, najnowsza wersja Springa przyniosła mnóstwo ekscytujących funkcjonalności. W tej książce opisanych zostanie wiele z wymienionych nowości, jak również wiele funkcji, które wprowadzono już we wczesnych wersjach Springa.

1.5. Podsumowanie

Teraz powinieneś już, Czytelniku, mieć niezłe wyobrażenie o tym, co Spring przynosi ze sobą. Celem Springa jest uczynienie programowania aplikacji biznesowych łatwiejszym i promowanie luźno wiązanego kodu. Fundamentem jest wstrzykiwanie zależności i programowanie aspektowe.

W tym rozdziale uzyskaliśmy przedsmak wstrzykiwania zależności w Springu. DI jest sposobem na powiązanie między sobą obiektów w aplikacji, tak że obiekty nie posiadają informacji o swoich zależnościach i sposobie ich implementacji. Zamiast same uzyskiwać połączenie z obiektami, od których zależą, otrzymują je, bez żadnej aktywności w nich samych. Ponieważ zależne obiekty często otrzymują informację o obiektach wstrzykiwanych za pośrednictwem interfejsów, ma miejsce luźne wiązanie.

Poza wstrzykiwaniem zależności rzuciliśmy okiem na obsługę AOP w Springu. AOP pozwala logikę, która normalnie byłaby rozrzucona po całej aplikacji, skupić w jednym miejscu — w aspekcie. Podczas wiązania ze sobą naszych beanów może być realizowane wplatanie aspektów w trakcie pracy, skutkujące nadaniem beanom nowej funkcjonalności.

Wstrzykiwanie zależności i AOP są podstawą działania Springa. Zatem konieczne jest, byś nauczył się używać tych podstawowych funkcji, by móc korzystać z pozostałej części możliwości tego frameworka. Treść tego rozdziału zaledwie musnęła powierzchnię tematu funkcjonalności DI oraz AOP w Springu. W następnych kilku rozdziałach zagłębimy się bardziej w tę tematykę.

Zatem bez dalszej zwłoki przejdźmy do rozdziału 2., by nauczyć się wiązania obiektów między sobą w Springu za pomocą wstrzykiwania zależności.

2 Tworzymy powiązania między komponentami

W tym rozdziale omówimy:

- Deklarowanie komponentów
- Wstrzykiwanie przez konstruktory i modyfikatory
- Tworzenie powiązań między komponentami
- Sterowanie przebiegiem tworzenia i usuwania komponentów

Czy zdarzyło Ci się, Czytelniku, zostać w kinie po projekcji wystarczająco długo, by widzieć wszystkie napisy na końcu? To niesamowite, jak wiele osób jest zaangażowanych w powstanie przyzwoitego filmu. Poza oczywistymi uczestnikami tego procesu — takimi jak aktorzy, scenarzyści, reżyserzy i producenci — pojawiają się tam też nie aż tak oczywiści muzycy, ekipa od efektów specjalnych i dyrektorzy artystyczni, że nie wspomnę o osobach kluczowych — montażyście dźwięku, projektantach kostiumów, makijażystach, koordynatorach kaskaderów, publicystach, pierwszym asystencie kamerzysty, drugim asystencie kamerzysty, projektantach scenografii, kierowniku produkcji i (być może najważniejszych) osobach zajmujących się kateringiem.

Teraz wyobraź sobie, Czytelniku, jak wyglądałby Twój ulubiony film, gdyby żadna z wymienionych powyżej osób nie odzywała się do pozostałych. Powiedzmy, że wszyscy dotarli do studio i każdy zaczął zajmować się swoimi sprawami bez kogoś, kto by wszystko koordynował. Gdyby reżyser zamknął się w sobie i nie powiedział „kręcimy", wtedy kamerzysta nie zacząłby filmować. To i tak prawdopodobnie nie miałoby znaczenia,

ponieważ aktorka odtwarzająca główną rolę nadal przebywałaby w swojej garderobie, a oświetlenie nie działałoby, ponieważ nikt nie zatrudnił kierownika produkcji. Może widziałeś film, który wygląda, jakby właśnie tak się stało. Jednak większość filmów (a przynajmniej te dobre) to wynik wspólnej pracy tysięcy ludzi, dążących do wspólnego celu, którym jest stworzenie kinowego hitu.

Pod tym względem wspaniałe oprogramowanie niewiele się różni. Każda niebanalna aplikacja jest zbudowana z wielu obiektów, które muszą współpracować ze sobą, by zrealizować jakieś zadanie biznesowe. Obiekty te muszą być poinformowane o sobie nawzajem i komunikować się między sobą, by zrealizować postawione przed nimi zadanie. Przykładowo, w aplikacji sklepu internetowego komponent zarządzający zamówieniami może potrzebować współpracy z komponentem zarządzania produktami i komponentem autoryzacji kart kredytowych. Zaś one wszystkie prawdopodobnie będą potrzebowały współpracy z komponentem dostępu do danych, by czytać z bazy danych i zapisywać do niej.

Lecz, jak widzieliśmy w rozdziale 1., tradycyjne podejście do tworzenia powiązań między obiektami aplikacji (za pomocą wbudowywania lub odwołań) prowadzi do skomplikowanego kodu, który jest trudny do ponownego użycia, a także do przeprowadzenia testów jednostkowych. W najlepszym razie obiekty te będą wykonywały więcej pracy, niż to potrzebne. W najgorszym ich silne powiązanie między sobą uniemożliwi ich ponowne wykorzystanie lub testowanie.

W Springu obiekty nie odpowiadają za odnajdywanie albo tworzenie innych obiektów, potrzebnych im do działania. Zamiast tego referencje do obiektów, z którymi współpracują, otrzymują przez kontener. Na przykład, komponent zarządzania zamówieniami może potrzebować komponentu autoryzacji kart kredytowych — lecz nie musi tworzyć komponentu autoryzacji. Musi tylko pokazać, że ma puste ręce, a otrzyma moduł autoryzacji, z którego będzie mógł korzystać podczas pracy.

Czynność tworzenia tych właśnie powiązań między obiektami aplikacji jest istotą wstrzykiwania zależności (DI) i jest często określana wiązaniem. W tym rozdziale poznamy podstawy wiązania beanów za pomocą Springa. Jako że DI jest najbardziej elementarnym działaniem Springa, z techniki tej będziesz korzystać prawie zawsze podczas tworzenia aplikacji na bazie Springa.

Istnieje wiele sposobów wiązania komponentów w Springu. Na początek zapoznajmy się z trzema najczęściej stosowanymi metodami konfiguracji kontenera Springa.

2.1. Poznajemy opcje konfiguracji Springa

Tak jak wspominałem w rozdziale 1., kontener Springa odpowiada za tworzenie komponentów w aplikacji i koordynuje wzajemne relacje pomiędzy obiektami z użyciem mechanizmu wstrzykiwania zależności. Ale to Twoim zadaniem jako programisty jest wskazanie, które komponenty ma utworzyć Spring i w jaki sposób mają być one ze sobą powiązane. Spring jest niezwykle elastyczny pod względem oferowanych specyfikacji wiązania komponentów i udostępnia trzy główne mechanizmy definiowania tych powiązań:

- jawna konfiguracja w plikach XML;
- jawna konfiguracja za pomocą klas Javy;
- niejawna konfiguracja z użyciem wyszukiwania komponentów i automatycznych powiązań.

Na pierwszy rzut oka może się wydawać, że istnienie trzech opcji konfiguracji komplikuje pracę ze Springiem. Ich możliwości w pewnym zakresie się pokrywają, a decyzja wyboru najlepszej opcji w danej sytuacji może wydawać się trudna i przytłaczająca. Ale nie denerwuj się — w wielu przypadkach wybór jest kwestią gustu i możesz zdecydować się na tę opcję, która najbardziej Ci odpowiada.

Naprawdę świetnie, że Spring oferuje nam tak duże możliwości wiązania komponentów. W pewnym momencie musimy jednak wybrać jedną z nich.

Nie ma tutaj prostej odpowiedzi, na co się zdecydować. Każdy wybór musi być odpowiedni zarówno dla Ciebie, jak i Twojego projektu. Ale kto powiedział, że musisz ograniczyć się do wyboru jednej opcji? Style konfiguracji Springa można ze sobą łączyć i dopasowywać. Można więc wybrać pliki XML do powiązania niektórych komponentów, pliki z konfiguracją Javy do wiązania innych, a pozostałe komponenty powiązać automatycznie za pomocą mechanizmu wyszukiwania w Springu.

Ja zalecam korzystanie z automatycznej konfiguracji tak często, jak to jest możliwe. Im mniej konfiguracji zadeklarujesz w sposób jawny, tym lepiej. Gdy wymagana jest jawna konfiguracja komponentów (na przykład do konfiguracji komponentów, których kodem źródłowym nie zarządzamy), polecam umieszczenie konfiguracji w plikach Java, które są bezpieczne ze względu na typ i oferują większe możliwości konfiguracyjne niż pliki XML. Konfiguracji XML najlepiej używać tylko w sytuacjach, kiedy mamy możliwość skorzystania z wygodnej przestrzeni nazw XML, która nie ma swojego odpowiednika w konfiguracji Javy.

W tym rozdziale przyjrzymy się szczegółowo każdej z tych technik, a w następnych rozdziałach będziemy z nich aktywnie korzystać. Zapoznamy się teraz pokrótce z kolejnymi technikami, a rozpoczniemy od mechanizmu automatycznej konfiguracji Springa.

2.2. Wykorzystujemy automatyczne wiązanie komponentów

W dalszej części tego rozdziału dowiesz się, jak zdefiniować wiązania w Springu za pomocą plików Java i XML. Niejednokrotnie znajdziemy zastosowanie dla tego typu jawnych technik wiązania komponentów. Nic nie przebije jednak prostoty oferowanej przez mechanizm automatycznej konfiguracji Springa. Po co mamy samodzielnie w jawny sposób wiązać komponenty, jeśli Spring może to za nas zrobić automatycznie?

Automatyczne wiązanie komponentów w Springu realizowane jest za pomocą dwóch mechanizmów:

- *Skanowania komponentów* — Spring automatycznie wyszukuje komponenty, które mają zostać utworzone w kontekście aplikacji.
- *Autowiązania* (ang. *Autowiring*) — Spring w sposób automatyczny rozwiązuje zależności komponentów.

Oba te mechanizmy, skanowania komponentów i autowiązania, świetnie ze sobą współdziałają, dając nam potężne możliwości, i pozwalają do minimum ograniczyć potrzebę jawnej konfiguracji.

Działanie tych mechanizmów zademonstruję na przykładzie komponentów wchodzących w skład systemu stereo. Rozpoczniemy od utworzenia interfejsu `CompactDisc` i jego implementacji, którą Spring wyszuka i uczyni komponentem. W kolejnym kroku utworzymy klasę `CDPlayer`, która po wyszukaniu zostanie wstrzyknięta do komponentu `CompactDisc`.

2.2.1. Tworzymy wyszukiwalne komponenty

W erze plików MP3 i streamingu muzyki płyty kompaktowe mogą się wydawać ciekawostką i archaizmem. Oczywiście nie w takim stopniu jak chociażby kasety magnetofonowe, Stereo 8 czy płyty winylowe. Jednak płyty CD są i tak coraz rzadziej spotykane i stanowią ostatni bastion fizycznej dystrybucji muzyki.

Mimo to przykład z płytą CD znakomicie ilustruje sposób działania wstrzykiwania zależności. Odtwarzacze płyt CD nie przedstawiają zbyt dużej wartości użytkowej, jeśli nie włożymy do nich płyty CD. Można by powiedzieć, że działanie odtwarzacza jest zależne od płyty CD.

Aby zilustrować ten przykład w realiach Springa, utwórzmy koncepcję płyty CD w języku Java. Na listingu 2.1. przedstawiono interfejs `CompactDisc`, stanowiący definicję płyty CD.

Listing 2.1. Interfejs CompactDisc nakreśla koncepcję płyty CD w języku Java

```
package soundsystem;
public interface CompactDisc {
   void play();
}
```

Szczegóły dotyczące interfejsu `CompactDisc` nie są istotne. Istotny jest sam fakt, że jest to interfejs. Interfejs definiuje kontrakt, poprzez który odtwarzacz może korzystać z płyty CD. W ten sposób zależności pomiędzy dowolną implementacją odtwarzacza a płytą CD zostają zredukowane do niezbędnego minimum.

Mimo to implementacja interfejsu `CompactDisc` jest nam potrzebna. W praktyce tych implementacji może być dużo więcej. W naszym przykładzie rozpoczniemy od jednej: klasy `SgtPeppers`, widocznej na listingu 2.2.

Listing 2.2. Klasa SgtPeppers opatrzona adnotacją @Component implementuje interfejs CompactDisc

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

@Component
public class SgtPeppers implements CompactDisc {
   private String title = "Sgt. Pepper's Lonely Hearts Club Band";
   private String artist = "The Beatles";

   public void play() {
```

```
        System.out.println("Odtwarzam utwór " + title + " artysty " + artist);
    }
}
```

Podobnie jak w przypadku interfejsu `CompactDisc`, szczegóły implementacji klasy `Sgt➥Peppers` też nie są w naszych rozważaniach istotne. Warto zwrócić uwagę na adnotację `@Component`, którą została opatrzona klasa `SgtPeppers`. Ta prosta adnotacja jest informacją dla Springa, aby utworzyć komponent dla tej klasy. Nie musimy konfigurować komponentu `SgtPeppers` w sposób jawny. Dzięki użyciu adnotacji `@Component` Spring zrobi to za nas automatycznie.

Należy pamiętać o tym, że skanowanie komponentów nie jest domyślnie włączone. Wciąż musimy w sposób jawny utworzyć konfigurację, która włączy mechanizm wyszukiwania klas opatrzonych adnotacją `@Component` i utworzy komponenty dla tych klas. Listing 2.3 zawiera minimalną klasę konfiguracji niezbędną do realizacji tego zadania.

Listing 2.3. Adnotacja @ComponentScan włącza mechanizm skanowania komponentów

```
package soundsystem;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan
public class CDPlayerConfig {
}
```

Klasa `CDPlayerConfig` definiuje springową specyfikację wiązania wyrażoną w języku Java. W sekcji 2.3 przyjrzymy się dokładniej konfiguracji Springa opartej na klasach javowych. W chwili obecnej konfiguracja `CDPlayerConfig` nie definiuje w sposób jawny żadnych komponentów. Opatrzona jest jednak adnotacją `@ComponentScan`, co włącza springowy mechanizm skanowania komponentów.

Jeśli do adnotacji `@ComponentScan` nie przekazano żadnych dodatkowych ustawień konfiguracji, domyślnym pakietem, w którym będą skanowane komponenty, jest pakiet klasy konfiguracji. Ponieważ klasa `CDPlayerConfig` znajduje się w pakiecie `soundsystem`, Spring przeskanuje ten pakiet i wszystkie jego pakiety potomne w poszukiwaniu klas opatrzonych adnotacją `@Component`. Powinien w ten sposób odnaleźć interfejs `CompactDisc` i utworzyć dla niego automatycznie komponent springowy.

Aby włączyć skanowanie komponentów za pomocą konfiguracji XML, dodajemy element `<context:component-scan>` z przestrzeni `context` w Springu. Listing 2.4 zawiera minimalną konfigurację XML pozwalającą wykonać to zadanie.

Listing 2.4. Włączanie skanowania komponentów za pomocą konfiguracji XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
```

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

<context:component-scan base-package="soundsystem" />

</beans>
```

Chociaż konfiguracja XML jest jedną z opcji włączenia mechanizmu skanowania komponentów, w tych rozważaniach skupię się na preferowanej metodzie konfiguracji opartej na klasach Javy. Osoby preferujące pliki XML ucieszy z pewnością fakt, że element `<context:component-scan>` posiada atrybuty i elementy podrzędne odpowiadające atrybutom, które zostaną opisane podczas pracy z adnotacją `@ComponentScan`.

Możesz mi wierzyć lub nie, ale te dwie zdefiniowane klasy w zupełności wystarczą do przetestowania opisywanych funkcjonalności. Działanie mechanizmu skanowania komponentów sprawdzimy za pomocą prostego testu JUnit. Test utworzy kontekst aplikacji Springa i zweryfikuje, czy komponent `CompactDisc` został rzeczywiście stworzony. Klasa `CDPlayerTest` widoczna na listingu 2.5 realizuje właśnie to sprawdzenie.

Listing 2.5. Sprawdzanie, czy komponent CompactDisc został odnaleziony w wyniku skanowania komponentów

```
package soundsystem;

import static org.junit.Assert.*;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=CDPlayerConfig.class)
public class CDPlayerTest {

   @Autowired
   private CompactDisc cd;

   @Test
   public void cdShouldNotBeNull() {
      assertNotNull(cd);
   }
}
```

Klasa `CDPlayerTest` wykorzystuje klasę Springa `SpringJUnit4ClassRunner` do automatycznego utworzenia kontekstu aplikacji po uruchomieniu testu. Adnotacja `@ContextConfi`
`↪guration` informuje Springa, że konfiguracja kontekstu ma zostać wczytana z klasy `CDPlayerConfig`. Klasa konfiguracji opatrzona jest adnotacją `@ComponentScan`, dzięki czemu otrzymany kontekst aplikacji powinien zawierać komponent `CompactDisc`.

Sprawdzamy to poprzez właściwość typu `CompactDisc` opatrzoną adnotacją `@Autowired`, która powoduje wstrzyknięcie komponentu `CompactDisc` do tworzonego testu. (Za chwilę powiem trochę więcej na temat adnotacji `@Autowired`). Na koniec prostym testem spraw-

dzamy, że wartością właściwości cd nie jest null. Pozytywny wynik testu oznacza, że Spring zdołał odnaleźć implementację CompactDisc, utworzył automatycznie jej komponent w kontekście aplikacji Springa i wstrzyknął do testu.

Test powinien zakończyć się sukcesem i zaświecić na zielono. Udało nam się zakończyć pierwsze proste ćwiczenie związane ze skanowaniem komponentów! Choć utworzyliśmy tylko pojedynczy komponent, taki sam nakład pracy byłby potrzebny przy skanowaniu i tworzeniu dowolnej liczby komponentów. Wszystkie klasy oznaczone adnotacją @Component i znajdujące się w pakiecie soundsystem oraz pakietach podrzędnych zostaną utworzone w postaci komponentów. Dodanie jednej linii zawierającej adnotację @ComponentScan w zamian za nieograniczoną liczbę automatycznie tworzonych komponentów to bez wątpienia świetny interes.

Przyjrzyjmy się teraz dokładniej adnotacjom @ComponentScan oraz @Component i zobaczmy, jakie dodatkowe możliwości oferują one w zakresie skanowania komponentów.

2.2.2. Nadajemy nazwy skanowanemu komponentowi

Wszystkim komponentom w springowym kontekście aplikacji nadawane są identyfikatory. Może nie jest to oczywiste, ale chociaż w poprzednim przykładzie nie nadaliśmy komponentowi SgtPeppers żadnego identyfikatora, otrzymał on identyfikator bazujący na nazwie klasy. A konkretnie — otrzymał on identyfikator sgtPeppers, czyli powstały poprzez zamianę w nazwie klasy pierwszej litery na małą.

Żeby nadać komponentowi inny identyfikator, wystarczy ustawić pożądaną nazwę jako wartość adnotacji @Component. Na przykład aby nadać komponentowi nazwę lonely ↪HeartsClub, musimy klasę SgtPeppers opatrzyć adnotacją w następujący sposób:

```
@Component("lonelyHeartsClub")
public class SgtPeppers implements CompactDisc {
...
}
```

Inną metodą nadania nazwy komponentowi jest zamiana adnotacji @Component na adnotację @Named, pochodzącą ze specyfikacji Java Dependency Injection (JSR-330), i wskazanie wybranego identyfikatora:

```
package soundsystem;
import javax.inject.Named;
@Named("lonelyHeartsClub")
public class SgtPeppers implements CompactDisc {
...
}
```

Adnotacja @Named stanowi w Springu alternatywę dla adnotacji @Component. Jest między nimi kilka subtelnych różnic, ale w większości przypadków mogą być one stosowane zamiennie.

Skoro już o tym mowa, to osobiście zdecydowanie wolę adnotację @Component. Głównie dlatego, że adnotacja @Named ma… że tak powiem… źle dobraną nazwę. Inaczej niż w przypadku adnotacji @Component, patrząc na nią, ciężko wywnioskować, do czego może służyć. Z tego powodu w tej książce adnotacja @Named nigdy więcej już się nie pojawi.

2.2.3. Ustawiamy pakiet bazowy dla skanowania komponentów

Do tej pory korzystaliśmy z adnotacji `@ComponentScan` bez żadnych atrybutów. Domyślnym pakietem bazowym dla skanowania komponentów jest w takiej sytuacji pakiet klasy konfiguracyjnej. Co zatem możemy zrobić, aby skanować inny pakiet? Albo skanować kilka pakietów bazowych?

Jednym z przypadków, gdy konieczne jest jawne ustawienie pakietu bazowego, jest przechowywanie całego kodu konfiguracyjnego w osobnym pakiecie, oddzielonego od reszty aplikacji. Domyślne ustawienie pakietu bazowego nie zadziała wtedy poprawnie.

Nie jest to jednak żaden problem. Żeby wybrać inny pakiet bazowy, ustawiamy jego nazwę jako atrybut `value` adnotacji `@ComponentScan`:

```
@Configuration
@ComponentScan("soundsystem")
public class CDPlayerConfig {}
```

Możemy podkreślić, że ustawiana wartość to nazwa pakietu bazowego, i w miejsce atrybutu `value` wykorzystać atrybut `basePackages`:

```
@Configuration
@ComponentScan(basePackages="soundsystem")
public class CDPlayerConfig {}
```

Nazwa atrybutu `basePackages` jest w liczbie mnogiej. Jeśli nasuwa Ci się więc pytanie, czy można wybrać większą liczbę pakietów bazowych, odpowiadam twierdząco. Wystarczy, że ustawimy jako wartość atrybutu `basePackages` tablicę pakietów, które chcemy przeszukiwać.

```
@Configuration
@ComponentScan(basePackages={"soundsystem", "video"})
public class CDPlayerConfig {}
```

Warto zauważyć, że w dotychczasowych przykładach nazwy pakietów bazowych podane były w postaci ciągu znaków. Nie ma w tym nic złego, poza faktem, że praktyka ta nie jest bezpieczna ze względu na typy. Zmiana nazw pakietów może doprowadzić do uszkodzenia ustawień pakietów bazowych.

Alternatywą dla nazw pakietów w postaci wartości typu `String` jest wykorzystanie innego atrybutu adnotacji `@ComponentScan`, który pozwala na ustawienie pakietów bazowych w oparciu o znajdujące się w nim klasy lub interfejsy:

```
@Configuration
@ComponentScan(basePackageClasses={CDPlayer.class, DVDPlayer.class})
public class CDPlayerConfig {}
```

Zwróć uwagę, że atrybut `basePackages` został zastąpiony przez atrybut `basePackageClasses`, a identyfikatory pakietów w postaci nazw złożonych z ciągów znaków zastąpiła tablica klas. Pakiety zawierające te klasy będą pakietami bazowymi skanowania komponentów.

W ostatnim przykładzie przy ustawianiu pakietu bazowego za pomocą atrybutu `basePackageClass` wykorzystaliśmy klasy komponentów. Moglibyśmy również utworzyć w wybranym pakiecie znacznik w postaci pustego interfejsu. Dzięki zastosowaniu takiego znacznika otrzymujemy bezpieczną konfigurację, odporną na zmianę struktury pakie-

tów, niepowiązaną równocześnie z kodem właściwej aplikacji (który to kod mógłby zostać przykładowo przeniesiony z pakietu przeznaczonego do skanowania).

Komponent `SgtPeppers` jest samowystarczalny i nie ma żadnych zależności. Gdyby wszystkie obiekty dostępne w aplikacji były takie jak on, sam mechanizm skanowania zapewniłby nam już poprawne funkcjonowanie mechanizmu powiązań. Działanie wielu obiektów uzależnione jest jednak od istnienia innych obiektów. Potrzebujemy więc sposobu wiązania skanowanych komponentów posiadających zależności. W tym celu przyjrzyjmy się autowiązaniu, drugiemu mechanizmowi automatycznej konfiguracji Springa.

2.2.4. Oznaczamy adnotacją komponenty przeznaczone do autowiązania

Mówiąc w skrócie, autowiązaniem nazywamy mechanizm, za pomocą którego Spring w sposób automatyczny rozwiązuje zależności komponentu z użyciem komponentów wyszukanych w kontekście aplikacji. Do oznaczenia obiektów poddanych autowiązaniu możemy wykorzystać adnotację `@Autowired`.

Jako przykład niech posłuży klasa `CDPlayer` widoczna na listingu 2.6. Konstruktor klasy jest opatrzony adnotacją `@Autowired`, dzięki czemu tworzenie komponentu `CDPlayer` następuje z użyciem tego konstruktora poprzez wstrzyknięcie zależności `CompactDisc`.

Listing 2.6. Wstrzykiwanie zależności CompactDisc do komponentu CDPlayer za pośrednictwem mechanizmu autowiązania

```
package soundsystem;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class CDPlayer implements MediaPlayer {
   private CompactDisc cd;

   @Autowired
   public CDPlayer(CompactDisc cd) {
      this.cd = cd;
   }

   public void play() {
      cd.play();
   }
}
```

Funkcjonalność adnotacji `@Autowired` nie ogranicza się do konstruktora. Można ją również wykorzystać do oznaczania metod typu setter. Zakładając, że w klasie `CDPlayer` istnieje metoda `setCompactDisc()`, możemy ją opatrzyć adnotacją:

```
@Autowired
public void setCompactDisc(CompactDisc cd) {
   this.cd = cd;
}
```

Po utworzeniu komponentu Spring spróbuje spełnić jego zależności, wyrażone za pomocą metod oznaczonych adnotacją `@Autowired`, właśnie takich jak `setCompactDisc`.

W praktyce metody typu setter nie wyróżnia nic szczególnego. Adnotacją `@Autowired` można opatrzyć dowolną metodę zdefiniowaną w klasie. Gdyby w klasie `CDPlayer` istniała metoda `insertDisc()`, adnotacja `@Autowired` działałaby na niej równie dobrze jak na metodzie `setCompactDisc()`:

```
@Autowired
public void insertDisc(CompactDisc cd) {
   this.cd = cd;
}
```

Niezależnie od tego, czy skorzystamy z konstruktora, metody typu setter, czy dowolnej innej metody, Spring spróbuje spełnić zależność wyrażoną za pomocą parametrów tej metody. Jeśli zależność ta spełniona jest przez tylko jeden komponent, to właśnie on zostanie wykorzystany.

Jeżeli żaden komponent nie spełnia zależności, Spring rzuci wyjątkiem przy tworzeniu kontekstu aplikacji. Możemy uniknąć powstania tego wyjątku, ustawiając w adnotacji `@Autowired` wartość atrybutu `required` na `false`:

```
@Autowired(required=false)
public CDPlayer(CompactDisc cd) {
   this.cd = cd;
}
```

Gdy wartością atrybutu `required` jest `false`, Spring podejmuje próbę autowiązania, lecz gdy wyszukanie zależności się nie powiedzie, nie rzuca wyjątkiem, ale komponent pozostaje niepowiązany. W takiej sytuacji należy jednak zachować należytą ostrożność. Niepowiązana właściwość przy braku sprawdzenia wartości `null` może doprowadzić do wystąpienia wyjątku `NullPointerException`.

W sytuacji, gdy zależność spełnia wiele komponentów, Spring rzuci wyjątkiem wskazującym na niejednoznaczność w wyborze komponentu autowiązania. W rozdziale 3. wyjaśnię, jak radzić sobie z niejednoznacznością w autowiązaniu.

`@Autowired` jest adnotacją dostępną tylko w Springu. Jeśli nie chcesz uzależniać kodu od springowych adnotacji, możesz ją zastąpić adnotacją `@Inject`:

```
package soundsystem;

import javax.inject.Inject;
import javax.inject.Named;

@Named
public class CDPlayer {
   ...

   @Inject
   public CDPlayer(CompactDisc cd) {
      this.cd = cd;
   }
   ...
}
```

Adnotacja `@Inject` wywodzi się ze specyfikacji Java Dependency Injection, tej samej, z której pochodzi adnotacja `@Named`. Spring umożliwia wykorzystanie zarówno adnotacji `@Inject`, jak i swojej własnej adnotacji `@Autowired`. Chociaż występują pomiędzy niby subtelne różnice, w większości sytuacji obie adnotacje można stosować zamiennie.

2.2.5. Weryfikujemy automatyczną konfigurację

Po opatrzeniu konstruktora klasy `CDPlayer` adnotacją `@Autowired` wiemy, że Spring wstrzyknie do niego automatycznie komponent `CompactDisc`. Upewnijmy się, że tak się naprawdę stanie. Zmieńmy test `CDPlayerTest` tak, aby odtwarzał płyty kompaktowe za pomocą komponentu `CDPlayer`:

```
package soundsystem;
import static org.junit.Assert.*;
import org.junit.Rule;
import org.junit.Test;
import org.junit.contrib.java.lang.system.StandardOutputStreamLog;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=CDPlayerConfig.class)
public class CDPlayerTest {

   @Rule
   public final StandardOutputStreamLog log =
      new StandardOutputStreamLog();

   @Autowired
   private MediaPlayer player;

   @Autowired
   private CompactDisc cd;

   @Test
   public void cdShouldNotBeNull() {
      assertNotNull(cd);
   }

   @Test
   public void play() {
      player.play();
      assertEquals(
         "Odtwarzam Sgt. Pepper's Lonely Hearts Club Band" +
         " autorstwa The Beatles\n",
         log.getLog());
   }
}
```

W powyższym przykładzie wstrzykujemy komponent `CompactDisc` do właściwości `cd`, a do właściwości `player` wstrzykujemy komponent `CDPlayer` (pod postacią bardziej ogólnego typu `MediaPlayer`). W metodzie testowej `play()` wywołujemy metodę `play()` obiektu `player` i sprawdzamy, czy uzyskaliśmy oczekiwany rezultat.

Testowanie kodu zawierającego wywołania metody `System.out.println()` nie jest proste. Z tego powodu w przykładzie korzystamy z klasy `StandardOutputStreamLog`, reguły dla JUnita pochodzącej z biblioteki System Rules (*http://stefanbirkner.github.io/system-rules/index.html*). Umożliwia ona wykonywanie asercji weryfikujących dane wysłane na konsolę. W naszym przykładzie sprawdzamy, czy wysłany został komunikat pochodzący z metody `SgtPeppers.play()`.

Poznałeś właśnie podstawy pracy z mechanizmem skanowania komponentów oraz mechanizmem autowiązań. Do tematu skanowania komponentów powrócimy ponownie w rozdziale 3., przy okazji rozwiązywania problemu z niejednoznacznością autowiązania.

W tym momencie odłóżmy jednak na bok skanowanie i autowiązanie komponentów i zobaczmy, jak w sposób jawny wiązać komponenty w Springu. Rozpoczniemy od prostej konfiguracji wykorzystującej kod w Javie.

2.3. Wiążemy kod za pomocą Javy

W większości przypadków rozwiązaniem preferowanym w Springu jest automatyczna konfiguracja, korzystająca ze skanowania i automatycznego wiązania komponentów. Jednak w niektórych sytuacjach jedynym wyjściem jest użycie jawnej konfiguracji. Przykładowo chcemy powiązać z naszą aplikacją komponenty pochodzące z zewnętrznej biblioteki. Nie posiadamy przy tym źródeł tej biblioteki, nie mamy zatem możliwości oznaczenia jej klas adnotacjami `@Component` i `@Autowired`. Zastosowanie automatycznej konfiguracji nie wchodzi więc w rachubę.

W takim wypadku musimy użyć jawnej konfiguracji. Istnieją dwie możliwości jawnej konfiguracji: za pomocą klas Javy (tzw. konfiguracja *JavaConfig*) oraz plików XML. W tej sekcji przyjrzymy się konfiguracji JavaConfig, a w następnej przejdziemy do konfiguracji opartej na plikach XML.

Jak wspominałem wcześniej, konfiguracja JavaConfig oferuje większe możliwości, jest bezpieczniejsza ze względu na typy i umożliwia wygodne refaktorowanie kodu. Wszystkie te dogodności zawdzięczamy temu, że jest to zwykły kod Javy, taki jak w innych miejscach naszej aplikacji.

Nie powinniśmy jednak traktować kodu JavaConfig tak jak kodu znajdującego się w pozostałych miejscach projektu. Kod konfiguracji jest koncepcyjnie oderwany od logiki biznesowej i kodu domeny aplikacji. Mimo że w obu przypadkach korzystamy z tego samego języka, kod JavaConfig służy do opisywania konfiguracji. Kod konfiguracji nie powinien zawierać żadnej logiki biznesowej ani nie powinien się wkradać w logikę biznesową innych plików. W związku z powyższym jest on często umieszczany w osobnym pakiecie, by nie pozostawiać żadnych wątpliwości co do jego przeznaczenia.

Zobaczmy więc, jak w jawny sposób skonfigurować ustawienia Springa z użyciem klas JavaConfig.

2.3.1. Tworzymy klasy konfiguracji

Na listingu 2.3 spotkaliśmy się po raz pierwszy z konfiguracją JavaConfig. Przypomnijmy sobie wygląd klasy `CDPlayerConfig` z tamtego przykładu:

```
package soundsystem;

import org.springframework.context.annotation.Configuration;

@Configuration
public class CDPlayerConfig {
}
```

Kluczowym elementem tworzenia klasy konfiguracji w języku Java jest opatrzenie jej adnotacją `@Configuration`. Adnotacja ta jest informacją, że opatrzona nią klasa jest konfiguracją zawierającą informacje na temat komponentów tworzonych w kontekście aplikacji Springa.

Do tej pory do wyszukiwania komponentów, które mają zostać utworzone przez Springa, używaliśmy mechanizmu skanowania komponentów. Nie istnieją żadne przeciwwskazania do korzystania równocześnie i ze skanowania komponentów, i z jawnej konfiguracji. W tej sekcji chcemy się jednak skupić na jawnej konfiguracji, usuńmy więc adnotację `@ComponentScan` z klasy `CDPlayerConfig`.

Usunięcie adnotacji sprawia, że klasa `CDPlayerConfig` przestaje działać. Uruchomienie w tym momencie testu `CDPlayerTest` zakończy się niepowodzeniem i wyrzuceniem wyjątku `BeanCreationException`. Pozytywny wynik testu uzależniony jest od udanego wstrzyknięcia komponentów `CDPlayer` i `CompactDisc`, a z powodu wyłączenia mechanizmu skanowania komponenty te nie mogą zostać utworzone.

Moglibyśmy naprawić nasz test, przywracając adnotację `@ComponentScan`. My chcemy się jednak skupić na jawnej konfiguracji i dowiedzieć się, jak powiązać komponenty `CDPlayer` i `CompactDisc` z użyciem konfiguracji JavaConfiga.

2.3.2. Deklarujemy prosty komponent

Aby zadeklarować komponent w konfiguracji JavaConfig, przygotowujemy metodę, która tworzy instancję pożądanego typu, i opatrujemy ją metodą `@Bean`. Przykładowo poniższy listing przedstawia deklarację komponentu `CompactDisc`:

```
@Bean
public CompactDisc sgtPeppers() {
   return new SgtPeppers();
}
```

Dzięki zastosowaniu adnotacji `@Bean` Spring wie, że obiekt zwracany przez metodę ma zostać zarejestrowany jako komponent w kontekście aplikacji. Ciało metody zawiera logikę, która prowadzi do utworzenia instancji tego komponentu.

Komponent otrzymuje domyślny identyfikator, którym jest nazwa metody opatrzonej adnotacją, więc w naszym przypadku jest to `sgtPeppers`. Jeśli uznamy, że nazwa nie jest odpowiednia, i zechcemy ją zmienić, zmieniamy nazwę metody lub wykorzystujemy atrybut `name` do nadania innego identyfikatora:

```
@Bean(name="lonelyHeartsClubBand")
public CompactDisc sgtPeppers() {
return new SgtPeppers();
}
```

Niezależnie od wybranego sposobu nazwania komponentu deklaracja jest zawsze bardzo prosta. Ciało metody zwraca nową instancję klasy `SgtPeppers`. Korzystamy z języka Java, więc tworząc nowy obiekt, mamy możliwość zastosowania wszystkich funkcji tego języka.

Puszczając wodze fantazji, możemy tam robić rzeczy tak szalone jak na przykład zwracanie losowo płyty spośród dostępnych możliwości:

```
@Bean
public CompactDisc randomBeatlesCD() {
   int choice = (int) Math.floor(Math.random() * 4);
   if (choice == 0) {
      return new SgtPeppers();
   } else if (choice == 1) {
      return new WhiteAlbum();
   } else if (choice == 2) {
      return new HardDaysNight();
   } else {
      return new Revolver();
   }
}
```

Pomyśl przez chwilę na temat różnych sposobów wykorzystania potęgi języka Java przy tworzeniu komponentów za pomocą metod opatrzonych adnotacją `@Bean`. Jak już skończysz, powrócimy do tematu i zobaczymy, jak wstrzyknąć komponent `CompactDisc` do klasy `CDPlayer`, korzystając z konfiguracji JavaConfig.

2.3.3. Wstrzykujemy zależności za pomocą konfiguracji JavaConfig

Zadeklarowany komponent `CompactDisc` był prosty i nie posiadał żadnych zewnętrznych zależności. Nadszedł czas, aby zadeklarować komponent `CDPlayer`, który jest uzależniony od klasy `CompactDisc`. Jak stworzyć właściwe wiązanie z użyciem konfiguracji JavaConfig?

Najprostszym sposobem wiązania komponentów jest wykorzystanie odpowiedniej metody komponentu. Poniżej widać przykład deklaracji komponentu `CDPlayer`:

```
@Bean
public CDPlayer cdPlayer() {
   return new CDPlayer(sgtPeppers());
}
```

Metoda `cdPlayer()`, podobnie jak metoda `sgtPeppers()`, opatrzona jest adnotacją `@Bean`. Oznacza to, że zwracana przez nią instancja klasy ma zostać zarejestrowana jako komponent w kontekście aplikacji Springa. Identyfikatorem komponentu będzie `cdPlayer`, czyli nazwa metody oznaczonej adnotacją.

Ciało metody `cdPlayer()` różni się trochę od metody `sgtPeppers()`. Instancja klasy `CDPlayer` nie jest tworzona z użyciem domyślnej metody, ale za pomocą konstruktora, który przyjmuje jako parametr obiekt typu `CompactDisc`.

Mogłoby się wydawać, że pozyskiwanie komponentu `CompactDisc` następuje poprzez wywołanie metody `sgtPeppers`, jednak nie jest to do końca prawdą. Metoda `sgtPeppers()` opatrzona jest adnotacją `@Bean`, co sprawia, że Spring przechwytuje wszystkie jej wywołania. Nie jest ona wywoływana za każdym razem, ale zwracany jest zarejestrowany komponent.

Przykładowo załóżmy, że rejestrujemy kolejny komponent CDPlayer, podobny do komponentu zarejestrowanego przez nas wcześniej:

```
@Bean
public CDPlayer cdPlayer() {
return new CDPlayer(sgtPeppers());
}
@Bean
public CDPlayer anotherCDPlayer() {
   return new CDPlayer(sgtPeppers());
}
```

Jeśli wywołanie metody sgtPeppers() przebiegałoby w sposób standardowy, każda klasa CDPlayer otrzymałaby własną instancję SgtPeppers. W odniesieniu do prawdziwych odtwarzaczy CD i płyt kompaktowych miałoby to jak najbardziej sens. Jeżeli posiadamy dwa odtwarzacze CD, nie ma fizycznej możliwości, by pojedynczą płytę kompaktową umieścić w obu odtwarzaczach jednocześnie.

W przypadku oprogramowania sytuacja jest inna. Nie ma powodu, dla którego nie możemy użyć tej samej instancji SgtPeppers w dowolnej liczbie komponentów. Wszystkie komponenty w Springu są domyślnie singletonami i nie musimy duplikować instancji, żeby móc spełnić zależności drugiego komponentu CDPlayer. Spring przechwytuje wywołanie metody sgtPeppers() i upewnia się, że zwrócony zostanie komponent powstały w wyniku wywołania tej metody przez Springa w trakcie tworzenia komponentu Com
↪pactDisc. W rezultacie oba komponenty typu CDPlayer otrzymają tę samą instancję SgtPeppers.

Zdaję sobie sprawę, że odwoływanie się do komponentu za pomocą nazwy metody może być mylące. Istnieje inny, bardziej przyjazny sposób:

```
@Bean
public CDPlayer cdPlayer(CompactDisc compactDisc) {
   return new CDPlayer(compactDisc);
}
```

W powyższym przykładzie metoda cdPlayers() przyjmuje CompactDisc jako parametr. Spring wywołuje metodę cdPlayer(), aby utworzyć komponent CDPlayer, i automatycznie wiąże instancję CompactDisc z metodą konfiguracji. Instancja ta może być potem w dowolny sposób wykorzystywana w metodzie. Dzięki tej technice metoda cdPlayer() może wstrzyknąć komponent CompactDisc do konstruktora CDPlayer, nie odwołując się bezpośrednio do nazwy metody zwracającej ten komponent, opatrzonej adnotacją @Bean.

Ten sposób odwoływania się do innych komponentów jest z reguły najlepszym wyborem, bo nie wymusza deklaracji komponentu CompactDisc w tej samej klasie konfiguracji. Co więcej, nie musi on być zadeklarowany z użyciem konfiguracji w Javie, ale można do tego zastosować adnotację lub plik XML. Mamy zatem możliwość eleganckiego rozbicia konfiguracji na klasy Javy oraz pliki XML, a równocześnie możliwość automatycznego skanowania i wiązania komponentów. Niezależnie od sposobu utworzenia komponentu CompactDisc, Spring z łatwością przekaże go do metody konfiguracji i utworzy komponent CDPlayer.

W każdym przypadku należy pamiętać, że chociaż korzystaliśmy z wstrzykiwania zależności przez konstruktor klasy `CDPlayer`, nie ma żadnych przeciwskazań, dla których nie moglibyśmy zastosować innej techniki DI. Możemy na przykład wstrzyknąć komponent `CompactDisc` za pomocą metody typu setter:

```
@Bean
public CDPlayer cdPlayer(CompactDisc compactDisc) {
   CDPlayer cdPlayer = new CDPlayer(compactDisc);
   cdPlayer.setCompactDisc(compactDisc);
   return cdPlayer;
}
```

Tak jak wspominałem, ciało metody opatrzonej adnotacją `@Bean` może w dowolny sposób wykorzystać wstrzykniętą zależność do utworzenia instancji komponentu. Wstrzykiwanie przez konstruktor i metodę typu setter to tylko dwa proste przykłady możliwości zastosowania metod opatrzonych adnotacją `@Bean`. Jedynym ograniczeniem są ograniczenia samej Javy.

2.4. Wiążemy komponenty za pomocą plików XML

Do tej pory korzystaliśmy jedynie z automatycznego skanowania i wiązania komponentów oraz jawnego tworzenia wiązań z użyciem klas Javy. Istnieje jednak inny sposób, mniej zalecany, ale mający długą historię związaną ze Springiem.

Od pierwszych wersji Springa podstawową metodą konfiguracji było użycie plików XML. Powstało w ten sposób nieprzebrane wprost bogactwo linii kodu. Wielu osobom Spring kojarzy się właśnie z plikami XML zawierającymi konfigurację.

Prawdą jest, że Spring był przez długi okres utożsamiany z plikami XML, miejmy jednak świadomość, że XML nie jest jedyną dostępną opcją konfiguracji Springa. Obecnie, gdy Spring otrzymał wsparcie dla konfiguracji automatycznej oraz konfiguracji JavaConfig, XML nie powinien być naszym pierwszym wyborem.

Mimo wszystko w oparciu o pliki XML powstało już wiele konfiguracji Springa. Ważne jest więc zrozumienie zasad ich działania. Mam nadzieję, że sekcja ta posłuży Ci jedynie do pracy z istniejącą konfiguracją, a przy tworzeniu nowej wykorzystasz inne dostępne opcje.

2.4.1. Tworzymy specyfikację konfiguracji XML

Zanim przystąpimy do wiązania komponentów Springa za pomocą plików XML, potrzebujemy utworzyć pustą specyfikację konfiguracji. W przypadku konfiguracji JavaConfig oznacza to wykorzystanie adnotacji `@Configuration`. Odpowiednikiem w konfiguracji XML jest utworzenie dokumentu z głównym elementem `<beans>`.

Najprostsza możliwa konfiguracja XML w przypadku Springa wygląda następująco:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
      http://www.springframework.org/schema/context">
   <!-- szczegóły konfiguracji znajdą się tutaj -->
</beans>
```

Widać od razu, że ta podstawowa konfiguracja jest już w tym momencie dużo bardziej złożona niż jej odpowiednik w konfiguracji JavaConfig. W klasie Javy wystarczyło dodanie adnotacji `@Configuration`, w pliku XML elementy konfiguracji Springa zdefiniowane są w kilku plikach schematów XML (XSD) i zadeklarowane w preambule pliku konfiguracji XML.

> **TWORZENIE KONFIGURACJI XML ZA POMOCĄ ŚRODOWISKA SPRING TOOL SUITE** Prostym sposobem tworzenia plików konfiguracji XML i zarządzania nimi jest wykorzystanie środowiska Spring Tool Suite (*https://spring.io/tools/sts*). Wystarczy wybrać w menu kolejno *File* (Plik)/*New* (Nowy)/*Spring Bean Configuration File* (Plik konfiguracji komponentów Springa), utworzyć plik XML konfiguracji i wybrać jedną z dostępnych przestrzeni nazw konfiguracji.

Podstawowe elementy XML służące do wiązania komponentów dostępne są w schemacie `spring-beans`, który zadeklarowaliśmy w tym pliku jako główną przestrzeń nazw. Element `<beans>`, główny element każdego pliku konfiguracji Springa, należy do tego schematu.

Dostępnych jest kilka innych schematów konfiguracji Springa z użyciem plików XML. W tej książce koncentruję się na automatycznej konfiguracji oraz konfiguracji za pomocą klas Javy. Postaram się jednak zasygnalizować możliwość wykorzystania innych schematów XML, gdy się taka pojawi.

Mamy już w tej chwili w pełni poprawną konfigurację Springa z użyciem XML. Jest ona jednocześnie bezużyteczna, gdyż nie zadeklarowaliśmy w niej żadnych komponentów. Tchnijmy w ten plik nieco życia, odtwarzając przykład z płytami CD, ale tym razem skorzystamy z konfiguracji XML.

2.4.2. Deklarujemy prosty komponent

Do deklaracji komponentu Springa w pliku XML wykorzystamy element `<bean>`, także pochodzący ze schematu `spring-beans`. Element `<bean>` jest odpowiednikiem adnotacji `@Bean` w konfiguracji JavaConfig. Możemy go użyć do deklaracji komponentu `CompactDisc`:

```
<bean class="soundsystem.SgtPeppers" />
```

Zadeklarowany komponent jest bardzo prosty. Klasę zastosowaną do deklaracji komponentu wskazaliśmy za pomocą atrybutu `class` i wyraziliśmy z wykorzystaniem w pełni kwalifikowalnej nazwy klasy.

Nie przypisaliśmy mu w sposób jawny żadnego identyfikatora, otrzymał więc nazwę zgodną z w pełni kwalifikowalną nazwą klasy. W naszym przypadku identyfikatorem tym jest `soundsystem.SgtPeppers#0`. `#0` jest numeracją służącą do rozróżnienia komponentów tego samego typu. Jeśli zadeklarowalibyśmy kolejny komponent typu `Sgt` ↪`Peppers` bez jawnego wskazania jego identyfikatora, otrzymałby on automatycznie identyfikator `soundsystem.SgtPeppers#1`.

Automatyczne nadawanie nazwy komponentom może się wydawać wygodne do momentu, gdy zajdzie potrzeba jawnego odwołania do wygenerowanych nazw. Dlatego dobrze jest zawsze samemu nadawać identyfikator (atrybut `id`) każdemu konfigurowanemu komponentowi:

```
<bean id="compactDisc" class="soundsystem.SgtPeppers" />
```

Z nadanej nazwy skorzystamy przy wiązaniu komponentu z komponentem `CDPlayer`.

> **ZMNIEJSZANIE OBJĘTOŚCI** Aby zmniejszyć rozmiar pliku XML, nadawaj nazwy komponentom tylko wtedy, gdy konieczne będzie odwołanie się do nich za pomocą nazwy (przykładowo kiedy musimy wstrzyknąć jego referencję do innego komponentu).

Zanim przejdziemy dalej, przyjrzyjmy się przez chwilę podstawowym cechom tej prostej deklaracji komponentu.

Pierwszą rzeczą, którą możemy zauważyć, jest fakt, że nie jesteśmy bezpośrednio odpowiedzialni za tworzenie instancji `SgtPeppers`, jak to miało miejsce w przypadku konfiguracji JavaConfig. Gdy Spring napotka element `<bean>`, tworzy komponent SgtPeppers, korzystając z jego domyślnego konstruktora. Tworzenie komponentów za pomocą konfiguracji XML jest dużo bardziej pasywne. Jednocześnie ma też dużo mniejsze możliwości niż konfiguracja korzystająca z klas Javy, która daje nam niemal nieograniczone możliwości tworzenia instancji komponentów.

Zwróćmy uwagę, że typ komponentu w elemencie `<bean>` wyrażany jest za pomocą atrybutu `class`, którego wartością jest ciąg znaków. Skąd mamy pewność, że wartość tego atrybutu w rzeczywistości odnosi się do prawdziwej klasy? Konfiguracja XML nie daje nam gwarancji ustawienia poprawnej wartości tego atrybutu w postaci błędów kompilacji. Nawet jeśli założymy, że wartość atrybutu `class` jest w tej chwili poprawna, to co się wydarzy, gdy zmienimy nazwę klasy?

> **SPRAWDZANIE POPRAWNOŚCI PLIKU XML ZA POMOCĄ ŚRODOWISKA IDE** Wykorzystanie środowiska IDE przystosowanego do pracy ze Springiem, jak na przykład Spring Tool Suite, w dużym stopniu ułatwia walidację konfiguracji Springa w plikach XML.

Wymieniłem zaledwie kilka powodów, dla których korzystanie z konfiguracji JavaConfig jest bardziej wskazane niż stosowanie konfiguracji XML. Zawsze pamiętaj o tych ograniczeniach, gdy nadejdzie pora na wybór typu konfiguracji. Kontynuujmy jednak proces poznawania konfiguracji XML w Springu, aby dowiedzieć się, w jaki sposób wstrzyknąć komponent `SgtPeppers` do `CDPlayer`.

2.4.3. Wstrzykujemy komponent przez konstruktor

Konfiguracja XML w Springu dopuszcza tylko jedną metodę deklaracji komponentu: użycie elementu `<bean>` i określenie atrybutu `class`. To wystarczy, aby Spring mógł rozpocząć pracę.

Istnieje kilka dostępnych opcji i stylów deklaracji wstrzykiwania zależności w plikach XML. Przy wstrzykiwaniu przez konstruktor mamy dwie możliwości:

- element `<constructor-arg>`;
- przestrzeń `c` wprowadzona w Springu 3.0.

Główną różnicą pomiędzy nimi jest rozmiar wytworzonego kodu. Za chwilę przekonasz się, że kod wykorzystujący element `<constructor-arg>` jest z reguły bardziej rozwlekły i trudniejszy do odczytania. Z drugiej strony oferuje więcej możliwości, niedostępnych w przypadku przestrzeni nazw `c`.

Porównamy obie konstrukcje na przykładzie wstrzykiwania zależności przez konstruktor w Springu. Na początek spójrzmy na wstrzykiwanie referencji komponentów.

WSTRZYKUJEMY REFERENCJE DO KOMPONENTÓW PRZEZ KONSTRUKTOR

Konstruktor komponentu `CDPlayer` przyjmuje implementację interfejsu `CompactDisc`. Jest to odpowiedni kandydat do wstrzykiwania zależności przez konstruktor.

Klasa zadeklarowanego wcześniej komponentu `SgtPeppers` implementuje interfejs `CompactDisc`. Referencja tego komponentu może więc zostać wstrzyknięta do klasy `CDPlayer`. W tym celu musimy zadeklarować komponent `CDPlayer` w pliku XML i wykorzystać referencję do komponentu `SgtPeppers` z użyciem jego identyfikatora:

```
<bean id="cdPlayer" class="soundsystem.CDPlayer">
   <constructor-arg ref="compactDisc" />
</bean>
```

Gdy Spring napotka element `<bean>`, utworzy instancję klasy `CDPlayer`. Element `<constructor-arg>` informuje, że do konstruktora tworzonego komponentu należy przekazać referencję komponentu o identyfikatorze `compactDisc`.

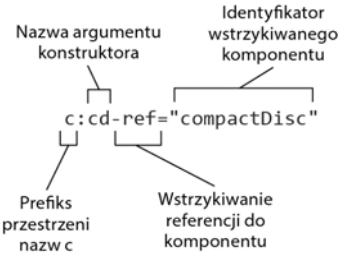
Możemy też użyć przestrzeni nazw `c` udostępnianej przez Springa. Przestrzeń ta została udostępniona w Springu 3.0, aby umożliwić bardziej zwięzłą formę deklaracji argumentów konstruktora w plikach XML. Żeby z niej skorzystać, zadeklarujmy jej schemat w preambule pliku XML:

```
 <?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:c="http://www.springframework.org/schema/c"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   ...
</beans>
```

Teraz gdy przestrzeń nazw i schemat zostały zadeklarowane, możemy je wykorzystać do deklaracji argumentów konstruktora:

```
<bean id="cdPlayer" class="soundsystem.CDPlayer"
   c:cd-ref="compactDisc" />
```

Zadeklarowaną przestrzeń nazw zastosowaliśmy w atrybucie elementu `<bean>`. Wygląda on dość dziwnie. Rysunek 2.1 ilustruje poszczególne składniki atrybutu.



Rysunek 2.1. Wstrzykiwanie referencji komponentu w postaci argumentu konstruktora z wykorzystaniem przestrzeni nazw c Springa

Nazwa atrybutu rozpoczyna się od `c:`, prefiksu przestrzeni nazw. Za prefiksem znajduje się nazwa argumentu konstruktora, który ma zostać powiązany. Nazwę atrybutu kończy `-ref`, co jest konwencją nazewniczą i informacją, że dowiązujemy komponent o nazwie `compactDisc`, a nie tekst `"compactDisc"`.

Od razu można zauważyć, że w porównaniu do elementu `<constructor-arg>` wykorzystanie atrybutów przestrzeni nazw `c` pozwala znacząco skrócić zapis. Jest to jeden z powodów, dla których bardzo lubię ten zapis. Nie dość, że jest trochę bardziej czytelny, to jeszcze ułatwia zmieszczenie kodu przykładów na kartach tej książki.

Z wykorzystaniem przestrzeni nazw `c` wiąże się jednak pewien nurtujący problem. W poprzednim przykładzie widzieliśmy, że odwołuje się on bezpośrednio do nazwy argumentu konstruktora. Odwołanie do nazwy parametru jest moim zdaniem niezbyt mądrym pomysłem, gdyż wymaga kompilacji kodu z symbolami debugowania zapisanymi w kodzie klasy. Jeśli zdecydujemy się zoptymalizować budowanie i pominąć te symbole, kod ten przestanie najprawdopodobniej działać.

Alternatywą może być odwołanie się do pozycji na liście parametrów:

```
<bean id="cdPlayer" class="soundsystem.CDPlayer"
   c:_0-ref="compactDisc" />
```

Widoczny w przykładzie atrybut przestrzeni nazw `c` wygląda jeszcze dziwniej niż poprzednio. Zamieniliśmy nazwę parametru na jego indeks, czyli `0`. Ponieważ XML nie pozwala na użycie cyfry jako pierwszego znaku atrybutu, musieliśmy zastosować prefiks w postaci znaku podkreślenia.

Wykorzystanie indeksu do identyfikacji argumentu konstruktora wydaje się lepszym rozwiązaniem niż korzystanie z jego nazwy. Nawet jeśli usuniemy symbole debugowania w procesie budowania, parametry zachowają swoją kolejność. W przypadku konstruktora wieloargumentowego może to być przydatne. Jednak w naszym przykładzie konstruktora jednoargumentowego mamy jeszcze jedną dodatkową opcję — nie musimy wcale identyfikować parametrów:

```
<bean id="cdPlayer" class="soundsystem.CDPlayer"
   c:_-ref="compactDisc" />
```

Jest to bez wątpienia najbardziej osobliwy atrybut przestrzeni nazw `c`. Nie określamy w nim ani indeksu, ani nazwy parametru. Podaliśmy tylko znak podkreślenia i sufiks `-ref`, aby wskazać, że wykonujemy wiązanie referencji.

Teraz, kiedy wiemy już, jak powiązać referencje do innych komponentów, sprawdźmy, jak wykonać wiązanie wartości tekstowych z użyciem konstruktora.

WSTRZYKUJEMY LITERAŁY PRZEZ KONSTRUKTOR

O wstrzykiwaniu zależności myślimy z reguły jako o rodzaju wiązania omawianym przeze mnie do tej pory — wstrzykiwaniu referencji do obiektów od nich zależnych. Czasem jednak potrzebujemy tylko skonfigurować obiekt za pomocą literału. W tym celu utworzymy nową implementację interfejsu `CompactDisc`:

```
package soundsystem;

public class BlankDisc implements CompactDisc {

   private String title;
   private String artist;

   public BlankDisc(String title, String artist) {
      this.title = title;
      this.artist = artist;
}

   public void play() {
      System.out.println("Odtwarzam utwór " + title + " autorstwa " + artist);
   }
}
```

W przeciwieństwie do klasy `SgtPepper`, w której zapisano „na sztywno" tytuł i artystę, ta implementacja interfejsu `CompactDisc` jest dużo bardziej elastyczna. Tak jak w przypadku pustej płyty CD, tak i w tym przypadku możemy nagrać na płytę utwory dowolnego artysty. Zmieńmy też istniejącą konfigurację komponentu `SgtPeppers`, by wykorzystywał tę właśnie klasę:

```
<bean id="compactDisc"
   class="soundsystem.BlankDisc">
   <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
   <constructor-arg value="The Beatles" />
</bean>
```

Ponownie używamy tutaj elementu `<constructor-arg>` do wstrzykiwania argumentów konstruktora. Tym razem jednak do wstrzyknięcia kolejnego komponentu zamiast atrybutu `ref` wykorzystujemy atrybut `value`. Oznacza to, że przypisana do niego wartość ma być traktowana dosłownie i przekazana do konstruktora.

Jak możemy to zapisać z użyciem przestrzeni nazw `c`? Jednym z możliwych rozwiązań jest odwołanie do argumentów konstruktora za pomocą nazwy:

```
<bean id="compactDisc"
   class="soundsystem.BlankDisc"
   c:_title="Sgt. Pepper's Lonely Hearts Club Band"
   c:_artist="The Beatles" />
```

Jak widzimy, wiązanie literałów z wykorzystaniem przestrzeni nazw `c` różni się od wiązania referencji brakiem sufiksu `-ref` w nazwie atrybutu. W podobny sposób moglibyśmy powiązać wartości literałów z użyciem indeksów parametrów:

```
<bean id="compactDisc"
   class="soundsystem.BlankDisc"
   c:_0="Sgt. Pepper's Lonely Hearts Club Band"
   c:_1="The Beatles" />
```

XML pozwala użyć co najwyżej jednego atrybutu o danej nazwie w ramach pojedynczego elementu. Nie ma więc możliwości zastosowania pojedynczego znaku podkreślenia, gdy konstruktor klasy przyjmuje więcej niż jeden argument. Możemy z niego korzystać tylko w przypadku konstruktorów jednoargumentowych. Aby zilustrować tę sytuację, załóżmy na chwilę, że klasa `BlankDisc` posiada konstruktor jednoargumentowy, który jako argument przyjmuje nazwę albumu. W takim przypadku moglibyśmy zadeklarować komponent w podany niżej sposób:

```
<bean id="compactDisc" class="soundsystem.BlankDisc"
   c:_="Sgt. Pepper's Lonely Hearts Club Band" />
```

Zarówno element `<constructor-arg>`, jak i atrybuty przestrzeni nazw `c` oferują takie same możliwości w zakresie wiązania referencji komponentów i literałów. Istnieje jednak przypadek, w którym możemy skorzystać tylko z pierwszego rozwiązania. Przyjrzyjmy się teraz wiązaniu kolekcji do argumentów konstruktora.

WIĄŻEMY KOLEKCJE

Zakładaliśmy do tej pory, że do opisu komponentów typu `CompactDisc` wystarczą tytuł i nazwa artysty. Jeśli byłyby to jedyne informacje dostępne na płytach CD, technologia ta nie miałaby szans na odniesienie sukcesu. Płytę kupujemy bowiem dla zapisanej na niej muzyki. Przeciętnie na płycie znajduje się około dwunastu utworów muzycznych.

Aby typ `CompactDisc` był odzwierciedleniem prawdziwej płyty CD, musi posiadać notację listy utworów. Przyjrzyjmy się nowej implementacji czystej płyty:

```
package soundsystem.collections;

import java.util.List;
import soundsystem.CompactDisc;

public class BlankDisc implements CompactDisc {

   private String title;
   private String artist;
   private List<String> tracks;

   public BlankDisc(String title, String artist, List<String> tracks) {
      this.title = title;
      this.artist = artist;
      this.tracks = tracks;
   }

   public void play() {
      System.out.println("Odtwarzam utwór " + title + " autorstwa " + artist);
      for (String track : tracks) {
         System.out.println("-Utwór: " + track);
      }
   }
}
```

Wprowadzona zmiana ma wpływ na sposób deklaracji komponentu w Springu. W deklaracji musimy teraz podać listę utworów.

Najprostszym rozwiązaniem jest nadanie liście wartości `null`. Argument konstruktora jest obowiązkowy, ale możemy do niego przekazać wartość `null`:

```
<bean id="compactDisc" class="soundsystem.BlankDisc">
   <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
   <constructor-arg value="The Beatles" />
   <constructor-arg><null/></constructor-arg>
</bean>
```

Element `<null/>` działa zgodnie z oczekiwaniami, przekazuje wartość `null` do konstruktora. Nie jest to może ładny sposób, ale zadziała w momencie wstrzykiwania zależności. Nie jest jednak idealny, bo przy wywołaniu metody `play()` wyrzucony zostanie błąd `NullPointerException`.

Lepszym rozwiązaniem byłoby przekazanie listy nazw utworów. Możemy to zrealizować na kilka różnych sposobów. Przykładowo możemy przekazać listę za pomocą elementu `<list>`:

```
<bean id="compactDisc" class="soundsystem.BlankDisc">
   <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
   <constructor-arg value="The Beatles" />
   <constructor-arg>
      <list>
         <value>Sgt. Pepper's Lonely Hearts Club Band</value>
         <value>With a Little Help from My Friends</value>
         <value>Lucy in the Sky with Diamonds</value>
         <value>Getting Better</value>
         <value>Fixing a Hole</value>
         <!-- ...pozostałe utwory zostały celowo pominięte... -->
      </list>
   </constructor-arg>
</bean>
```

Element `<list>` jest potomkiem elementu `<constructor-arg>` i świadczy o tym, że do konstruktora przekazywana jest lista wartości. Element `<value>` umożliwia ustawienie wartości pojedynczego elementu listy.

Podobnie możemy ustalić listę referencji komponentów, zamieniając element `<value>` na `<ref>`. Przypuśćmy, że istnieje klasa `Discography`, reprezentująca dyskografię i posiadająca następujący konstruktor:

```
public Discography(String artist, List<CompactDisc> cds) { ... }
```

Deklaracja komponentu `Discography` mogłaby wyglądać tak:

```
<bean id="beatlesDiscography"
   class="soundsystem.Discography">
   <constructor-arg value="The Beatles" />
   <constructor-arg>
      <list>
         <ref bean="sgtPeppers" />
         <ref bean="whiteAlbum" />
         <ref bean="hardDaysNight" />
         <ref bean="revolver" />
         ...
```

```
      </list>
   </constructor-arg>
</bean>
```

Element `<list>` możemy wykorzystać do wiązania argumentu konstruktora typu `java.util.List`. W podobny sposób możemy użyć elementu `<set>`:

```
<bean id="compactDisc" class="soundsystem.BlankDisc">
   <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
   <constructor-arg value="The Beatles" />
   <constructor-arg>
      <set>
         <value>Sgt. Pepper's Lonely Hearts Club Band</value>
         <value>With a Little Help from My Friends</value>
         <value>Lucy in the Sky with Diamonds</value>
         <value>Getting Better</value>
         <value>Fixing a Hole</value>
         <!-- ...pozostałe utwory zostały celowo pominięte... -->
      </set>
   </constructor-arg>
</bean>
```

Istnieją subtelne różnice pomiędzy elementami `<set>` i `<list>`. Główną różnicą jest typ kolekcji tworzonej przez Springa, odpowiednio `java.util.Set` i `java.util.List`. Wartości w kolekcji typu `Set` nie mogą się powtarzać, a ich kolejność może nie być respektowana. W obu przypadkach elementy `<set>` jak i `<list>` mogą zostać powiązane zarówno do kolekcji typu `List`, jak i `Set`, a nawet do tablicy.

Wiązanie kolekcji to jedna z przewag elementu `<constructor-arg>` nad atrybutami przestrzeni nazw `c`. Nie istnieje w tym przypadku prosty odpowiednik przedstawionych przykładów wiązania kolekcji.

Jest jeszcze kilka innych niuansów dotyczących wykorzystywania obu konstrukcji w zakresie wstrzykiwania zależności. Informacje podane w tej sekcji powinny Ci umożliwić ich samodzielne poznanie, gdy zajdzie taka potrzeba. Pamiętaj też o mojej wcześniejszej poradzie, aby do konfiguracji Springa używać klas Javy, a nie plików XML. Zostawiam zatem temat wstrzykiwania zależności, a przystępuję do omawiania wiązania właściwości za pomocą konfiguracji XML.

2.4.4. Ustawiamy właściwości

Do tej pory do konfiguracji klas `CDPlayer` i `BlankDisc` używaliśmy wyłącznie wstrzykiwania przez konstruktor. Klasy te nie miały więc metod typu setter, które pozwalałyby na ustawienie ich właściwości. Zobaczmy teraz, jak działa w Springu wstrzykiwanie przez właściwości z konfiguracją w pliku XML. Załóżmy, że nasza nowa, przygotowana na wstrzykiwanie właściwości, wersja klasy `CDPlayer` wygląda następująco:

```
package soundsystem;

import org.springframework.beans.factory.annotation.Autowired;
import soundsystem.CompactDisc;
import soundsystem.MediaPlayer;

public class CDPlayer implements MediaPlayer {
```

```
    private CompactDisc compactDisc;

    @Autowired
    public void setCompactDisc(CompactDisc compactDisc) {
        this.compactDisc = compactDisc;
    }

    public void play() {
        compactDisc.play();
    }
}
```

WYBÓR POMIĘDZY WSTRZYKIWANIEM PRZEZ KONSTRUKTOR A WSTRZYKIWANIEM PRZEZ WŁAŚCIWOŚCI Osobiście stosuję następującą zasadę. Ze wstrzykiwania przez konstruktor korzystam przy wymaganych (twardych) zależnościach, natomiast ze wstrzykiwania przez właściwości korzystam w przypadku zależności opcjonalnych. W obliczu powyższej reguły wydawać się może, że zarówno tytuł, artysta, jak i lista utworów są twardymi zależnościami klasy `BlankDisc` i wstrzykiwanie przez konstruktor powinno być preferowaną metodą. Można by jednak dyskutować, czy `CompactDisc` jest twardą, czy opcjonalną zależnością klasy `CDPlayer`. Ja uważam, że jest to twarda zależność, ale ktoś inny mógłby uznać, że nawet odtwarzacz CD może w jakimś zakresie działać nawet bez włożonej płyty CD.

Teraz gdy klasa `CDPlayer` nie ma żadnego konstruktora (poza niejawnie zdefiniowanym konstruktorem domyślnym), nie ma również żadnych twardych zależności. Deklaracja tego komponentu może więc wyglądać następująco:

```
<bean id="cdPlayer"
    class="soundsystem.CDPlayer" />
```

Spring nie będzie miał żadnych problemów z utworzeniem tego komponentu. Test jednostkowy `CDPlayerTest` zakończy się jednak niepowodzeniem i wywołaniem wyjątku `NullPointerException`, bo do klasy `CDPlayer` nie została wstrzyknięta właściwość `compact` ↪`Disc`. Można to naprawić, wykonując następującą zmianę w pliku XML:

```
<bean id="cdPlayer"
    class="soundsystem.CDPlayer">
    <property name="compactDisc" ref="compactDisc" />
</bean>
```

Element `<property>` pełni przy wstrzykiwaniu zależności przez właściwość funkcję odpowiadającą funkcji pełnionej przez element `<constructor-arg>` przy wstrzykiwaniu przez konstruktor. W podanym przykładzie odwołuje się za pomocą atrybutu `ref` do komponentu o identyfikatorze `compactDisc`, który ma zostać wstrzyknięty do właściwości o tej samej nazwie (za pośrednictwem metody `setCompactDisc()`). Teraz uruchomienie testów powinno się zakończyć powodzeniem.

Może Cię też zainteresować fakt, że podobnie jak w przypadku konstruktorów, gdzie alternatywą dla elementu `<constructor-arg>` były atrybuty przestrzeni nazw `c`, tak dla właściwości alternatywą dla elementu `<property>` są atrybuty przestrzeni nazw `p`.

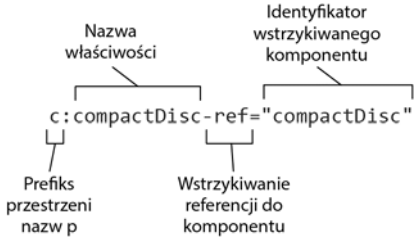
Do włączenia przestrzeni nazw p służy deklaracja odpowiedniego schematu w preambule pliku XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:p="http://www.springframework.org/schema/p"
   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>
```

Przestrzeń nazw p możemy też wykorzystać do wiązania właściwości compactDisc:

```
<bean id="cdPlayer"
   class="soundsystem.CDPlayer"
   p:compactDisc-ref="compactDisc" />
```

Atrybuty przestrzeni nazw p podlegają podobnej konwencji nazewniczej jak ta znana z atrybutów przestrzeni nazw c. Rysunek 2.2 ilustruje poszczególne składowe nazwy atrybutu.



Rysunek 2.2. Wstrzykiwanie referencji do komponentu do właściwości za pomocą przestrzeni nazw p Springa

Nazwa atrybutu poprzedzona jest prefiksem p:, co wskazuje, że ustawiamy wartość właściwości (ang. *property*). Po prefiksie znajduje się nazwa wstrzykiwanej właściwości. Nazwę atrybutu kończy sufiks -ref, dzięki któremu Spring wie, że wiązana jest referencja do komponentu, a nie literał.

WSTRZYKUJEMY WŁAŚCIWOŚCI O WARTOŚCIACH BĘDĄCYCH LITERAŁAMI

Wstrzyknięcie literałów do właściwości następuje w podobny sposób jak w przypadku argumentów konstruktora. Spójrzmy ponownie na komponent BlankDisc. Tym razem jednak przeprowadzimy jego konfigurację z użyciem wstrzykiwania przez właściwość, a nie konstruktor. Nowa wersja klasy BlankDisc wygląda następująco:

```
package soundsystem;

import java.util.List;
import soundsystem.CompactDisc;

public class BlankDisc implements CompactDisc {
   private String title;
   private String artist;
   private List<String> tracks;

   public void setTitle(String title) {
      this.title = title;
```

```
    }

    public void setArtist(String artist) {
        this.artist = artist;
    }

    public void setTracks(List<String> tracks) {
        this.tracks = tracks;
    }

    public void play() {
        System.out.println("Odtwarzam " + title + " autorstwa " + artist);
        for (String track : tracks) {
            System.out.println("-Utwór: " + track);
        }
    }
}
```

Podanie żadnej z tych właściwości nie jest obowiązkowe. Komponent `BlankDisc` moglibyśmy utworzyć bez ustawiania jakichkolwiek właściwości:

```
<bean id="reallyBlankDisc"
    class="soundsystem.BlankDisc" />
```

Oczywiście zastosowanie takiej konfiguracji wiązania komponentów nie zadziała prawidłowo po uruchomieniu. Metoda `play()` informuje nas, że odtwarza utwór `null` autorstwa `null`, a chwilę potem wyrzucany jest wyjątek `NullPointerException`, spowodowany brakiem utworów. Dlatego powinniśmy jednak powiązać te właściwości. W tym celu wykorzystamy atrybut `value` elementu `<property>`:

```
<bean id="compactDisc"
    class="soundsystem.BlankDisc">
    <property name="title"
        value="Sgt. Pepper's Lonely Hearts Club Band" />
    <property name="artist" value="The Beatles" />
    <property name="tracks">
        <list>
            <value>Sgt. Pepper's Lonely Hearts Club Band</value>
            <value>With a Little Help from My Friends</value>
            <value>Lucy in the Sky with Diamonds</value>
            <value>Getting Better</value>
            <value>Fixing a Hole</value>
            <!-- ...pozostałe utwory pominięte celowo, aby skrócić listing... -->
        </list>
    </property>
</bean>
```

Poza ustawieniami atrybutu `value` właściwości `title` i `artist` warto zwrócić uwagę na ustawienia właściwości `tracks`. Wykorzystaliśmy tam zagnieżdżony element `<list>`, analogicznie jak przy wiązaniu listy utworów za pomocą elementu `<constructor-arg>`.

Ten sam cel możemy osiągnąć za pomocą atrybutów przestrzeni nazw `p`:

```
<bean id="compactDisc"
    class="soundsystem.BlankDisc"
    p:title="Sgt. Pepper's Lonely Hearts Club Band"
    p:artist="The Beatles">
```

```
   <property name="tracks">
      <list>
         <value>Sgt. Pepper's Lonely Hearts Club Band</value>
         <value>With a Little Help from My Friends</value>
         <value>Lucy in the Sky with Diamonds</value>
         <value>Getting Better</value>
         <value>Fixing a Hole</value>
         <!-- ...pozostałe utwory pominięto celowo, aby skrócić listing... -->
      </list>
   </property>
</bean>
```

Tak jak w przypadku atrybutów przestrzeni nazw `c`, tak i tutaj jedyną różnicą pomiędzy wiązaniem referencji a wiązaniem literałów jest obecność lub brak sufiksu `-ref`. Brak sufiksu wskazuje na wiązanie literałów.

Pamiętaj jednak, że przestrzeni nazw `p` nie możemy wykorzystać do wiązania kolekcji. Nie istnieje niestety wygodny sposób określenia listy wartości (bądź referencji do komponentów) z użyciem tej przestrzeni nazw. Możemy jednak uprościć tworzenie komponentu `BlankDisc` za pomocą przestrzeni nazw `util`.

Na początek zadeklarujmy przestrzeń nazw `util` i jej schemat w pliku XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:util="http://www.springframework.org/schema/util"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/util
      http://www.springframework.org/schema/util/spring-util.xsd">
   ...
</beans>
```

Jednym z elementów udostępnianych przez tę przestrzeń nazw jest `<util:list>`, który pozwala na utworzenie listy komponentów. Użycie elementu `<util:list>` umożliwia wyniesienie listy utworów z wnętrza komponentu `BlankDisc` i umieszczenie w oddzielnym komponencie:

```
<util:list id="trackList">
   <value>Sgt. Pepper's Lonely Hearts Club Band</value>
   <value>With a Little Help from My Friends</value>
   <value>Lucy in the Sky with Diamonds</value>
   <value>Getting Better</value>
   <value>Fixing a Hole</value>
   <!-- ...pozostałe utwory pominięto, aby skrócić listing... -->
</util:list>
```

Teraz możemy powiązać komponent `trackList` z właściwością `tracks` komponentu `BlankDisc` w taki sam sposób jak wszystkie pozostałe komponenty:

```
<bean id="compactDisc"
   class="soundsystem.BlankDisc"
   p:title="Sgt. Pepper's Lonely Hearts Club Band"
   p:artist="The Beatles"
   p:tracks-ref="trackList" />
```

Element <util:list> to tylko jeden spośród kilku elementów udostępnionych w przestrzeni nazw util. Tabela 2.1 zawiera opis wszystkich dostępnych elementów.

Tabela 2.1. Elementy w przestrzeni nazw util w Springu

Element	Opis
<util:constant>	Tworzy powiązanie z polem public static danego typu i udostępnia jako komponent.
<util:list>	Tworzy komponent będący listą (java.util.List) wartości lub referencji.
<util:map>	Tworzy komponent będący mapą (java.util.Map) wartości lub referencji.
<util:properties>	Tworzy komponent właściwości java.util.Properties.
<util:property-path>	Tworzy odnośnik do właściwości (lub zagnieżdżonej właściwości) i udostępnia jako komponent.
<util:property-path>	Tworzy komponent będący zbiorem (java.util.Set) wartości lub referencji.

Czasami może się pojawić potrzeba wykorzystania elementów przestrzeni nazw util. W tej chwili jednak, podsumowując ten rozdział, spójrzmy na sposób łączenia różnych typów konfiguracji: automatycznej, JavaConfig oraz XML.

2.5. Importujemy i łączymy konfiguracje

W typowej aplikacji springowej z reguły zachodzi potrzeba wykorzystania jednocześnie zarówno automatycznej, jak i niejawnej konfiguracji. Nawet jeśli do jawnej konfiguracji preferujesz użycie klas Javy, w niektórych sytuacjach zastosowanie plików XML może być najlepszym rozwiązaniem.

Na szczęście różne opcje konfiguracji wzajemnie się nie wykluczają. Możemy w dowolny sposób łączyć skanowanie i autowiązanie komponentów z konfiguracją w klasach Javy i (lub) w plikach XML. W sekcji 2.2.1 zobaczyliśmy, że niezbędne jest chociaż częściowe użycie jawnej konfiguracji, aby można było włączyć mechanizmy skanowania i autowiązanie komponentów.

Pierwszą rzeczą, którą musisz wiedzieć przy łączeniu różnych stylów konfiguracji, jest to, że przy autowiązaniu nie jest istotny sposób, w jaki powstały komponenty. Autowiązanie uwzględnia wszystkie komponenty dostępne w kontenerze Springa, niezależnie od tego, czy ich deklaracja znajduje się w klasach Javy, plikach XML lub czy zostały wyszukane przez mechanizm skanowania komponentów.

Pojawia się pytanie, jak możemy się odwołać do komponentów zadeklarowanych za pomocą innej metody jawnej konfiguracji. Poznamy najpierw metodę odwołania się z poziomu konfiguracji JavaConfig do komponentów skonfigurowanych w plikach XML.

2.5.1. Odwołujemy się do konfiguracji XML z poziomu konfiguracji JavaConfig

Przez chwilę załóżmy, że nasza konfiguracja CDPlayerConfig stała się duża, trudna w utrzymaniu i chcemy ją rozbić na mniejsze elementy. Oczywiście w naszym przypadku klasa zawiera tylko dwa komponenty, trudno więc powiedzieć, że jest skomplikowana. Możemy poudawać, że te dwa komponenty to o dwa za dużo.

Rozwiążemy nasz problem, wyodrębniając komponent `BlankDisc` z klasy `CDPlayer ➥Config` do osobnej klasy `CDConfig`:

```
package soundsystem;

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

@Configuration
public class CDConfig {
   @Bean
   public CompactDisc compactDisc() {
      return new SgtPeppers();
   }
}
```

Następnie, po przeniesieniu metody `compactDisc()` z klasy `CDPlayerConfig`, musimy poznać sposób, by połączyć obie konfiguracje. Jedną z możliwości jest zaimportowanie konfiguracji `CDPlayerConfig` za pomocą adnotacji `@Import`:

```
package soundsystem;

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

@Configuration
@Import(CDConfig.class)
public class CDPlayerConfig {
   @Bean
   public CDPlayer cdPlayer(CompactDisc compactDisc) {
      return new CDPlayer(compactDisc);
   }
}
```

Jeszcze lepszym rozwiązaniem byłoby usunięcie adnotacji `@Import` z klasy `CDPlayer ➥Config` do nowo utworzonej klasy `SoundSystemConfig`, której zadaniem byłoby importowanie konfiguracji z obu plików:

```
package soundsystem;

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

@Configuration
@Import({CDPlayerConfig.class, CDConfig.class})
public class SoundSystemConfig {
}
```

W jednym i drugim przypadku udało się nam rozdzielić konfigurację komponentów `CDPlayer` i `BlankDisc`. Załóżmy teraz, że z jakiegoś powodu chcesz skonfigurować komponent `BlankDisc` w pliku XML:

```
<bean id="compactDisc"
   class="soundsystem.BlankDisc"
   c:_0="Sgt. Pepper's Lonely Hearts Club Band"
   c:_1="The Beatles">
```

```
    <constructor-arg>
      <list>
        <value>Sgt. Pepper's Lonely Hearts Club Band</value>
        <value>With a Little Help from My Friends</value>
        <value>Lucy in the Sky with Diamonds</value>
        <value>Getting Better</value>
        <value>Fixing a Hole</value>
        <!-- ...pozostałe utwory pominięte w celu skrócenia listingu... -->
      </list>
    </constructor-arg>
</bean>
```

Teraz gdy deklaracja komponentu `BlankDisc` znajduje się w pliku XML, jak możemy ją wczytać wraz z całą konfiguracją zadeklarowaną w klasach Javy?

Odpowiedzią jest adnotacja `@ImportResource`. Załóżmy, że komponent `BlankDisc` zadeklarowany został w pliku o nazwie *cd-config.xml*, znajdującym się w katalogu głównym ścieżki klas. Możemy zmienić klasę `SoundSystemConfig` tak, aby korzystała z adnotacji `@ImportResource`:

```
package soundsystem;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.ImportResource;

@Configuration
@Import(CDPlayerConfig.class)
@ImportResource("classpath:cd-config.xml")
public class SoundSystemConfig {
}
```

Oba komponenty — `CDPlayer` skonfigurowany za pomocą konfiguracji JavaConfig oraz `BlankDisc` skonfigurowany z użyciem pliku XML — zostaną wczytane przez kontener Springa. Metoda komponentu `CDPlayer` opatrzona adnotacją `@Bean` przyjmuje parametr `CompactDisc`, więc komponent `BlankDisc` zostanie do niego powiązany, mimo że został skonfigurowany w pliku XML.

Spójrzmy na ten przykład raz jeszcze, ale tym razem odwołamy się do konfiguracji JavaConfig z poziomu pliku XML.

2.5.2. Odwołujemy się do konfiguracji JavaConfig z poziomu konfiguracji XML

Przypuśćmy, że pracujesz z konfiguracją Springa opartą na pliku XML i stwierdzasz, iż plik zrobił się zbyt duży, nieczytelny i trudny w utrzymaniu. Tak jak i w poprzednim przykładzie, mamy do czynienia z zaledwie dwoma komponentami, nie jest to więc najgorsza sytuacja. Chcąc się uratować przed lawiną znaków mniejszości i większości, charakterystycznych dla plików XML, postanawiasz rozbić konfigurację XML na mniejsze części.

Na przykładzie konfiguracji JavaConfig pokazałem, jak rozbić konfigurację zapisaną w klasach Javy, korzystając z adnotacji `@Import` oraz `@ImportResource`. Odpowiednikiem tych adnotacji w plikach XML jest element `<import>`.

Przykładowo chcemy dla komponentu `BlankDisc` wydzielić osobny plik konfiguracji o nazwie *cd-config.xml*, podobnie jak we wcześniejszym przykładzie dla adnotacji `@Import ➥Resource`. Możemy odwołać się do tego pliku z poziomu konfiguracji XML za pomocą elementu `<import>`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:c="http://www.springframework.org/schema/c"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <import resource="cd-config.xml" />
   <bean id="cdPlayer"
      class="soundsystem.CDPlayer"
      c:cd-ref="compactDisc" />
</beans>
```

Teraz załóżmy, że przenosimy deklarację komponentu `BlankDisc` z pliku XML do konfiguracji w klasie `CDPlayer`. W jaki sposób możemy odwołać się z poziomu konfiguracji XML do konfiguracji JavaConfig?

Jak się okazuje, odpowiedź nie jest oczywista. Element `<import>` działa wyłącznie z konfiguracją umieszczoną w innych plikach XML. Nie istnieje żaden inny element XML, którego zadaniem byłoby zaimportowanie klas JavaConfig.

Istnieje na szczęście znany Ci już element, który możemy wykorzystać do wczytania konfiguracji z klas Javy. Elementem tym jest `<bean>`. Aby zaimportować klasę JavaConfig do konfiguracji XML, musimy ją zadeklarować w postaci komponentu:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:c="http://www.springframework.org/schema/c"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean class="soundsystem.CDConfig" />
   <bean id="cdPlayer"
      class="soundsystem.CDPlayer"
      c:cd-ref="compactDisc" />
</beans>
```

I w ten właśnie sposób obie konfiguracje — jedna wyrażona w pliku XML a druga w klasie Javy — zostały połączone. Podobnie jak poprzednio, moglibyśmy utworzyć nowy plik konfiguracyjny niezawierający deklaracji żadnych komponentów. Jego zadaniem byłoby tylko łączenie konfiguracji. Moglibyśmy usunąć komponent `CDConfig` z poprzedniej konfiguracji XML i dodać kolejny plik konfiguracji, służący do ich połączenia.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:c="http://www.springframework.org/schema/c"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
    <bean class="soundsystem.CDConfig" />
    <import resource="cdplayer-config.xml" />
</beans>
```

Niezależnie od tego, czy korzystam z konfiguracji powiązań za pomocą JavaConfig, czy plików XML, tworzę z reguły główną konfigurację, która, tak jak w pokazanych przykładach, łączy ze sobą dwie lub więcej klas konfiguracji albo plików XML. I właśnie ten główny plik konfiguracji jest zazwyczaj miejscem, w którym włączam skanowanie komponentów (za pomocą elementu `<context:component-scan>` bądź adnotacji `@Component ↪Scan`). Z tej praktyki będziemy niejednokrotnie korzystać w przykładach w tej książce.

2.6. Podsumowanie

W sercu Spring Frameworka znajduje się kontener Springa. Odpowiada on za zarządzanie cyklem życia komponentów w aplikacji i tworzenie nowych komponentów oraz dba o spełnienie ich zależności, dzięki czemu mogą wykonywać przewidzianą dla nich pracę.

W tym rozdziale poznałeś trzy sposoby wiązania komponentów: automatyczną konfigurację, jawną konfigurację w klasach Javy oraz jawną konfigurację w plikach XML. Wybór jest dowolny, bo każda z opisanych technik pozwala zdefiniować komponenty w aplikacji Springa i określić ich wzajemne relacje.

Gorąco polecam w jak największym stopniu korzystać z automatycznej konfiguracji, co pozwala zminimalizować koszt utrzymania pojawiający się przy użyciu jawnej konfiguracji. Kiedy jednak zajdzie potrzeba zastosowania jawnej konfiguracji Springa, zalecam wybór konfiguracji opartej na klasach Javy — oferuje ona większe możliwości, jest bezpieczniejsza ze względu na typy i prostsza w modyfikacji niż konfiguracja bazująca na plikach XML. Moje osobiste preferencje w tym zakresie będą widoczne przy wyborze technik wiązania komponentów w przykładach pojawiających się w tej książce.

Wstrzykiwanie zależności jest esencją pracy ze Springiem, a techniki opisane w tym rozdziale odgrywają kluczową rolę w niemal wszystkich zagadnieniach poruszonych na łamach tej książki. W następnym rozdziale, uzbrojony w zdobytą wiedzę, poznasz bardziej zaawansowane techniki wiązania komponentów, które umożliwią Ci jeszcze lepsze wykorzystanie możliwości kontenera Springa.

3 Zaawansowane opcje wiązania

W tym rozdziale omówimy:

- Profile w Springu
- Warunkową deklarację komponentów
- Niejednoznaczność w autowiązaniach
- Zasięg komponentów
- Język wyrażeń Springa (SpEL)

W poprzednim rozdziale poznałeś podstawowe techniki wiązania komponentów. Zdobyte w nim informacje wykorzystasz niejednokrotnie w swojej codziennej pracy. Ale wiązanie komponentów nie ogranicza się tylko do technik opisanych w rozdziale 2. Spring trzyma parę asów w rękawie, które w razie potrzeby umożliwiają zaawansowane wiązanie komponentów.

W tym rozdziale opiszę kilka z tych zaawansowanych technik. Nie są one aż tak powszechnie stosowane, co wcale nie oznacza, że są mniej wartościowe.

3.1. Środowiska i profile

Jednym z najtrudniejszych etapów wytwarzania oprogramowania jest przeniesienie aplikacji z jednego środowiska na drugie. Niektóre wybory dokonane w trakcie tworzenia kodu nie będą właściwe lub będą bezużyteczne po wdrożeniu aplikacji na środowisko produkcyjne. Konfiguracja bazy danych, algorytmy szyfrowania oraz integracja

z zewnętrznymi systemami to tylko kilka z przykładów potencjalnych różnic pomiędzy środowiskami.

Rozważmy dla przykładu konfigurację bazy danych. W środowisku deweloperskim powszechnym wyborem jest wbudowana baza danych wypełniona testowymi danymi. Przykładowo w klasie konfiguracji Springa możemy wykorzystać klasę `EmbeddedDatabase➥Builder` w metodzie opatrzonej adnotacją `@Bean`.

```
@Bean(destroyMethod="shutdown")
public DataSource dataSource() {
   return new EmbeddedDatabaseBuilder()
      .addScript("classpath:schema.sql")
      .addScript("classpath:test-data.sql")
      .build();
}
```

Wynikiem działania tej metody jest utworzenie komponentu typu `javax.sql.DataSource`. Najciekawszy jest jednak *sposób, w jaki* to jest zrealizowane. Klasa `EmbeddedDatabase➥Builder` przygotowuje wbudowaną bazę danych Hypersonic o schemacie skonfigurowanym w pliku *schema.sql* i z danymi testowymi wczytanymi z *pliku test-data.sql*.

Podana konfiguracja źródła danych sprawdza się świetnie w środowisku deweloperskim, gdy odpalamy testy integracyjne albo uruchamiamy aplikację w celu ręcznego testowania. Za każdym razem przy uruchomieniu aplikacji stan początkowy naszej bazy danych jest identyczny.

To, co czyni komponent `dataSource` utworzony z użyciem klasy `EmbeddedDatabase➥Builder` idealnym rozwiązaniem na czas programowania, dyskwalifikuje go równocześnie w warunkach produkcyjnych. W środowisku produkcyjnym lepszym rozwiązaniem byłoby pobranie komponentu `DataSource` z kontenera za pomocą JNDI. W takim przypadku nasza metoda powinna wyglądać następująco:

```
@Bean
public DataSource dataSource() {
   JndiObjectFactoryBean jndiObjectFactoryBean = new JndiObjectFactoryBean();
   jndiObjectFactoryBean.setJndiName("jdbc/myDS");
   jndiObjectFactoryBean.setResourceRef(true);
   jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
   return (DataSource) jndiObjectFactoryBean.getObject();
}
```

Pobranie źródła danych z JNDI pozwala kontenerowi na podjęcie decyzji o sposobie utworzenia komponentu, wliczając w to obsługę puli połączeń zarządzanej przez kontener. Jak wcześniej wspominałem, wykorzystanie źródła danych zarządzanego przez JNDI przydaje się bardziej w środowisku produkcyjnym, a jego użycie na potrzeby testów integracyjnych lub testów manualnych wiązałoby się z niepotrzebnym komplikowaniem konfiguracji.

Równocześnie w środowisku testowym (ang. *Quality Assurance* — QA) moglibyśmy wykorzystać jeszcze inne ustawienia źródła danych. W poniższym przykładzie konfigurujemy pulę połączeń Commons DBCP:

```
@Bean(destroyMethod="close")
public DataSource dataSource() {
   BasicDataSource dataSource = new BasicDataSource();
```

```
    dataSource.setUrl("jdbc:h2:tcp://dbserver/~/test");
    dataSource.setDriverClassName("org.h2.Driver");
    dataSource.setUsername("sa");
    dataSource.setPassword("password");
    dataSource.setInitialSize(20);
    dataSource.setMaxActive(30);
    return dataSource;
}
```

Jak widać, wszystkie trzy zaprezentowane wersje metody `dataSource()` różnią się od siebie. Ich jedyną wspólną cechą jest to, że zwracają komponent typu `javax.sql.DataSource`, ale wszystkie realizują to zadanie z wykorzystaniem zupełnie odmiennej strategii.

Chcę podkreślić, że celem tych rozważań nie jest pokazanie, jak należy konfigurować źródło danych (do tego wrócimy w rozdziale 10.). Definicja komponentu `DataSource` nie jest aż tak prosta, jak się nam mogło początkowo wydawać. Jest to dobry przykład na odmienną implementację komponentu pomiędzy środowiskami. Musimy znaleźć sposób, który umożliwi nam taką konfigurację komponentu `DataSource`, aby dla każdego środowiska wybrana została jego najbardziej odpowiednia implementacja.

Jednym z możliwych sposobów byłoby skonfigurowanie każdego komponentu w osobnej klasie (lub pliku XML) konfiguracji, a następnie na etapie budowania projektu (na przykład z wykorzystaniem profili Mavena) wybór klasy, która ma zostać skompilowana we wdrażanej aplikacji. Niedogodnością tego rozwiązania jest konieczność przebudowania aplikacji dla każdego środowiska. Przebudowa aplikacji przy przenoszeniu aplikacji ze środowiska deweloperskiego do środowiska testowego może nie być jeszcze takim problemem. Ale wymóg przebudowania projektu na potrzeby środowiska produkcyjnego po przejściu testów wiąże się z ryzykiem powstania nowych błędów i doprowadzenia członków zespołu QA do zawału.

Na szczęście Spring udostępnia rozwiązanie problemu, które nie wymaga przebudowywania kodu.

3.1.1. Konfigurujemy komponenty profilu

Rozwiązanie udostępniane przez Springa nie różni się aż tak bardzo od wspomnianego rozwiązania dotyczącego etapu budowania. Tutaj również na podstawie środowiska podejmowana jest decyzja, które komponenty mają zostać utworzone, a które nie. Różnica polega na tym, że podejmowanie decyzji nie następuje na etapie budowania projektu, lecz Spring robi to dopiero w momencie uruchomienia aplikacji. W rezultacie możliwe jest wykorzystanie tego samego pliku wynikowego (na przykład pliku WAR) we wszystkich środowiskach bez potrzeby przebudowywania.

W wersji 3.1 Springa pojawiły się profile komponentów. Jeśli chcesz z nich korzystać, musisz zgromadzić wszystkie definicje komponentów do jednego bądź większej liczby profili i upewnić się, że po wdrożeniu na każde środowisko aktywny jest właściwy profil.

W konfiguracji za pomocą klas Javy możemy określić, do którego profilu należy dany komponent, oznaczając go adnotacją `@Profile`. Klasa definiująca nasz przykładowy komponent `DataSource`, używający wbudowanej bazy danych, może wyglądać następująco:

```
package com.myapp;
import javax.activation.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
@Configuration @Profile("dev")
public class DevelopmentProfileConfig {
   @Bean(destroyMethod="shutdown")
   public DataSource dataSource() {
      return new EmbeddedDatabaseBuilder()
         .setType(EmbeddedDatabaseType.H2)
         .addScript("classpath:schema.sql")
         .addScript("classpath:test-data.sql")
         .build();
   }
}
```

Głównym elementem, na który chciałbym zwrócić uwagę, jest adnotacja `@Profile`, zastosowana na poziomie klasy. Dzięki tej adnotacji Spring wie, że komponenty w tej klasie mają być tworzone tylko w środowisku deweloperskim (gdy aktywny jest profil `dev`). Jeśli profil `dev` nie jest aktywny, metody opatrzone adnotacjami `@Bean` są ignorowane.

Zdefiniujmy też osobną klasę konfiguracji dla środowiska produkcyjnego:

```
package com.myapp;

import javax.activation.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration
@Profile("prod")
public class ProductionProfileConfig {

   @Bean
   public DataSource dataSource() {
      JndiObjectFactoryBean jndiObjectFactoryBean = new JndiObjectFactoryBean();
      jndiObjectFactoryBean.setJndiName("jdbc/myDS");
      jndiObjectFactoryBean.setResourceRef(true);
      jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
      return (DataSource) jndiObjectFactoryBean.getObject();
   }
}
```

Komponent zdefiniowany w tej klasie nie zostanie utworzony, dopóki aktywnym profilem nie będzie `prod`.

W Springu 3.1 użycie adnotacji `@Profile` było dozwolone jedynie na poziomie klasy. Zmieniło się to jednak w wersji 3.2, w której dodano możliwość jej wykorzystania na poziomie metody. Pozwala to na umieszczenie obu deklaracji komponentów w pojedynczej klasie konfiguracji, tak jak pokazano na listingu 3.1.

Listing 3.1. Adnotacja @Profile umożliwia wiązanie komponentów w oparciu o aktywne profile

```
package com.myapp;

import javax.activation.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration
public class DataSourceConfig {

   @Bean(destroyMethod="shutdown")
   @Profile("dev") <-- Wiązanie dla profilu "dev"
   public DataSource embeddedDataSource() {
      return new EmbeddedDatabaseBuilder()
         .setType(EmbeddedDatabaseType.H2)
         .addScript("classpath:schema.sql")
         .addScript("classpath:test-data.sql")
         .build();
   }

   @Bean
   @Profile("prod") <-- Wiązanie dla profilu "prod"
   public DataSource jndiDataSource() {
      JndiObjectFactoryBean jndiObjectFactoryBean = new JndiObjectFactoryBean();
      jndiObjectFactoryBean.setJndiName("jdbc/myDS");
      jndiObjectFactoryBean.setResourceRef(true);
      jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
      return (DataSource) jndiObjectFactoryBean.getObject();
   }
}
```

Obie deklaracje komponentu `DataSource` znajdują się w tym samym pliku i tworzone są tylko wtedy, gdy aktywny jest odpowiadający im profil. Możesz się zastanawiać, co się dzieje z komponentami niezdefiniowanymi w zasięgu aktywnego profilu. Odpowiedź jest prosta: komponenty, które nie są przypisane do żadnego profilu, są wczytywane zawsze, niezależnie od jego ustawień.

KONFIGURUJEMY PROFILE W PLIKACH XML

Innym sposobem deklaracji komponentów uzależnionych od ustawień profilu jest użycie konfiguracji XML. Służy do tego atrybut `profile` elementu `<beans>`. Przykładowo jeśli chcemy zdefiniować komponent `DataSource` w pliku XML tak, aby na czas programowania korzystał z wbudowanej bazy danych, możemy utworzyć plik XML o następującej treści:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
```

```
   xsi:schemaLocation=" http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
   profile="dev">
   <jdbc:embedded-database id="dataSource">
      <jdbc:script location="classpath:schema.sql" />
      <jdbc:script location="classpath:test-data.sql" />
   </jdbc:embedded-database>
</beans>
```

W podobny sposób moglibyśmy przygotować osobny plik konfiguracyjny z ustawieniami profilu dla środowiska produkcyjnego (wartość `profile="prod"`) i utworzyć komponent `DataSource` za pośrednictwem JNDI, a następnie kolejny plik na potrzeby środowiska testowego (wartość `profile="qa"`). Wszystkie te pliki XML znajdą się w jednym pliku wynikowym (na przykład w pliku WAR), ale wykorzystywany będzie tylko ten, który odpowiada aktywnemu profilowi.

Nie ma jednak potrzeby tworzenia wielu plików XML dla każdego środowiska. Możemy wykorzystać opcję definiowania wielu elementów podrzędnych `<beans>` wewnątrz nadrzędnego elementu `<beans>`. Umożliwia to zebranie wszystkich komponentów polegających na użyciu profili wewnątrz pojedynczego pliku XML, tak jak na listingu 3.2.

Listing 3.2. Elementy <beans> mogą pojawiać się wielokrotnie na potrzeby poszczególnych profili

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xmlns:jdbc=http://www.springframework.org/schema/jdbc
   xmlns:jee=http://www.springframework.org/schema/jee
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="
      http://www.springframework.org/schema/jee
      http://www.springframework.org/schema/jee/spring-jee.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <beans profile="dev">                                    Komponenty profilu "dev"
      <jdbc:embedded-database id="dataSource">
         <jdbc:script location="classpath:schema.sql" />
         <jdbc:script location="classpath:test-data.sql" />
      </jdbc:embedded-database>
   </beans>
   <beans profile="qa">                                     Komponenty profilu "qa"
      <bean id="dataSource"
         class="org.apache.commons.dbcp.BasicDataSource"
         destroy-method="close"
         p:url="jdbc:h2:tcp://dbserver/~/test"
         p:driverClassName="org.h2.Driver"
         p:username="sa"
         p:password="password"
         p:initialSize="20"
```

```
        p:maxActive="30" />
   </beans>
<beans profile="prod">                                    Komponenty profilu "prod"
      <jee:jndi-lookup id="dataSource"
         jndi-name="jdbc/myDatabase"
         resource-ref="true"
         proxy-interface="javax.sql.DataSource" />
   </beans>
</beans>
```

Pomijając fakt, że wszystkie komponenty są teraz zdefiniowane wewnątrz tego samego pliku XML, efekt jest identyczny jak przy rozdzieleniu profili na osobne pliki. Dostępne są trzy komponenty — typem każdego z nich jest `javax.sql.DataSource`, a identyfikatorem `dataSource`. Po uruchomieniu aplikacji utworzony zostanie jednak tylko jeden komponent, w zależności od tego, który profil jest aktywny.

Nasuwa się pytanie: jak uaktywnić wybrany profil?

3.1.2. Aktywujemy profil

Kiedy Spring sprawdza, który profil jest aktywny, bierze pod uwagę dwie niezależne właściwości: `spring.profiles.active` oraz `spring.profiles.default`. Ustawienie właściwości `spring.profiles.active` powoduje, że jej wartość staje się wartością aktywnego profilu. Jeśli jednak właściwość ta nie zostanie ustawiona, Spring sprawdzi wartość właściwości `spring.profiles.default`. Jeżeli żadna z tych właściwości nie jest ustawiona, nie zostanie wybrany żaden profil i utworzone będą tylko komponenty niezdefiniowane w ramach jakiegokolwiek profilu.

Istnieje kilka sposobów ustawiania właściwości:

- jako parametry inicjalizacji klasy `DispatcherServlet`;
- jako parametry kontekstu aplikacji internetowej;
- jako wpisy JNDI;
- jako zmienne środowiskowe;
- jako właściwości systemowe JVM;
- za pomocą adnotacji `@ActiveProfiles` w klasie testów integracyjnych.

Wybór najodpowiedniejszej kombinacji właściwości `spring.profiles.active` i `spring.profiles.default` pozostawiam Tobie, drogi Czytelniku.

Podejście, które mi osobiście najbardziej pasuje, to ustawienie wartości `spring.profiles.default` na profil deweloperski w parametrze klasy `DispatcherServlet` i w kontekście serwletu (na potrzeby klasy `ContextLoaderListener`). Właściwość `spring.profiles.default` można ustawić w pliku *web.xml* aplikacji, tak jak pokazano na listingu 3.3.

Listing 3.3. Ustawianie domyślnego profilu w pliku web.xml aplikacji

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
   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_2_5.xsd">
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/root-context.xml</param-value>
   </context-param>
   <context-param>
    <param-name>spring.profiles.default</param-name>
     <param-value>dev</param-value>
   </context-param>
   <listener>
      <listener-class>
   org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>
   <servlet>
      <servlet-name>appServlet</servlet-name>
      <servlet-class>
         org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
         <param-name>spring.profiles.default</param-name>
         <param-value>dev</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>appServlet</servlet-name>
      <url-pattern>/</url-pattern>
   </servlet-mapping>
</web-app>
```

Ustawianie domyślnego profilu dla kontekstu

Ustawianie domyślnego profilu dla serwletu

Po skonfigurowaniu właściwości `spring.profiles.default` w podany sposób każdy programista może pobrać kod aplikacji za pomocą systemu kontroli wersji i uruchomić w trybie deweloperskim (na przykład z użyciem wbudowanej bazy danych) bez jakiejkolwiek dodatkowej konfiguracji.

Następnie po wdrożeniu aplikacji na środowisko testowe, produkcyjne lub dowolne inne osoba odpowiedzialna za wdrożenie może ustawić wartość `spring.profiles.active` za pomocą właściwości systemowych, zmiennych środowiskowych albo interfejsu JNDI. Po ustawieniu wartości `spring.profiles.active` właściwość `spring.profiles.default` nie jest już brana pod uwagę.

W nazwie właściwości `spring.profiles.active` oraz `spring.profiles.default` pojawia się słowo `profiles` (profile) w liczbie mnogiej. Oznacza to możliwość jednoczesnej aktywacji kilku profili poprzez podanie jako wartości ich nazw oddzielonych przecinkiem. Równoczesne aktywowanie profili `dev` i `prod` nie ma zbytniego sensu, ale możemy włączyć wiele ortogonalnych profili.

KORZYSTAMY Z PROFILI DO TESTOWANIA

Kiedy uruchamiamy testy integracyjne, chcemy je często przeprowadzić z użyciem takiej samej konfiguracji (albo jej podzbioru), jaka zostanie wykorzystana w środowisku produkcyjnym. Jeśli jednak konfiguracja odwołuje się do komponentów znajdujących się w profilach, musimy znaleźć sposób, aby wybrać profil na czas trwania testu.

Spring udostępnia nam adnotację `@ActiveProfiles`, pozwalającą określić, które profile mają być aktywne po uruchomieniu testu. Profilem, który jest często wykorzystywany w testach integracyjnych, jest profil deweloperski. Przykładowo poniżej znajduje się fragment klasy testu opatrzonej adnotacją `@ActiveProfiles`, która ustawia profil deweloperski.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={PersistenceTestConfig.class})
@ActiveProfiles("dev")
public class PersistenceTest {
   ...
}
```

Profile Springa to wspaniałe narzędzie do definiowania komponentów w zależności od aktywnego profilu. W Springu w wersji 4. pojawił się jednak bardziej uniwersalny mechanizm warunkowej definicji komponentów, który oferuje dużo większą elastyczność w tym zakresie. Przyjrzyjmy się teraz warunkowej definicji komponentów z użyciem adnotacji `@Conditional`.

3.2. Warunkowe komponenty

Przypuśćmy, że jeden lub większa liczba komponentów ma być skonfigurowana wtedy i tylko wtedy, gdy w ścieżce klas aplikacji będzie dostępna pewna biblioteka. Albo że dany komponent ma zostać utworzony tylko wtedy, gdy zadeklarowany jest również inny komponent. Możemy też chcieć utworzyć komponent jedynie wówczas, kiedy ustawiona jest pewna zmienna środowiskowa.

Do momentu wydania Springa w wersji 4. osiągnięcie tego poziomu konfiguracji warunkowej było dość skomplikowane, ale w Springu 4 wprowadzono nową adnotację, `@Conditional`, którą można stosować na metodach wraz z adnotacją `@Bean`. Jeśli warunek przypisany do adnotacji jest prawdziwy, komponent zostanie utworzony. W przeciwnym wypadku komponent jest ignorowany.

Przypuśćmy, że istnieje klasa o nazwie `MagicBean`, której instancja ma powstać tylko wtedy, gdy ustawiona jest zmienna środowiskowa `magic`. Jeżeli nie ma takiej zmiennej, komponent `MagicBean` ma zostać zignorowany. Na listingu 3.4 widzimy, jak skonfigurować komponent `MagicBean` za pomocą adnotacji `@Conditional`.

Listing 3.4. Warunkowa konfiguracja komponentu

```
@Bean
@Conditional(MagicExistsCondition.class) <-------- Warunkowe tworzenie komponentu
public MagicBean magicBean() {
   return new MagicBean();
}
```

Jak widać na listingu, do adnotacji `@Conditional` przekazana jest klasa opisująca warunek — w tym przypadku jest to `MagicExistsCondition`. Adnotacja powiązana jest z interfejsem `Condition`:

```
public interface Condition {
   boolean matches(ConditionContext ctxt,
      AnnotatedTypeMetadata metadata);
}
```

Klasa przekazana do adnotacji `@Conditional` może mieć dowolny typ implementujący interfejs `Condition`. Interfejs ten jest bardzo prosty i wymaga implementacji tylko jednej metody `matches()`. Komponent oznaczony adnotacją `@Conditional` zostanie utworzony wyłącznie wtedy, gdy metoda zwróci wartość `true`.

W naszym przypadku musimy utworzyć implementację klasy `Condition`, której działanie uzależnione jest od istnienia zmiennej środowiskowej `magic`. Listing 3.5 przedstawia klasę `MagicExistsCondition`, która implementuje interfejs `Condition` i rozwiązuje postawiony problem.

Listing 3.5. Sprawdzanie istnienia zmiennej środowiskowej magic

```
package com.habuma.restfun;

import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;
import org.springframework.util.ClassUtils;

public class MagicExistsCondition implements Condition {

   public boolean matches(
      ConditionContext context, AnnotatedTypeMetadata metadata) {
      Environment env = context.getEnvironment();
      return env.containsProperty("magic"); <-- Sprawdzanie właściwości "magic"
   }
}
```

Przedstawiona metoda `matches()` jest prosta, ale posiada duże możliwości. Wykorzystuje implementację `Environent`, pozyskaną przy użyciu interfejsu `ConditionContext`, i za jej pomocą sprawdza, czy zmienna środowiskowa `magic` została ustawiona. W podanym przykładzie wartość tej zmiennej jest nieważna, musi ona po prostu istnieć. Wtedy metoda `matches()` zwróci wartość `true`, co spowoduje utworzenie wszystkich komponentów oznaczonych adnotacją `@Conditional` o warunku zdefiniowanym przez klasę `MagicExistsCondition`.

Z drugiej strony, jeśli zmienna `magic` nie istnieje, warunek nie zostanie spełniony, metoda `matches()` zwróci wartość `false` i oznaczone komponenty nie zostaną utworzone.

W przypadku klasy `MagicExistsCondition` wykorzystywany jest tylko komponent `Environment` z kontekstu `ConditionContext`, ale warunek uwzględniać może dużo więcej elementów. Metoda `matches()` przyjmuje argumenty `ConditionContext` oraz `Annotated`
`↪TypeMetadata`, z których możemy korzystać przy ustalaniu warunku.

`ConditionContext` jest interfejsem, który w skrócie wygląda tak:

```
public interface ConditionContext {
   BeanDefinitionRegistry getRegistry();
   ConfigurableListableBeanFactory getBeanFactory();
```

```
   Environment getEnvironment();
   ResourceLoader getResourceLoader();
   ClassLoader getClassLoader();
}
```

Interfejs ten umożliwia:

- sprawdzenie definicji komponentów zwróconych za pomocą metody `getRegistry()` rejestru `BeanDefinitionRegistry`;
- sprawdzenie obecności komponentów oraz wgląd we właściwości komponentu za pomocą fabryki `ConfigurableListableBeanFactory` zwróconej przez metodę `getBeanFactory()`;
- sprawdzenie obecności i wartości zmiennych środowiskowych za pomocą obiektu typu `Environment` pozyskanego przy użyciu metody `getEnvironment()`;
- odczytanie i sprawdzenie zawartości zasobów wczytanych przez obiekt typu `ResourceLoader`, zwrócony za pomocą metody `getResourceLoader()`;
- wczytanie i sprawdzenie obecności klas z użyciem obiektu typu `ClassLoader` pobranego za pomocą metody `getClassLoader()`.

`AnnotatedTypeMetadata` oferuje możliwość sprawdzenia adnotacji dodanych w metodzie opatrzonej adnotacją `@Bean`. `AnnotatedTypeMediadata` podobnie jak `ConditionContext` jest interfejsem i wygląda następująco:

```
public interface AnnotatedTypeMetadata {
   boolean isAnnotated(String annotationType);
   Map<String, Object> getAnnotationAttributes(String annotationType);
   Map<String, Object> getAnnotationAttributes(
      String annotationType, boolean classValuesAsString);
   MultiValueMap<String, Object> getAllAnnotationAttributes(
      String annotationType);
   MultiValueMap<String, Object> getAllAnnotationAttributes(
      String annotationType, boolean classValuesAsString);
}
```

Metoda `isAnnotated()` pozwala sprawdzić, czy badana metoda opatrzona adnotacją `@Bean`, oznaczona jest adnotacją jakiegoś wybranego typu. Pozostałe metody umożliwiają sprawdzenie atrybutów wszystkich adnotacji zastosowanych do metody oznaczonej adnotacją `@Bean`.

Co ciekawe, począwszy od Springa 4, adnotacja `@Profile` została poddana refaktoringowi i w swej implementacji wykorzystuje adnotację `@Conditional` i interfejs `Condition`. Jako kolejny przykład zobaczmy więc implementację `@Profile` w Springu 4.

Wygląda ona tak:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Documented
@Conditional(ProfileCondition.class)
public @interface Profile {
   String[] value();
}
```

Adnotacja `@Profile` składa się z adnotacji `@Conditional`, która jako warunek przyjmuje klasę `ProfileConditional`. Klasa `ProfileCondition` implementuje interfejs `Condition` i podejmując decyzję o wyborze profilu, uwzględnia kilka różnych czynników pochodzących z interfejsów `ConditionContext` i `AnnotatedTypeMetadata` (listing 3.6).

Listing 3.6. Klasa ProfileCondition sprawdza, czy komponent odpowiada wybranemu profilowi

```
class ProfileCondition implements Condition {

public boolean matches(
   ConditionContext context, AnnotatedTypeMetadata metadata) {
      if (context.getEnvironment() != null) {
         MultiValueMap<String, Object> attrs =
            metadata.getAllAnnotationAttributes(Profile.class.getName());
         if (attrs != null) {
            for (Object value : attrs.get("value")) {
               if (context.getEnvironment()
                  .acceptsProfiles(((String[]) value))) {
                  return true;
               }
            }
            return false;
         }
      }
      return true;
   }
}
```

Jak widzimy, klasa `ProfileCondition` pobiera mapę wszystkich atrybutów adnotacji `@Profile`, wykorzystując w tym celu interfejs `AnnotatedTypeMetadata`. Następnie sprawdza w sposób jawny wartość atrybutu `value`, zawierającego nazwę profilu komponentu. Na koniec pobiera środowisko `Environment`, korzystając z interfejsu `ConditionContext`, aby sprawdzić, czy profil jest aktywny (za pomocą metody `acceptsProfiles()`).

3.3. Radzimy sobie z niejednoznacznościami w autowiązaniach

W rozdziale 2. nauczyłeś się korzystać z mechanizmu autowiązań, aby zlecić Springowi wykonanie za Ciebie całej pracy związanej z wstrzykiwaniem zależności komponentów przez konstruktor lub właściwości. Autowiązanie redukuje znacznie nakład pracy związany z przygotowywaniem jawnej konfiguracji do spięcia wszystkich komponentów aplikacji ze sobą, co stanowi wielką pomoc w wykonywaniu tych zadań.

Autowiązanie działa jednak tylko wtedy, gdy do wiązania pasuje dokładnie jeden komponent. Jeśli pasujących komponentów jest więcej, powstała niejednoznaczność uniemożliwia Springowi przeprowadzenie autowiązania właściwości, konstruktora albo parametrów metody.

Żeby zilustrować sytuację niejednoznaczności autowiązania, załóżmy, że adnotacją `@Autowired` opatrzona jest następująca metoda `setDessert()`:

```
@Autowired
public void setDessert(Dessert dessert) {
this.dessert = dessert;
}
```

W naszym przykładzie `Dessert` (deser) jest interfejsem, a implementują go trzy klasy — `Cake` (ciastko), `Cookies` (ciasteczka) oraz `IceCream` (lody):

```
@Component
public class Cake implements Dessert { ... }
@Component
public class Cookies implements Dessert { ... }
@Component
public class IceCream implements Dessert { ... }
```

Wszystkie trzy implementacje są opatrzone adnotacją `@Component`, więc zostaną wyszukane na etapie skanowania komponentów i utworzone jako komponenty w kontekście aplikacji Springa. Następnie Spring próbuje powiązać parametr typu `Dessert` w metodzie `setDessert()`. Nie ma tu jednak jednoznacznego wyboru. Większość z nas nie miałaby pewnie problemu z wybraniem deseru z listy, Spring jednak tego nie potrafi. Nie ma innej możliwości, jak poddać się i rzucić wyjątkiem. Tym wyjątkiem jest `NoUniqueBeanDefinitionException`:

```
nested exception is
org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [com.desserteater.Dessert] is defined:
expected single matching bean but found 3: cake,cookies,iceCream
```

Podany przykład z deserem jest oczywiście zmyślony i służy tylko zilustrowaniu problemu, który się pojawia przy niejednoznaczności autowiązania. W praktyce problem ten jest spotykany dużo rzadziej, niż mogłoby się nam wydawać. Mimo że jest on realny, zdecydowanie częściej mamy tylko jedną implementację danego interfejsu i autowiązanie działa doskonale.

Jeśli jednak nie uda się nam uniknąć niejednoznaczności, Spring oferuje kilka opcji rozwiązania tego problemu. Możemy ustawić jedną z implementacji jako główny wybór lub wykorzystać kwantyfikatory, aby ograniczyć możliwość wyboru do pojedynczego kandydata.

3.3.1. Wybieramy główny komponent

Może tak jak ja lubisz wszystkie rodzaje deserów? Ciasto… ciasteczka… lody… wszystko jest takie pyszne. Ale co, jeżeli staniesz przed koniecznością wyboru tylko jednego deseru? Który z nich lubisz najbardziej?

Gdy deklarujemy komponenty, możemy uniknąć niejednoznaczności autowiązania, wybierając jeden z komponentów jako główny. Wtedy, gdy pojawi się niejednoznaczność, wybrany zostanie właśnie ten komponent. Moglibyśmy go nazwać „ulubionym" komponentem.

Przyjmijmy, że Twoim ulubionym deserem są lody. W Springu ulubiony wybór oznaczamy adnotacją `@Primary`. Adnotację `@Primary` możemy zastosować razem z adnotacją `@Component` dla komponentów wyszukiwanych w ramach skanowania lub razem

z adnotacją `@Bean` dla komponentów zadeklarowanych w konfiguracji Java. Poniżej pokazano przykład deklaracji komponentu `IceCream` jako komponentu głównego:

```
@Component
@Primary
public class IceCream implements Dessert { ... }
```

Jeśli komponent `IceCream` byłby zadeklarowany w sposób jawny w konfiguracji Java, metoda oznaczona adnotacją `@Bean` mogłaby wyglądać następująco:

```
@Bean
@Primary
public Dessert iceCream() {
return new IceCream();
}
```

Jeżeli korzystasz z deklaracji w pliku XML, również masz taką możliwość. Element `<bean>` posiada atrybut `primary`, służący do określenia komponentów głównych:

```
<bean id="iceCream"
class="com.desserteater.IceCream"
primary="true" />
```

Nieważne, w jaki sposób wybierzemy komponent główny — efekt będzie taki sam. Podpowiadamy Springowi, który komponent ma wybrać, jeśli pojawi się niejednoznaczność.

Ta metoda działa doskonale do momentu, gdy wybierzemy więcej niż jeden komponent główny. Na przykład niech klasa `Cake` wygląda następująco:

```
@Component
@Primary
public class Cake implements Dessert { ... }
```

Mamy teraz dwa główne komponenty typu `Dessert`: `Cake` oraz `IceCream`. Powoduje to ponownie problemy z niejednoznacznością wyboru. Wcześniej Spring nie potrafił wybrać kandydata spośród wielu komponentów, teraz również nie umie wybrać spośród wielu kandydatów głównych. Kiedy ustalimy kilku kandydatów głównych, żaden z nich nim nie zostanie.

Musimy skorzystać z bardziej wyrafinowanych mechanizmów eliminowania niejednoznaczności. Nadszedł czas na zapoznanie się z kwalifikatorami.

3.3.2. Kwalifikujemy autowiązane komponenty

Ograniczeniem komponentów głównych jest to, że adnotacja `@Primary` pozwala na dokonanie wielokrotnego i niejednoznacznego wyboru. Jej zadanie polega tylko na wskazaniu preferowanej opcji. Po wyborze więcej niż jednego komponentu głównego nie mamy zbytniej możliwości dalszego zawężenia możliwości wyboru.

Kwalifikatory Springa pozwalają na zawężenie wyboru spośród wszystkich kandydatów, aby w końcowym rozrachunku można było uzyskać pojedynczy komponent spełniający określone warunki. Jeżeli po zastosowaniu wszystkich kwalifikatorów niejednoznaczność w dalszym ciągu występuje, możemy użyć dodatkowych kwalifikatorów.

Głównym sposobem pracy z kwalifikatorami jest wykorzystanie adnotacji `@Qualifier`. Można ją stosować razem z adnotacjami `@Autowired` i `@Inject` w miejscu wstrzykiwania komponentów w celu wskazania pożądanego wyboru. Załóżmy, że chcemy, by do metody `setDessert()` wstrzyknięty został komponent `IceCream`:

```
@Autowired
@Qualifier("iceCream")
public void setDessert(Dessert dessert) {
   this.dessert = dessert;
}
```

Jest to najprostszy przykład kwalifikatora. Parametrem przekazanym do adnotacji `@Qualifier` jest identyfikator komponentu, który chcemy wstrzyknąć. Wszystkie klasy oznaczone adnotacją `@Component` mają nadane identyfikatory, a ich nazwą jest nazwa klasy z pierwszą literą zamienioną na małą.

Zatem kwalifikator `@Qualifier("iceCream")` odnosi się do komponentu typu `IceCream` utworzonego w wyniku skanowania komponentów.

W rzeczywistości jest to trochę bardziej złożone. Mówiąc konkretniej, `@Qualifier ↪("iceCream")` odwołuje się do komponentu, którego kwalifikatorem jest ciąg znaków `"iceCream"`. Wszystkim komponentom, którym nie nadano żadnych kwalifikatorów, nadane są kwalifikatory domyślne, odpowiadające nazwą ich identyfikatorom. W ten sposób do metody `setDessert()` wstrzyknięty zostanie komponent o identyfikatorze `"ice ↪Cream"`. Jest nim komponent o identyfikatorze `iceCream` utworzony w wyniku odnalezienia klasy `IceCream` na etapie skanowania komponentów.

Opieranie się na kwalifikacji z użyciem domyślnego identyfikatora komponentu jest proste, ale może prowadzić do kilku problemów. Co by się stało po zmianie nazwy klasy `IceCream` na `Gelato`, czyli gdybyśmy zmienili nasz deser z lodów na galaretkę? W tym przypadku zarówno identyfikator klasy, jak i domyślny kwalifikator zamienią się w `gelato`, który nie odpowiada kwalifikatorowi użytemu w metodzie `setDessert()`. Zatem proces autowiązania się nie powiedzie.

Problem polega na tym, że kwalifikator przypisany do metody `setDessert()` jest ściśle powiązany z nazwą klasy wstrzykiwanego komponentu. Dowolna zmiana nazwy tej klasy sprawi, że kwalifikator przestanie działać.

TWORZENIE WŁASNYCH KWALIFIKATORÓW

W kwalifikatorach nie musimy wcale wykorzystywać identyfikatorów komponentów, możemy zastosować własne kwalifikatory. Wystarczy, że umieścimy adnotację `@Qualifier` w deklaracji komponentu. Przykładowo możemy ją umieścić obok adnotacji `@Component`:

```
@Component
@Qualifier("cold")
public class IceCream implements Dessert { ... }
```

W tym przykładzie do komponentu przypisujemy kwalifikator `cold` (zimne). Kwalifikator ten nie jest w żaden sposób powiązany z nazwą klasy, możemy ją zatem w dowolny sposób zmieniać, nie uszkadzając autowiązań. Dopóki korzystamy z kwalifikatora `cold` w miejscu wstrzykiwania, wszystko będzie w dalszym ciągu działało bez zarzutu:

```
@Autowired
@Qualifier("cold")
public void setDessert(Dessert dessert) {
this.dessert = dessert;
}
```

Warto zauważyć, że adnotacji `@Qualifier` możemy użyć wraz z adnotacją `@Bean` w jawnej definicji komponentów w konfiguracji JavaConfig:

```
@Bean
@Qualifier("cold")
public Dessert iceCream() {
   return new IceCream();
}
```

W definicji własnych wartości `@Qualifier` dobrą praktyką jest wykorzystanie jakiejś cechy lub deskryptywnego opisu komponentu, a nie przypadkowej nazwy. W naszym przykładzie opisaliśmy komponent `IceCream` z użyciem pojęcia `cold`. Informację tę możemy odczytać jako zwrot typu „Podaj mi zimny deser", co dobrze opisuje wstrzykiwanie komponentu `IceCream`. W podobnym stylu możemy określić kwalifikatory komponentów `Cake` jako `soft` (miękkie), a `Cookies` jako `crispy` (chrupiące).

DEFINIOWANIE WŁASNYCH ADNOTACJI KWALIFIKATORÓW

Kwalifikatory oparte na nazwach cech są lepsze od tych bazujących na identyfikatorach komponentów. Jeśli jednak jedną cechę dzieli wiele komponentów, możemy ponownie napotkać wcześniejsze problemy. Przypuśćmy, że utworzyliśmy nowy komponent, mrożony lizak:

```
@Component
@Qualifier("cold")
public class Popsicle implements Dessert { ... }
```

O nie! Teraz mamy dwa „zimne" desery. Powrócił problem z niejednoznacznością autowiązania komponentów. Potrzebne nam są dodatkowe kwalifikatory, dzięki którym zawęzimy selekcję do pojedynczego komponentu.

Rozwiązaniem mogłoby być dodanie kolejnej adnotacji `@Qualifier` w miejscu wstrzykiwania zależności i w definicji komponentu. Lody są przecież nie tylko zimne, ale też kremowe (`creamy`). Klasa `IceCream` wyglądałaby tak:

```
@Component
@Qualifier("cold")
@Qualifier("creamy")
public class IceCream implements Dessert { ... }
```

W klasie `Popsicle` dodalibyśmy również kolejną adnotację `@Qualifier`, aby zaznaczyć, że mrożone lizaki są owocowe (`fruity`):

```
@Component
@Qualifier("cold")
@Qualifier("fruity")
public class Popsicle implements Dessert { ... }
```

Następnie w punkcie wstrzykiwania ograniczylibyśmy wybór komponentu jedynie do `IceCream`:

```
@Autowired
@Qualifier("cold")
@Qualifier("creamy")
public void setDessert(Dessert dessert) {
   this.dessert = dessert;
}
```

Jest tylko jeden mały problem. Język Java nie pozwala na użycie więcej niż jednej adnotacji tego samego typu na pojedynczym elemencie[1]. Próba uruchomienia powyższego kodu zakończy się błędem kompilacji. Nie mamy możliwości ograniczenia listy kandydatów do pojedynczego wyboru za pomocą adnotacji `@Qualifier` (przynajmniej nie bezpośrednio).

Tym, co możemy w tej sytuacji zrobić, jest utworzenie własnej adnotacji kwalifikatora, reprezentującej cechę, która ma być wykorzystana przy kwalifikacji komponentów. W tym celu musimy stworzyć adnotację opatrzoną adnotacją `@Qualifier`, a następnie w miejscu adnotacji `@Qualifier("cold")` użyć własnej adnotacji `@Cold`:

```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,
   ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Cold { }
```

W podobny sposób możesz utworzyć nową adnotację `@Creamy` jako zamiennik dla adnotacji `@Qualifier("creamy")`:

```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,
   ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Creamy { }
```

Analogicznie tworzymy adnotacje dla pozostałych cech: `@Soft`, `@Crispy` i `@Fruity`, i stosujemy je w miejscach, w których wcześniej znajdowała się adnotacja `@Qualifier`. Adnotacje oznaczone adnotacją `@Qualifier` przejmują jej cechy. W praktyce same stają się prawdziwymi adnotacjami kwalifikacji.

Spójrzmy teraz ponownie na klasę `IceCream` i zastosujmy na niej adnotacje `@Cold` i `@Creamy`:

```
@Component
@Cold
@Creamy
public class IceCream implements Dessert { ... }
```

Klasę `Popsicle` opatrujemy adnotacjami `@Cold` i `@Fruity`:

```
@Component
@Cold
@Fruity
public class Popsicle implements Dessert { ... }
```

[1] Java 8 daje możliwość wielokrotnego użycia tych samych adnotacji, ale tylko wtedy, gdy one same opatrzone są adnotacją `@Repeatable`. Springowa adnotacja `@Qualifier` nie jest jednak opatrzona tą adnotacją.

Ostatnim krokiem jest wybór takiej kombinacji adnotacji kwalifikatora w punkcie wstrzykiwania, który doprowadzi do wskazania tego jednego pożądanego komponentu. Przykładowo aby wskazać komponent `IceCream`, oznaczamy adnotacjami metodę `set➥Dessert()` w poniższy sposób:

```
@Autowired
@Cold
@Creamy
public void setDessert(Dessert dessert) {
   this.dessert = dessert;
}
```

Definiowanie własnych adnotacji umożliwia równoczesne zastosowanie wielu kwalifikatorów bez ograniczeń i błędów ze strony kompilatora Javy. Użycie własnych adnotacji jest też bezpieczniejsze pod względem typów niż użycie podstawowej wersji adnotacji `@Qualifier` i kwalifikatora w postaci ciągu znaków.

Przyjrzyjmy się bliżej metodzie `setDessert()` i adnotacjom, które ją opisują. W żadnym miejscu nie mówimy bezpośrednio, że chcemy do tej metody dowiązać komponent `IceCream`. Do wskazania pożądanego komponentu wykorzystujemy opisujące go cechy. Lody są chłodne i kremowe, co możemy opisać za pomocą adnotacji `@Cold` i `@Creamy`. W ten sposób metoda `setDessert()` nie jest bezpośrednio powiązana z żadną konkretną implementacją interfejsu `Dessert`. W tym konkretnym przypadku odpowiedni byłby dowolny komponent posiadający obie wskazane cechy. Tak się akurat składa, że spośród dostępnych implementacji deserów jedynie lody spełniają wszystkie wymagania.

W tej oraz w poprzedniej sekcji poznałeś kilka sposobów rozszerzania Springa o własne adnotacje. Własną adnotację warunkową możemy przygotować poprzez utworzenie nowej adnotacji i oznaczenie jej adnotacją `@Conditional`. Aby przygotować własny kwalifikator, tworzymy nową adnotację i oznaczamy ją adnotacją `@Qualifier`. Podobnie możemy wykorzystać inne adnotacje springowe i skonstruować z nich własną adnotację o specjalnym przeznaczeniu.

Teraz przyjrzyjmy się deklaracji komponentów w różnych zasięgach.

3.4. Ustalamy zasięg komponentów

Domyślnie wszystkie komponenty tworzone w Springu są singletonami. Oznacza to, że niezależnie od tego, ile razy dany komponent jest wstrzykiwany do innych komponentów, zawsze wykorzystywana jest ta sama jego instancja.

W większości przypadków jest to najlepsze rozwiązanie. Koszt tworzenia i usuwania z pamięci instancji obiektów, których celem jest wykonanie prostych zadań, staje się nieuzasadniony, jeśli obiekty te nie przechowują żadnego stanu i mogą być wielokrotnie wykorzystane w ramach aplikacji.

Czasem się jednak zdarza, że musimy pracować z klasą mutowalną, która przechowuje jakiś stan. Wtedy jej ponowne użycie nie jest już bezpieczne. W takim przypadku zadeklarowanie klasy w postaci singletonu prawdopodobnie nie jest najlepszym pomysłem, bo obiekt ten może zawierać niepożądane dane i przy wielokrotnym wykorzystaniu może doprowadzić do powstania nieoczekiwanych problemów.

Spring udostępnia kilka zasięgów deklaracji komponentów, między innymi:

- *Singleton* — jedna instancja komponentu tworzona dla całej aplikacji;
- *Prototype* (prototyp) — jedna instancja komponentu tworzona za każdym razem, gdy komponent jest wstrzykiwany lub pobierany z kontekstu aplikacji Springa;
- *Session* (sesja) — w aplikacji internetowej jedna instancja obiektu utworzona dla każdej sesji;
- *Request* (żądanie) — w aplikacji internetowej jedna instancja obiektu utworzona dla każdego żądania.

Domyślnym zasięgiem jest singleton, ale jak już wspominałem, nie jest to idealne rozwiązanie w przypadku typów mutowalnych. Wybór alternatywnego zasięgu jest możliwy dzięki użyciu adnotacji `@Scope`, zastosowanej w połączeniu z adnotacjami `@Component` lub `@Bean`.

Jeśli korzystasz ze skanowania komponentów do wyszukiwania i deklarowania komponentów, możesz zmienić zasięg komponentu na prototyp za pomocą odpowiedniej adnotacji `@Scope`:

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class Notepad { ... }
```

Wykorzystaliśmy tutaj stałą `SCOPE_PROTOTYPE` klasy `ConfigurableBeanFactory`. Moglibyśmy też skorzystać z adnotacji `@Scope("prototype")`, ale użycie stałej jest bezpieczniejsze i mniej podatne na błędy.

W przypadku konfiguracji w języku Java odpowiednikiem powyższej deklaracji komponentu `Notepad` jest użycie adnotacji `@Scope` w połączeniu z adnotacją `@Bean`:

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public Notepad notepad() {
   return new Notepad();
}
```

W konfiguracji XML użylibyśmy atrybutu `scope` elementu `<bean>`:

```
<bean id="notepad"
   class="com.myapp.Notepad"
   scope="prototype" />
```

Niezależnie od sposobu deklaracji, za każdym razem, gdy następuje wstrzyknięcie komponentu `Notepad` do innego komponentu lub gdy jest on pobierany z kontekstu aplikacji Springa, tworzona jest jego nowa instancja. W rezultacie każdy otrzymuje swoją własną instancję notatnika.

3.4.1. Zasięg żądania oraz sesji

W aplikacji internetowej przydatne może być utworzenie komponentu współdzielonego w zasięgu danego żądania bądź sesji. Przykładowo w typowej aplikacji e-commerce może istnieć komponent reprezentujący koszyk zakupowy użytkownika. Jeśli utworzylibyśmy koszyk w postaci singletonu, wszyscy użytkownicy dodawaliby przedmioty do

tego samego koszyka. Z drugiej strony, jeżeli koszyk zostałby zadeklarowany w postaci prototypu, to produkty umieszczone w koszyku w jednym obszarze aplikacji nie byłyby dostępne w innej, w której wstrzyknięta zostałaby inny instancja koszyka.

W przypadku komponentu koszyka zakupowego najrozsądniejszym rozwiązaniem wydaje się wykorzystanie zasięgu sesji, gdyż jest on w największym stopniu powiązany z danym użytkownikiem. Aby umieścić komponent w zasięgu sesji, stosujemy adnotację `@Scope` bardzo podobnie jak przy zasięgu prototypu:

```
@Component
@Scope(
   value=WebApplicationContext.SCOPE_SESSION,
   proxyMode=ScopedProxyMode.INTERFACES)
public ShoppingCart cart() { ... }
```

Wartość atrybutu `value` ustawiamy tutaj na stałą `SCOPE_SESSION` (o wartości `session`) klasy `WebApplicationContext`. Dzięki temu ustawieniu Spring wie, że powinien utworzyć osobną instancję komponentu `ShoppingCart` dla każdej sesji w aplikacji internetowej. Pojawi się wiele instancji komponentu `ShoppingCart`, ale w ramach danej sesji powstanie tylko jedna instancja. W ramach jednej sesji utworzony komponent będzie więc singletonem.

Warto zauważyć, że adnotacja `@Scope` posiada atrybut `proxyMode`, który w naszym przykładzie ma wartość `ScopedProxyMode.INTERFACES`. Użycie tego atrybutu pozwala rozwiązać problem powstały po wstrzyknięciu komponentu działającego w zasięgu sesji lub żądania do singletona. Zanim jednak zajmę się opisem atrybutu `proxyMode`, spójrzmy na przykładowy scenariusz ilustrujący problem.

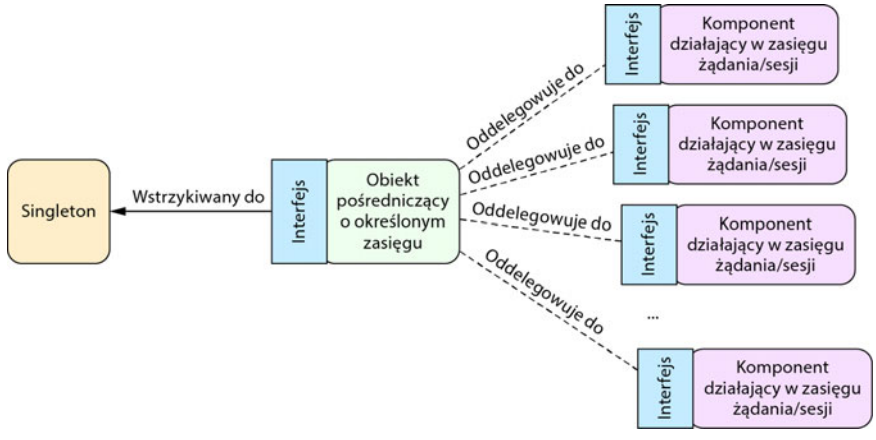
Przypuśćmy, że chcemy wstrzyknąć komponent `ShoppingCart` do metody typu setter w komponencie `StoreService`, który jest singletonem:

```
@Component
public class StoreService {

   @Autowired
   public void setShoppingCart(ShoppingCart shoppingCart) {
      this.shoppingCart = shoppingCart;
   }
   ...
}
```

Ponieważ komponent `StoreService` jest singletonem, zostanie utworzony przez Springa w trakcie wczytywania kontekstu aplikacji. Po jego utworzeniu Spring podejmie próbę wstrzyknięcia koszyka `ShoppingCart` do metody `setShoppingCart()`. Ale komponent `ShoppingCart` jest zdefiniowany w zasięgu sesji, która w tym momencie jeszcze nie istnieje. Jego instancja zostanie utworzona dopiero po wejściu użytkownika na stronę i wygenerowaniu sesji.

Co więcej, powstanie wiele instancji koszyka, po jednej dla każdego użytkownika. Nie chcemy, aby Spring wstrzyknął jakąś dowolną pojedynczą instancję komponentu `ShoppingCart` do singletona `StoreService`. Chcemy, by `StoreService` pracował z instancją `ShoppingCart` utworzoną dla sesji obowiązującej w aktualnie przetwarzanym żądaniu.

W związku z tym Spring nie wstrzykuje do singletona `StoreService` właściwego komponentu `ShoppingCart`, ale obiekt pośredniczący (proxy), co przedstawia listing 3.2. Obiekt pośredniczący udostępnia te same metody co komponent `ShoppingCart`, klasa `StoreService` może go więc traktować jak zwyczajny obiekt koszyka. Wywołanie tych metod powoduje opóźnione (lazy) rozwiązanie właściwego adresata metod i oddelegowanie wywołania do komponentu `ShoppingCart`. Spójrzmy na rysunek 3.1.



Rysunek 3.1. Obiekty pośredniczące o określonym zasięgu umożliwiają opóźnione wstrzykiwanie komponentów działających w zasięgu żądania oraz sesji

Spróbujmy teraz zrozumieć działanie obiektu pośredniczącego o określonym zasięgu (scoped proxy) i przyjrzyjmy się roli atrybutu `proxyMode`. Zgodnie z konfiguracją z wcześniejszego przykładu atrybut `proxyMode` ma wartość `ScopedProxyMode.INTERFACES`, co oznacza, że obiekt pośredniczący implementuje interfejs `ShoppingCart` i oddelegowuje wywołania metod do odpowiedniego komponentu.

Jest to bardzo dobre rozwiązanie (i idealny model pracy obiektu pośredniczącego) tak długo, jak długo `ShoppingCart` jest interfejsem, a nie klasą. Jeśli jednak `ShoppingCart` jest klasą, Spring nie będzie mógł utworzyć obiektu pośredniczącego opartego na interfejsie. W tym celu do wygenerowania obiektu pośredniczącego opartego na klasie musi użyć biblioteki CGLib. My musimy wtedy zmienić wartość ustawienia `proxyMode` na `ScopedProxyMode.TARGET_CLASS`, co oznacza, że obiekt pośredniczący ma zostać wygenerowany jako rozszerzenie klasy docelowej.

Musisz wiedzieć, że chociaż w tym przykładzie omawiam zasięg sesji, te same wyzwania pojawiają się przy wiązaniu komponentów o zasięgu żądania. One również muszą być wstrzykiwane jako obiekty pośredniczące o określonym zasięgu.

3.4.2. Deklarujemy obiekty pośredniczące o określonym zasięgu za pomocą XML

Jeśli do deklaracji komponentów w zasięgu sesji lub żądania wykorzystujemy konfigurację XML, nie mamy oczywiście możliwości skorzystania z adnotacji `@Scope` i atrybutu `proxyMode`. Atrybut `scope` elementu `<bean>` umożliwia ustawienie zasięgu komponentu, jak jednak możemy ustawić tryb obiektu pośredniczącego?

Umożliwia nam to nowy element, pochodzący z przestrzeni nazw `aop` Springa:

```
<bean id="cart"
   class="com.myapp.ShoppingCart"
   scope="session">
   <aop:scoped-proxy />
</bean>
```

Element `<aop:scoped-proxy />` jest odpowiednikiem atrybutu `proxyMode` adnotacji `@Scope` w konfiguracji XML w Springu. Jest to informacja dla Springa, aby ten utworzył dla komponentu obiekt pośredniczący o określonym zasięgu. Ustawienia domyślne do stworzenia obiektu pośredniczącego klasy docelowej wykorzystują bibliotekę CGLib. Możemy to zmienić za pomocą atrybutu `proxy-target-class`, który po ustawieniu wartości `false` umożliwia wygenerowanie obiektu pośredniczącego opartego na interfejsie:

```
<bean id="cart"
   class="com.myapp.ShoppingCart"
   scope="session">
   <aop:scoped-proxy proxy-target-class="false" />
</bean>
```

Żeby użyć elementu `<aop:scoped-proxy>`, musimy najpierw zadeklarować przestrzeń nazw `aop` w konfiguracji XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xsi:schemaLocation="
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
      ...
</beans>
```

Więcej na temat przestrzeni nazw `aop` w Springu dowiesz się w rozdziale 4., gdy zajmiemy się programowaniem aspektowym w Springu. Bieżący rozdział zakończę, przedstawiając jedną z najbardziej zaawansowanych opcji autowiązania w Springu: język wyrażeń Springa (SpEL).

3.5. Wstrzykujemy wartości w czasie wykonywania

Gdy rozmawiamy na temat wstrzykiwania zależności i wiązania, myślimy z reguły o wiązaniu referencji komponentu do właściwości lub argumentu konstruktora innego komponentu. Jest to najczęściej wiązanie ze sobą dwóch obiektów.

Wiązaniem komponentów może być też wiązanie zwykłej wartości do właściwości komponentu bądź argumentu konstruktora. Kilka różnych przykładów wiązania wartości omówiłem w rozdziale 2., na przykład wiązanie nazwy albumu do konstruktora albo właściwości tytułu komponentu `BlankDisc`. Wiązanie w klasie `BlankDisc` mogło wyglądać następująco:

```
@Bean
public CompactDisc sgtPeppers() {
   return new BlankDisc(
      "Sgt. Pepper's Lonely Hearts Club Band",
      "The Beatles");
}
```

Wprawdzie osiągnęliśmy zamierzony cel, ustawiając tytuł i nazwę artysty dla komponentu `BlankDisc`, ale zrobiliśmy to za pomocą wartości ustawionych na sztywno w klasie konfiguracji. W przypadku konfiguracji XML sytuacja byłaby dokładnie taka sama:

```
<bean id="sgtPeppers"
   class="soundsystem.BlankDisc"
   c:_title="Sgt. Pepper's Lonely Hearts Club Band"
   c:_artist="The Beatles" />
```

W niektórych przypadkach ustawianie wartości na sztywno może być dobrym rozwiązaniem. Z reguły chcemy jednak uniknąć zaszywania takich wartości w konfiguracji i umożliwić ich pobranie po uruchomieniu aplikacji. Spring oferuje dwie możliwości pobrania wartości w trakcie wywołania aplikacji:

- symbole zastępcze właściwości;
- język wyrażeń Springa (ang. *Spring Expression Language* — SpEL).

Za chwilę przekonasz się, że zastosowanie obu technik jest bardzo podobne, choć różni je zarówno cel powstania, jak i zachowanie. Na początek spojrzymy na prostszą technikę, symbole zastępcze właściwości, a następnie przejdziemy do języka SpEL, który oferuje dużo większe możliwości.

3.5.1. Wstrzykujemy zewnętrzne wartości

Najprostszym sposobem rozwiązania zewnętrznych wartości w Springu jest zadeklarowanie źródła właściwości i ich pobranie za pośrednictwem interfejsu `Environment`. Na listingu 3.7 widzimy podstawową klasę konfiguracji Springa, która zależności komponentu `BlankDisc` wypełnia za pomocą zewnętrznych właściwości.

Listing 3.7. Użycie adnotacji @PropertySource i interfejsu Environment

```
package com.soundsystem;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;

@Configuration
@PropertySource("classpath:/com/soundsystem/app.properties")
public class ExpressiveConfig {

   @Autowired
   Environment env;

   @Bean
```

Deklaracja źródła właściwości

```
   public BlankDisc disc() {
      return new BlankDisc(
         env.getProperty("disc.title"),
         env.getProperty("disc.artist"));
   }
}
```

Pobieranie wartości właściwości

W powyższym przykładzie adnotacja `@PropertySource` odwołuje się do pliku o nazwie `app.properties` w ścieżce klas. Jego zawartość może wyglądać następująco:

```
disc.title=Sgt. Peppers Lonely Hearts Club Band
disc.artist=The Beatles
```

Właściwości z tego pliku wczytywane są przez środowisko Springa `Environment`, z którego mogą być później pobrane. W międzyczasie w metodzie `disc()` tworzymy nowy obiekt `BlankDisc`; argumentami przekazywanymi do konstruktora są wartości pobrane ze środowiska za pomocą metody `getProperty()`.

WIĘCEJ INFORMACJI O ŚRODOWISKU SPRINGA

Skoro mówimy o interfejsie `Environment`, warto wiedzieć, że metoda `getProperty()`, którą wykorzystaliśmy na listingu 3.7, nie jest jedyną dostępną metodą umożliwiającą pobieranie wartości właściwości. Metoda `getProperty()` jest przeciążona i dostępna w czterech wariantach:

- `String getProperty(String key);`
- `String getProperty(String key, String defaultValue);`
- `T getProperty(String key, Class<T> type);`
- `T getProperty(String key, Class<T> type, T defaultValue).`

Dwie pierwsze wersje metody zwracają zawsze wartość typu String. Pierwsza wersja to metoda, którą poznałeś na listingu 3.7. Możemy jednak lekko zmodyfikować metodę tworzącą komponent, aby zwracała ustalone przez nas domyślne wartości, jeśli podane właściwości nie zostały zdefiniowane:

```
@Bean
public BlankDisc disc() {
   return new BlankDisc(
      env.getProperty("disc.title", "Rattle and Hum"),
      env.getProperty("disc.artist", "U2"));
}
```

Kolejne dwie wersje metody `getProperty()` działają podobnie jak dwie pierwsze, ale umożliwiają zwrócenie właściwości innego typu niż `String`. Przypuśćmy, że pobieramy wartość reprezentującą liczbę utrzymywanych połączeń w puli. Po pobraniu wartości w postaci ciągu znaków musielibyśmy dokonać konwersji do typu `Integer`, zanim moglibyśmy z niej skorzystać. Dzięki wykorzystaniu jednej z przeciążonych wersji metody `getProperty()` konwersja ta jest przeprowadzana automatycznie:

```
int connectionCount =
   env.getProperty("db.connection.count", Integer.class, 30);
```

Interfejs Environment oferuje kilka dodatkowych metod związanych z obsługą właściwości. Użycie którejkolwiek z metod getProperty() bez określenia wartości domyślnej skutkuje zwróceniem wartości null, jeśli właściwość ta nie została zdefiniowana. Żeby wymusić zdefiniowanie właściwości, możemy zastosować metodę getRequiredProperty():

```
@Bean
public BlankDisc disc() {
   return new BlankDisc(
      env.getRequiredProperty("disc.title"),
      env.getRequiredProperty("disc.artist"));
}
```

W powyższym przykładzie, jeśli któraś z wartości właściwości disc.title lub disc.artist nie zostanie zdefiniowana, wyrzucony zostanie wyjątek IllegalStateException.

Jeżeli chcemy tylko sprawdzić obecność właściwości, możemy wykorzystać metodę containsProperty() interfejsu Environment:

```
boolean titleExists = env.containsProperty("disc.title");
```

Mamy też możliwość przekształcenia właściwości w klasę wybranego typu. Służy do tego metoda getPropertyAsClass():

```
Class<CompactDisc> cdClass =
   env.getPropertyAsClass("disc.class", CompactDisc.class);
```

Odchodząc trochę od tematu właściwości — interfejs Enviroment pozwala też sprawdzić, które profile są aktywne:

- String[] getActiveProfiles() — zwraca tablicę nazw aktywnych profili.
- String[] getDefaultProfiles() — zwraca tablicę nazw domyślnych profili.
- boolean acceptsProfiles(String... profiles) — zwraca true, jeśli środowisko wspiera wybrany profil lub profile.

Działanie metody acceptsProfiles() poznałeś na listingu 3.6. W tamtym przykładzie z kontekstu ConditionContext pobraliśmy środowisko Environment i wykorzystaliśmy metodę acceptsProfiles() do sprawdzenia, czy profil przypisany do wybranego komponentu jest aktywny i czy komponent ma zostać utworzony. Metod interfejsu Enviro
↪nment operujących na profilach nie stosuje się może zbyt często, ale warto wiedzieć, że istnieje taka możliwość.

Pobieranie właściwości bezpośrednio za pomocą środowiska Environment jest pomocne, zwłaszcza wtedy, gdy do wiązania komponentów używamy konfiguracji JavaConfig. Jednak Spring oferuje też opcję wiązania właściwości z wykorzystaniem symboli zastępczych, rozwiązywanych za pomocą źródła właściwości.

ROZWIĄZUJEMY SYMBOLE ZASTĘPCZE WŁAŚCIWOŚCI

Spring od samego początku swego istnienia wspiera opcję wynoszenia właściwości do zewnętrznych plików i przekazywania ich wartości do komponentów z użyciem symboli zastępczych. W wiązaniach w Springu wartościami symboli zastępczych są nazwy właściwości otoczone znakami ${ … }. Przykładowo rozwiązywanie argumentów konstruktora klasy BlankDisc zadeklarowanej w pliku XML może wyglądać następująco:

```
<bean id="sgtPeppers"
   class="soundsystem.BlankDisc"
   c:_title="${disc.title}"
   c:_artist="${disc.artist}" />
```

Argument konstruktora o nazwie `title` przyjmuje wartość właściwości `disc.title`. Argument `artist` powiązany jest z wartością właściwości `disc.artist`. W ten sposób konfiguracja XML nie przechowuje żadnych wartości ustawionych na sztywno. Zamiast tego wartości rozwiązywane są z zewnętrznego źródła, pochodzącego spoza pliku konfiguracji. (Za chwilę powiem, jak następuje rozwiązywanie tych właściwości).

Jeśli przy tworzeniu i inicjowaniu komponentów aplikacji polegamy na skanowaniu komponentów oraz mechanizmie autowiązania, nie istnieje plik ani klasa konfiguracji, w których można określić symbole zastępcze. Zamiast tego wykorzystujemy adnotację `@Value` w sposób zbliżony do tego, w jaki korzystamy z adnotacji `@Autowired`. Przykładowy konstruktor w klasie `BlankDisc` może wyglądać tak:

```
public BlankDisc(
   @Value("${disc.title}") String title,
   @Value("${disc.artist}") String artist) {
   this.title = title;
   this.artist = artist;
}
```

Aby korzystać z wartości symboli zastępczych, musimy skonfigurować jeden z komponentów `PropertyPlaceholderConfigurer` lub `PropertySourcesPlaceholderConfigurer`. Począwszy od Springa 3.1, preferowanym wyborem jest `PropertySourcesPlaceholder`
`➥Configurer`, gdyż do rozwiązywania symboli zastępczych wykorzystuje interfejs `Environment` i zbiór jego źródeł właściwości.

Poniższa metoda pozwala skonfigurować komponent `PropertySourcesPlaceholder`
`➥Configurer` za pomocą konfiguracji Java:

```
@Bean
public
static PropertySourcesPlaceholderConfigurer placeholderConfigurer() {
   return new PropertySourcesPlaceholderConfigurer();
}
```

Jeśli preferujesz konfigurację w plikach XML, wybrany komponent możesz pobrać z użyciem elementu `<context:property-placeholder>` z przestrzeni nazw `context` Springa:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">
   <context:property-placeholder />
</beans>
```

Rozwiązywanie zewnętrznych właściwości jest jedną z metod pobierania wartości opóźnionego do czasu wykonania aplikacji. Jego możliwości związane są jednak głów-

nie z rozwiązywaniem właściwości za pomocą nazwy za pośrednictwem interfejsu `Environment` i źródeł właściwości. Język wyrażeń Springa (SpEL) umożliwia ogólniejsze sposoby wyliczania wartości wstrzykiwanych podczas działania aplikacji.

3.5.2. Tworzymy powiązania z użyciem języka wyrażeń Springa (SpEL)

W wersji 3. Springa pojawił się język wyrażeń Springa (*Spring Expression Language* — SpEL), potężny i zarazem zwięzły sposób wiązania wartości z właściwościami lub argumentami konstruktora komponentu, korzystający z wyrażeń wyliczanych w trakcie wywołania aplikacji. Język wyrażeń umożliwia nam wykonywanie niesamowitych operacji na wiązanych komponentach, których przeprowadzenie w inny sposób byłoby dużo trudniejsze (a czasem nawet niemożliwe).

SpEL trzyma w zanadrzu wiele sztuczek, takich jak:

- możliwość odwoływania się do komponentów za pomocą ich identyfikatorów;
- wywoływanie metod i uzyskiwanie dostępu do właściwości obiektów;
- matematyczne, relacyjne i logiczne operacje na wartościach;
- dopasowywanie wyrażeń regularnych;
- manipulacje na kolekcjach.

W dalszej części książki wykorzystamy język wyrażeń nie tylko do wstrzykiwania zależności, ale też do innych zadań. Spring Security pozwala na przykład zdefiniować za pomocą języka SpEL ograniczenia bezpieczeństwa. Jeśli w aplikacji Spring MVC korzystasz z systemu szablonów Thymeleaf w widokach, szablony te pozwalają za pomocą wyrażeń SpEL odwołać się do danych w modelu.

Rozpoczniemy od kilku przykładów wyrażeń SpEL i sposobów powiązania ich z komponentami. Następnie przyjrzymy się bliżej niektórym prostym wyrażeniom, które można łączyć ze sobą i tworzyć dzięki temu bardziej skomplikowane wyrażenia.

KILKA PRZYKŁADÓW WYRAŻEŃ SPEL

SpEL jest tak elastycznym językiem wyrażeń, że spisanie wszystkich sposobów jego wykorzystania spowodowałoby przekroczenie liczby stron przewidzianych dla całej tej książki. Mam jednak wystarczającą ilość miejsca na pokazanie kilku podstawowych przykładów, które mogą Cię zainspirować przy tworzeniu własnych wyrażeń.

Pierwszą rzeczą, którą musisz wiedzieć o wyrażeniach SpEL, jest to, że ograniczone są przez znaki `#{ … }`, podobnie jak symbole zastępcze ograniczone przez znaki `${ … }`. Poniżej zamieściłem przykład najprawdopodobniej jednego z najprostszych wyrażeń SpEL:

```
#{1}
```

Pomijając znaki `#{ … }`, to wszystko, co znajduje się wewnątrz, jest ciałem wyrażenia SpEL. W tym przypadku jest to stała numeryczna. Nie jest dla chyba żadnym zaskoczeniem, że wynikiem tego wyrażenia jest liczba `1`.

Oczywiście w prawdziwej aplikacji nie będziemy korzystać z tak prostych wyrażeń. Dużo częściej spotkamy się z bardziej interesującymi wyrażeniami, takimi jak:

```
#{T(System).currentTimeMillis()}
```

Wynikiem tego wyrażenia jest aktualny czas, podany w milisekundach, liczony w chwili rozwiązywania tego wyrażenia. Operator `T()` służy do przedstawienia klasy `java.lang.System` w postaci typu, aby umożliwić wywołanie metody statycznej `currentTimeMillis()`.

Wyrażenia SpEL umożliwiają też odwołania do innych komponentów oraz ich właściwości. Przykładowo wynikiem poniższego wyrażenia jest wartość właściwości `artist` komponentu o identyfikatorze `sgtPeppers`:

```
#{sgtPeppers.artist}
```

Istnieje również możliwość odwołania do właściwości systemu za pośrednictwem obiektu:

```
#{systemProperties['disc.title']}
```

To tylko kilka prostych przykładów wyrażeń SpEL. W tym rozdziale poznasz ich więcej. Teraz jednak pomyślmy, jak możemy wykorzystać wyrażenia przy wiązaniu elementów.

Jeśli korzystamy z mechanizmu skanowania komponentów, możemy do wstrzykiwania właściwości i argumentów konstruktora użyć adnotacji `@Value`, którą stosowaliśmy już wcześniej, przy okazji omawiania symboli zastępczych właściwości. Nie wykorzystujemy jednak wyrażeń w postaci symboli zastępczych, a wyrażenia SpEL. Poniższy przykład pokazuje, jak mógłby wyglądać konstruktor klasy `BlankDisc`, który pobiera dane o tytule i nazwie artysty z właściwości systemowych:

```
public BlankDisc(
   @Value("#{systemProperties['disc.title']}") String title,
   @Value("#{systemProperties['disc.artist']}") String artist) {
   this.title = title;
   this.artist = artist;
}
```

W konfiguracji XML możemy przekazać wyrażenia SpEL do atrybutu `value` elementów `<property>` lub `<constructor-arg>` albo jako wartość przekazaną do atrybutów przestrzeni nazw `p` bądź `c`. Poniżej widać przykładową deklarację XML komponentu `BlankDisc` o argumentach konstruktora ustawianych za pomocą wyrażeń SpEL:

```
<bean id="sgtPeppers"
   class="soundsystem.BlankDisc"
   c:_title="#{systemProperties['disc.title']}"
   c:_artist="#{systemProperties['disc.artist']}" />
```

Teraz, po obejrzeniu kilku prostych przykładów wstrzykiwania wartości rozwiązywanych za pomocą wyrażeń SpEL, poznasz podstawowe wyrażenia wspierane przez ten język.

WYRAŻAMY WARTOŚCI W POSTACI LITERAŁÓW

Widziałeś już przykład wyrażenia SpEL, którego wynikiem była wartość liczbowa. Wynikiem może być też jednak liczba zmiennoprzecinkowa, ciąg znaków lub wartość boolowska.

Poniższy przykład przedstawia wyrażenie SpEL, którego wartością jest liczba zmiennoprzecinkowa:

```
#{3.14159}
```

Liczby można również wyrazić w notacji naukowej. Wynikiem tego przykładu jest liczba 98 700:

```
#{9.87E4}
```

Wartością wyrażenia SpEL może być także ciąg znaków typu `String`:

```
#{'Hello'}
```

Literały `true` i `false` typu `Boolean` rozwiązywane są do ich wartości `Boolean`. Na przykład:

```
#{false}
```

Praca z literałami za pomocą wyrażeń SpEL nie wygląda zbyt interesująco. Nie potrzebujemy korzystać z wyrażeń, jeśli chcemy tylko przypisać właściwości wartość `1` lub `false`. Przyznaję, że wyrażenia SpEL złożone w całości z literałów nie przynoszą zbyt dużej korzyści. Należy jednak pamiętać, że bardziej interesujące wyrażenia SpEL złożone są z prostszych wyrażeń, dobrze jest więc wiedzieć, jak pracować z literałami. Będą nam one potrzebne przy konstruowaniu bardziej skomplikowanych wyrażeń.

ODWOŁUJEMY SIĘ DO KOMPONENTÓW, WŁAŚCIWOŚCI I METOD

Kolejną prostą funkcjonalnością oferowaną przez wyrażenia SpEL jest odwoływanie się do innych komponentów za pośrednictwem identyfikatorów. Możemy na przykład powiązać komponent z właściwością innego komponentu za pomocą identyfikatora ID jako wyrażenia SpEL (w tym przypadku komponentu o identyfikatorze `sgtPeppers`):

```
#{sgtPeppers}
```

Teraz za pomocą wyrażenia spróbujmy odwołać się do właściwości `artist` komponentu `sgtPeppers`:

```
#{sgtPeppers.artist}
```

Pierwsza część ciała wyrażenia odnosi się do komponentu o identyfikatorze `sgtPeppers`. Po kropce znajduje się referencja do właściwości `artist` obiektu.

Możemy nie tylko odwoływać się do właściwości komponentu, ale również wywoływać jego metody. Przypuśćmy, że istnieje komponent o identyfikatorze `artistSelector`. Możemy wywołać metodę `selectArtist()` tego komponentu za pomocą następującego wyrażenia:

```
#{artistSelector.selectArtist()}
```

Możemy też wywoływać metody kaskadowo na wartościach zwróconych po poprzednim wywołaniu. Przypuśćmy, że metoda `selectArtist()` zwraca wartość typu `String`, co umożliwia nam wywołanie metody `toUpperCase()` i otrzymanie nazwy artysty pisanej dużymi literami:

```
#{artistSelector.selectArtist().toUpperCase()}
```

Powyższy przykład działa doskonale, dopóki metoda `selectArtist()` nie zwróci wartości `null`. Możemy uchronić się przed wyrzuceniem wyjątku `NullPointerException` za pomocą operatora bezpieczeństwa typów:

```
#{artistSelector.selectArtist()?.toUpperCase()}
```

W wywołaniu metody `toUpperCase()` nie stosujemy samotnej kropki (`.`), ale poprzedzamy ją znakiem zapytania (`?.`). Operator ten upewnia się, że wartość po jego lewej stronie nie jest równa `null`, zanim podejmie próbę odwołania się do wartości po prawej stronie. Tak więc jeśli metoda `selectArtist()` zwróci wartość `null`, język wyrażeń SpEL nie podejmie nawet próby wywołania metody `toUpperCase()`. Wartością całego wyrażenia będzie `null`.

TYPY W WYRAŻENIACH

Kluczem w pracy z metodami i stałymi klasy w języku SpEL jest operator `T()`. Przykładowo aby wyrazić klasę `Math` języka Java za pomocą wyrażenia SpEL, musimy wykorzystać operator `T()` w następujący sposób:

```
T(java.lang.Math)
```

Wynikiem działania operatora `T()` jest obiekt typu `Class` reprezentujący klasę `java.lang.Math`. Możemy go nawet powiązać z właściwością komponentu typu `Class`. Prawdziwą korzyścią z użycia operatora `T()` jest dostęp do metod i stałych statycznych utworzonego typu.

Załóżmy, że musimy powiązać wartość pi z właściwością komponentu. Możemy to uzyskać za pomocą następującej sztuczki:

```
T(java.lang.Math).PI
```

W podobny sposób możemy wywoływać metody statyczne na typie rozwiązanym za pomocą operatora `T()`. Przykład użycia operatora `T()` widziałeś już przy wywołaniu metody `System.currentTimeMillis()`. Poniżej znajduje się kolejny przykład, który generuje wartość losową z przedziału od 0 do 1.

```
T(java.lang.Math).random()
```

OPERATORY SPEL

Język SpEL oferuje kilka operatorów, z których możemy korzystać przy budowaniu wyrażeń. W tabeli 3.1 zamieściłem ich listę i krótkie podsumowanie.

Tabela 3.1. Operatory SpEL umożliwiające manipulację wartościami wyrażeń

Typ operatora	Operatory
Arytmetyczne	`+`, `-`, `*`, `/`, `%`, `^`
Porównania	`<`, `lt`, `>`, `gt`, `==`, `eq`, `<=`, `le`, `>=`, `ge`
Logiczne	`and`, `or`, `not`, `|`
Warunkowe	`?:` (trójargumentowy), `?:` (Elvis)
Wyrażenia regularne	`matches`

Prostym przykładem wykorzystania jednego z przedstawionych operatorów jest poniższe wyrażenie:

```
#{2 * T(java.lang.Math).PI * circle.radius}
```

Jest to nie tylko świetny przykład użycia operatora mnożenia (*), ale także prezentacja sposobu konstruowania złożonych wyrażeń poprzez łączenie prostych elementów. W podanym przykładzie liczba pi jest przemnażana przez 2, a następnie mnożona przez wartość właściwości `radius` (promień) komponentu o identyfikatorze `circle` (koło). W rezultacie otrzymujemy obwód koła zdefiniowanego za pomocą komponentu `circle`.

W podobny sposób możemy policzyć powierzchnię koła, korzystając z operatora potęgowania (^):

```
#{T(java.lang.Math).PI * circle.radius ^ 2}
```

W tym przypadku operator potęgowania służy do obliczenia kwadratu promienia koła.

Gdy pracujemy z wartościami typu String, operator + umożliwia łączenie ciągów znaków identycznie, jak ma to miejsce w Javie:

```
#{disc.title + ' by ' + disc.artist}
```

Język SpEL udostępnia też operatory porównania wartości w wyrażeniu. Jak można zauważyć w tabeli 3.1, operatory porównania dostępne są w dwóch postaciach: symbolicznej i tekstowej. W większości zastosowań obie te postaci są równoważne i możemy je wykorzystywać zgodnie z własnymi preferencjami.

Na przykład aby porównać ze sobą dwie liczby, możemy użyć podwójnego operatora równości (==):

```
#{counter.total == 100}
```

Możemy też jednak skorzystać z operatora tekstowego `eq`:

```
#{counter.total eq 100}
```

W obu przypadkach wynik jest ten sam. Jest nim wartość typu `Boolean`: `true`, jeśli właściwość `counter.total` jest równa 100, lub `false` w przeciwnym wypadku.

Język SpEL oferuje także operator trójargumentowy, działający podobnie jak jego javowy odpowiednik. Na przykład wynikiem poniższego wyrażenia jest ciąg „Zwycięzca!", jeżeli zachodzi warunek `scoreboard.score > 100`, albo „Przegrany" w przeciwnym wypadku:

```
#{scoreboard.score > 1000 ? "Zwycięzca!" : "Przegrany"}
```

Operator trójargumentowy jest często używany do sprawdzania, czy jakaś wartość jest równa `null`, i jeśli jest, ustawiana jest w jej miejsce inna domyślna wartość. Poniższe wyrażenie sprawdza, czy wartość właściwości `disc.title` jest równa `null`, i jeśli jest, wynikiem wyrażenia staje się ciąg „Rattle and Hum".

```
#{disc.title ?: 'Rattle and Hum'}
```

Ten typ wyrażenia jest często nazywany operatorem *Elvis*. Operator ten jest też wykorzystywany jako emotikonka i stąd pochodzi jego dziwna nazwa. Znak zapytania ma bowiem odwzorowywać charakterystyczny kształt włosów Elvisa Presleya[2].

[2] Nie wiń mnie. To nie ja wymyśliłem tą nazwę. Ale musisz przyznać, że ten operator trochę przypomina włosy Elvisa.

WYRAŻENIA REGULARNE

W pracy z tekstem przydatne może być sprawdzenie, czy tekst odpowiada jakiemuś wzorcowi. Język SpEL umożliwia wykorzystanie wyrażeń regularnych za pomocą operatora `matches`. Operator `matches` próbuje zastosować wyrażenie regularne (zapisane po jego prawej stronie) na wartości typu `String` (zapisanej jako argument po lewej stronie). Wynikiem operacji dopasowywania jest wartość typu `Boolean`: `true`, jeśli wartość odpowiada wzorcowi, i `false` w przeciwnym wypadku.

Przypuśćmy, że chcemy sprawdzić, czy ciąg zawiera poprawny adres e-mail. W takim przypadku moglibyśmy zastosować wzorzec podobny do podanego poniżej:

```
#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.pl'}
```

Wyjaśnienie zawiłości tego tajemniczego wyrażenia wykracza poza temat tej książki. Zdaję też sobie sprawę, że podane wyrażenie nie jest wystarczająco rozbudowane, by pokryć wszystkie możliwe scenariusze. Jest jednak w pełni wystarczające do przedstawienia działania operatora `matches`.

KOLEKCJE

Kilka z najciekawszych sztuczek związanych z wyrażeniami SpEL dotyczy pracy z kolekcjami i tablicami. Najprostszym przykładem jest odwołanie do pojedynczego elementu na liście:

```
#{jukebox.songs[4].title}
```

Wynikiem tego wyrażenia jest właściwość `title` piątego elementu (indeksy liczymy od zera) kolekcji `songs` (piosenki) komponentu o identyfikatorze `jukebox` (szafa grająca).

Możemy trochę urozmaicić ten przykład, wybierając piosenkę w sposób losowy:

```
#{jukebox.songs[T(java.lang.Math).random() *
   jukebox.songs.size()].title}
```

Okazuje się, że operator `[]` służący do pobrania pojedynczego elementu z tablicy lub kolekcji może też służyć do pobrania pojedynczego znaku z ciągu typu `String`. Na przykład:

```
#{'To jest test'[5]}
```

Jest to odwołanie do szóstego (indeks liczony od zera) znaku w podanym ciągu, czyli litery `s`.

Język SpEL udostępnia również operator wyboru (`.?[]`), który umożliwia filtrowanie kolekcji i wybór podzbioru jej elementów. Przypuśćmy, że chcemy otrzymać listę wszystkich piosenek zespołu Aerosmith dostępnych w szafie grającej (czyli tych, dla których atrybut `artist` ma wartość `Aerosmith`). Poniższe wyrażenie wykorzystuje operator wyboru do zwrócenia tej listy:

```
#{jukebox.songs.?[artist eq 'Aerosmith']}
```

Jak widać, operator wyboru przyjmuje w nawiasach kwadratowych inne wyrażenie. Następuje iteracja po kolejnych piosenkach z listy i dla każdego elementu obliczana jest wartość wskazanego wyrażenia. Jeśli wartością wyrażenia jest `true`, element umiesz-

czany jest w nowej tablicy. W przeciwnym wypadku jest pomijany. W podanym przykładzie wyrażenie sprawdza, czy właściwość `artist` danej piosenki ma wartość `Aerosmith`.

SpEL oferuje jeszcze dwa inne operatory wyboru: `.^[]`, który pozwala wybrać pierwszy pasujący element, oraz `.$[]`, umożliwiający wybór ostatniego pasującego elementu. Spójrzmy na kolejny przykład, w którym wybieramy pierwszą znalezioną piosenkę zespołu Aerosmith.

```
#{jukebox.songs.^[artist eq 'Aerosmith']}
```

Istnieje jeszcze operator projekcji (`.![]`), który umożliwia przetworzenie elementów jednej kolekcji i utworzenie nowej kolekcji. Przypuśćmy, że nie chcemy kolekcji obiektów piosenek, ale kolekcję wszystkich ich tytułów. Podane wyrażenie spowoduje utworzenie nowej kolekcji w oparciu o właściwość `title`:

```
#{jukebox.songs.![title]}
```

Oczywiście operator projekcji można połączyć z dowolnym innym operatorem, wliczając w to operator wyboru. Możemy przykładowo uzyskać listę wszystkich tytułów piosenek zespołu Aerosmith:

```
#{jukebox.songs.?[artist eq 'Aerosmith'].![title]}
```

Zaprezentowane tutaj przykłady to zaledwie przedsmak możliwości oferowanych przez język SpEL. Na łamach tej książki jeszcze niejednokrotnie będziemy z niego korzystać, zwłaszcza w rozdziale poświęconym bezpieczeństwu.

W tej chwili podsumuję krótko dotychczasowe rozważania na ten temat i przekażę małe ostrzeżenie. Wyrażenia języka Spring są poręcznymi i potężnymi narzędziami umożliwiającymi wstrzykiwanie obiektów do komponentów Springa. Tworzenie złożonych, „sprytnych" wyrażeń może się nam czasem wydawać kuszące. Nie należy jednak konstruować zbyt „mądrych" wyrażeń. Im są one „mądrzejsze" i „sprytniejsze", tym bardziej rośnie potrzeba ich właściwego przetestowania. Niestety, wyrażenia SpEL zapisywane są w postaci ciągów znaków typu `String` i ich testowanie nie zawsze jest proste. Z tego powodu chciałbym Cię zachęcić do tworzenia prostych wyrażeń, aby ich testowanie nie było aż tak istotne.

3.6. Podsumowanie

W tym rozdziale omówiłem wiele zagadnień. Wykorzystywana była wiedza z rozdziału 2., przekazana przy opisywaniu technik wiązania komponentów, ale także poszerzyłeś swoją wiedzę o nowe, zaawansowane techniki.

Rozpoczęliśmy od użycia profili Springa do rozwiązania typowego problemu z rozróżnianiem komponentów w zależności od środowiska, na które jest wdrożona aplikacja. Dzięki rozwiązywaniu komponentów powiązanych z danym środowiskiem na etapie wykonania aplikacji, poprzez porównanie tego środowiska z aktywnym profilem lub profilami, Spring ma możliwość wykorzystania tego samego pliku wynikowego na różnych środowiskach bez potrzeby przebudowania kodu.

Komponenty przypisane do profili to tylko jeden ze sposobów tworzenia komponentów w trakcie wywołania aplikacji. Spring 4 udostępnia bardziej ogólną metodę

określania, czy komponenty mają (czy nie mają) zostać utworzone, w zależności od spełniania pewnego zdefiniowanego warunku. Służy do tego adnotacja `@Conditional`, powiązana z implementacją interfejsu Springa `Condition`, która udostępnia deweloperom potężny i elastyczny mechanizm warunkowego tworzenia komponentów.

Poznałeś też dwie techniki postępowania w sytuacji niejednoznaczności autowiązania: komponenty główne i kwalifikatory. Ustawienie wybranego komponentu jako komponentu głównego jest bardzo proste, ale ograniczone w swoim działaniu. Z tego powodu do zawężenia listy kandydatów autowiązania do pojedynczego komponentu zastosowaliśmy mechanizm kwalifikatorów. Dodatkowo poznałeś sposób tworzenia własnej adnotacji kwalifikatora do określenia komponentu za pomocą jego cechy.

Chociaż większość tworzonych w Springu komponentów to singletony, w niektórych sytuacjach bardziej odpowiednie mogą być inne strategie tworzenia komponentów. Standardowo Spring pozwala na konstruowanie singletonów, prototypów, komponentów o zasięgu żądania oraz komponentów o zasięgu sesji. Podczas deklaracji tych dwóch ostatnich dowiedziałeś się kilku rzeczy na temat tworzenia obiektów pośredniczących o określonym zasięgu, opartych na klasach lub interfejsach.

Na koniec przyglądaliśmy się językowi wyrażeń Springa, który umożliwia rozwiązywanie wartości wstrzykiwanych do właściwości komponentów w trakcie wywoływania aplikacji.

Z wiedzą na temat wiązania komponentów możemy przejść do kolejnego tematu, czyli programowania aspektowego (ang. *aspect-oriented programming* — AOP). Tak jak wzorzec DI pomaga zminimalizować zależności pomiędzy współpracującymi ze sobą komponentami, tak programowanie aspektowe pomaga zmniejszyć zależności między komponentami a zadaniami wpływającymi na pracę wielu komponentów w aplikacji. W następnym rozdziale zagłębimy się w proces tworzenia i pracy z aspektami w Springu.

4 Aspektowy Spring

W tym rozdziale omówimy:

- Podstawy programowania aspektowego
- Tworzenie aspektów z POJO
- Stosowanie adnotacji `@AspectJ`
- Wstrzykiwanie zależności do aspektów AspectJ

Gdy piszę ten rozdział, w Teksasie (gdzie mieszkam) lato trwa w najlepsze. I jak to w Teksasie bywa, od kilku dni mamy rekordowo wysokie temperatury. Jest strasznie gorąco. Podczas takiej pogody klimatyzacja jest koniecznością. Jednak wadą klimatyzacji jest zużycie energii elektrycznej, a za energię elektryczną trzeba zapłacić. Niewiele możemy zrobić w celu uniknięcia wydatków na to, by mieszkać w chłodnych i komfortowych warunkach. To dlatego w każdym domu jest zamontowany licznik energii elektrycznej, który rejestruje wszystkie zużyte kilowaty, i raz w miesiącu ktoś przychodzi odczytać ten licznik, aby zakład energetyczny wiedział, jaki wystawić nam rachunek.

Teraz wyobraźmy sobie, co by się stało, gdyby taki licznik zniknął i nikt nie przychodził, by sprawdzić nasze zużycie energii elektrycznej. Załóżmy, że do każdego właściciela domu należałoby skontaktowanie się z zakładem energetycznym i zgłoszenie swojego zużycia energii. Choć jest możliwe, że jakiś obsesyjny właściciel domu uważnie rejestrowałby zużycie światła, telewizji czy klimatyzacji, większość by się tym nie przejmowała.

Ta forma systemu płatności za energię elektryczną mogłaby być świetna dla klientów, lecz byłaby daleka od ideału z punktu widzenia zakładów energetycznych.

Kontrola zużycia energii elektrycznej jest ważną funkcją, lecz nie zajmuje najwyższej pozycji w świadomości większości właścicieli domów. Koszenie trawników, odkurzanie dywanów czy sprzątanie łazienki to czynności, w które posiadacz domu jest aktywnie zaangażowany. Natomiast kontrola zużycia elektryczności w domu jest z punktu widzenia właściciela domu czynnością pasywną (choć byłoby cudownie, gdyby koszenie trawników też było czynnością pasywną — zwłaszcza w takie gorące dni).

Niektóre funkcje systemów informatycznych przypominają liczniki elektryczne w naszych domach. Pewne funkcje muszą być wykonane w wielu punktach aplikacji, ale nie chcemy ich jawnie wywoływać w każdym z tych miejsc. Rejestrowanie transakcji, ich bezpieczeństwo i zarządzanie nimi są ważne, lecz czy powinny być czynnościami, w których aktywnie uczestniczą obiekty naszej aplikacji? Czy może lepiej by było, aby obiekty naszej aplikacji pozostały skoncentrowane na problemach z dziedziny biznesowej, do jakich zostały zaprojektowane, i pozostawiły pewne aspekty do obsługi przez kogoś innego?

Funkcje, które przenikają wiele miejsc w aplikacji, noszą w branży programistycznej nazwę **zagadnień przecinających**. W typowej sytuacji takie zagadnienia przecinające są koncepcyjnie rozdzielone od biznesowej logiki aplikacji (choć często bezpośrednio w nią wbudowane). Oddzielenie zagadnień przecinających od logiki biznesowej jest miejscem, gdzie do pracy włącza się **programowanie aspektowe** (AOP).

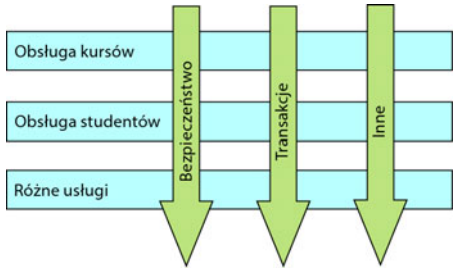
W rozdziale 2. nauczyliśmy się, jak używać wstrzykiwania zależności do zarządzania obiektami naszej aplikacji i ich konfigurowania. Podobnie jak wstrzykiwanie zależności pozwala rozdzielić od siebie poszczególne obiekty aplikacji, AOP pozwala oddzielić zagadnienia przecinające od obiektów, których one dotyczą.

Logowanie jest popularnym przykładem zastosowania aspektów. Jednak nie jest to jedyne zastosowanie, w którym korzystne będzie użycie aspektów. Podczas lektury tej książki zobaczymy kilka praktycznych zastosowań aspektów, w tym transakcje deklaracyjne, bezpieczeństwo oraz pamięć podręczną.

W tym rozdziale zostanie omówiona obsługa aspektów w Springu, w tym sposób deklarowania zwykłych klas, aby stały się aspektami, oraz możliwość zastosowania adnotacji podczas tworzenia aspektów. Dodatkowo dowiemy się, w jaki sposób przy użyciu AspectJ — innej popularnej implementacji AOP — możemy uzupełnić działanie środowiska AOP Springa. Lecz najpierw, zanim się zajmiemy transakcjami, bezpieczeństwem i pamięcią podręczną, dowiedzmy się, w jaki sposób aspekty są implementowane w Springu, zaczynając od paru zupełnych podstaw AOP.

4.1. Czym jest programowanie aspektowe

Jak wcześniej wspomnieliśmy, aspekty pomagają zamykać w modułach zagadnienia przecinające. W skrócie, zagadnienie przecinające można opisać jako dowolny mechanizm, którego wpływ jest używany na wielu miejscach w aplikacji. Bezpieczeństwo na przykład jest zagadnieniem przecinającym, w tym sensie, że wiele metod w aplikacji może podlegać stosowaniu reguł bezpieczeństwa. Na rysunku 4.1 pokazano obrazową ilustrację zagadnień przecinających.



Rysunek 4.1. Aspekty zamykają w modułach zagadnienia przecinające, które dotyczą logiki zawartej w wielu spośród obiektów aplikacji

Na rysunku pokazano typową aplikację, która jest podzielona na moduły. Głównym zadaniem każdego z modułów jest realizowanie usług ze swojej szczególnej dziedziny. Lecz każdy moduł potrzebuje także podobnych mechanizmów pomocniczych, takich jak bezpieczeństwo czy zarządzanie transakcjami.

Popularną obiektową techniką ponownego użycia tej samej funkcjonalności jest stosowanie dziedziczenia albo delegacji. Ale dziedziczenie może prowadzić do zburzenia hierarchii obiektów, jeśli ta sama klasa bazowa jest stosowana w całej aplikacji. Z kolei stosowanie delegacji bywa uciążliwe, ponieważ może być potrzebne skomplikowane wywołanie delegacji.

Aspekty stanowią alternatywę dla dziedziczenia i delegacji, która w wielu sytuacjach może być bardziej eleganckim rozwiązaniem. Korzystając z techniki AOP, nadal definiujemy popularne mechanizmy w jednym miejscu, lecz za pomocą deklaracji definiujemy, gdzie i jak mechanizmy te zostaną zastosowane, bez konieczności modyfikowania klasy, do której mają zastosowanie nowe mechanizmy. Odtąd zagadnienia przekrojowe możemy zamknąć w modułach w postaci specjalnych klas zwanych **aspektami**. Wynikają z tego dwie korzyści. Po pierwsze, logika dla każdego zagadnienia znajduje się w jednym miejscu, zamiast być rozrzuconą po całym kodzie. Po drugie, nasze moduły usługowe będą bardziej uporządkowane, ponieważ zawierają jedynie kod dotyczący ich głównego zadania (albo podstawowych funkcji), zaś zagadnienia drugorzędne zostały przeniesione do aspektów.

4.1.1. Definiujemy terminologię dotyczącą AOP

Podobnie jak w przypadku innych technologii, wraz z rozwojem AOP powstał specyficzny żargon opisujący tę technikę programistyczną. Aspekty są często opisywane za pomocą pojęć takich jak „porada", „punkt przecięcia" czy „punkt złączenia". Na rysunku 4.2 pokazano, jak są ze sobą powiązane te zagadnienia.

Niestety, wiele spośród pojęć używanych do opisywania AOP jest dalekie od intuicyjności. Są one jednak obecnie częścią pojęcia „programowanie aspektowe" i musimy się z nimi zapoznać w celu zrozumienia tej nowej techniki programistycznej. Zanim przejdziemy do praktyki, nauczmy się języka, w którym będziemy rozmawiać.

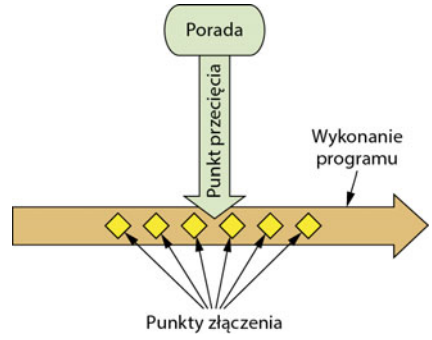
PORADA

Gdy osoba odczytująca licznik pokaże się w naszym domu, jej zadaniem jest poinformowanie zakładu energetycznego o wskazywanej liczbie kilowatogodzin. Z pewnością osoba ta ma listę domów, które musi odwiedzić, zaś informacje, które dostarcza

zakładowi energetycznemu, są ważne. Lecz głównym zadaniem osoby odczytującej liczniki jest sama czynność notowania zużycia energii.

Podobnie, aspekty mają główne zadanie — czynność, której wykonanie jest celem ich istnienia. W terminologii programowania aspektowego zadanie aspektu jest nazywane **poradą**.

Porada jest dla aspektu definicją zarówno czynności, która ma zostać wykonana, jak i właściwego momentu jej wykonania. Czy aspekt powinien być wykonany przed wywołaniem metody? Po jej wykonaniu? Zarówno przed wywołaniem metody, jak i po niej? A może tylko wtedy, gdy metoda zgłosi wyjątek?



Rysunek 4.2. Funkcjonalność aspektu (porada) jest wplatana do wykonania programu w jednym lub więcej punktach złączenia

Aspekty w Springu mogą działać z pięcioma rodzajami porad:

- *Before* — Funkcjonalność porady jest wykonywana przed wywołaniem metody z poradą.
- *After* — Funkcjonalność porady jest wykonywana po zakończeniu działania metody z poradą, niezależnie od wyniku jej działania.
- *After-returning* — Funkcjonalność porady jest wykonywana po prawidłowym zakończeniu metody z poradą.
- *After-throwing* — Funkcjonalność porady jest wykonywana po zgłoszeniu wyjątku przez metodę z poradą.
- *Around* — Porada realizuje tę samą funkcjonalność zarówno przed wywołaniem, jak i po zakończeniu metody z poradą.

PUNKTY ZŁĄCZENIA

Zakład energetyczny obsługuje wiele domów, prawdopodobnie wręcz całe miasto. W każdym domu zamontowany jest licznik energii elektrycznej, zatem każdy dom jest potencjalnym celem dla osoby odczytującej liczniki. Potencjalnie osoba taka mogłaby odczytywać wiele różnych przyrządów pomiarowych, lecz w ramach swoich obowiązków służbowych powinna przyjąć za cel właśnie liczniki energii zamontowane w domach.

W podobny sposób w naszej aplikacji mogą być tysiące miejsc, w których moglibyśmy zastosować poradę. Miejsca te są nazywane **punktami złączenia**. Punkt złączenia jest taką pozycją w przebiegu wykonania programu, w której może zostać wpięty aspekt. Takim punktem może być wywołanie metody, zgłoszenie wyjątku czy nawet modyfikacja zawartości pola. Są to pozycje, w których kod aspektu może zostać włączony w normalny przebieg działania naszej aplikacji, wzbogacając tym jej działanie.

PUNKTY PRZECIĘCIA

Nie jest możliwe, by jedna osoba odczytująca liczniki zdołała odwiedzić wszystkie domy, którym elektryczność dostarcza ten sam zakład energetyczny. Pojedynczej osobie jest przypisany jedynie pewien podzbiór wszystkich domów. Podobnie porada nie musi

koniecznie dołączać aspektu do każdego punktu złączenia w aplikacji. **Punkty przecięcia** pozwalają zawęzić listę punktów złączenia, do których zastosowany będzie aspekt.

Jeśli porada definiowała *czynność* i *moment jej wykonania* przez aspekt, to punkty przecięcia definiują *miejsce*, w którym czynność ta zostanie wykonana. Definicja punktu przecięcia dopasowuje jeden lub więcej punktów złączenia, w których należy wpleść poradę. Często określamy takie punkty przecięcia za pomocą jawnego wskazania nazw metod i klas albo za pomocą wyrażeń regularnych, które dopasowują do wzorców nazwy klas i metod. Niektóre frameworki aspektowe pozwalają na tworzenie dynamicznych punktów przecięcia, w których decyzja, czy zastosować poradę, jest podejmowana w trakcie działania, na przykład na podstawie wartości parametrów metod.

ASPEKTY

Gdy osoba odczytująca liczniki rozpoczyna dzień pracy, wie zarówno, co należy do jej obowiązków (kontrola zużycia energii elektrycznej), jak również z których domów powinna zbierać tę informację. Zatem wie ona wszystko, co potrzeba, by wywiązać się ze swoich obowiązków.

Aspekt jest sumą porady i punktów przecięcia. Wzięte razem porada i punkty przecięcia definiują wszystko, co można powiedzieć o aspekcie — jaką czynność wykonuje, gdzie i kiedy.

WPROWADZENIA

Wprowadzenie pozwala nam dodawać nowe metody lub atrybuty do istniejących klas. Na przykład możemy skonstruować klasę z poradą, o nazwie `Auditable`. Będzie ona przechowywała informację o momencie ostatniej modyfikacji obiektu. Może to być klasa tak prosta, że będzie posiadała tylko jedną metodę, nazwaną `setLastModified(Date)` i zmienną o zasięgu instancji, która będzie przechowywała stan tej klasy. Taką nową metodę i zmienną możemy wprowadzić do istniejących klas bez konieczności ich modyfikowania, wzbogacając je o nowe działanie i zmienną stanu.

WPLATANIE

Wplatanie jest procesem zastosowania aspektu do obiektu docelowego w celu utworzenia nowego obiektu z pośrednikiem. Aspekty są wplatane do docelowych obiektów we wskazanych punktach złączenia. Wplatanie może mieć miejsce w różnych momentach cyklu życia docelowego obiektu:

- *W czasie kompilacji* — Aspekty zostają wplecione, gdy klasa docelowa jest kompilowana. W tym celu potrzebny jest specjalny kompilator. Wplatający kompilator w AspectJ wplata aspekty w ten sposób.
- *W czasie ładowania klasy* — Aspekty zostają wplecione, gdy klasa docelowa jest ładowana do maszyny wirtualnej Javy (JVM). W tym celu potrzebny jest specjalny `ClassLoader`, który rozszerza kod bajtowy docelowej klasy, zanim zostanie ona wprowadzona do aplikacji. Opcja *wplatania w czasie ładowania* (ang. *load-time weaving* — LTW) wprowadzona w wersji 5 AspectJ realizuje wplatanie aspektów w ten sposób.

- *W czasie działania* — Aspekty zostają wplecione w jakimś momencie podczas działania aplikacji. W typowej sytuacji kontener aspektowy generuje dynamicznie obiekt pośredniczący, który jest połączony z obiektem docelowym za pomocą delegacji podczas wplatania aspektów. Z tego sposobu korzysta Spring AOP podczas wplatania aspektów.

To wiele nowych pojęć, z którymi trzeba się zapoznać. Wracając do rysunku 4.1, zobaczymy, że porada zawiera zachowanie zagadnienia przecinającego, które ma być włączone do obiektów aplikacji. Punktami złączenia są wszystkie punkty w przebiegu działania aplikacji, które są potencjalnymi miejscami zastosowania porady. Najważniejszym, co należy tu sobie uświadomić, jest fakt, że punkty przecięcia definiują, które punkty złączenia będą brane pod uwagę.

Gdy już się troszkę oswoiliśmy z najbardziej podstawową częścią terminologii dotyczącej programowania aspektowego, przekonajmy się, jak te podstawowe koncepcje AOP zostały zaimplementowane w Springu.

4.1.2. Obsługa programowania aspektowego w Springu

Nie wszystkie aspektowe frameworki są skonstruowane tak samo. Mogą się różnić tym, jak bogaty model punktów złączeń posiadają. Niektóre pozwalają stosować porady na poziomie modyfikacji pól, podczas gdy inne udostępniają jedynie punkty złączeń związane z wywołaniem metod. Mogą się także różnić sposobem i momentem wplatania aspektów. Niezależnie od konkretnego przypadku, możliwość tworzenia punktów przecięcia, definiujących punkty złączenia, w których powinny zostać wplecione aspekty, jest tym, co stanowi aspektowy framework.

Ponieważ książka ta jest poświęcona frameworkowi Spring, skupimy się na Spring AOP. Mimo to występuje wiele podobieństw między projektami Spring i AspectJ, zaś obsługa AOP w Springu zawiera wiele zapożyczeń z projektu AspectJ.

Obsługa AOP w Springu ma cztery odmiany:

- Klasyczna obsługa AOP w Springu na bazie obiektów pośredniczących.
- Aspekty zbudowane z „czystych" POJO.
- Aspekty sterowane adnotacją `@AspectJ`.
- Wstrzykiwane aspekty z AspektJ (dostępne we wszystkich wersjach Springa).

Pierwsze trzy pozycje są odmianami implementacji AOP w Springu. Programowanie aspektowe w Springu bazuje na dynamicznych obiektach pośredniczących. W efekcie obsługa AOP w Springu jest ograniczona do przechwytywania metod.

Słowo *klasyczne* zwykle budzi pozytywne skojarzenia. Klasyczne samochody, klasyczny turniej golfowy czy też klasyczna Coca-Cola — wszystkie są czymś dobrym. Lecz klasyczny model programowania aspektowego w Springu wcale nie jest zbyt dobry. Oczywiście, był czymś świetnym jak na swoje czasy. Lecz obecnie istnieją w Springu znacznie bardziej uporządkowane i łatwiejsze sposoby pracy z aspektami. W porównaniu do prostego deklaracyjnego modelu AOP i AOP opartego na adnotacjach klasyczny model AOP w Springu robi wrażenie ociężałego i nadmiernie skomplikowanego. W związku z tym nie będę omawiał *klasycznego* AOP w Springu.

Przestrzeń nazw `aop` umożliwia przekształcenie zwykłych obiektów POJO w aspekty. W praktyce obiekty te udostępniają tylko metody wywoływane w reakcji na napotkane punkty przecięcia. Niestety technika ta wymaga zastosowania konfiguracji w pliku XML. Jest to jednak prosty sposób przekształcenia obiektu w aspekt za pomocą deklaracji.

Mamy też możliwość konfiguracji AOP w Springu z wykorzystaniem adnotacji „zapożyczonych" z AspectJ. W rzeczywistości w dalszym ciągu korzystamy z modelu AOP opartego na obiektach pośredniczących, stosujemy jednak model programistyczny niemal identyczny jak w prawdziwych aspektach opatrzonych adnotacjami AspectJ. Ten styl AOP przynosi nam też taką korzyść, że jest możliwy bez stosowania konfiguracji XML.

Jeśli Twoje potrzeby odnośnie do AOP wykraczają poza proste przechwytywanie metod (obejmują na przykład przechwytywanie konstruktorów lub właściwości), warto rozważyć użycie aspektów AspectJ. W tym przypadku czwarta opcja na liście daje nam możliwość wstrzykiwania wartości do tych aspektów.

W tym rozdziale dowiemy się więcej o powyższych technikach aspektowych w Springu. Jednak zanim rozpoczniemy, ważne, byśmy zrozumieli kilka kluczowych zagadnień w aspektowym frameworku Spring.

PORADA SPRINGA JEST ZAPISANA W JAVIE

Wszystkie porady, jakie utworzymy w Springu, są zapisane w standardowych klasach Javy. W ten sposób czerpiemy korzyść z możliwości konstruowania naszych aspektów w tym samym zintegrowanym środowisku programistycznym (IDE), którego na co dzień używamy do pisania programów w Javie. Punkty przecięcia, które definiują miejsca, gdzie porady powinny zostać zastosowane, mogą być zapisane za pomocą adnotacji lub w plikach XML konfiguracji Springa. To oznacza, że zarówno kod aspektów, jak i składnia konfiguracji będą naturalne dla programistów Javy.

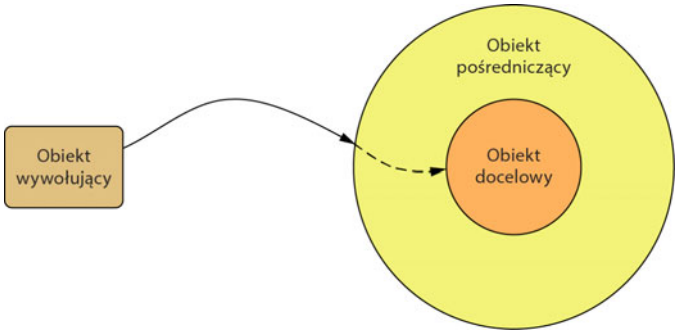
Inaczej jest w AspectJ. Choć obecnie AspectJ obsługuje już aspekty oparte na adnotacjach, AspectJ pojawił się jako rozszerzenie języka Java. Takie podejście ma zarówno zalety, jak i wady. Posiadanie specjalizowanego języka do obsługi aspektów zwiększa możliwości takiego języka i pozwala na precyzyjniejszą kontrolę jego zachowania, a także bogatszy zestaw narzędzi aspektowych. Lecz osiągamy to kosztem konieczności nauki nowego narzędzia i nowej składni.

SPRING DOŁĄCZA PORADY DO OBIEKTÓW W TRAKCIE PRACY

Spring wplata aspekty do zarządzanych przez niego komponentów w trakcie pracy, opakowując je w klasy pośredniczące. Jak pokazano na rysunku 4.3, klasa pośrednicząca zawiera docelowy komponent, przechwytuje wywołania metod z poradą i przekierowuje wywołania tych metod do komponentu docelowego.

W czasie pomiędzy przechwyceniem przez obiekt pośredniczący wywołania metody a momentem wywołania metody komponentu docelowego obiekt pośredniczący realizuje logikę aspektu.

Spring nie tworzy obiektu z pośrednikiem, dopóki aplikacja nie będzie potrzebowała określonego komponentu. Jeśli korzystamy z `ApplicationContext`, obiekt z pośrednikiem



Rysunek 4.3. Aspekty w Springu zostały zaimplementowane jako obiekty pośredniczące, które opakowują obiekt docelowy. Obiekt pośredniczący przechwytuje wywołania metod, wykonuje dodatkową logikę aspektu, a następnie wywołuje metodę docelową

zostanie utworzony, gdy kontekst będzie ładował wszystkie należące do niego komponenty z `BeanFactory`. Ponieważ Spring tworzy obiekty pośredniczące w czasie działania, korzystanie z AOP w Springu nie zmusza nas do stosowania specjalnego kompilatora, który umożliwiałby wplatanie aspektów.

SPRING OBSŁUGUJE JEDYNIE PUNKTY ZŁĄCZENIA ZWIĄZANE Z METODAMI

Jak wspomnieliśmy wcześniej, w poszczególnych implementacjach AOP stosowane są różne modele punktów złączenia. Ponieważ Spring bazuje na dynamicznych obiektach pośredniczących, obsługuje tylko punkty złączenia związane z metodami. W tym różni się od niektórych innych frameworków aspektowych, takich jak AspectJ czy JBoss, które poza punktami złączenia związanymi z metodami oferują także obsługę punktów złączenia związanych z polami oraz z konstruktorami. Nieobecność w Springu punktów przecięcia związanych z polami uniemożliwia nam tworzenie porad o bardzo dużej precyzji, takich jak przejęcie aktualizacji pola w obiekcie. Zaś bez punktów przecięcia związanych z konstruktorami nie dysponujemy sposobem, by zastosować poradę podczas tworzenia instancji beana.

Przechwytywanie wywołań metod powinno zaspokoić większość, jeśli nie wszystkie, z naszych potrzeb. Jeśli okaże się, że potrzebujemy czegoś więcej niż tylko przechwytywanie wywołań metod, będziemy mogli uzupełnić funkcjonalność AOP w Springu za pomocą AspectJ.

Teraz mamy już ogólne pojęcie, do czego służy programowanie aspektowe i w jaki sposób jest obsługiwane przez Springa. Nadszedł czas, by zakasać rękawy i wziąć się za tworzenie aspektów w Springu. Zacznijmy od deklaracyjnego modelu AOP w Springu.

4.2. Wybieramy punkty złączenia za pomocą punktów przecięcia

Jak wcześniej wspominaliśmy, punktów przecięcia używamy do wskazania miejsca, w którym powinna zostać zastosowana porada aspektu. Obok porady, punkty przecięcia stanowią najbardziej podstawowe składniki aspektu. Jest zatem ważne, byśmy wiedzieli, w jaki sposób wiązać punkty przecięcia.

Programując aspektowo w Springu, definiujemy punkty przecięcia za pomocą pochodzącego z frameworka AspectJ języka wyrażeń punktów przecięcia. Jeśli jesteśmy już

zaznajomieni ze środowiskiem AspectJ, wówczas definiowanie punktów przecięcia w Springu wyda się nam naturalne. Lecz w razie gdyby AspectJ był dla nas frameworkiem nowym, ten podrozdział może służyć jako szybki samouczek pisania punktów przecięcia w stylu AspectJ. Poszukującym bardziej szczegółowego omówienia frameworka AspectJ oraz pochodzącego z tego frameworka języka wyrażeń punktów przecięcia z całego serca polecam drugie wydanie książki *AspectJ in Action*, którą napisał Ramnivas Laddad.

Najważniejszym, co powinniśmy wiedzieć na temat punktów przecięcia w stylu AspectJ w zastosowaniu do programowania aspektowego w Springu, jest fakt obsługiwania przez Springa jedynie podzbioru desygnatorów punktów przecięcia dostępnych w AspectJ. Przypomnijmy sobie, że Spring AOP bazuje na obiektach pośredniczących i niektóre wyrażenia opisujące punkty przecięcia niczego nie wnoszą dla programowania aspektowego bazującego na obiektach pośredniczących. W tabeli 4.1 zawarto zestawienie desygnatorów punktów przecięcia pochodzących z AspectJ, które są obsługiwane przez Spring AOP.

Tabela 4.1. Spring do definiowania aspektów wykorzystuje język wyrażeń punktów przecięcia pochodzący z AspectJ

Desygnator w stylu AspectJ	Opis
`args()`	Ogranicza dopasowanie punktów złączenia do wywołań metod, których argumenty są instancjami określonych typów.
`@args()`	Ogranicza dopasowanie punktów złączenia do wywołań metod, których argumenty są opatrzone adnotacjami określonych typów.
`execution()`	Dopasowuje punkty złączenia, które są wywołaniami metod.
`this()`	Ogranicza dopasowanie punktów złączenia do takich, które posiadają w obiekcie pośredniczącym AOP referencję do beana określonego typu.
`target()`	Ogranicza dopasowanie punktów złączenia do takich, w których obiekt docelowy jest instancją określonego typu.
`@target()`	Ogranicza dopasowanie do punktów złączenia, w których klasa wywoływanego obiektu jest opatrzona adnotacją określonego typu.
`within()`	Ogranicza dopasowanie do punktów złączenia będących instancjami określonych typów.
`@within()`	Ogranicza dopasowanie do punktów złączenia będących instancjami typów, które są opatrzone określoną adnotacją (w zastosowaniu dla Spring AOP wywołania metod zadeklarowanych w typach opatrzonych określoną adnotacją).
`@annotation`	Ogranicza dopasowanie punktów złączenia do takich, w których obiekt będący przedmiotem działania punktów złączenia jest opatrzony określoną adnotacją.

Próba użycia któregoś z desygnatorów pochodzących z AspectJ, niewymienionego w powyższej tabeli, będzie skutkowała zgłoszeniem wyjątku `IllegalArgumentException`.

Przeglądając listę obsługiwanych desygnatorów, zwróćmy uwagę, że `execution` jest jedynym desygnatorem, który faktycznie realizuje dopasowanie. Wszystkich pozostałych używamy do ograniczania takich dopasowań. To znaczy, że podstawowy jest desygnator `execution`, którego użyjemy w każdej definicji punktu przecięcia, jaką napiszemy. Pozostałych będziemy używać do ograniczania zasięgu punktu przecięcia.

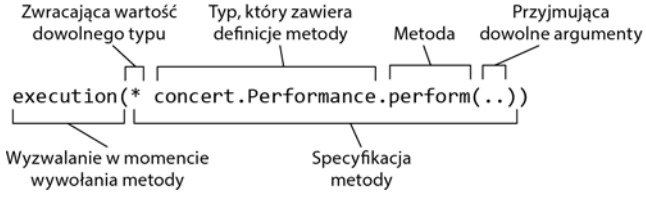
4.2.1. Piszemy definicje punktów przecięcia

Do zaprezentowania aspektów w Springu potrzebujemy czegoś, co będzie tematem punktu przecięcia aspektu. W tym celu zadeklarujmy interfejs `Performance`:

```
package concert;
public interface Performance {
   public void perform();
}
```

Interfejs ten reprezentuje dowolną formę występu na żywo, jak na przykład sztuka, film lub koncert. Załóżmy, że chcemy utworzyć aspekt, którego zadaniem będzie wywołanie utworzonej metody `perform()`.

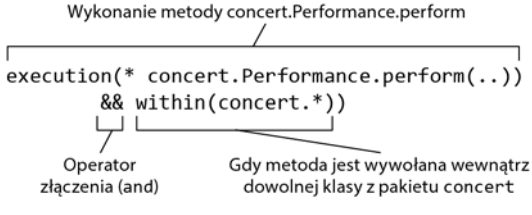
Na rysunku 4.4 pokazano wyrażenie, które pozwala zastosować poradę za każdym wywołaniem metody `perform()`.



Rysunek 4.4. Wybieramy metodę perform(), zdefiniowaną w interfejsie Performance, za pomocą wyrażenia punktu przecięcia w stylu AspectJ

Użyliśmy desygnatora `execution()`, by wybrać metodę `perform()`, należącą do interfejsu Performance. Specyfikacja metody rozpoczyna się od gwiazdki, która oznacza, że nie ma dla nas znaczenia, jaki będzie typ wartości zwróconej przez metodę. Następnie podajemy pełną, kwalifikowaną nazwę klasy oraz nazwę metody, którą chcemy wybrać. Dla listy parametrów metody użyliśmy podwójnej kropki (`..`), co oznacza, że punkt przecięcia może wybrać dowolną spośród metod `play()`, niezależnie od tego, jakie parametry przyjmują poszczególne z nich.

Teraz załóżmy, że chcemy zawęzić zasięg tego punktu przecięcia jedynie do pakietu `concert`. W takiej sytuacji możemy ograniczyć dopasowanie, dodając do wyrażenia desygnator `within()`, jak pokazano na rysunku 4.5.



Rysunek 4.5. Ograniczamy zasięg punktu przecięcia za pomocą desygnatora within()

Zauważmy, że użyliśmy operatora `&&`, by połączyć desygnatory `execution()` oraz `within()` za pomocą relacji *koniunkcji* (w której warunkiem dopasowania punktu przecięcia jest dopasowanie obydwu desygnatorów). Podobnie, mogliśmy użyć operatora `||`, by utworzyć relację *alternatywy*. Z kolei operator `!` pozwala zanegować wynik działania desygnatora.

Ponieważ znak & jest symbolem specjalnym w języku XML, możemy swobodnie zastąpić notację && operatorem and, gdy zapisujemy specyfikację punktów przecięcia w pliku konfiguracyjnym XML Springa. Podobnie, możemy użyć operatorów or oraz not, zastępując nimi, odpowiednio, notację || i !.

4.2.2. Wybieramy komponenty w punktach przecięcia

Spring wprowadził nowy desygnator bean(), rozszerzający listę zawartą w tabeli 4.1. Pozwala on wskazywać komponenty za pomocą ich nazw w wyrażeniu punktu przecięcia. Desygnator bean() przyjmuje jako argument nazwę komponentu i ogranicza działanie punktu przecięcia do tego konkretnego komponentu.

Przykładowo, rozważmy poniższy punkt przecięcia:

```
execution(* concert.Performance.perform()) and bean('woodstock')
```

Informujemy tu Springa, że chcemy zastosować poradę aspektu do wykonania metody perform() interfejsu Performance, lecz ograniczając się do wywołań z komponentu o nazwie woodstock.

Możliwość zawężenia punktu przecięcia do określonego komponentu może być cenna w niektórych sytuacjach, lecz możemy także użyć negacji, by zastosować aspekt do wszystkich komponentów, z wyjątkiem posiadającego określoną nazwę:

```
execution(* concert.Performance.perform()) and !bean('woodstock')
```

W tym wypadku porada aspektu zostanie wpleciona do wszystkich komponentów, które mają nazwę różną od woodstock.

Teraz, gdy omówiliśmy podstawy pisania punktów przecięcia, zmierzmy się z pisaniem porad i deklarowaniem aspektów, które będą z tych punktów przecięcia korzystały.

4.3. Tworzenie aspektów z użyciem adnotacji

Kluczową funkcjonalnością wprowadzoną w AspectJ 5 jest możliwość wykorzystania adnotacji do tworzenia aspektów. We wcześniejszych wersjach pisanie aspektów w AspectJ wymagało nauki specjalnego rozszerzenia języka Java. Model programowania oparty na adnotacjach AspectJ pozwala w prosty sposób, za pomocą kilku adnotacji, przekształcić dowolną klasę w aspekt.

Zdefiniowaliśmy już wcześniej interfejs Performance jako temat punktu przecięcia. Teraz za pomocą adnotacji AspectJ utwórzmy aspekt.

4.3.1. Definiujemy aspekt

Występ artystyczny nie jest prawdziwym występem bez udziału widowni. A może jednak jest? Patrząc z perspektywy występu, widownia jest wprawdzie istotna, ale nie najważniejsza dla samego faktu odbycia się tego występu. Są to dwie zupełnie odrębne sprawy. Z tego względu sensowne wydaje się zdefiniowanie widowni w postaci aspektu nałożonego na występ.

Listing 4.1. przedstawia klasę Audience (widownia) będącą definicją tego aspektu.

Listing 4.1. Klasa Audience: aspekt obserwujący występ

```
package concert;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class Audience {
   @Before("execution(** concert.Performance.perform(..))") <-- Przed występem
   public void silenceCellPhones() {
      System.out.println("Widzowie wyciszają telefony komórkowe");

   }

   @Before("execution(** concert.Performance.perform(..))") <-- Przed występem
   public void takeSeats() {
      System.out.println("Widzowie zajmują miejsca");
   }

   @AfterReturning("execution(** concert.Performance.perform(..))") <-- Po występie
   public void applause() {
      System.out.println("Brawooo! Oklaski!");
   }

   @AfterThrowing("execution(** concert.Performance.perform(..))") <-- Po nieudanym występie
   public void demandRefund() {
      System.out.println("Buu! Oddajcie pieniądze za bilety!");
   }
}
```

Zwróć uwagę na to, że klasa `Audience` została opatrzona adnotacją `@Aspect`. Adnotacja ta sprawia, że `Audience` nie jest już zwykłym obiektem POJO, ale aspektem. Metody zdefiniowane w ramach tej klasy zostały opatrzone adnotacjami, które określają szczegóły działania aspektu.

W klasie `Audience` zdefiniowaliśmy cztery metody opisujące zachowania widowni w trakcie występu. Przed rozpoczęciem występu publiczność musi zająć miejsca (metoda `takeSeats()`) i wyciszyć telefony komórkowe (`silenceCellPhones()`). Jeśli występ jest udany, publiczność wyraża swoje uznanie brawami (`applause()`). Jeżeli jednak występ zawiedzie oczekiwania widowni, publiczność wystąpi o zwrot pieniędzy (`demandRefund()`).

Jak widzisz, metody zostały opatrzone adnotacjami porad, aby określić moment ich wywołania. W AspectJ udostępniono pięć adnotacji pozwalających na zdefiniowanie porady. Przedstawiam je wszystkie w tabeli 4.2.

Klasa `Audience` korzysta z trzech spośród pięciu dostępnych adnotacji porad. Metody `takeSeats()` oraz `silenceCellPhones()` zostały opatrzone adnotacją `@Before`, dzięki czemu są wywoływane przed rozpoczęciem występu. Metoda `applause()` opatrzona jest adnotacją `@AfterReturning`, zostanie więc wywołana po zakończeniu udanego występu. Z kolei metoda `demandRefund()` została opatrzona adnotacją `@AfterThrowing`, co oznacza, że zostanie wywołana, jeśli w trakcie występu nastąpi wyrzucenie jakiegoś wyjątku.

Tabela 4.2. Spring wykorzystuje adnotacje AspectJ do deklaracji metod porad

Adnotacja	Porada
`@After`	Metoda porady wywoływana jest po zakończeniu działania metody lub wyrzuceniu wyjątku.
`@AfterReturning`	Metoda porady wywoływana jest po zakończeniu działania metody.
`@AfterThrowing`	Metoda porady wywoływana jest po wyrzuceniu wyjątku przez metodę.
`@Around`	Metoda porady „opakowuje" metodę.
`@Before`	Metoda porady wywoływana jest przed wywołaniem metody.

Jak łatwo zauważyć, do każdej z powyższych adnotacji przekazano jako wartość wyrażenie punktów przecięcia. Dla wszystkich czerech metod ustawiony jest ten sam punkt. Każda z metod mogłaby mieć przypisane inne wyrażenie punktu przecięcia, w naszym przypadku nie jest to jednak konieczne. Jeżeli przyjrzysz się bliżej wyrażeniu, to zauważysz, że jego uruchomienie następuje po wywołaniu metody `perform()` klasy `Performance`.

Kod na listingu 4.2 ma jedną wyraźną słabość — wyrażenie punktów przecięcia zostało powtórzone czterokrotnie. Powielanie kodu jest złą praktyką. Wolelibyśmy zdefiniować je w jednym miejscu, a następnie odwoływać się do tej definicji w pozostałych miejscach.

Na szczęście jest na to sposób. Adnotacja `@Pointcut` umożliwia zdefiniowanie punktu przecięcia w klasie aspektu w jednym miejscu i wielokrotne jego wykorzystanie. Listing 4.2 przedstawia zaktualizowany aspekt Audience, który korzysta z tak zdefiniowanego punktu przecięcia.

Listing 4.2. Deklarowanie często wykorzystywanych wyrażeń punktów przecięcia za pomocą adnotacji @Pointcut

```
package concert;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience {

   @Pointcut("execution(** concert.Performance.perform(..))") <-- Definiujemy nazwany punkt przecięcia
   public void performance() {}

   @Before("performance()") <-- Przed występem
   public void silenceCellPhones() {
      System.out.println("Widzowie wyciszają telefony komórkowe");
}

   @Before("performance()") <-- Przed występem
   public void takeSeats() {
      System.out.println("Widzowie zajmują miejsca");
   }
```

```
    @AfterReturning("performance()" ) <-------------------------- Po występie
    public void applause() {
       System.out.println("Brawooo! Oklaski!");
    }

    @AfterThrowing("performance()") <---------------------------- Po nieudanym występie
     public void demandRefund() {
       System.out.println("Buu! Oddajcie pieniądze za bilety!");
    }
}
```

Metodę `performance()` w klasie `Audience` opatrzyliśmy adnotacją `@Pointcut`. Adnotacja ta jako wartość przyjmuje wyrażenie punktu przecięcia dokładnie w ten sam sposób jak przy wcześniejszych definicjach adnotacji porad. W praktyce zastosowanie adnotacji `@Pointcut` na metodzie `performance()` spowodowało rozszerzenie języka wyrażeń. Dzięki temu możemy wykorzystać wyrażenie `performance()` w wyrażeniach punktów przecięcia i wyraźnie uprościć ich kod. Używamy go następnie w adnotacjach wszystkich czterech metod.

Ciało metody `performance()` nie jest istotne i powinno być puste. Metoda pełni jedynie funkcję znacznika, do którego można było podczepić adnotację `@Pointcut`.

Zauważ, że poza adnotacjami i pustą metodą `performance()` ciało klasy `Audience` jest w zasadzie zwykłym plikiem POJO. Metody tej klasy można wywoływać i testować jednostkowo, tak jak metody dowolnej innej klasy Javy. Jednym słowem, klasa `Audience` jest zwykłą klasą Javy, która po zastosowaniu adnotacji stała się aspektem.

Podobnie jak dowolną inną klasę Javy, tak i klasę `Audience` można powiązać jako komponent w Springu:

```
@Bean
public Audience audience() {
   return new Audience();
}
```

Jeśli definicję zakończylibyśmy w tym momencie, klasa Audience byłaby zwykłym komponentem w kontenerze Springa. Mimo że została opatrzona adnotacjami AspectJ, nie byłaby traktowana jako aspekt, jeżeli nie istniałoby coś, co zinterpretuje te adnotacje i utworzy obiekty pośredniczące, które dopiero ją w ten aspekt przekształcą.

Jeśli korzystasz z konfiguracji JavaConfig, możesz włączyć tworzenie automatycznych pośredników za pomocą adnotacji `@EnableAspectJAutoProxy` na poziomie klasy w pliku konfiguracyjnym. Listing 4.3. pokazuje, jak włączyć mechanizm tworzenia automatycznych pośredników.

Listing 4.3. Włączenie tworzenia automatycznych pośredników w celu obsługi aspektów AspectJ za pomocą konfiguracji JavaConfig

```
package concert;

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

@Configuration
```

```
@EnableAspectJAutoProxy ←——— Włączamy mechanizm tworzenia automatycznych pośredników
@ComponentScan
public class ConcertConfig {

   @Bean
   public Audience audience() { ←——————————— Deklarujemy komponent Audience
      return new Audience();
   }
}
```

Jeśli do konfiguracji powiązań komponentów w Springu wykorzystujesz plik XML, musisz użyć elementu `<aop:aspectj-autoproxy>` z przestrzeni nazw `aop`. Konfiguracja XML na listingu 4.4 pokazuje, jak to zrobić.

Listing 4.4. Włączenie tworzenia automatycznych pośredników w celu obsługi aspektów AspectJ za pomocą konfiguracji JavaConfig

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:aop="http://www.springframework.org/schema/aop" ←——┐ Deklaracja przestrzeni nazw
   xsi:schemaLocation="http://www.springframework.org/schema/aop │ aop w Springu
      http://www.springframework.org/schema/aop/spring-aop.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

   <context:component-scan base-package="concert" />
   <aop:aspectj-autoproxy /> ←——— Włączenie tworzenia automatycznych pośredników
   <bean class="concert.Audience" /> ←——— Deklaracja komponentu Audience
</beans>
```

Niezależnie od tego, czy korzystasz z konfiguracji w klasach Javy, czy w plikach XML, mechanizm automatycznych pośredników tworzy za pomocą adnotacji `@Aspect` obiekty pośredniczące opakowujące inne komponenty, dla których nastąpiło dopasowanie punktów przecięcia. W naszym przykładzie obiekt pośredniczący powstanie dla komponentu `Performance`, w przypadku którego metody porad wywołane zostaną przed wywołaniem metody `perform()` i po nim.

Istotne jest zrozumienie tego, że mechanizm automatycznych pośredników AspectJ w Springu używa adnotacji `@AspectJ` tylko jako oznaczeń wykorzystywanych przy tworzeniu aspektów. Pod spodem korzystamy w dalszym ciągu ze springowych aspektów opartych na obiektach pośredniczących. Jest to ważne, bo ogranicza możliwość użycia aspektów do inwokacji metod. Aby wykorzystać pełny potencjał aspektów AspectJ, musimy zrezygnować z aspektów opartych na obiektach pośredniczących, a zamiast nich użyć środowiska uruchomieniowego AspectJ.

Do tej pory definiowaliśmy aspekty za pomocą różnych metod dla porad `before` i `after`. Tabela 4.2 zawierała też jednak inny rodzaj porady: poradę `around` (wokół). Sposób pracy z tą poradą jest na tyle inny od metod pokazanych przeze mnie wcześniej, że warto spędzić trochę czasu, by nauczyć się z tej porady korzystać.

4.3.2. Tworzymy porady around

Porada typu `around` jest poradą udostępniającą największe możliwości. Pozwala na utworzenie logiki w pełni opakowującej metodę, na której zastosowano poradę. W swym działaniu przypomina użycie w ramach jednej metody porad typu `before` i `after`.

Spójrzmy na przykład porady `around` na listingu 4.5. W tym celu przepiszemy kod aspektu `Audience`. Tym razem w miejscu oddzielnych metod `before` i `after` wykorzystamy pojedynczą metodę porady `around`.

Listing 4.5. Implementacja aspektu Audience z użyciem porady around

```
package concert;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience {

   @Pointcut("execution(** concert.Performance.perform(..))") <-- Deklaracja nazwanego punktu przecięcia
   public void performance() {}

   @Around("performance()") <-- Metoda porady around
   public void watchPerformance(ProceedingJoinPoint jp) {
      try {
         System.out.println("Widzowie wyciszają telefony komórkowe");
         System.out.println("Widzowie zajmują miejsca");
         jp.proceed();
         System.out.println("Brawooo! Oklaski!");
      } catch (Throwable e) {
         System.out.println("Buu! Oddajcie pieniądze za bilety!");
      }
   }
}
```

Adnotacja `@Around` jest informacją, że metoda `watchPerformance()` ma być wywołana jako porada typu around dla punktu przecięcia `performance()`. Zastosowanie tej porady spowoduje, że przed rozpoczęciem występu publiczność wyciszy telefony i zajmie miejsca, a po zakończeniu przedstawienia nagrodzi występujących brawami. Podobnie jak we wcześniejszej wersji tego aspektu, jeśli w czasie występu wyrzucony zostanie jakiś wyjątek, publiczność upomni się o zwrot pieniędzy.

Jak widzisz, za pomocą porady `around` osiągnęliśmy ten sam efekt, co wcześniej za pomocą osobnych porad `before` i `after`. Wykonaliśmy jednak wszystkie czynności za pomocą jednej metody, a wcześniejsza implementacja rozłożona była na cztery osobne metody.

Pierwszym, co zauważamy w tej nowej metodzie porady, jest fakt, że otrzymuje ona parametr `ProceedingJoinPoint`. Obiekt ten jest konieczny, abyśmy byli w stanie wywołać docelową metodę z wnętrza naszej porady. Metoda porady wykonuje wszystkie swoje zadania, po czym, gdy jest gotowa do przekazania sterowania do metody docelowej, wywoła metodę `proceed()` obiektu `ProceedingJoinPoint`.

Zwróćmy uwagę, że sprawą krytyczną jest, abyśmy pamiętali o umieszczeniu w poradzie wywołania metody `proceed()`. Gdybyśmy o tym szczególe zapomnieli, w efekcie nasza porada blokowałaby dostęp do metody docelowej. Może byłby to efekt zgodny z naszymi zamierzeniami, lecz jest znacznie bardziej prawdopodobne, że naprawdę chcielibyśmy, aby metoda docelowa została w jakimś momencie wywołana.

Jeszcze jedną interesującą informacją jest fakt, że podobnie jak istnieje możliwość zablokowania dostępu do metody docelowej przez pominięcie wywołania metody `pro ↪ceed()`, możemy także umieścić w poradzie wielokrotne wywołanie tej metody. Jednym z powodów, by tak postąpić, może być implementowanie logiki ponawiania próby wykonania pewnej czynności w sytuacji, gdy realizująca ją metoda docelowa zakończyła się niepowodzeniem.

4.3.3. Przekazujemy parametry do porady

Jak dotąd nasze aspekty miały prostą budowę i nie przyjmowały żadnych parametrów. Jedynym wyjątkiem była metoda `watchPerformance()`, którą napisaliśmy w celu realizowania przykładowej porady `around`. Otrzymywała ona parametr `ProceedingJoinPoint`. Poza tym jednym przypadkiem nie zajmowaliśmy naszym poradom uwagi kontrolą wartości parametrów przekazywanych do metody docelowej. Było to jednak jak najbardziej poprawne, ponieważ metoda `perform()`, dla której pisaliśmy porady, nie przyjmowała parametrów.

Zdarzają się jednak sytuacje, gdy może się okazać pożyteczne, by porada nie tylko opakowywała metodę, lecz także kontrolowała wartości parametrów przekazywanych do tej metody.

Jako przykład wykorzystajmy kod klasy `BlankDisc` z sekcji 2.4.4. W obecnej implementacji metoda `play()` cyklicznie przełącza utwory i odtwarza kolejno za pośrednictwem metody `playTrack()`. Moglibyśmy jednak wywoływać metodę `playTrack()` bezpośrednio, aby odtworzyć pojedyncze utwory.

Przypuśćmy, że chcemy wiedzieć, ile razy został odtworzony dany utwór. Jednym z możliwych rozwiązań jest zmiana implementacji metody `playTrack()`, tak by zliczała każde kolejne odtworzenie utworu. Jednak liczenie odsłon jest czynnością całkowicie niezwiązaną z odtwarzaniem utworów, nie leży więc w zakresie zadań metody `play ↪Track()`.

Do zliczania odsłon utworów utworzymy osobną klasę, `TrackCount`, która będzie aspektem realizującym poradę dla metody `playTrack()`. Na listingu 4.6 znajduje się przykład takiego aspektu.

Listing 4.6. Użycie sparametryzowanego aspektu do zliczania liczby odtworzeń utworu

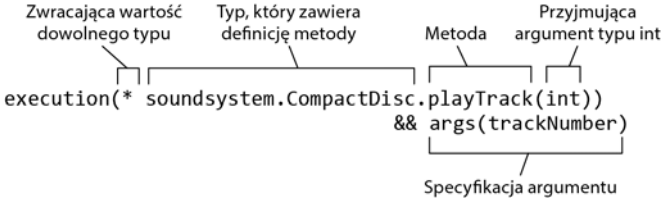
```
package soundsystem;

import java.util.HashMap;
import java.util.Map;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
```

```
public class TrackCounter {

   private Map<Integer, Integer> trackCounts =
      new HashMap<Integer, Integer>();
   @Pointcut(
      "execution(* soundsystem.CompactDisc.playTrack(int)) " +
      "&& args(trackNumber)")
   public void trackPlayed(int trackNumber) {}

   @Before("trackPlayed(trackNumber)")
public void countTrack(int trackNumber) {
      int currentCount = getPlayCount(trackNumber);
      trackCounts.put(trackNumber, currentCount + 1);
   }

   public int getPlayCount(int trackNumber) {
      return trackCounts.containsKey(trackNumber)
         ? trackCounts.get(trackNumber) : 0;
   }
}
```

Porada dla metody playTrack()

Zwiększ licznik przed odtworzeniem utworu

Tak jak w przypadku innych utworzonych do tej pory aspektów, ten również wykorzystuje adnotację `@Pointcut` do zdefiniowania nazwanych punktów przecięcia oraz adnotację `@Before` do zadeklarowania metody, która ma być poradą typu before. Istnieje jednak pewna różnica. Punkt przecięcia deklaruje także parametry, które mają zostać przekazane do metody porady. Rysunek 4.6 rozbija wyrażenie przecięcia na części składowe, aby wskazać miejsce deklaracji parametrów.



Rysunek 4.6. Deklarujemy parametr w wyrażeniu punktu przecięcia, który ma być przekazany do metody porady

Warto zwrócić uwagę na kwalifikator `args(trackNumber)` w wyrażeniu punktu przecięcia. Wskazuje on, że dowolna liczba całkowita przekazana jako argument metody `play ↪Track()` ma być następnie przekazana do porady. Nazwa parametru, `trackNumber`, odpowiada również parametrowi w sygnaturze metody punktu przecięcia.

Parametr przekazywany jest do metody porady, oznaczonej adnotacją `@Before` i wskazującej na nazwany punkt przecięcia `trackPlayed(trackNumber)`. Następuje dopasowanie parametru w punkcie przecięcia z parametrem o tej samej nazwie w metodzie porady i uzupełnienie ścieżki parametru z nazwanego punktu przecięcia do metody porady.

Teraz możemy skonfigurować klasy `BlankDisc` oraz `TrackCounter` jako komponenty w konfiguracji Springa i włączyć mechanizm automatycznego tworzenia obiektów pośredniczących, co pokazano na listingu 4.7.

Listing 4.7. Konfigurowanie klasy TrackCounter w celu zliczania liczby odtworzeń utworu

```
package soundsystem;

import java.util.ArrayList;
import java.util.List;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy <-- Włączenie AspectJ
public class TrackCounterConfig {

   @Bean
   public CompactDisc sgtPeppers() { <-- Komponent CompactDisc
      BlankDisc cd = new BlankDisc();
      cd.setTitle("Sgt. Pepper's Lonely Hearts Club Band");
      cd.setArtist("The Beatles");
      List<String> tracks = new ArrayList<String>();
      tracks.add("Sgt. Pepper's Lonely Hearts Club Band");
      tracks.add("With a Little Help from My Friends");
      tracks.add("Lucy in the Sky with Diamonds");
      tracks.add("Getting Better");
      tracks.add("Fixing a Hole");
      // ...pozostałe utwory zostały celowo pominięte...
      cd.setTracks(tracks);
      return cd;
   }

   @Bean
   public TrackCounter trackCounter() { <-- Komponent TrackCounter
      return new TrackCounter();
   }
}
```

Na koniec sprawdzimy, czy wszystko działa, za pomocą prostego testu zamieszczonego na listingu 4.8. W teście odtworzymy kilka utworów i sprawdzimy wartość licznika odtworzeń za pośrednictwem komponentu `TrackCounter`.

Listing 4.8. Testowanie aspektu TrackCounter

```
package soundsystem;

import static org.junit.Assert.*;
import org.junit.Assert;
import org.junit.Rule;
import org.junit.Test;
import org.junit.contrib.java.lang.system.StandardOutputStreamLog;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
```

```
@ContextConfiguration(classes=TrackCounterConfig.class)
public class TrackCounterTest {

   @Rule
   public final StandardOutputStreamLog log =
      new StandardOutputStreamLog();

   @Autowired
   private CompactDisc cd;

   @Autowired
   private TrackCounter counter;
   @Test
   public void testTrackCounter() {
      cd.playTrack(1); <-- Odtwarzamy jakieś utwory
      cd.playTrack(2);
      cd.playTrack(3);
      cd.playTrack(3);
      cd.playTrack(3);
      cd.playTrack(3);
      cd.playTrack(7);
      cd.playTrack(7);

      assertEquals(1, counter.getPlayCount(1)); <-- Sprawdzamy oczekiwane wartości liczników
      assertEquals(1, counter.getPlayCount(2));
      assertEquals(4, counter.getPlayCount(3));
      assertEquals(0, counter.getPlayCount(4));
      assertEquals(0, counter.getPlayCount(5));
      assertEquals(0, counter.getPlayCount(6));
      assertEquals(2, counter.getPlayCount(7));
   }
}
```

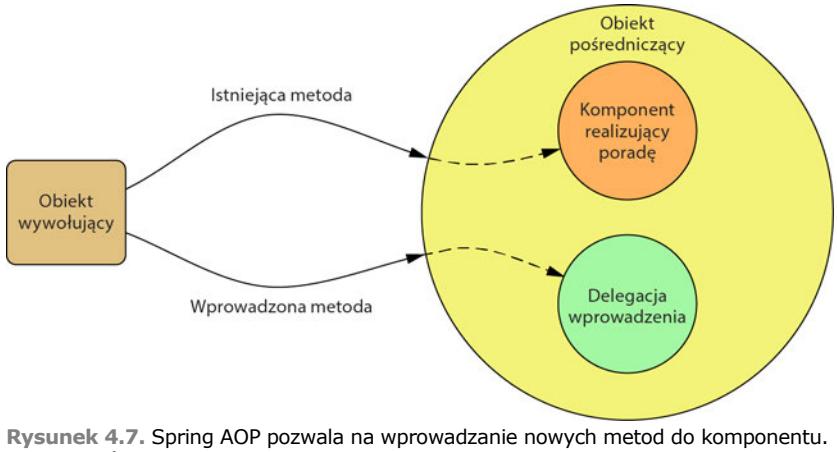
Aspekty, które poznawaliśmy do tej pory, opakowywały istniejące metody na obiektach poddawanych poradom. Zobaczmy, jak stworzyć aspekty, które dodają całkiem nowe funkcjonalności do obiektów.

4.3.4. Wprowadzenia z użyciem adnotacji

W niektórych językach, jak Ruby czy Groovy, istnieje pojęcie klas otwartych. Umożliwiają one dodawanie nowych metod do obiektu albo klasy bez konieczności bezpośredniego modyfikowania definicji takiego obiektu czy też klasy. Niestety, Java nie jest językiem tak dynamicznym. Gdy klasa została skompilowana, niewiele można zrobić w celu dodania do niej nowej funkcjonalności.

Lecz jeśli się dobrze zastanowić, czy nie to właśnie staraliśmy się robić w tym rozdziale za pomocą aspektów? Oczywiście, nie dodaliśmy do obiektów żadnej nowej metody, lecz dodawaliśmy nowe mechanizmy obok metod wcześniej definiowanych przez odpowiednie obiekty. Jeśli aspekt może opakowywać istniejące metody w dodatkowe mechanizmy, czemu nie dodać nowych metod do obiektu? Właściwie, korzystając z aspektowej koncepcji zwanej **wprowadzeniem**, aspekty mogą dodawać zupełnie nowe metody do beanów w Springu.

Przypomnijmy sobie, że w Springu aspekty są obiektami pośredniczącymi, które implementują te same interfejsy, co komponent docelowy. Jaki byłby skutek, gdyby, poza tymi wspólnymi interfejsami, obiekt pośredniczący implementował jeszcze jakiś inny interfejs? Wówczas każdy komponent realizujący poradę aspektu sprawiałby wrażenie, jakby także implementował ten nowy interfejs. Dzieje się tak, mimo że klasa, której instancją jest nasz komponent, nie zawiera implementacji tego dodatkowego interfejsu. Na rysunku 4.7 pokazano, jak to działa.



Rysunek 4.7. Spring AOP pozwala na wprowadzanie nowych metod do komponentu. Obiekt pośredniczący przechwytuje wywołania i deleguje do innego obiektu, który implementuje daną metodę

Na rysunku 4.7 możemy zauważyć, że w momencie wywołania wprowadzonego interfejsu obiekt pośredniczący deleguje wywołanie do pewnego innego obiektu, który zawiera implementację tego nowego interfejsu. W efekcie uzyskujemy pojedynczy komponent, którego implementacja jest rozdzielona pomiędzy więcej niż jedną klasę.

By wcielić ten pomysł w życie, powiedzmy, że chcemy wprowadzić do wszystkich implementacji występów poniższy interfejs `Encoreable` („bisowalna”):

```
package concert;

public interface Encoreable {
  void performEncore();
}
```

Pomijając dyskusję o tym, czy „bisowalna” jest prawdziwym słowem, potrzebujemy sposobu, by dodać ten interfejs do wszystkich implementacji interfejsu `Performance`. Zakładamy, że możemy zajrzeć do każdej implementacji interfejsu `Performance` i zmodyfikować je wszystkie tak, aby implementowały także interfejs `Encoreable`. Z projektowego punktu widzenia może to jednak nie być najlepsze rozwiązanie. Nie wszystkie występy muszą się kończyć bisem. Co więcej, może nawet okazać się niemożliwe, by zmienić wszystkie implementacje interfejsu `Performance`. Szczególnie jeśli pracujemy z implementacjami tworzonymi przez osoby trzecie, możemy nie mieć dostępu do kodu źródłowego.

Szczęśliwie wprowadzenia przez aspekty mogą nam pomóc sobie z tym poradzić, nie wymagając poświęcania decyzji projektowych lub inwazyjnych działań względem istniejących implementacji. Aby skorzystać z tego mechanizmu, musimy utworzyć nowy aspekt:

```
package concert;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class EncoreableIntroducer {
   @DeclareParents(value="concert.Performance+",
      defaultImpl=DefaultEncoreable.class)
   public static Encoreable encoreable;
}
```

Jak widać, klasa `EncoreableIntroducer` jest aspektem. Ale w odróżnieniu od aspektów, które do tej pory widzieliśmy, nie udostępnia porady `before`, `after` ani `around`. Zamiast tego dodaje interfejs `Encoreable` do komponentów typu `Performance` dzięki zastosowaniu adnotacji `@DeclareParents`.

Adnotacja `@DeclareParents` składa się z trzech części:

- Atrybut `value` identyfikuje typ komponentów, które mają implementować interfejs. W tym przypadku są to wszystkie komponenty, które implementują interfejs `Performance`. (Znak plusa na końcu oznacza, że chodzi o podtyp `Performance`, a nie o interfejs sam w sobie).
- Atrybut `defaultImpl` identyfikuje klasy, które dostarczą implementacji dla wprowadzenia. W naszym przykładzie będzie to klasa `DefaultEncoreable`.
- Statyczna właściwość opatrzona adnotacją `@DeclareParents` określa wprowadzany interfejs. W tym przypadku jest to interfejs `Encoreable`.

Tak jak w przypadku wszystkich innych aspektów, tak i tutaj musimy zadeklarować klasę `EncoreableIntroducer` jako komponent w kontekście aplikacji Springa.

```
<bean class="concert.EncoreableIntroducer" />
```

Dokończeniem tego zadania zajmie się mechanizm automatycznego tworzenia obiektów pośredniczących. Gdy Spring odnajdzie komponent opatrzony adnotacją `@Aspect`, automatycznie utworzy obiekt pośredniczący, który oddeleguje wywołanie do komponentu z pośrednikiem lub obiektu implementującego wprowadzenie, zależnie od tego, do kogo należy wołana metoda.

Adnotacje w połączeniu z automatycznym tworzeniem obiektów pośredniczących umożliwiają wygodne tworzenie aspektów w Springu. Model ten jest prosty i wymaga jedynie minimalnej konfiguracji. Deklaracja aspektów z użyciem adnotacji posiada niestety jedną wyraźną wadę: musimy mieć możliwość dodania adnotacji do klasy porady. Żeby to było możliwe, potrzebujemy dostępu do kodu źródłowego tej klasy.

Jeśli nie mamy dostępu do kodu źródłowego albo nie chcemy umieszczać adnotacji AspectJ w kodzie, Spring oferuje nam inną metodę pracy z aspektami. Zobaczmy teraz, w jaki sposób możemy deklarować adnotacje z wykorzystaniem konfiguracji w plikach XML.

4.4. Deklarujemy aspekty w języku XML

We wcześniejszych rozdziałach przyznałem, że wolę użycie adnotacji od konfiguracji w klasach Javy, a konfiguracja JavaConfig odpowiada mi bardziej niż konfiguracja oparta na plikach XML. Jeśli jednak chcemy zadeklarować aspekty bez użycia adnotacji do oznaczenia klas porad, nie mamy innego wyjścia, jak skorzystać z konfiguracji XML.

Przestrzeń nazw `aop` w Springu udostępnia kilka użytecznych elementów przy deklaracji aspektów w plikach XML. Ich wykaz znajduje się w tabeli 4.3.

Tabela 4.2. Elementy konfiguracyjne Spring AOP umożliwiają nieinwazyjną deklarację aspektów

Element konfiguracji AOP	Zastosowanie
`<aop:advisor>`	Definiuje doradcę aspektowego.
`<aop:after>`	Definiuje aspektową poradę `after` (niezależną od wyniku działania metody zaopatrzonej w poradę).
`<aop:after-returning>`	Definiuje aspektową poradę `after-returning` (po pomyślnym zakończeniu działania metody).
`<aop:after-throwing>`	Definiuje aspektową poradę `after-throwing` (po zgłoszeniu wyjątku przez metodę).
`<aop:around>`	Definiuje aspektową poradę `around` (zarówno przed wykonaniem metody, jak i po zakończeniu jej działania).
`<aop:aspect>`	Definiuje aspekt.
`<aop:aspect-autoproxy>`	Przełącza w tryb aspektów sterowanych adnotacjami z użyciem `@AspectJ`.
`<aop:before>`	Definiuje aspektową poradę `before` (przed wykonaniem metody).
`<aop:config>`	Element nadrzędnego poziomu w konfiguracji aspektowej. Większość elementów `<aop:*>` powinna znajdować się wewnątrz elementu `<aop:config>`.
`<aop:declare-parents>`	Wprowadza do obiektów z poradą dodatkowe interfejsy, implementowane w przezroczysty sposób.
`<aop:pointcut>`	Definiuje punkt przecięcia.

Mieliśmy już okazję poznać element `<aop:aspectj-autoproxy>` i dowiedzieć się, w jaki sposób umożliwia on włączenie mechanizmu automatycznego tworzenia obiektów pośredniczących dla klas porad opatrzonych adnotacjami AspectJ. Pozostałe elementy przestrzeni `aop` pozwalają zadeklarować aspekty bezpośrednio w konfiguracji Springa bez potrzeby wykorzystania adnotacji.

Spójrzmy na przykład na klasę `Audience`. Tym razem usuniemy z niej wszystkie adnotacje AspectJ:

```
public class Audience {

   public void silenceCellPhones() {
      System.out.println("Widzowie wyciszają telefony komórkowe");

   }

   public void takeSeats() {
      System.out.println("Widzowie zajmują miejsca");
   }
```

```
    public void applause() {
        System.out.println("Brawooo! Oklaski!");
    }

    public void demandRefund() {
        System.out.println("Buu! Oddajcie pieniądze za bilety!");
    }
}
```

Jak widzimy, klasa `Audience`, pozbawiona adnotacji AspectJ, niczym szczególnym się nie wyróżnia. Jest to podstawowa klasa Javy, zawierająca kilka metod. Możemy także zarejestrować tę klasę jako beana w kontekście aplikacji Springa.

Mimo skromnego wyglądu klasy `Audience` niezwykłe w niej jest to, że ma ona wszystko, czego potrzeba do utworzenia porady AOP. Potrzebuje ona jedynie odrobiny pomocy, aby stała się dokładnie takim aspektem, o jaki nam chodzi.

4.4.1. Deklarujemy porady before i after

Moglibyśmy przywrócić wszystkie adnotacje AspectJ, ale nie to jest celem tej sekcji. Użyjemy kilku elementów z przestrzeni nazw Springa `aop`, aby przekształcić pozbawioną adnotacji klasę `Audience` z powrotem w aspekt. Na listingu 4.9 znajduje się kod potrzebny do przeprowadzenia tej operacji.

Listing 4.9. Pozbawiona adnotacji klasa Audience, zadeklarowana jako aspekt z użyciem konfiguracji XML

```
<aop:config>
   <aop:aspect ref="audience">          <-- Referencja do komponentu audience
      <aop:before pointcut=             <-- Przed występem
         "execution(** concert.Performance.perform(..))"
         method="takeSeats" />

      <aop:before pointcut=             <-- Przed występem
         "execution(** concert.Performance.perform(..))"
         method="turnOffCellPhones" />

      <aop:after-returning pointcut=    <-- Po występie
         "execution(** concert.Performance.perform(..))"
         method="applause" />

      <aop:after-throwing pointcut=     <-- Po nieudanym występie
         "execution(** concert.Performance.perform(..))"
         method="demandRefund" />
  </aop:aspect>
</aop:config>
```

Pierwszym, co zauważamy w temacie elementów konfiguracyjnych Spring AOP, jest fakt, że prawie wszystkie z nich powinny być użyte wewnątrz kontekstu elementu `<aop:config>`. Jest od tej reguły kilka wyjątków, lecz podczas deklarowania komponentów jako aspektów zawsze zaczynamy od elementu `<aop:config>`.

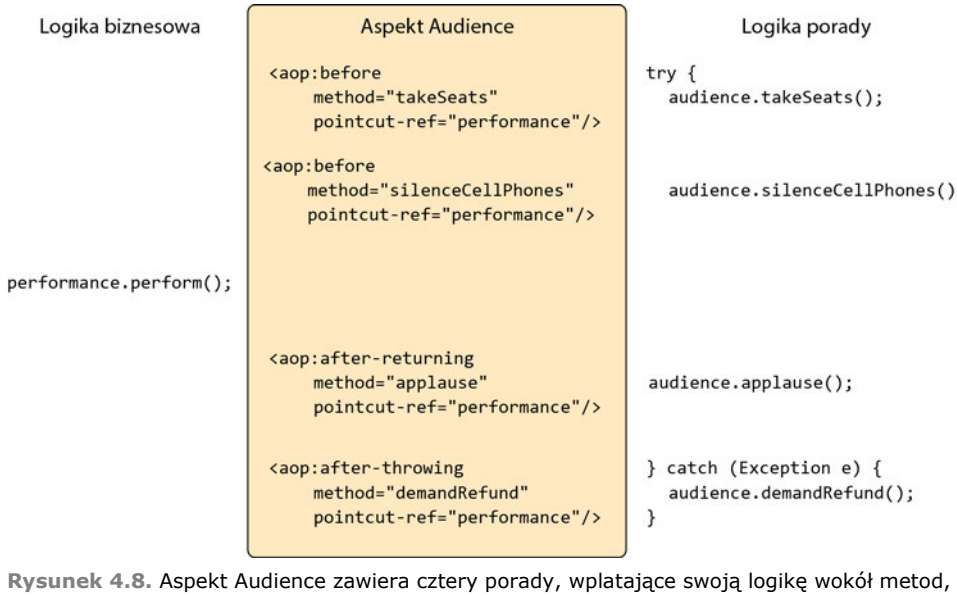
Wewnątrz elementu `<aop:config>` możemy zadeklarować jeden albo więcej aspektów, doradców lub punktów przecięcia. Na listingu 4.9 za pomocą elementu `<aop:aspect>`

zadeklarowaliśmy pojedynczy punkt przecięcia. Atrybut `ref` zawiera referencję do komponentu w postaci POJO, który będzie użyty jako dostawca funkcjonalności aspektu — w tym wypadku jest to komponent `audience`. Komponent, do którego referencję zawiera atrybut `ref`, będzie dostarczał metody wywoływane przez każdą z porad w aspekcie.

Warto zauważyć, że komponent ten może być dowolnego typu, musi tylko udostępniać metody wywoływane w wyznaczonych punktach przecięcia. Dzięki temu konfiguracja AOP w Springu oparta na plikach XML jest bardzo pomocna, gdy jako porad używamy typów zdefiniowanych w zewnętrznej bibliotece. Nawet wtedy, gdy nie mamy możliwości opatrzenia ich adnotacjami AspectJ.

Aspekt posiada cztery różne porady. Dwa elementy `<aop:before>` definiują porady *realizowane przed wykonaniem metody*, które wywołają metody `takeSeats()` oraz `silenceCellPhones()` (zadeklarowane z użyciem atrybutu `method`) komponentu `Audience`, przed wykonaniem jakiejkolwiek metody dopasowanej do punktu przecięcia. Element `<aop:after-returning>` definiuje *poradę realizowaną po powrocie z metody*, która wywoła metodę `applause()` po punkcie przecięcia. Jednocześnie element `<aop:after-throwing>` definiuje *poradę realizowaną po zgłoszeniu wyjątku*, która wywoła metodę `demandRefund()`, jeśli zostanie zgłoszony jakiś wyjątek. Na rysunku 4.8 pokazano, jak logika porady jest wplatana między logikę biznesową.

We wszystkich elementach porad atrybut `pointcut` definiuje punkt przecięcia, w którym zostanie zastosowana porada. Wartość, jaką podamy w atrybucie `pointcut`, jest punktem przecięcia zdefiniowanym w składni wyrażeń punktów przecięcia w stylu AspectJ.



Rysunek 4.8. Aspekt Audience zawiera cztery porady, wplatające swoją logikę wokół metod, które zostaną dopasowane do punktu przecięcia

Zapewne zwróci naszą uwagę fakt, że wszystkie elementy porad posiadają taką samą wartość atrybutu `pointcut`. To dlatego, że wszystkie porady mają zostać zastosowane w tym samym punkcie przecięcia.

Z problemem duplikacji kodu spotkaliśmy się już wcześniej, przy omawianiu porad opatrzonych adnotacjami AspectJ. Udało nam się go wyeliminować dzięki zastosowaniu adnotacji `@Pointcut`. W przypadku deklaracji opartej na plikach XML musimy jednak użyć elementu `<aop:pointcut>`. Listing 4.10 przedstawia sposób wyodrębnienia powtarzających się wyrażeń punktów przecięcia do pojedynczej deklaracji, a następnie wykorzystanie jej we wszystkich elementach porad.

Listing 4.10. Definiujemy nazwany punkt przecięcia za pomocą elementu <aop:pointcut>

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut id="performance" expression=
        "execution(** concert.Performance.perform(..))"
        />

    <aop:before
        pointcut-ref="performance"
        method="takeSeats" />
    <aop:before
        pointcut-ref="performance"
        method="silenceCellPhones" />
    <aop:after-returning
        pointcut-ref="performance"
        method="applause" />
    <aop:after-throwing
        pointcut-ref="performance"
        method="demandRefund" />
  </aop:aspect>
</aop:config>
```

Definicja punktu przecięcia

Referencje do punktu przecięcia

Teraz punkt przecięcia jest zdefiniowany w jednym miejscu, do którego odwołuje się wiele elementów porad. Element `<aop:pointcut>` definiuje punkt przecięcia o nazwie `performance`. Jednocześnie wszystkie elementy porad zostały zmienione na odwołania do nazwanego punktu przecięcia za pomocą atrybutu `pointcut-ref`.

Jak pokazano na listingu 4.10, element `<aop:pointcut>` definiuje punkt przecięcia, do którego mogą się odwoływać wszystkie porady wewnątrz tego samego elementu `<aop:aspect>`. Lecz możemy także zdefiniować punkty przecięcia, które będą widoczne z wielu aspektów. W tym celu musielibyśmy umieścić element `<aop:pointcut>` bezpośrednio wewnątrz elementu `<aop:config>`.

4.4.2. Deklarujemy poradę around

Obecna implementacja aspektu `Audience` działa rewelacyjnie, lecz podstawowe porady `before` i `after` mają pewne ograniczenia. W szczególności dużym wyzwaniem jest wymiana informacji między parą porad `before` i `after` bez uciekania się do przechowywania tej informacji w zmiennych składowych.

Przykładowo, wyobraźmy sobie, że poza wyłączeniem telefonów komórkowych i oklaskiwaniem wykonawców po zakończeniu, chcielibyśmy, by widzowie zerknęli na zegarki i poinformowali nas o czasie trwania danego występu. Jedynym sposobem osiągnięcia tego za pomocą porad `before` i `after` jest zanotowanie momentu rozpoczęcia w poradzie `before` i raportowanie czasu trwania w poradzie `after`. Lecz musielibyśmy przechować moment rozpoczęcia w zmiennej składowej. Ponieważ zaś komponent `Audience` jest instancją klasy singletonowej, nie byłoby to rozwiązaniem bezpiecznym ze względu na pracę wielowątkową, gdybyśmy w taki sposób przechowywali informację o stanie.

Pod tym względem porada `around` ma przewagę nad parą porad `before` i `after`. Za pomocą porady `around` możemy osiągnąć ten sam efekt, co za pomocą osobnych porad `before` i `after`, lecz wykonując wszystkie czynności za pomocą jednej metody. Dzięki umieszczeniu w jednej metodzie całego zbioru porad nie ma potrzeby przechowywania informacji o stanie w zmiennej składowej.

Przykładowo spójrzmy na listing 4.11. i rozważmy nową klasę `Audience`, która zawiera pojedynczą metodę `watchPerformance()`, pozbawioną wszelkich adnotacji.

Listing 4.11. Metoda watchPerformance() realizuje funkcjonalność aspektowej porady around

```
package concert;

import org.aspectj.lang.ProceedingJoinPoint;

public class Audience {

   public void watchPerformance(ProceedingJoinPoint jp) {
      try {
         System.out.println("Widzowie wyciszają telefony komórkowe.");
         System.out.println("Widzowie zajmują miejsca.");

         jp.proceed();

         System.out.println("Brawooo! Oklaski!");
      } catch (Throwable t) {
         System.out.println("Buu! Oddajcie pieniądze za bilety!");
      }
   }
}
```

Przed występem

Przejście do metody opatrzonej poradą

Po występie

Po nieudanym występie

W przypadku aspektu `audience` metoda `watchPerformance()` zawiera całą funkcjonalność wcześniejszych czterech metod realizujących porady, lecz tym razem zawartych w jednej, włącznie z tym, że jest odpowiedzialna za obsługę zgłoszonych przez nią samą wyjątków.

Deklaracja porady `around` nie różni się znacząco od deklaracji innych rodzajów porad. Musimy jedynie posłużyć się elementem `<aop:around>`, jak na listingu 4.12.

Listing 4.12. Deklarujemy poradę around w pliku XML za pomocą elementu <aop:around>

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut id="performance" expression=
```

```
        "execution(** concert.Performance.perform(..))"
        />

    <aop:around ◄──────────────────────────────── Deklaracja porady around
        pointcut-ref="performance"
        method="watchPerformance()" />
  </aop:aspect>
</aop:config>
```

Podobnie jak w przypadku elementów XML dotyczących innych porad, element `<aop:around>` otrzymuje punkt przecięcia oraz nazwę metody realizującej poradę. Użyliśmy tu tego samego punktu przecięcia co wcześniej, lecz atrybutowi `method` nadaliśmy wartość wskazującą na nową metodę `watchPerformance()`.

4.4.3. Przekazujemy parametry do porady

W sekcji 4.3.3 wykorzystaliśmy adnotacje AspectJ do utworzenia aspektu, który aktualizował licznik odtworzeń utworów nagranych na płycie kompaktowej. Spójrzmy teraz na listing 4.13, aby dowiedzieć się, jak przygotować analogiczną konfigurację z użyciem plików XML. Na początek usuńmy wszystkie adnotacje `@AspectJ` z klasy `TrackCounter`.

Listing 4.13. Wersja klasy TrackCounter pozbawiona adnotacji

```
package soundsystem;

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

public class TrackCounter {

   private Map<Integer, Integer> trackCounts =
      new HashMap<Integer, Integer>();

public void countTrack(int trackNumber) { ◄──── Metoda, która ma być zadeklarowana
      int currentCount = getPlayCount(trackNumber);   jako porada before
      trackCounts.put(trackNumber, currentCount + 1);
   }

   public int getPlayCount(int trackNumber) {
      return trackCounts.containsKey(trackNumber)
         ? trackCounts.get(trackNumber) : 0;
   }
}
```

Klasa `TrackCounter` pozbawiona adnotacji AspectJ może się wydawać „goła”. W chwili obecnej klasa nie zliczy żadnych odtworzeń, dopóki w sposób jawny nie wywołamy metody `countTrack()`. Za pomocą niewielkiej konfiguracji w pliku XML przywrócimy jednak działanie klasy `TrackCounter` jako aspektu.

Listing 4.14 przedstawia pełną konfigurację Springa, która uwzględnia deklarację komponentów `TrackCounter` oraz `BlankDisc`, a także ustawia rolę klasy `TrackCounter` jako aspektu.

Listing 4.14. Konfigurujemy klasę TrackCounter jako sparametryzowany aspekt za pomocą pliku XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xsi:schemaLocation=
      "http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="trackCounter"
      class="soundsystem.TrackCounter" /> <-- Komponent TrackCounter
   <bean id="cd" class="soundsystem.BlankDisc"> <-- Komponent BlankDisc
      <property name="title"
         value="Sgt. Pepper's Lonely Hearts Club Band" />
      <property name="artist" value="The Beatles" />
      <property name="tracks">
         <list>
            <value>Sgt. Pepper's Lonely Hearts Club Band</value>
            <value>With a Little Help from My Friends</value>
            <value>Lucy in the Sky with Diamonds</value>
            <value>Getting Better</value>
            <value>Fixing a Hole</value>
            <!-- ...pozostałe utwory zostały celowo pominięte... -->
         </list>
      </property>
   </bean>

   <aop:config>
      <aop:aspect ref="trackCounter"> <-- Deklarujemy TrackCounter jako aspekt
         <aop:pointcut id="trackPlayed" expression=
            "execution(* soundsystem.CompactDisc.playTrack(int))
            and args(trackNumber)" />
         <aop:before
            pointcut-ref="trackPlayed"
            method="countTrack"/>
      </aop:aspect>
   </aop:config>
</beans>
```

Jak widzisz, wykorzystujemy te same elementy XML z przestrzeni nazw `aop` co poprzednio. Za ich pomocą deklarujemy obiekt POJO jako aspekt. Jedyną istotną różnicą jest to, że wyrażenie punktu przecięcia zawiera teraz parametr, który ma zostać przekazany do metody porady. Jeśli wyrażenie to porównamy z wyrażeniem z listingu 4.6, okaże się, że są niemal identyczne. Jedyną różnicą jest użycie słowa kluczowego `and` w miejsce `&&` (ponieważ znak `&` jest interpretowany jako początek encji w języku XML).

Przećwiczyliśmy tworzenie kilku podstawowych aspektów za pomocą konfiguracji XML i przestrzeni nazw `aop`. Teraz sprawdźmy, w jaki sposób przestrzeń nazw `aop` umożliwia deklarację aspektów wprowadzających.

4.4.4. Wprowadzamy nową funkcjonalność przez aspekty

Wcześniej, w sekcji 4.3.4, dowiedziałeś się, jak użyć adnotacji AspectJ `@DeclareParents` do magicznego wprowadzania nowych metod do komponentów. Technika wprowadzania nie jest jednak unikalna dla AspectJ. Podobne rezultaty możemy uzyskać za pomocą elementu `<aop:declare-parents>` z przestrzeni nazw `aop` Springa.

Poniższy fragment kodu XML jest odpowiednikiem utworzonego wcześniej wprowadzenia opartego na AspectJ:

```
<aop:aspect>
   <aop:declare-parents
      types-matching="concert.Performance+"
      implement-interface="concert.Encoreable"
      default-impl="concert.DefaultEncoreable"
   />
</aop:aspect>
```

Jak sugeruje znaczenie nazwy elementu `<aop:declare-parents>`, pozwala on na zadeklarowanie, że komponent, którego dotyczy porada, otrzyma nowe obiekty nadrzędne w hierarchii obiektów. W tym konkretnym wypadku deklarujemy za pomocą atrybutu `types-matching`, że typy pasujące do interfejsu `Performance` powinny posiadać także, jako klasę nadrzędną, interfejs `Encoreable` (wskazany przez atrybut `implement-interface`). Ostatnią sprawą do rozstrzygnięcia jest położenie implementacji metod interfejsu `Encoreable`.

Istnieją dwa sposoby, by wskazać implementację wprowadzonego interfejsu. W tym wypadku korzystamy z atrybutu `default-impl`, by wprost wskazać implementację za pomocą jej w pełni kwalifikowanej nazwy klasy. Alternatywnie, moglibyśmy wskazać implementację za pomocą atrybutu `delegate-ref`:

```
<aop:declare-parents
  types-matching="concert.Performance+"
  implement-interface="concert.Encoreable"
  delegate-ref="encoreableDelegate"
/>
```

Atrybut `delegate-ref` odwołuje się do komponentu w Springu jako delegacji wprowadzenia. Zakładamy tu, że komponent o nazwie `encoreableDelegate` istnieje w kontekście Springa:

```
<bean id="encoreableDelegate"
      class="concert.DefaultEncoreable"/>
```

Różnica między bezpośrednim wskazaniem delegacji za pomocą atrybutu `default-impl` oraz pośrednim przez atrybut `delegate-ref` polega na tym, że w tym drugim rozwiązaniu mamy komponent Springa, który sam może podlegać wstrzykiwaniu, otrzymać poradę albo w jakiś inny sposób zostać skonfigurowany przez Springa.

4.5. Wstrzykujemy aspekty z AspectJ

Choć Spring AOP jest wystarczającym rozwiązaniem dla wielu zastosowań aspektów, wypada słabo w porównaniu z rozwiązaniem AOP, jakim jest AspectJ. AspectJ obsługuje wiele typów punktów przecięcia, które nie są możliwe w Spring AOP.

Punkty przecięcia w konstruktorach, na przykład, są przydatne, gdy potrzebujemy zastosować poradę podczas tworzenia obiektów. W odróżnieniu od konstruktorów w niektórych innych językach obiektowych, konstruktory w Javie różnią się od zwykłych metod. Z tego powodu bazująca na obiektach pośredniczących obsługa programowania aspektowego w Springu okazuje się zdecydowanie niewystarczająca, gdy chcemy zastosować poradę podczas tworzenia obiektu.

W znacznej większości aspekty w AspectJ są niezależne od Springa. Choć mogą być wplatane do aplikacji opartych na Javie, w tym aplikacji w Springu, zastosowanie aspektów z AspectJ wprowadza odrobinę zamieszania po stronie Springa.

Jednak każdy dobrze zaprojektowany i znaczący aspekt prawdopodobnie będzie zależał od innych klas, które będą go wspomagały podczas działania. Jeśli aspekt jest zależny od jednej lub więcej klas w trakcie realizacji porady, możemy z poziomu tego aspektu tworzyć instancje takich współpracujących obiektów. Albo, jeszcze lepiej, możemy posłużyć się wstrzykiwaniem zależności w Springu, by wstrzykiwać komponenty do aspektów w AspectJ.

By to zobrazować, utwórzmy kolejny aspekt dla występów. Zatem utworzymy aspekt krytyka, który ogląda występy i po ich zakończeniu wystawia krytyczną recenzję. Aspekt ten, zdefiniowany na listingu 4.15, nazwiemy `CriticAspect`.

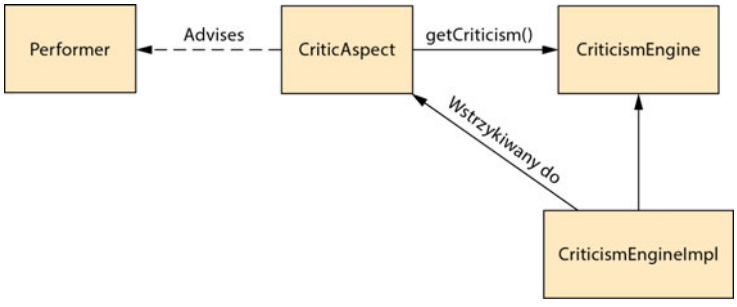
Listing 4.15. Implementacja w AspectJ krytyka występu

```
package concert;

public aspect CriticAspect {
  public CriticAspect() {}

  pointcut performance() : execution(* perform(..));

  afterReturning() : performance() {
    System.out.println(criticismEngine.getCriticism());
  }

private CriticismEngine criticismEngine;
  public void setCriticismEngine(CriticismEngine criticismEngine) { <-- Wstrzyknięty obiekt
    this.criticismEngine = criticismEngine;
  }
}
```

Głównym zadaniem aspektu `CriticAspect` jest komentowanie występu po jego zakończeniu. Punkt przecięcia `performance()` z listingu 4.15 zostanie dopasowany do metody `perform()`. W połączeniu z poradą `afterReturning()` otrzymujemy aspekt, który reaguje na zakończenie występu.

Tym, co okazuje się interesujące w listingu 4.15, jest fakt, że krytyk nie komentuje występu sam we własnym zakresie. Zamiast tego aspekt `CriticAspect` współpracuje

z obiektem `CriticismEngine`, wywołując jego metodę `getCriticism()`, aby uzyskać krytyczny komentarz po występie. By uniknąć niepotrzebnego wiązania między aspektem `CriticAspect` i obiektem `CriticismEngine`, aspekt `CriticAspect` otrzymuje referencję do obiektu `CriticismEngine` za pomocą wstrzykiwania przez metodę dostępową. Relacja ta została zobrazowana na rysunku 4.9.



Rysunek 4.9. Aspekty też potrzebują wstrzykiwania. Spring może wstrzykiwać zależności do aspektów w AspectJ, zupełnie jakby to były zwykłe komponenty

`CriticismEngine` jest tylko interfejsem, który deklaruje prostą metodę `getCriticism()`. Listing 4.16 przedstawia implementację tego interfejsu.

Listing 4.16. Implementacja interfejsu CriticismEngine, z którego korzysta aspekt JudgeAspect

```
package concert;

public class CriticismEngineImpl implements CriticismEngine {
  public CriticismEngineImpl() {}

  public String getCriticism() {
    int i = (int) (Math.random() * criticismPool.length);

    return criticismPool[i];
  }

  // wstrzyknięty obiekt
  private String[] criticismPool;
  public void setCriticismPool(String[] criticismPool) {
    this.criticismPool = criticismPool;
  }
}
```

Klasa `CriticismEngineImpl` implementuje interfejs `CriticismEngine`, losowo wybierając krytyczny komentarz z listy wstrzykniętej krytyki. Klasa ta może zostać zadeklarowana jako bean w Springu za pomocą poniższego kodu w języku XML:

```
<bean id="criticismEngine"
   class="concert.CriticismEngineImpl">
   <property name="criticisms">
      <list>
         <value>Najgorsze przedstawienie w historii! </value>
         <value>Śmiałem się, płakałem, a potem zorientowałem się, że jestem na niewłaściwym
         ↪przedstawieniu.</value>
```

```
        <value>Musicie zobaczyć ten występ!</value>
      </list>
   </property>
</bean>
```

Jak dotąd nieźle. Mamy już implementację interfejsu `CriticismEngine`, z którego będzie korzystał aspekt `CriticAspect`. Wszystko, co pozostało, to powiązanie klasy `CriticismEngineImpl` z aspektem `CriticAspect`.

Zanim zademonstrujemy, jak zrealizować wstrzykiwanie, powinniśmy wiedzieć, że aspekty w AspectJ mogą być wplatane do naszej aplikacji zupełnie bez angażowania Springa. Jednak jeśli chcemy użyć wstrzykiwania zależności w Springu, by wstrzykiwać klasy współpracujące do aspektu w AspectJ, musimy zadeklarować aspekt jako element `<bean>` w konfiguracji Springa. Poniższa deklaracja elementu `<bean>` realizuje wstrzykiwanie beana `criticismEngine` do aspektu `CriticAspect`:

```
<bean class="com.springinaction.springidol.CriticAspect"
    factory-method="aspectOf">
  <propertyname="criticismEngine" ref="criticismEngine"/>
</bean>
```

W dużej części ta deklaracja elementu `<bean>` nie różni się w istotny sposób od wszystkich innych deklaracji komponentów, jakie występują w Springu. Jedyna poważna różnica polega na użyciu atrybutu `factory-method`. Normalnie instancje komponentów w Springu tworzy kontener Springa, lecz aspekty w AspectJ są tworzone przez bibliotekę uruchomieniową AspectJ. Do momentu, gdy Spring uzyska możliwość wstrzyknięcia komponentu typu `CriticismEngine` do aspektu `CriticAspect`, istnieje już instancja klasy `CriticAspect`.

Ponieważ Spring nie odpowiada za tworzenie instancji aspektu `CriticAspect`, nie możemy po prostu zadeklarować klasy `CriticAspect` jako komponentu w Springu. Zamiast tego potrzebujemy sposobu, by Spring uzyskał uchwyt do instancji klasy `CriticAspect`, która została właśnie utworzona przez AspectJ, tak abyśmy mogli wstrzyknąć do niej obiekt `CriticismEngine`. Zgodnie z konwencją, wszystkie aspekty w AspectJ posiadają statyczną metodę `aspectOf()`, która zwraca singleton będący instancją aspektu. Zatem, by uzyskać instancję aspektu, musimy użyć atrybutu `factory-method`, by wywołać metodę `aspectOf()`, zamiast próbować wywoływać konstruktor klasy `CriticAspect`.

W skrócie, Spring nie korzysta z deklaracji `<bean>`, jakiej używaliśmy wcześniej, do tworzenia instancji klasy `CriticAspect` — instancja ta została już utworzona przez bibliotekę uruchomieniową AspectJ. Zamiast tego Spring otrzymuje referencję do aspektu przez metodę `aspectOf()` fabryki, a następnie realizuje wstrzykiwanie do niego zależności zgodnie z przepisem w elemencie `<bean>`.

4.6. Podsumowanie

Programowanie aspektowe jest potężnym uzupełnieniem programowania obiektowego. Dzięki aspektom możemy grupować zachowania aplikacji, dotychczas rozproszonych po całej aplikacji, w modułach wielokrotnego użytku. Możemy wówczas zadeklarować,

gdzie i w jaki sposób dane zachowanie będzie zastosowane. Pozwala to na ograniczenie niepotrzebnego powielania kodu i pozwala, aby podczas konstruowania klas skupić się na ich głównych funkcjach.

Spring zapewnia aspektowe środowisko uruchomieniowe, które pozwala nam na dodawanie aspektów wokół wywołań metod. Nauczyliśmy się, jak możemy wplatać porady przed wywołaniem metody, po jej wywołaniu oraz wokół niego, a także dodać dostosowane zachowanie do obsługi wyjątków.

Mamy możliwość podjęcia kilku decyzji co do sposobu użycia aspektów przez naszą aplikację w Springu. Wiązanie porad i punktów przecięcia w Springu jest znacznie prostsze dzięki dodaniu obsługi adnotacji `@AspectJ` i uproszczonemu schematowi konfiguracji.

Na koniec, zdarzają się sytuacje, gdy Spring AOP jest mechanizmem niewystarczającym i musimy przejść na AspectJ, aby korzystać z aspektów o większych możliwościach. Na wypadek takich sytuacji zerknęliśmy na sposób użycia Springa, by wstrzykiwać zależności do aspektów w AspectJ.

Do tego momentu omówiliśmy podstawy frameworka Spring. Dowiedzieliśmy się, jak skonfigurować kontener Springa i jak zastosować aspekty do obiektów zarządzanych przez Springa. Te podstawowe techniki dają świetną możliwość tworzenia aplikacji złożonych z luźno powiązanych obiektów.

Teraz przejdziemy do bardziej zaawansowanych tematów i dowiesz się, jak tworzyć prawdziwe aplikacje. W następnym rozdziale rozpoczniemy tworzenie aplikacji internetowej w Springu.

Część II

Spring w sieci

Spring wykorzystywany jest często do tworzenia aplikacji internetowych. W części II dowiesz się więc, jak użyć frameworka Spring MVC do utworzenia warstwy widoku naszej aplikacji.

W rozdziale 5., „Budowanie aplikacji internetowych za pomocą Springa", poznasz podstawy Spring MVC, frameworka sieciowego zbudowanego na bazie Springa. Nauczysz się tworzyć kontrolery Spring MVC do obsługi żądań sieciowych i zobaczysz, jak w przezroczysty sposób podpinać parametry żądania pod obiekty biznesowe, dbając jednocześnie o poprawność danych i obsługę błędów.

Rozdział 6., „Generowanie widoków", kontynuuje temat rozpoczęty w rozdziale 5., pokazując, jak wykorzystać dane z modelu utworzone w kontrolerach Spring MVC i wygenerować je w postaci kodu HTML serwowanego użytkownikowi w przeglądarce. W rozdziale tym pojawią się rozważania o stronach JavaServer Pages (JSP), Apache Tiles oraz szablonach Thymeleaf.

W rozdziale 7., „Zaawansowane możliwości Spring MVC", nauczysz się kilku bardziej zaawansowanych technik wykorzystywanych w tworzeniu aplikacji internetowych, takich jak ustawianie niestandardowych opcji konfiguracji, obsługa przesyłania plików w żądaniach wieloczęściowych, praca z wyjątkami oraz przekazywanie danych pomiędzy żądaniami za pośrednictwem atrybutów jednorazowych.

Z rozdziału 8., „Praca ze Spring Web Flow", dowiesz się, jak budować oparte na przepływach, konwersacyjne aplikacje sieciowe, używając frameworka Spring Web Flow.

Jako że bezpieczeństwo jest istotnym aspektem każdej aplikacji, w rozdziale 9., „Zabezpieczanie Springa", omówimy wreszcie użycie frameworka Spring Security w celu ochrony informacji aplikacji.

5 Budowanie aplikacji internetowych za pomocą Springa

W tym rozdziale omówimy:

- Odwzorowywanie żądań na kontrolery Springa
- Przezroczyste podpinanie parametrów formularzy
- Walidację przesyłanych danych

Jako programista aplikacji biznesowych Javy z pewnością miałeś okazję napisać parę aplikacji internetowych. Wielu programistów Javy koncentruje się wyłącznie na aplikacjach internetowych. Jeśli masz za sobą tego typu doświadczenia, doskonale zdajesz sobie sprawę z wyzwań związanych z tworzeniem takich aplikacji. Szczególnie ważnymi zagadnieniami do rozwiązania są: zarządzanie stanem, schematy działania (ang. *workflow*) i walidacja. A bezstanowy charakter protokołu HTTP wcale tego nie ułatwia.

Framework sieciowy Springa został stworzony do rozwiązywania tego typu problemów. Jest on oparty na wzorcu Model-Widok-Kontroler (ang. *Model-View-Controller* — MVC), dzięki czemu budowane aplikacje sieciowe są elastyczne i luźno powiązane w takim samym stopniu jak sam Spring Framework.

W tym rozdziale poznamy podstawowe możliwości frameworka internetowego Springa (Spring MVC). Skoncentrujemy się na wykorzystaniu adnotacji do tworzenia kontrolerów obsługujących różne żądania HTTP, parametry oraz formularze.

Zanim jednak zagłębimy się w szczegóły Spring MVC, przyjrzyjmy się mu na wyższym poziomie i wykonajmy czynności konieczne do rozpoczęcia pracy z tym frameworkiem.

5.1. Wprowadzenie do Spring MVC

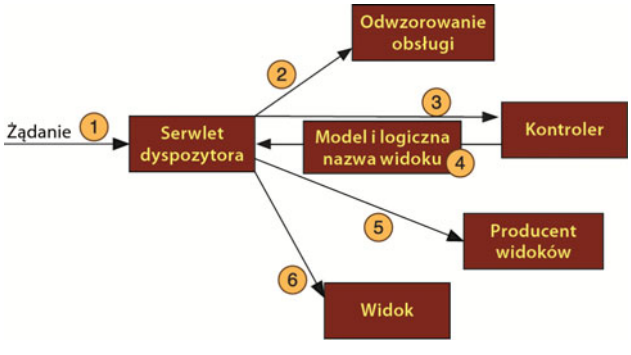
Jedną z ulubionych gier moich dzieci jest „Pułapka na myszy". Gra polega na przepuszczeniu małej stalowej kulki przez szereg dziwacznych urządzeń w celu „odpalenia" pułapki na myszy. Kulka przechodzi przez różne misternie skonstruowane instalacje, stacza się po kręconej rampie, odbija od trampoliny, wiruje na miniaturowym diabelskim kole, a nawet zostaje wykopana z wiadra przez gumowy but. Wszystko to po to, aby uruchomić pułapkę na biedną plastikową mysz.

Framework internetowy Springa na pierwszy rzut oka może przypominać „Pułapkę na myszy". Różnica polega na tym, że zamiast przemieszczać kulkę pomiędzy rampami, trampolinami i diabelskimi kołami, Spring przemieszcza żądania pomiędzy serwletem dyspozytora (ang. *dispatcher servlet*), odwzorowaniami obsługi, kontrolerami i producentami widoków (ang. *view resolvers*). Nie daj się jednak zwieść pozorom. Każdy komponent Spring MVC spełnia określone zadanie. I nie jest to aż takie skomplikowane. Zacznijmy naszą eksplorację frameworka od analizy cyklu życia żądania, rozpoczynając od momentu jego wysłania, przez podróż po komponentach Spring MVC, a kończąc na odpowiedzi wracającej do klienta.

5.1.1. Cykl życia żądania

Za każdym razem, gdy użytkownik klika odnośnik lub wysyła formularz w przeglądarce internetowej, mamy do czynienia z żądaniem. Żądanie można porównać do posłańca. Podobnie jak praca listonosza czy kuriera, praca żądania polega na przeniesieniu informacji z jednego miejsca w drugie.

Praca ta, wbrew pozorom, nie jest wcale taka lekka. W czasie między opuszczeniem przeglądarki a powrotem z odpowiedzią żądanie robi sobie kilka przystanków, zbierając i zostawiając trochę informacji na każdym z nich. Na rysunku 5.1 pokazano miejsca tych przystanków w podróży przez Spring MVC.



Rysunek 5.1. Zanim informacja przenoszona w żądaniu zakończy się pożądanym rezultatem, ma na swej drodze kilka postojów

Kiedy żądanie opuszcza przeglądarkę ❶, przenosi informację o poleceniu wydanym przez użytkownika. Informacja taka składa się przynajmniej z żądanego adresu URL. Żądanie może też przenosić dodatkowe dane, jak na przykład informację wysłaną przez użytkownika za pomocą formularza.

Pierwszym przystankiem w podróży żądania jest serwlet dyspozytora Springa. Podobnie jak wiele frameworków sieciowych opartych na Javie, Spring MVC kieruje wszystkie żądania do pojedynczego serwletu kontrolera frontowego (ang. *front controller servlet*). Kontroler frontowy jest popularnym wzorcem aplikacji, w którym pojedynczy serwlet deleguje odpowiedzialność za przetwarzanie żądania innym komponentom aplikacji. W przypadku Spring MVC kontrolerem frontowym jest serwlet dyspozytora.

Rola serwletu dyspozytora polega na przesłaniu żądania do kontrolera Spring MVC. Kontroler jest komponentem Springa, który przetwarza żądanie. Typowa aplikacja może jednak posiadać kilka kontrolerów. Serwlet dyspozytora potrzebuje zatem pomocy przy podejmowaniu decyzji, do którego z nich wysłać żądanie. Zwraca się o nią do jednego lub kilku odwzorowań obsługi ❷ w celu ustalenia następnego przystanku żądania. Przy podejmowaniu decyzji odwzorowanie obsługi zwróci szczególną uwagę na adres URL żądania.

Po wybraniu odpowiedniego kontrolera ❸ serwlet dyspozytora wysyła do niego żądanie. Na tym etapie żądanie wypakowuje swój ładunek (informację wysłaną przez użytkownika) i czeka cierpliwie na przetworzenie informacji przez kontroler (w rzeczywistości, rola dobrze zaprojektowanego kontrolera ogranicza się do delegacji odpowiedzialności za logikę biznesową któremuś z obiektów usług).

Działania wykonane przez kontroler często powodują potrzebę przesłania informacji z powrotem do użytkownika i wyświetlenia jej w przeglądarce. Informację taką określa się jako **model**. Wysłanie surowej informacji do użytkownika to jednak mało — powinna ona przybrać przyjazną dla użytkownika formę, z reguły będzie to kod HTML. Do jego wygenerowania potrzebny jest **widok**, najczęściej strona Java Server Page (JSP).

Jedną z ostatnich czynności wykonywanych przez kontroler jest przygotowanie danych modelu i identyfikacja nazwy widoku, który powinien wygenerować kod wyjściowy. Kontroler wysyła wtedy żądanie, wraz z modelem i nazwą widoku, z powrotem do serwletu dyspozytora ❹.

Aby kontroler nie był przywiązany do danego widoku, nazwa widoku przesyłana do serwletu dyspozytora nie identyfikuje bezpośrednio konkretnej strony JSP. Co więcej, nie sugeruje nawet, że widok musi być stroną JSP. Zamiast tego przekazuje tylko logiczną nazwę, która posłuży do odszukania konkretnego widoku, który wygeneruje wynik. Serwlet dyspozytora skonsultuje się z producentem widoków ❺ w celu przekształcenia logicznej nazwy widoku na konkretną implementację widoku, która może, ale nie musi być stroną JSP.

Na etapie, kiedy serwlet dyspozytora wie już, który widok wygeneruje wynik, rola żądania jest prawie skończona. Ostatnim przystankiem żądania jest implementacja widoku (❻, z reguły jest to strona JSP), której dostarcza dane modelu. To już koniec. Widok użyje danych modelu do wygenerowania wyjścia, które zostanie przesłane z powrotem do klienta za pomocą (nie aż tak bardzo zapracowanego) obiektu odpowiedzi ❼.

Jak widzisz, podróż, jaką odbywa żądanie od momentu jego wysłania do zwrócenia odpowiedzi, składa się z kilku etapów. Najwięcej z nich dotyczy działania frameworka Spring MVC w ramach komponentów pokazanych na rysunku 5.1. W tym rozdziale będziemy się głównie zajmować tworzeniem kontrolerów. Najpierw jednak przyjrzyjmy się przez chwilę podstawowym komponentom Spring MVC.

5.1.2. Konfiguracja Spring MVC

Kiedy spojrzymy na rysunek 5.1, może się nam wydawać, że konieczna będzie konfiguracja wielu elementów frameworka. Na szczęście dzięki ulepszeniom wprowadzonym w ostatnich wersjach Springa rozpoczęcie pracy ze Spring MVC jest bardzo łatwe. Zaczniemy od najprostszego możliwego podejścia do konfiguracji Spring MVC — wykonamy tylko minimum pracy potrzebnej do skorzystania z tworzonych kontrolerów. Dodatkowe dostępne opcje konfiguracji poznasz w rozdziale 7.

KONFIGURUJEMY SERWLET DYSPOZYTORA

Serwlet dyspozytora odgrywa centralną rolę w Spring MVC. To tam w pierwszej kolejności trafia żądanie we frameworku i to on odpowiada za przekierowanie tego żądania do wszystkich pozostałych komponentów.

W przeszłości serwlety takie jak serwlet dystrybutora konfigurowano w pliku *web.xml*, zawartym w pliku WAR aplikacji internetowej. Z całą pewnością jest to jedna z możliwości konfiguracji serwletu dystrybutora. Jednak dzięki ostatnim ulepszeniom wprowadzonym w specyfikacji Servlet 3 i Spring 3.1 nie jest to jedyna dostępna opcja. Nie jest to też opcja, z której będziemy korzystać w tym rozdziale.

Zamiast pliku *web.xml* do konfiguracji serwletu dystrybutora użyjemy konfiguracji opartej na klasach Javy. Potrzebna nam klasa konfiguracji znajduje się na listingu 5.1.

Listing 5.1. Konfigurujemy serwlet dystrybutora

```
package spittr.config;
import org.springframework.web.servlet.support.
   AbstractAnnotationConfigDispatcherServletInitializer;

public class SpittrWebAppInitializer extends
   AbstractAnnotationConfigDispatcherServletInitializer {

@Override
   protected String[] getServletMappings() {
      return new String[] { "/" };
   }

   @Override
   protected Class<?>[] getRootConfigClasses() {
      return new Class<?>[] { RootConfig.class };
   }

   @Override
   protected Class<?>[] getServletConfigClasses() {
      return new Class<?>[] { WebConfig.class }; }
   }
}
```

....Odwzorowujemy serwlet dystrybutora na /

Wskazujemy klasę konfiguracji

Za chwilę zagłębimy się w szczegóły listingu 5.1, ale teraz być może zastanawiasz się, co oznacza słowo `spittr`. Utworzona klasa ma nazwę `SpittrWebAppInitializer` i znajduje się w pakiecie `spittr.config`. Za moment (w punkcie 5.1.3) wyjaśnię znaczenie tego słowa, na razie musi Ci wystarczyć informacja, że aplikację nazwiemy Spittr.

Do zrozumienia działania listingu 5.1 musisz wiedzieć tylko tyle, że do konfiguracji serwletu dystrybutora i kontekstu aplikacji Springa w kontekście serwletu wystarczy dowolna klasa rozszerzająca `AbstractAnnotationConfigDispatcherServletInitializer`.

AbstractAnnotationConfigDispatcherServletInitializer — wyjaśnione

Jeśli naprawdę chcesz poznać bardziej szczegółowe wyjaśnienia, oto one. W środowisku Servlet 3.0 kontener szuka w ścieżce klas wszystkich klas implementujących interfejs `javax.servlet.ServletContainerInitializer`. Jeżeli je znajdzie, używa ich do konfiguracji kontenera serwletów.

Spring dostarcza implementację tego interfejsu o nazwie `ServletContainerInitializer`, która z kolei wyszukuje wszystkie klasy implementujące `WebApplicationInitializer` i oddelegowuje do nich pracę związaną z konfiguracją. Spring 3.2 udostępnia wygodną bazową implementację interfejsu `WebApplicationInitializer` pod postacią klasy abstrakcyjnej `Abstract↪AnnotationConfigDispatcherServletInitializer`. Klasa `SpittrWebAppInitializer` rozszerza klasę `AbstractAnnotationConfigDispatcherServletInitializer` (i w ten sposób implementuje interfejs `WebApplicationInitializer`), po wdrożeniu na kontener obsługujący specyfikację Servlet 3.0 zostanie więc automatycznie wyszukana i wykorzystana do konfiguracji kontekstu serwletu.

Pomimo swojej długiej nazwy klasa `AbstractAnnotationConfigDispatcherServletInitializer` jest niezwykle prosta w użyciu. Na listingu 5.1 widzimy, że klasa `SpittrWebAppInitializer` nadpisuje trzy metody.

Pierwsza metoda, `getServletMappings()`, zwraca tablicę złożoną z jednej lub większej liczby ścieżek, którym przypisano odwzorowanie serwletu dystrybutora. W naszym przypadku jest to `/`, co oznacza, że jest to domyślny serwlet w naszej aplikacji. Obsłuży on wszystkie żądania przychodzące do aplikacji.

W celu zrozumienia dwóch innych metod musimy najpierw poznać relację pomiędzy serwletem dystrybutora a serwletem nasłuchującym `ContextLoaderListener`.

HISTORIA DWÓCH KONTEKSTÓW APLIKACJI

Gdy serwlet dystrybutora `DispatcherServlet` rozpoczyna działanie, tworzy kontekst aplikacji Springa i wypełnia go komponentami zadeklarowanymi we wskazanych klasach lub plikach konfiguracyjnych. Na listingu 5.1 wykorzystaliśmy metodę `getServletConfig↪Classes()`, aby przy pomocy serwletu dystrybutora załadować do kontekstu aplikacji komponenty zdefiniowane w klasie konfiguracji `WebConfig` (używając konfiguracji JavaConfig).

W aplikacjach internetowych Springa dostępny jest też często inny kontekst aplikacji. Kontekst ten tworzony jest za pomocą serwletu nasłuchującego `ContextLoaderListener`. Zadaniem serwletu dystrybutora jest załadowanie komponentów internetowych, takich jak kontrolery, producenci widoków oraz odwzorowania obsługi. Serwlet nasłuchujący

odpowiada natomiast za wczytywanie pozostałych komponentów. Są to z reguły komponenty warstwy pośredniej oraz warstwy danych, odpowiadające za funkcjonowanie warstwy logicznej aplikacji.

Zarówno serwlet dystrybutora `DispatcherServlet`, jak i serwlet nasłuchujący `Context➥LoaderListener` tworzone są przez klasę `AbstractAnnotationConfigDispatcherServlet➥Initializer`. Klasy konfiguracji opatrzone adnotacją `@Configuration` zwrócone przez metodę `getServletConfigClasses()` definiują komponenty dla kontekstu aplikacji serwletu dystrybutora. Równocześnie klasy opatrzone adnotacją `@Configuration` zwrócone przez metodę `getRootConfigClasses()` służą do konfiguracji kontekstu aplikacji dla serwletu nasłuchującego `ContextLoaderListener`.

W tym przypadku konfiguracja główna została zdefiniowana w pliku `RootConfig`, a deklaracja konfiguracji serwletu dystrybutora w klasie `WebConfig`. Za chwilę się dowiesz, jak wyglądają obie te klasy.

Musisz pamiętać, że konfiguracja serwletu dystrybutora za pomocą klasy `Abstract➥AnnotationConfigDispatcherServletInitializer` jest alternatywą dla tradycyjnych plików *web.xml*. Moglibyśmy równocześnie korzystać zarówno z konfiguracji w pliku *web.xml*, jak i podklasy `AbstractAnnotationConfigDispatcherServletInitializer`, ale nie jest to konieczne.

Możliwość konfiguracji serwletu dystrybutora za pomocą klasy zamiast pliku web.xml uzależniona jest jedynie od serwera, na który wdrażamy aplikację. Musi on wspierać serwlety w wersji 3.0. Takim serwerem jest przykładowo Apache Tomcat w wersji 7.0 lub nowszej. Ostateczna wersja specyfikacji Servlet 3.0 powstała w grudniu 2009 roku, istnieje więc duże prawdopodobieństwo, że korzystasz już z odpowiedniego kontenera serwletów.

Jeśli jednak Twój kontener nie wspiera serwletów w wersji 3.0, konfiguracja serwletu dystrybutora z użyciem podklasy `AbstractAnnotationConfigDispatcherServletInitializer` nie będzie możliwa. Jedyną możliwością jest wtedy wykorzystanie pliku *web.xml*. Konfiguracji za pomocą pliku *web.xml* oraz innym opcjom konfiguracji przyjrzymy się bliżej w rozdziale 7. Na tę chwilę zajmiemy się klasami konfiguracji `WebConfig` i `RootConfig`, z których skorzystaliśmy na listingu 5.1, i dowiesz się, jak za ich pomocą włączyć obsługę Spring MVC.

WŁĄCZAMY SPRING MVC

Istnieje kilka sposobów konfigurowania serwletu dystrybutora, mamy też więcej niż jeden sposób na włączenie komponentów Spring MVC. W projektach, zwłaszcza tych starszych, wykorzystujących konfigurację opartą na plikach XML do włączania obsługi adnotacji Spring MVC służy element `<mvc:annotation-drive>`.

O elemencie `<mvc:annotation-drive>` i innych opcjach konfiguracji Spring MVC dowiesz się więcej w rozdziale 7. Teraz jednak skorzystamy z prostej konfiguracji Spring MVC, działającej w oparciu o klasy Javy.

Najprostszym sposobem konfiguracji Spring MVC jest utworzenie klasy opatrzonej adnotacją `@EnableWebMvc`:

```
package spittr.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@EnableWebMvc
public class WebConfig {
}
```

Konfiguracja ta zadziała i włączy Spring MVC. Nie jest jednak zbyt użyteczna:

- Producent widoków nie jest skonfigurowany. W ten sposób wykorzystany zostanie domyślny producent widoków `BeanNameViewResolver`. Tworzy on widoki w oparciu o wyszukane komponenty implementujące interfejs `View`, których identyfikator odpowiada nazwie widoku.
- Skanowanie komponentów nie jest włączone. W rezultacie jedynym sposobem na to, by Spring znalazł jakiekolwiek kontrolery, jest ich jawne zadeklarowanie w konfiguracji.
- W obecnej postaci serwlet dystrybutora odwzorowywany jest jako domyślny serwlet aplikacji obsługujący *wszystkie* żądania, także te o statycznych zasobach typu obrazek i arkusz kalkulacyjny (co w większości przypadków nie jest najlepszym pomysłem).

Żeby klasa `WebConfig` stała się użyteczna, musimy do niej dodać minimalną konfigurację Spring MVC. Nowa wersja klasy `WebConfig` znajduje się na listingu 5.2.

Listing 5.2. Minimalna użyteczna konfiguracja Spring MVC

```
package spittr.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.
   DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.
   WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.
   InternalResourceViewResolver;

@Configuration
@EnableWebMvc <-- Włączamy Spring MVC
@ComponentScan("spitter.web") <-- Włączamy skanowanie komponentów
public class WebConfig extends WebMvcConfigurerAdapter {
   @Bean
   public ViewResolver viewResolver() {
      InternalResourceViewResolver resolver = <-- Konfigurujemy producenta widoków JSP
         new InternalResourceViewResolver();
      resolver.setPrefix("/WEB-INF/views/");
      resolver.setSuffix(".jsp");
      resolver.setExposeContextBeansAsAttributes(true); return resolver;
```

```
    }

    @Override
    public void configureDefaultServletHandling (
        DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}
```

Konfigurujemy obsługę statycznych zasobów

Pierwszą rzeczą, na jaką możesz zwrócić uwagę, patrząc na listing 5.2, jest fakt, że klasa `WebConfig` została opatrzona adnotacją `@ComponentScan`. Dzięki temu włączone zostało skanowanie pakietu `spitter.web` w poszukiwaniu komponentów. Wkrótce zobaczysz, że tworzone kontrolery opatrzymy adnotacją `@Controller`, dzięki czemu też będą kandydatami do skanowania komponentów. W rezultacie nie musimy w jawny sposób deklarować żadnych kontrolerów w klasach konfiguracji.

Następnie dodajemy komponent `ViewResolver`. Mówiąc dokładnie, jest to obiekt klasy `InternalResourceViewResolver`. W rozdziale 6. powiem więcej na temat producentów widoku. W tej chwili musisz wiedzieć tylko tyle, że producent widoku służy do wyszukiwania plików JSP poprzez dodawanie prefiksu i sufiksu do nazw widoków (na przykład widok o nazwie `home` zostanie odczytany jako */WEB-INF/views/home.jsp*).

Na koniec klasa `WebConfig` rozszerza klasę `WebMvcConfigurerAdapter` i nadpisuje jej metodę `configureDefaultServletHandling()`. Wywołanie metody `enable()` na danym obiekcie `DefaultServletHandlerConfigurer` spowoduje, że żądania do zasobów statycznych nie będą już obsługiwane, a zostaną przesłane do domyślnego serwletu kontenera serwletów.

Po ustawieniu konfiguracji `WebConfig` zastanówmy się, co z klasą `RootConfig`. W tym rozdziale koncentrujemy się na tworzeniu aplikacji internetowych. Konfiguracja ustawień internetowych zachodzi w kontekście aplikacji utworzonym przez serwlet dystrybutora. W tej chwili pozostawimy więc względnie prostą konfigurację `RootConfig`:

```
package spittr.config;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.ComponentScan.Filter;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@ComponentScan(basePackages={"spitter"},
    excludeFilters={
        @Filter(type=FilterType.ANNOTATION, value=EnableWebMvc.class)
    })
public class RootConfig {
}
```

Jedynym istotnym elementem w klasie `RootConfig` jest adnotacja `@ComponentScan`, którą ta klasa została opatrzona. Na łamach tej książki pojawi się mnóstwo okazji, aby zamieścić w tej klasie różne komponenty niezwiązane z ustawieniami internetowymi.

Jesteśmy już niemal gotowi do rozpoczęcia budowy aplikacji internetowej z wykorzystaniem Spring MVC. Otwartym pytaniem pozostaje, jaką aplikację będziemy tworzyć.

5.1.3. Wprowadzenie do aplikacji Spittr

Podejmujemy próbę wejścia w społecznościową grę sieciową, tworząc prosty mikroblog. Pod wieloma względami nasza aplikacja będzie przypominać swój pierwowzór, Twitter. W trakcie tworzenia aplikacji dodamy do niej pewne elementy. Użyjemy do tego oczywiście Springa.

Zaczerpniemy z Twittera niektóre pomysły i zaimplementujemy je w Springu, nadając naszej aplikacji nazwę roboczą Spitter. Pójdźmy o krok dalej i stosując wzorzec nazewnictwa popularny w serwisach typu Flickr, usuńmy z nazwy literę „e". Otrzymujemy w rezultacie nazwę „Spittr". Ta nazwa będzie też przydatna przy odróżnianiu nazwy aplikacji od klasy domeny, której nadamy nazwę `Spitter`.

Aplikacja Spittr posiada dwie główne koncepcje domeny: **spittersów** (użytkowników aplikacji) i **spittle'e** (krótkie aktualizacje statusu publikowane przez użytkowników). W tej książce przy rozwijaniu funkcjonalności aplikacji będziemy opierać się na tych dwóch koncepcjach. W tym rozdziale zbudujemy warstwę internetową aplikacji, utworzymy kontrolery do wyświetlania spittle'ów i obsłużymy formularze rejestrujące nowych spittersów.

Wszystko jest przygotowane. Skonfigurowaliśmy serwlet dystrybutora, włączyliśmy kluczowe komponenty Spring MVC i zdefiniowaliśmy nasz cel. Przejdźmy teraz do sedna tego rozdziału: obsługi żądań internetowych z użyciem kontrolerów Spring MVC.

5.2. Tworzymy prosty kontroler

W Spring MVC kontrolery są po prostu klasami zawierającymi metody oznaczone adnotacją `@RequestMapping` do określenia rodzajów żądań, które chcemy obsłużyć.

Rozpocznijmy od podstaw i wyobraźmy sobie klasę kontrolera, która obsługuje żądania o stronę / i generuje stronę główną aplikacji. Klasa `HomeController`, widoczna na listingu 5.3, jest przykładem najprostszej klasy kontrolera Spring MVC.

Listing 5.3. HomeController: przykład najprostszego kontrolera

```
package spittr.web;

import static org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller <-------------------------------- Deklarujemy, że klasa ma być kontrolerem
public class HomeController {
   @RequestMapping(value="/", method=GET) <---------- Obsługujemy żądania typu GET na adres /
   public String home() {
              return "home"; <-------------------------------- Nazwą widoku jest home
   }
}
```

Zauważ, że kontroler `HomeController` został opatrzony adnotacją `@Controller`. Choć oczywiste jest, że ta adnotacja służy do deklaracji kontrolera, nie ma ona zbyt wiele wspólnego ze Spring MVC.

`@Controller` jest stereotypem utworzonym w oparciu o adnotację `@Component`. Jego jedynym zadaniem w tym miejscu jest wykorzystanie możliwości skanowania komponentów. Kontroler `HomeController` opatrzony jest adnotacją `@Controller`, więc skaner komponentów automatycznie go wyszuka i zadeklaruje w postaci komponentu w kontekście aplikacji Springa.

Ten sam efekt otrzymalibyśmy więc, oznaczając klasę `HomeController` adnotacją `@Component`, ale wtedy zadanie klasy nie byłoby równie oczywiste.

Jedyna metoda klasy `HomeController`, metoda `home()`, opatrzona została adnotacją `@RequestMapping`. Atrybut `value` określa ścieżkę żądań, które ta metoda ma obsługiwać, a atrybut `method` — obsługiwaną metodę http. W naszym przypadku wywołanie metody `home()` spowodują wszystkie żądania HTTP typu `GET` przychodzące na ścieżkę `/`.

Jak widzimy, metoda `home()` nie robi zbyt wiele: zwraca tylko ciąg znaków `"home"`. Ten ciąg zostanie zinterpretowany przez Spring MVC jako nazwa generowanego widoku. Serwlet dystrybutora zwróci się do producenta widoków z prośbą o odwzorowanie tej nazwy na rzeczywisty widok.

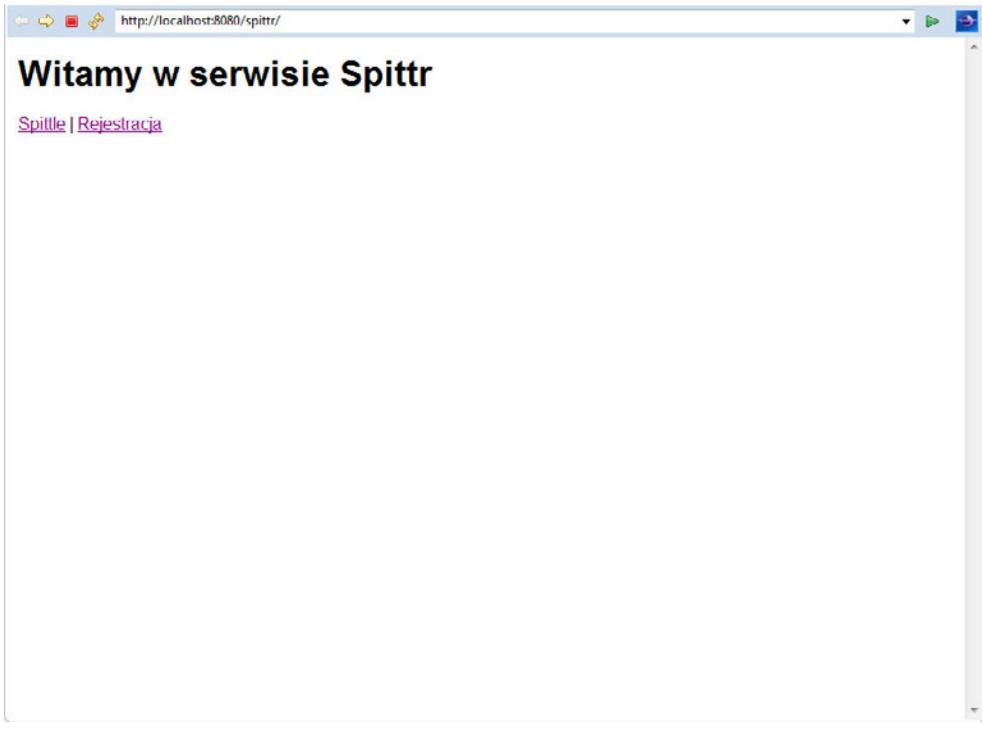
Przy uwzględnieniu ustalonej przez nas konfiguracji z użyciem klasy `InternalResource` ➥`ViewResolver` nazwa widoku `"home"` zostanie rozwiązana jako plik JSP */WEB-INF/views/home.jsp*. Na początku nasza strona domowa będzie raczej prosta, co widać na listingu 5.4.

Listing 5.4. Strona domowa aplikacji Spittr zdefiniowana w postaci prostego pliku JSP

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
   <head>
      <title>Spittr</title>
      <link rel="stylesheet"
            type="text/css"
            href="<c:url value="/resources/style.css" />" >
   </head>
   <body>
      <h1>Witamy w serwisie Spittr</h1>
      <a href="<c:url value="/spittles" />">Spittle</a> |
      <a href="<c:url value="/spitter/register" />">Rejestracja</a>
   </body>
</html>
```

W kodzie tego pliku JSP nie ma nic szczególnie interesującego. Strona wyświetla powitanie użytkownika aplikacji i udostępnia dwa linki: jeden do widoku listy spittle'ów, a drugi do rejestracji w serwisie. Aktualny widok strony głównej jest dostępny na rysunku 5.2.

Przed końcem tego rozdziału zaimplementujemy metody kontrolera obsługujące te żądania. W tej chwili jednak wykonajmy kilka żądań do kontrolera, aby sprawdzić jego działanie. Najbardziej oczywistym sposobem testowania kontrolera wydaje się zbudo-



Rysunek 5.2. Strona domowa aplikacji Spittr

wanie i wdrożenie aplikacji, a następnie ręczne odwiedzenie strony. Lepszym rozwiązaniem jest jednak przygotowanie testu automatycznego, który zwraca wyniki szybciej i w bardziej spójny sposób. Przygotujmy zatem test kontrolera `HomeController`.

5.2.1. Testujemy kontroler

Spójrzmy raz jeszcze na kontroler `HomeController`. Jeśli mocno zmrużysz oczy, tak mocno, że adnotacje przestaną być widoczne, zobaczysz zwykłe POJO. Wiesz już, że testowanie POJO jest proste. Przetestujemy więc kontroler `HomeController` za pomocą prostego testu, zamieszczonego na listingu 5.5.

Listing 5.5. HomeControllerTest: testujemy HomeController

```
package spittr.web;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import spittr.web.HomeController;

public class HomeControllerTest {
   @Test
   public void testHomePage() throws Exception {
      HomeController controller = new HomeController();
      assertEquals("home", controller.home());
   }
}
```

Test na listingu 5 5 jest bardzo prosty — sprawdza jedynie, co się dzieje po wywołaniu metody `home()`. Wywołuje bezpośrednio tę metodę i sprawdza, czy zwracany jest ciąg znaków `"home"`. W żaden sposób nie sprawdza jednak elementów, które czynią metodę `home()` metodą kontrolera. Nie sprawdzamy także w żaden sposób, czy wywołanie metody następuje w wyniku żądania `GET` na ścieżce `/`. Nie weryfikujemy też, czy zwracany przez metodę ciąg `"home"` jest rzeczywiście nazwą widoku.

W wersji 3.2 Springa pojawiła się możliwość sprawdzania kontrolerów Spring MVC jako rzeczywistych kontrolerów, a nie zwykłych plików POJO. Spring pozwala również wykorzystać atrapę (ang. *mocking*) mechanizmów Spring MVC i wywoływania żądań http na kontrolerach. Umożliwia to testowanie kontrolerów bez uruchamiania serwera internetowego i przeglądarki internetowej.

Zademonstrujemy prawidłowy sposób testowania kontrolerów w Spring MVC, przepisując test `HomeControllerTest` tak, aby wykorzystywał nowe funkcje testowania Spring MVC. Listing 5.6 przedstawia taką nową wersję testu.

Listing 5.6. Nowa wersja testu HomeControllerTest

```
package spittr.web;

import static
   org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
   org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
import static
   org.springframework.test.web.servlet.setup.MockMvcBuilders.*;
import org.junit.Test;
import org.springframework.test.web.servlet.MockMvc;
import spittr.web.HomeController;

public class HomeControllerTest {

   @Test
   public void testHomePage() throws Exception {
      HomeController controller = new HomeController();
      MockMvc mockMvc = <-- Ustawiamy atrapę MockMvc
         standaloneSetup(controller).build();
      mockMvc.perform(get("/")) <-- Wywołujemy żądanie GET /
         .andExpect(view().name("home")); <-- Oczekujemy zwrócenia widoku home
      }
}
```

Mimo że nowa wersja testu jest o kilka linii dłuższa od swojego poprzednika, testuje kontroler `HomeController` w sposób bardziej kompletny. Nie wywołuje metody `home()` bezpośrednio, żeby sprawdzić zwróconą wartość. Ta wersja testu wywołuje żądanie `GET` o zasób `/` i sprawdza, czy zwracany jest widok `home`. Rozpoczynamy od przekazania instancji kontrolera `HomeController` do metody `MockMvcBuilders.standaloneSetup()` i wywołania metody `build()` w celu utworzenia instancji atrapy `MockMvc`. Następnie wywoływane jest żądanie `GET` do zasobu `/` i weryfikacja nazwy zwróconego widoku.

5.2.2. Definiujemy obsługę żądań na poziomie klasy

Teraz gdy już mamy test dla kontrolera `HomeController`, możemy przeprowadzić jego modyfikacje bez obawy, że coś się zepsuje. Jedną z rzeczy, które warto zrobić, jest rozbicie adnotacji `@RequestMapping` poprzez umieszczenie odwzorowania ścieżki na poziomie klasy. Na listingu 5.7 pokazuję, jak to wykonać.

Listing 5.7. Rozdzielamy adnotację @RequestMapping w kontrolerze HomeController

```
package spittr.web;
import static org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/") <-- Odwzorowujemy kontroler na /
public class HomeController {
   @RequestMapping(method=GET) <-- Obsługujemy żądania GET
   public String home() {
      return "home"; <-- Nazwą widoku jest home
   }
}
```

W tej nowej wersji kontrolera `HomeController` ścieżka została umieszczona w adnotacji `@RequestMapping` na poziomie klasy, a deklaracja pożądanej metody http pozostała na poziomie metody. Za każdy razem, gdy adnotacja `@RequestMapping` zadeklarowana jest na klasie kontrolera, dotyczy wszystkich zadeklarowanych w tej klasie metod obsługujących żądania. Wszystkie adnotacje `@RequestMapping` zadeklarowane na poziomie metod obsługujących żądania uzupełniają informacje zawarte w adnotacji umieszczonej na poziomie klasy.

W klasie `HomeController` dostępna jest tylko jedna metoda obsługująca żądania. Opatrująca ją adnotacja `@RequestMapping` w połączeniu z adnotacją na poziomie klasy powoduje, że metoda `home()` obsłuży wszystkie żądania `GET` do zasobu `/`.

Innymi słowy, nic się nie zmieniło. Poprzenosiliśmy kilka rzeczy, ale klasa `Home`
`↪Controller` nadal działa tak samo. Utworzyliśmy wcześniej test, możemy się więc upewnić, że nic rzeczywiście nie zepsuliśmy.

Skoro już modyfikujemy ustawienia adnotacji `@RequestMapping`, możemy wykonać jeszcze jedną modyfikację kontrolera `HomeController`. Atrybut `value` przyjmuje wartość typu `String`. Do tej pory podawaliśmy jednak tylko jedną wartość `"/"`. Ale możemy podać kolejną wartość ścieżki `/homepage`, do której mają być odwzorowywane żądania poprzez zmianę adnotacji `@RequestMapping` na poziomie klasy:

```
@Controller
@RequestMapping({"/", "/homepage"})
public class HomeController {
   ...
}
```

Teraz metoda `home()` kontrolera `HomeController` wywoływana jest przez wszystkie żądania `GET` do zasobów `/` oraz `/homepage`.

5.2.3. Przekazujemy dane modelu do widoku

W tej chwili kontroler `HomeController` stanowi doskonały przykład bardzo prostego kontrolera. Większość kontrolerów nie jest jednak taka prosta. W aplikacji Spittr potrzebna nam będzie strona wyświetlająca listę ostatnio dodanych spittle'ów. Potrzebna nam więc będzie nowa metoda do jej obsługi.

Na początek zdefiniujemy repozytorium dostępu do danych. Aby odseparować dane i nie zagłębiać się w szczegóły dotyczące bazy danych, zdefiniujemy repozytorium w postaci interfejsu, a implementacją zajmiemy się później (w rozdziale 10.). W tym momencie potrzebny jest nam tylko interfejs do pobierania listy spittle'ów. Dobrym punktem startu jest zdefiniowany poniżej interfejs `SpittleRepository`:

```
package spittr.data;
import java.util.List;
import spittr.Spittle;

public interface SpittleRepository {
   List<Spittle> findSpittles(long max, int count);
}
```

Metoda `findSpittles()` przyjmuje dwa parametry. Parametr `max` reprezentuje maksymalny identyfikator spittle'a spośród tych, które mają zostać zwrócone. Parametr `count` wskazuje z kolei, ile obiektów typu `Spittle` ma zostać zwróconych. Aby otrzymać listę 20 ostatnich obiektów typu `Spittle`, możemy wywołać metodę `findSpittles()`:

```
List<Spittle> recent =
   spittleRepository.findSpittles(Long.MAX_VALUE, 20);
```

Klasa `Spittle`, którą zaraz zdefiniujemy, będzie na początku bardzo prosta. Zawierać będzie właściwości do przechowywania wiadomości, znacznika czasu oraz lokalizacji w postaci szerokości i długości geograficznej, z której spittle został wysłany (listing 5.8).

Listing 5.8. Klasa Spittle: przechowuje wiadomość, znacznik czasu oraz lokalizację

```
package spittr;
import java.util.Date;

public class Spittle {
   private final Long id;
   private final String message;
   private final Date time;
   private Double latitude;
   private Double longitude;

   public Spittle(String message, Date time) {
      this(message, time, null, null);
   }

   public Spittle(
      String message, Date time, Double longitude, Double latitude) {
      this.id = null;
      this.message = message;
      this.time = time;
      this.longitude = longitude;
```

```
        this.latitude = latitude;
    }

    public long getId() {
        return id;
    }

    public String getMessage() {
        return message;
    }

    public Date getTime() {
        return time;
    }

    public Double getLongitude() {
        return longitude;
    }

    public Double getLatitude() {
        return latitude;
    }

    @Override
    public boolean equals(Object that) {
        return EqualsBuilder.reflectionEquals(this, that, "id", "time");
    }

    @Override public int hashCode() {
        return HashCodeBuilder.reflectionHashCode(this, "id", "time");
    }
}
```

Klasa `Spittle` to przede wszystkim POJO — nie jest to więc nic skomplikowanego. Jedyną godną uwagi rzeczą jest wykorzystanie biblioteki Apache Commons Lang w celu utworzenia prostej implementacji metod `equals()` i `hashCode()`. Poza tym, że mają one ogólne walory użytkowe, przydadzą się też w pisaniu testów do metod obsługujących żądania w kontrolerze.

Skoro już jesteśmy przy temacie testowania, pójdźmy do przodu i napiszmy test dla naszej nowej metody kontrolera. Listing 5.9 wykorzystuje springową klasę `MockMvc` do weryfikacji zachowania nowej metody obsługi żądań.

Listing 5.9. Testujemy, czy kontroler SpittleController obsługuje żądania GET do zasobu /spittles

```
@Test
public void shouldShowRecentSpittles() throws Exception {
    List<Spittle> expectedSpittles = createSpittleList(20);
    SpittleRepository mockRepository =                          <-- Tworzymy atrapę repozytorium
        mock(SpittleRepository.class);
    when(mockRepository.findSpittles(Long.MAX_VALUE, 20))
        .thenReturn(expectedSpittles);

    SpittleController controller = new SpittleController(mockRepository);
```

```
    SpittleController controller = new SpittleController(mockRepository);
    MockMvc mockMvc = standaloneSetup(controller)          <-- Tworzymy atrapę Spring MVC
        .setSingleView(
            new InternalResourceView("/WEB-INF/views/spittles.jsp"))
        .build();
    mockMvc.perform(get("/spittles"))                      <-- Pobieramy listę spittle'ów
        .andExpect(view().name("spittles"))
        .andExpect(model().attributeExists("spittleList"))
        .andExpect(model().attribute("spittleList",        <-- Weryfikujemy wyniki
            hasItems(expectedSpittles.toArray())));
}

...

private List<Spittle> createSpittleList(int count) {
    List<Spittle> spittles = new ArrayList<Spittle>();
    for (int i=0; i < count; i++) {
        spittles.add(new Spittle("Spittle " + i, new Date()));
    }
    return spittles;
}
```

Test rozpoczynamy od utworzenia atrapy implementacji interfejsu `SpittleRepository`, która zwróci listę 20 obiektów typu `Spittle`, pobranych za pomocą metody `findSpit➥tles()`. Następnie wstrzykujemy to repozytorium do nowej instancji kontrolera `Spittle➥Controller` i przygotowujemy instancję `MockMvc` do wykorzystania w tym kontrolerze.

Warto zauważyć, że w przeciwieństwie do testu `HomeControllerTest` ten test wywołuje metodę `setSingleVew` na klasie budującej `MockMvc`. W ten sposób framework obsługujący mechanizm atrap nie będzie próbował samodzielnie rozwiązywać nazwy widoku zwróconej z kontrolera. W wielu przypadkach nie jest to konieczne. Ale w tej metodzie kontrolera nazwa widoku będzie podobna do ścieżki żądania. Domyślne działanie spowoduje wystąpienie błędu, bo ścieżka do widoku będzie mylona ze ścieżką do kontrolera. Rzeczywista ścieżka powstała przy konstruowaniu `InternalResourceView` nie jest w tym teście istotna, ustawiamy ją jednak, aby zachować spójność z konfiguracją producenta widoków `InternalResourceViewResolver`.

Test kończy się wywołaniem żądania `GET` do zasobu `/spittles` i weryfikacją, że nazwą widoku jest `spittles`, a model posiada atrybut o nazwie `spittleList` o oczekiwanej zawartości.

Oczywiście uruchomienie testu w tym momencie spowoduje wystąpienie błędu. Test nie tylko nie zadziała, ale się nawet nie skompiluje. Nie przygotowaliśmy jeszcze bowiem kontrolera `SpittleController`. Utwórzmy ten kontroler, żeby spełniał oczekiwania wyrażone w teście z listingu 5.9. Przykładowa implementacja kontrolera `Spittle➥Controller` znajduje się poniżej, na listingu 5.10.

Listing 5.10. SpittleController: umieszczamy listę ostatnich spittle'ów w modelu

```
package spittr.web;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RequestMethod;
import spittr.Spittle;
import spittr.data.SpittleRepository;

@Controller
@RequestMapping("/spittles")
public class SpittleController {

   private SpittleRepository spittleRepository;

   @Autowired
          public SpittleController(                    <-- Wstrzykujemy repozytorium SpittleRepository
      SpittleRepository spittleRepository) {
      this.spittleRepository = spittleRepository;
   }

   @RequestMapping(method=RequestMethod.GET)
   public String spittles(Model model) {
      model.addAttribute(
         spittleRepository.findSpittles(           <-- Dodajemy listę spittle'ów do modelu
            Long.MAX_VALUE, 20));
      return "spittles";                             <-- Zwracamy nazwę widoku
   }
}
```

Jak widzisz, kontroler `SpittleController` posiada konstruktor opatrzony adnotacją `@Auto`
`↪wired`, który oczekuje na wstrzyknięcie repozytorium `SpittleRepository`. To repozytorium jest potem wykorzystywane w metodzie `spittles()` do pobrania listy ostatnich spittle'ów.

Zauważ, że metoda `spittles()` przyjmuje obiekt typu `Model` jako parametr. Dzięki temu zabiegowi metoda może wypełnić wstrzyknięty model listą obiektów klasy `Spittle` pobranych z repozytorium. Klasa `Model` jest w praktyce mapą (kolekcją par klucz-wartość) przekazywaną do widoku, co umożliwia wyświetlenie listy obiektów użytkownikom aplikacji. Kiedy metoda `addAttribute()` jest wywoływana bez określenia klucza, klucz tworzony jest automatycznie na podstawie typu obiektu przechowywanego jako jego wartość. W naszym przykładzie typem obiektu jest `List<Spittle>`, nazwą klucza będzie więc `spittleList`.

Ostatnią czynnością wykonywaną przez metodę `spittles()` jest zwrócenie ciągu znaków `spittles`, reprezentującego nazwę widoku, w którym mają być wyświetlone dane zapisane w modelu.

Możemy też oczywiście ustalić klucz modelu w sposób jawny. Przykładowo poniższa wersja metody `spittles()` jest odpowiednikiem metody zdefiniowanej na listingu 5.10:

```
@RequestMapping(method=RequestMethod.GET)
public String spittles(Model model) {
   model.addAttribute("spittleList",
      spittleRepository.findSpittles(Long.MAX_VALUE, 20));
   return "spittles";
}
```

W metodzie możemy też wykorzystać typ niezwiązany ze Springiem i w miejscu klasy `Model` zastosować mapę `java.util.Map`. Poniżej znajduje się kolejna, równoznaczna z poprzednimi, wersja metody `spittles()`:

```
@RequestMapping(method=RequestMethod.GET)
public String spittles(Map model) {
   model.put("spittleList",
      spittleRepository.findSpittles(Long.MAX_VALUE, 20));
   return "spittles";
}
```

Świetnie nam idzie tworzenie kolejnych implementacji metody `spittles()`, wprowadźmy więc jeszcze jedną zmianę:

```
@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles() {
   return spittleRepository.findSpittles(Long.MAX_VALUE, 20));
}
```

Ta wersja różni się trochę od poprzednich. Nie zwracamy w niej nazwy widoku ani nie ustawiamy widoku w sposób jawny, a zamiast tego zwracamy listę obiektów typu `Spittle`. Kiedy metoda obsługująca żądania zwraca obiekt lub kolekcję obiektów, zwracana wartość umieszczana jest w modelu, a klucz modelu ustalany jest na podstawie zwracanego typu (w naszym przykładzie nazwą klucza jest ponownie `spittleList`).

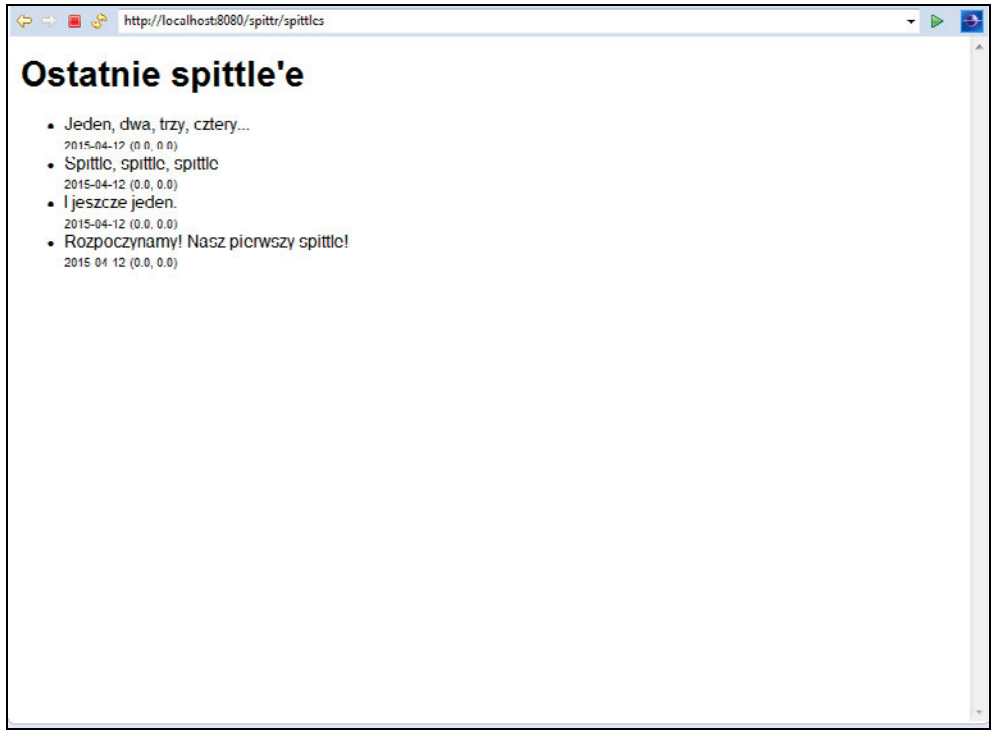
Nazwa widoku ustalana jest w oparciu o ścieżkę żądania. Metoda ta obsługuje żądania `GET` do zasobu `/spittles`, nazwą widoku jest więc `spittles` (slash znajdujący się na początku ścieżki jest usuwany).

Niezależnie od wybranej implementacji metody `spittles()` wynik jest zawsze ten sam. Lista obiektów typu `Spittle` zapisywana jest w modelu pod kluczem `spittleList` i przekazywana do widoku o nazwie `spittles`. Biorąc pod uwagę ustawioną konfigurację producenta `InternalResourceViewResolver`, widokiem tym jest szablon JSP zapisany w pliku */WEB-INF/views/spittles.jsp*.

Teraz gdy dane są już zapisane w modelu, w jaki sposób możemy się do nich odwołać w pliku JSP? Okazuje się, że w przypadku widoków JSP dane modelu kopiowane są do żądania w postaci atrybutów. Dzięki temu plik `spittles.jsp` pozwala wyświetlić listę spittle'ów za pomocą znacznika `<c:forEach>`, pochodzącego z biblioteki znaczników JSTL (*JavaServer Pages Standard Library*):

```
<c:forEach items="${spittleList}" var="spittle" >
   <li id="spittle_<c:out value="spittle.id"/>">
      <div class="spittleMessage">
         <c:out value="${spittle.message}" />
      </div>
      <div>
         <span class="spittleTime"><c:out value="${spittle.time}" /></span>
         <span class="spittleLocation">
            (<c:out value="${spittle.latitude}" />,
            <c:out value="${spittle.longitude}" />)
         </span>
      </div>
   </li>
</c:forEach>
```

Rysunek 5.3 przedstawia wygląd tej strony w przeglądarce.



Rysunek 5.3. Utworzony w kontrolerze model danych Spittle udostępniany jest w postaci parametrów żądania i wyświetlany w postaci listy w oknie przeglądarki

Chociaż kontroler `SpittleController` jest nadal prosty, stanowi duży krok naprzód w stosunku do kontrolera `HomeController`. Żaden z tych kontrolerów nie zawiera jednak obsługi formularza. Rozszerzmy kontroler `SpittleController` o obsługę danych wejściowych przesłanych przez klienta.

5.3. Obsługujemy dane wejściowe

Niektóre aplikacje służą tylko do odczytu. Użytkownicy chodzą po stronie w oknach przeglądarek i czytają zawartość wysłaną przez serwer.

Na szczęście nie jest to jedyny scenariusz. Wiele aplikacji daje swoim użytkownikom możliwość interakcji ze stroną i wysyłania danych z powrotem na serwer. Bez tej możliwości internet nie byłby tym samym miejscem.

Spring MVC udostępnia kilka sposobów przekazywania danych do metody obsługującej żądania w kontrolerze, takich jak:

- parametry zapytania,
- parametry formularza,
- zmienne ścieżki.

Za chwilę poznasz metody tworzenia kontrolerów z wykorzystaniem wszystkich tych mechanizmów obsługi danych wejściowych. Rozpoczniemy od parametrów zapytania, najprostszego sposobu przesyłania danych na serwer przez użytkownika aplikacji.

5.3.1. Pobieramy parametry zapytania

Jednym z zadań aplikacji Spittr jest wyświetlenie listy spittle'ów. W tej chwili kontroler `SpittleController` wyświetla jedynie listę ostatnich spittle'ów. Nie udostępnia możliwości poruszania się po historii utworzonych spittle'ów. Jeśli chcemy zaoferować użytkownikom taką możliwość, musimy poznać sposób przekazywania parametrów, co pozwoli nam wskazać zbiór wyświetlanych spittle'ów.

Musimy zdecydować, jak chcemy zrealizować to zadanie. Załóżmy, że przeglądamy stronę z listą spittle'ów posortowaną od najnowszych do najstarszych. Wtedy pierwszy spittle na następnej stronie powinien mieć identyfikator *mniejszy* od identyfikatora ostatniego spittle'a na bieżącej stronie. Zatem aby wyświetlić następną stronę ze spittle'ami, musimy przekazać identyfikator bezpośrednio mniejszy od ostatniego spittle'a na aktualnej stronie. Możemy też przekazać parametr z liczbą splittle'ów wyświetlanych na stronie.

Do implementacji stronicowania potrzebna nam będzie metoda obsługująca żądanie, która przyjmuje następujące parametry:

- `before` (wskazujący identyfikator obiektu `Spittle`, większy od identyfikatorów wszystkich zwróconych wyników);
- `count` (wskazujący liczbę zwracanych spittle'ów).

W tym celu zamienimy implementację metody `spittles()` utworzoną na listingu 5.10 na nową wersję metody zawierającą obsługę parametrów `before` i `count`. Rozpoczniemy od dodania testu metody `spittles()` uwzględniającego te nowe funkcjonalności, co pokazuję na listingu 5.11.

Listing 5.11. Nowa metoda testowania stronicowanej listy spittle'ów

```
@Test
public void shouldShowPagedSpittles() throws Exception {
   List<Spittle> expectedSpittles = createSpittleList(50);
   SpittleRepository mockRepository =
      mock(SpittleRepository.class);
   when(mockRepository.findSpittles(238900, 50))   <-- Oczekujemy parametrów max i count
      .thenReturn(expectedSpittles);
   SpittleController controller =
      new SpittleController(mockRepository);
   MockMvc mockMvc = standaloneSetup(controller)
      .setSingleView(
         new InternalResourceView("/WEB-INF/views/spittles.jsp"))
      .build();
   mockMvc.perform(get("/spittles?max=238900&count=50"))   <-- Przekazujemy parametry max i count
      .andExpect(view().name("spittles"))
      .andExpect(model().attributeExists("spittleList"))
      .andExpect(model().attribute("spittleList",
         hasItems(expectedSpittles.toArray())));
}
```

Podstawowa różnica pomiędzy tą metodą testową a wersją z listingu 5.9 polega na tym, że wysyła ona żądanie GET o zasób /spittles, przekazując równocześnie wartości parametrów max i count. Test sprawdza działanie metody obsługującej żądanie, gdy oba parametry są ustawione, a wcześniejszy sprawdzał działanie metody bezparametrowej. Teraz gdy oba testy są już gotowe, możemy dowolnie modyfikować kontroler bez obawy o to, że któryś ze zdefiniowanych scenariuszy przestanie działać:

```
@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles(
   @RequestParam("max") long max,
   @RequestParam("count") int count) {
   return spittleRepository.findSpittles(max, count);
}
```

Jeśli metoda spittles() w kontrolerze SpittleController ma obsługiwać zarówno żądania zawierające parametry max i count, jak i te ich niezawierające, musimy ją zmodyfikować tak, aby przyjmując parametry, nadawała im też domyślne wartości Long.MAX_
➥VALUE i 20, gdy nie zostaną w sposób jawny podane. W tym celu wykorzystamy atrybut defaultValue adnotacji @RequestParam:

```
@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles(
   @RequestParam(value="max", defaultValue=MAX_LONG_AS_STRING) long max,
   @RequestParam(value="count", defaultValue="20") int count) {
   return spittleRepository.findSpittles(max, count);
}
```

Teraz, jeżeli nie ustawimy parametru max, przyjmie on jako wartość domyślną maksymalną liczbę typu Long. Parametrami zapytań są zawsze wartości typu String, atrybut defaultValue musi więc również otrzymać wartość tego typu. Nie możemy zatem wykorzystać wartości Long.MAX_VALUE, a zamiast tego zdefiniujemy stałą typu String, w której zapiszemy wartość Long.MAX_VALUE w postaci ciągu znaków. Zmienną tę nazwiemy MAX_
➥LONG_AS_STRING:

```
private static final String MAX_LONG_AS_STRING = Long.toString(Long.MAX_VALUE);
```

Atrybut defaultValue ma typ String, ale przy wiązaniu do parametru max metody spittles() zostanie skonwertowany na wartość typu Long.

Gdy wartość parametru count nie zostanie przekazana, ustawiana jest jej domyślna wartość, w naszym przypadku równa 20.

Parametry zapytań są popularną metodą przekazywania informacji do kontrolera w żądaniu. Innym często spotykanym sposobem, zwłaszcza w kontrolerach zorientowanych na dostęp do zasobów, jest przekazywanie parametrów jako elementu ścieżki żądania. Zobaczmy, jak wykorzystać zmienne ścieżki żądania do pobrania danych wejściowych.

5.3.2. Pobieramy dane wejściowe za pośrednictwem parametrów ścieżki

Załóżmy, że aplikacja ma dawać możliwość wyświetlania pojedynczego spittle'a, wybranego na podstawie identyfikatora. Jednym z rozwiązań jest utworzenie metody obsługi żądania, opatrzonej adnotacją `@RequestMapping` i przyjmującej ten identyfikator w postaci parametru zapytania:

```
@RequestMapping(value="/show", method=RequestMethod.GET)
public String showSpittle(
   @RequestParam("spittle_id") long spittleId,
   Model model) {
   model.addAttribute(spittleRepository.findOne(spittleId));
   return "spittle";
}
```

Ta metoda obsłuży żądania takie jak `/spittles/show?spittle_id=12345`. Rozwiązanie takie działa, nie jest jednak idealne z perspektywy zarządzania zasobami. Dużo lepszym sposobem jest identyfikacja zasobu w oparciu o ścieżkę URL, a nie o parametry zapytania. Ogólną zasadą jest niestosowanie parametrów zapytań do identyfikacji zasobów. Żądanie `GET` do zasobu `/spittles/12345` jest lepsze od odwołania `/spittles/show?spittle_id=12345`. Pierwsze odwołanie wskazuje zasób, który ma zostać pobrany. Drugie określa operację z użyciem parametru — co jest w zasadzie wywołaniem RPC za pośrednictwem HTTP.

Pamiętając o zasadzie zorientowania na zasoby, przygotujmy test uwzględniający to wymaganie. Na listingu 5.12 znajduje się nowy test, który sprawdza obsługę żądań zorientowanych na zasoby w kontrolerze `SpittleController`.

Listing 5.12. Testujemy żądanie pobrania spittle'a w oparciu o identyfikator w zmiennej ścieżki

```
@Test
public void testSpittle() throws Exception {
Spittle expectedSpittle = new Spittle("Hello", new Date());
SpittleRepository mockRepository = mock(SpittleRepository.class);
when(mockRepository.findOne(12345)).thenReturn(expectedSpittle);

SpittleController controller = new SpittleController(mockRepository);
MockMvc mockMvc = standaloneSetup(controller).build();

mockMvc.perform(get("/spittles/12345"))   <-- Żądanie pobrania zasobu na podstawie ścieżki
   .andExpect(view().name("spittle"))
   .andExpect(model().attributeExists("spittle"))
   .andExpect(model().attribute("spittle", expectedSpittle));
}
```

W teście ustawiamy atrapę repozytorium, kontroler i obiekt `MockMvc`, podobnie jak w innych testach zamieszczonych w tym rozdziale. Najważniejszym elementem tego testu jest kilka ostatnich linii, w których wywoływane jest żądanie `GET` do zasobu `/spittles/12345` i następuje weryfikacja, czy nazwą widoku jest `spittle` oraz czy przekazywany jest do niego oczekiwany obiekt `Spittle`. Test nie powiedzie się, ponieważ nie

utworzyliśmy jeszcze metody obsługującej żądania tego typu. Możemy to jednak naprawić, dodając nową metodę do kontrolera `SpittleController`.

Jak do tej pory wszystkie metody kontrolera mapowane są (poprzez adnotację `@RequestMapping`) do statycznie zdefiniowanej ścieżki. Jeśli chcesz, aby ten test przeszedł, musisz utworzyć adnotację `@RequestMapping`, zawierającą zmienną ścieżki reprezentującą identyfikator spittle'a.

Spring MVC umożliwia pracę ze zmiennymi ścieżki dzięki użyciu symboli zastępczych w adnotacji `@RequestMapping`. Symbolami zastępczymi są nazwy otoczone przez pary nawiasów klamrowych ({ i }). Symbole zastępcze mogą zawierać dowolne wartości, podczas gdy inne elementy ścieżki muszą dokładnie pasować do obsługiwanego żądania.

Poniższa metoda wykorzystuje symbole zastępcze do przyjęcia identyfikatora spittle'a jako elementu ścieżki:

```
@RequestMapping(value="/{spittleId}", method=RequestMethod.GET)
public String spittle(
   @PathVariable("spittleId") long spittleId,
   Model model) {
   model.addAttribute(spittleRepository.findOne(spittleId));
   return "spittle";
}
```

Może na przykład obsłużyć żądanie do zasobu `/spittles/12345` ścieżki testowanej na listingu 5.12.

Metoda `spittle()` zawiera parametr `spittleId`, opatrzony adnotacją `@PathVariable ➥("spittleId")`. Oznacza to, że dowolna wartość wstawiona w miejsce symbolu zastępczego ścieżki zostanie przekazana do parametru `spittleId` metody obsługującej żądanie. Jeśli żądanie korzysta z metody `GET`, aby odwołać się do zasobu `/spittles/54321`, to wartością `spittleId` stanie się `54321`.

Zauważ, że fraza `spittleId` jest w powyższym przykładzie powtarzana kilkakrotnie: w ścieżce adnotacji `@RequestMapping`, w atrybucie `value` adnotacji `@PathVariable` i ponownie jako nazwa parametru metody. W sytuacji, gdy nazwa parametru metody jest taka sama jak nazwa symbolu zastępczego, możemy wykorzystać ustawienia domyślne i pominąć wartość atrybutu `value` w adnotacji `@PathVariable`:

```
@RequestMapping(value="/{spittleId}", method=RequestMethod.GET)
public String spittle(@PathVariable long spittleId, Model model) {
   model.addAttribute(spittleRepository.findOne(spittleId));
   return "spittle";
}
```

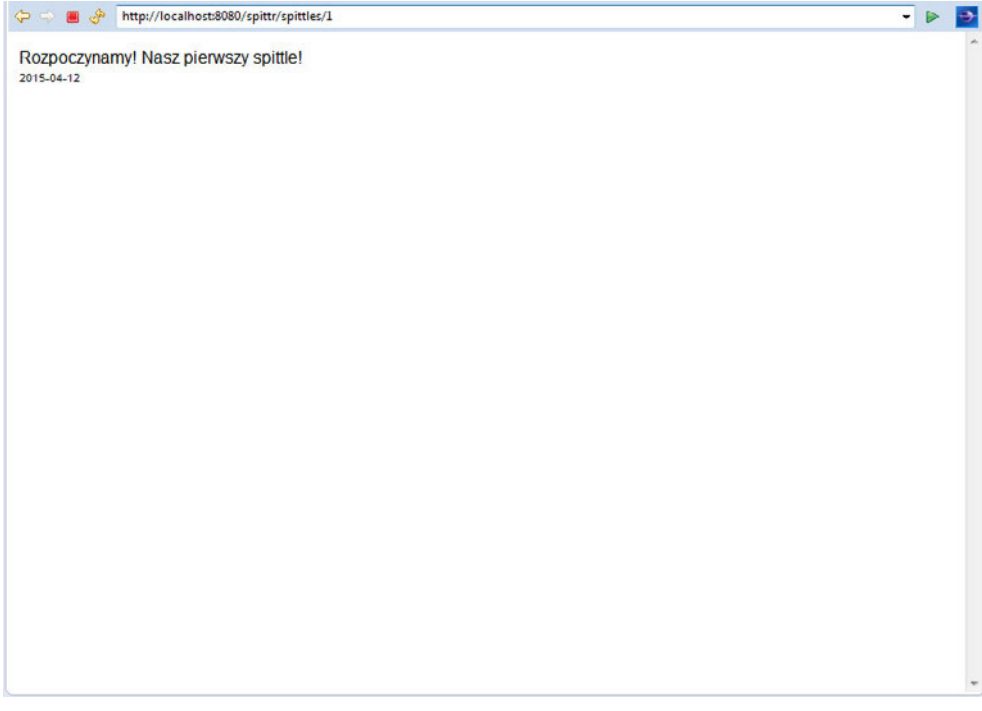
Do adnotacji `@PathVariable` nie przekazano atrybutu `value`, Spring zakłada więc, że nazwa symbolu zastępczego jest taka sama jak nazwa parametru metody. Dzięki temu zabiegowi kod jest prostszy, bo nie duplikujemy nazwy symbolu zastępczego więcej razy, niż jest to niezbędne. Uważaj jednak: jeżeli postanowisz zmienić nazwę parametru, musisz również zmienić nazwę powiązanego z nim symbolu zastępczego.

Metoda `spittle()` przekazuje parametr do metody `findOne()` repozytorium `Spittle ➥Repository`, aby odnaleźć pojedynczy obiekt `Spittle` i dodać go do modelu. Kluczem modelu jest `spittle` ustalany na podstawie typu przekazanego do metody `addAttribute()`.

Dane w obiekcie `Spittle` mogą być generowane w widoku za pośrednictwem klucza `spittle` atrybutu request (klucza o takiej samej nazwie, jaką ma klucz dostępny w modelu). Oto fragment widoku JSP, który służy do wyświetlenia spittle'a:

```
<div class="spittleView">
   <div class="spittleMessage"><c:out value="${spittle.message}" /></div>
   <div>
      <span class="spittleTime"><c:out value="${spittle.time}" /></span> </div>
   </div>
```

Na rysunku 5.4 możesz zobaczyć, że widok ten nie zawiera nic niezwykłego.



Rysunek 5.4. Wyświetlamy spittle'a w przeglądarce

Parametry zapytania i parametry ścieżki sprawdzają się doskonale do przekazywania w żądaniu małej ilości danych. Często jednak potrzebujemy przekazać dużo danych (na przykład pochodzących z formularza), a wtedy parametry zapytania okazują się niezręczne w użyciu i zbyt ograniczone. Zobaczmy, jak tworzyć metody kontrolera do obsługi danych wysłanych z formularza.

5.4. Przetwarzamy formularze

Rola aplikacji internetowych nie kończy się z reguły na wypychaniu zawartości do użytkownika. Większość aplikacji pozwala użytkownikom na interakcję z aplikacją poprzez wypełnianie formularzy i przesyłanie danych za ich pomocą. Kontrolery MVC Springa doskonale nadają się do przetwarzania formularzy i serwowania zawartości.

Praca z formularzem składa się z dwóch etapów: wyświetlania formularza (listing 5.13) i przetwarzania danych przesłanych przez użytkownika za ich pomocą. W aplikacji Spittr potrzebny nam będzie formularz rejestracji nowego użytkownika. Utworzymy nowy kontroler `SpitterController` z pojedynczą metodą obsługującą żądanie wyświetlania formularza rejestracji.

Listing 5.13. SpitterController: wyświetlamy formularz, aby umożliwić użytkownikom rejestrację w aplikacji

```
package spittr.web;
import static org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import spittr.Spitter;
import spittr.data.SpitterRepository;

@Controller
@RequestMapping("/spitter")
public class SpitterController {
   @RequestMapping(value="/register", method=GET) <-- Obsługujemy żądanie GET o zasób /spitter/register
      public String showRegistrationForm() {
      return "registerForm";
   }
}
```

Adnotacja `@RequestMapping` metody `showRegistrationForm()`, wraz z adnotacją `@Request ↪Mapping` ustawioną na poziomie klasy, wskazuje, że metoda obsługuje żądania `GET` do zasobu `/spitter/register`. Metoda jest prosta, nie pobiera żadnych danych wejściowych i zwraca tylko widok `registerForm`. Przy uwzględnieniu konfiguracji klasy `InternalResource ↪ViewResolver` widok ten zadeklarowany jest w pliku JSP `/WEB-INF/views/register ↪Form.jsp` i zawiera formularz rejestracji.

Pomimo swej prostoty metoda `showRegistrationForm()` powinna być przetestowana. Tworzony przez nas test będzie równie prosty jak sama metoda (listing 5.14).

Listing 5.14. Testujemy metodę wyświetlającą formularz

```
@Test
public void shouldShowRegistration() throws Exception {
   SpitterController controller = new SpitterController();
   MockMvc mockMvc = standaloneSetup(controller).build(); <-- Ustawiamy obiekt MockMvc
   mockMvc.perform(get("/spitter/register"))
         .andExpect(view().name("registerForm")); <-- Sprawdzamy widok registerForm
}
```

Ta metoda testu bardzo przypomina test dla metody strony głównej kontrolera. Wywołuje żądanie `GET` do zasobu `/spitter/register` i sprawdza, czy otrzymany widok ma nazwę `registerForm`.

Powróćmy teraz do tego widoku. Nazwą widoku jest `registerForm`, musimy więc utworzyć plik JSP o nazwie `registerForm.jsp`. W pliku tym musi się znajdować element HTML `<form>`, za pomocą którego użytkownik będzie mógł wprowadzić informacje umożliwiające jego zarejestrowanie w aplikacji. W przykładzie wykorzystamy plik o zawartości pokazanej na listingu 5.15.

Listing 5.15. Plik JSP wyświetlający formularz rejestracji

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" contentType="text/html; charset=UTF-8" %>
<html>
   <head>
      <title>Spittr</title>
      <link rel="stylesheet" type="text/css" href="<c:url value="/resources/style.css" />" >
   </head>
   <body>
      <h1>Rejestracja</h1>
      <form method="POST">
         Imię: <input type="text" name="firstName" /><br/>
         Nazwisko: <input type="text" name="lastName" /><br/>
         Nazwa użytkownika: <input type="text" name="username" /><br/>
         Hasło: <input type="password" name="password" /><br/>
         <input type="submit" value="Zarejestruj" />
      </form>
   </body>
</html>
```

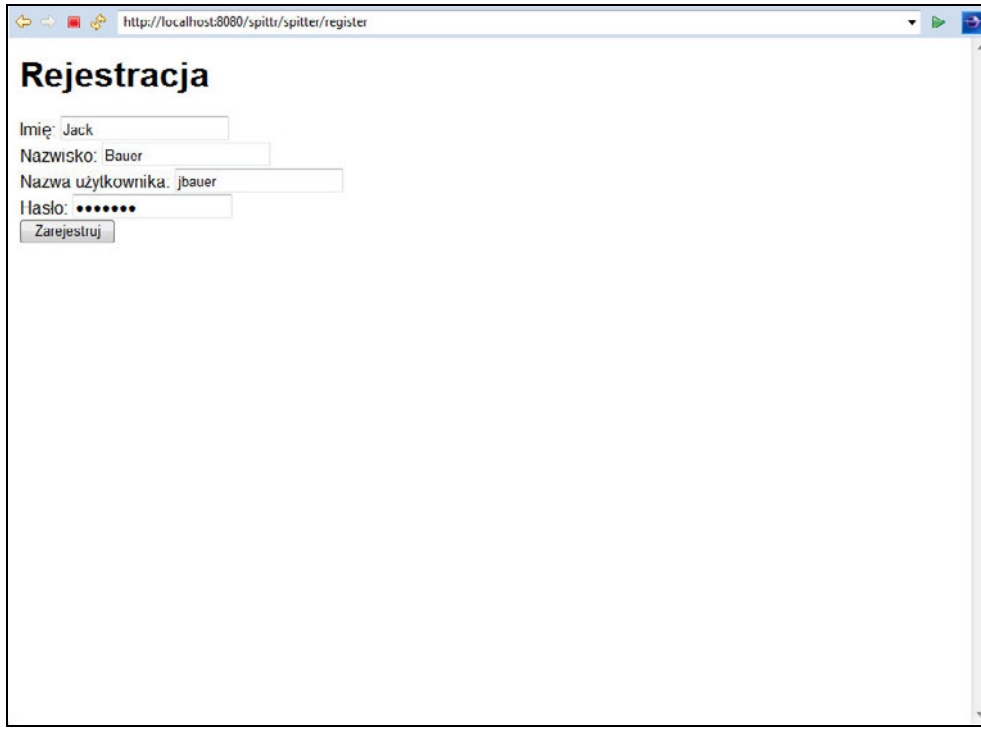
Plik ten jest bardzo prosty. Zawiera pola formularza HTML służące do wprowadzenia imienia, nazwiska, nazwy użytkownika i hasła oraz przycisk do wysyłania formularza. Wejście na stronę powoduje wyświetlenie widoku przypominającego ten przedstawiony na rysunku 5.5.

Zauważ, że znacznik `<form>` nie zawiera parametru `action`. Z tego powodu formularz po zatwierdzeniu zostanie wysłany na ten sam adres URL, pod którym znajduje się formularz, czyli na `/spitters/register`.

Oznacza to konieczność obsługi żądania `POST` dla tego adresu. Dodajmy więc nową metodę w kontrolerze `SpitterController`, służącą do obsługi danych wysłanych przez formularz.

5.4.1. Tworzymy kontroler do obsługi formularza

W trakcie przetwarzania żądania `POST` z formularza rejestracji kontroler przyjmuje dane wysłane z tego formularza i zapisuje je do obiektu `Spitter`. Na koniec, aby zapobiec wielokrotnemu przesłaniu tego samego formularza (na przykład gdy użytkownik kliknie przycisk odświeżania okna przeglądarki), przekierowuje użytkownika na stronę nowo utworzonego profilu użytkownika. Zachowanie to zostało odwzorowane i sprawdzone w teście `shouldProcessRegistration()`, przedstawionym na listingu 5.16.



Rysunek 5.5. Strona rejestracji udostępnia formularz przetwarzany przez kontroler SpitterController w celu dodania nowego użytkownika do aplikacji

Listing 5.16. Testujemy metodę obsługi formularza

```
@Test
public void shouldProcessRegistration() throws Exception {
   SpitterRepository mockRepository =
      mock(SpitterRepository.class);                          ← Tworzymy atrapę repozytorium
   Spitter unsaved = new Spitter("jbauer", "24hours", "Jack", "Bauer");
   Spitter saved = new Spitter(24L, "jbauer", "24hours", "Jack", "Bauer");
   when(mockRepository.save(unsaved)).thenReturn(saved);

   SpitterController controller =
      new SpitterController(mockRepository);
   MockMvc mockMvc = standaloneSetup(controller).build();     ← Ustawiamy obiekt MockMvc

   mockMvc.perform(post("/spitter/register")                  ← Wywołujemy żądanie
      .param("firstName", "Jack")
      .param("lastName", "Bauer")
      .param("username", "jbauer")
      .param("password", "24hours"))
      .andExpect(redirectedUrl("/spitter/jbauer"));
   verify(mockRepository, atLeastOnce()).save(unsaved);       ← Sprawdzamy, czy wystąpił zapis danych
}
```

Test ten jest bardziej skomplikowany niż test wyświetlania formularza rejestracji. Po utworzeniu atrapy implementacji repozytorium `SpitterRepository` i kolejno kontrolera

oraz obiektu klasy `MockMvc` metoda `shouldProcessRegistration()` wywołuje żądanie `POST` do zasobu `/spitter/register`. Dane o użytkowniku przekazujemy w postaci parametrów żądania `POST`, co pozwala imitować przesyłanie formularza.

Po zakończeniu operacji korzystającej z metody `POST` zalecane jest przekierowanie użytkownika na inną stronę, aby odświeżenie okna przeglądarki nie skutkowało przypadkową ponowną wysyłką formularza. Stworzony przez nas test zakłada, że żądanie zakończy się przekierowaniem na stronę `/spitter/jbauer`, ścieżkę URL strony profilu nowo utworzonego użytkownika.

Na koniec weryfikujemy, czy atrapa repozytorium `SpitterRepository` została użyta do zapisania danych pochodzących z formularza.

Przygotujmy teraz implementację metody kontrolera obsługującej ten test. Analizując kod testu `shouldProcessRegistration()`, mogliśmy odnieść wrażenie, że do spełnienia testu wymagany jest spory nakład pracy. Na listingu 5.17 możemy jednak zaobserwować, że nie musimy robić aż tak wiele.

Listing 5.17. Obsługujemy formularz rejestracji nowego użytkownika

```
package spittr.web;

import static org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import spittr.Spitter;
import spittr.data.SpitterRepository;

@Controller
@RequestMapping("/spitter")
public class SpitterController {
   private SpitterRepository spitterRepository;

   @Autowired
        public SpitterController(                    <-- Wstrzykujemy repozytorium SpitterRepository
      SpitterRepository spitterRepository) { this.spitterRepository = spitterRepository;
   }

   @RequestMapping(value="/register", method=GET)
   public String showRegistrationForm() {
      return "registerForm";
   }

   @RequestMapping(value="/register", method=POST)
   public String processRegistration(Spitter spitter) {
      spitterRepository.save(spitter);               <-- Zapisujemy obiekt Spitter
     return "redirect:/spitter/" +                   <-- Przekierowujemy na stronę profilu
         spitter.getUsername();
   }
}
```

Metoda `showRegistrationForm()` jest wciąż dostępna na swoim miejscu. Pojawiła się jednak nowa metoda, `processRegistration()`. Jako parametr przekazujemy do niej obiekt

Spitter. Obiekt ten posiada właściwości firstName, lastName, username oraz password, które zostaną wypełnione danymi pochodzącymi z parametrów żądania o tych samych nazwach.

Metoda processRegistration() po przekazaniu obiektu typu Spitter wywołuje metodę save() repozytorium SpitterRepository, wstrzykniętego przez konstruktor do kontrolera SpitterController.

Ostatnią rzeczą wykonywaną przez metodę processRegistration() jest zwrócenie ciągu znaków zawierającego nazwę widoku. Ten widok specyfikacji różni się od obserwowanych wcześniej. Zwracamy tu nie tylko nazwę widoku do rozwiązania przez producenta widoków, ale także specyfikację przekierowania.

Dzięki zastosowaniu w specyfikacji widoku prefiksu redirect: klasa InternalResource
➥ViewResolver wie, że ma wykonać przekierowanie, a nie zwracać widok o podanej nazwie. W tym przypadku użytkownik zostanie przekierowany na swoją stronę profilu. Na przykład jeśli właściwość username obiektu Spitter ma wartość jbauer, to nastąpi przekierowanie na adres /spitter/jbauer.

Warto wiedzieć, że oprócz opcji redirect: klasa InternalResourceViewResolver rozpoznaje też prefiks forward:. Po napotkaniu specyfikacji widoku poprzedzonej prefiksem forward: żądanie nie jest przekierowywane, a przekazywane do danej ścieżki URL.

Świetnie! W tym momencie test z listingu 5.16 powinien zakończyć się powodzeniem. Ale to jeszcze nie koniec. Wykonujemy przekierowanie na stronę profilu użytkownika, powinniśmy więc do kontrolera SpitterController dodać metodę obsługującą żądania o stronę profilu. Zadanie to wykona przedstawiona poniżej metoda showSpitter
➥Profiler():

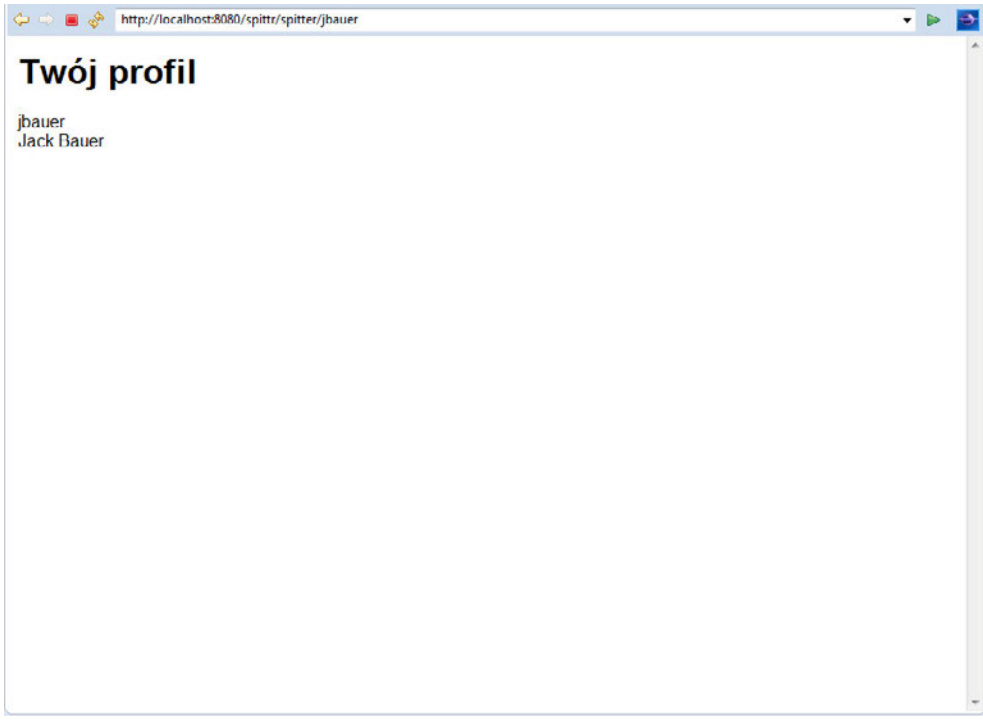
```
@RequestMapping(value="/{username}", method=GET)
public String showSpitterProfile(
   @PathVariable String username, Model model) {
   Spitter spitter = spitterRepository.findByUsername(username);
   model.addAttribute(spitter);
   return "profile";
}
```

Metoda showSpitterProfile() pobiera obiekt Spitter z repozytorium SpitterRepository z wykorzystaniem nazwy użytkownika. Obiekt Spitter jest dodawany do modelu i zwraca tekst profile, nazwę widoku dla widoku profilu. Tak jak wszystkie widoki pokazane w tym rozdziale, ten również będzie prosty:

```
<h1>Twój profil</h1>
<c:out value="${spitter.username}" /><br/>
<c:out value="${spitter.firstName}" />
<c:out value="${spitter.lastName}" />
```

Rysunek 5.6 pokazuje stronę profilu wygenerowaną w oknie przeglądarki.

Co się stanie, jeżeli w formularzu nie zostaną przesłane parametry username lub password? Albo jeśli wartości firstName bądź lastName będą puste lub zbyt krótkie? Przyjrzyjmy się teraz procesowi dodawania walidacji przesłanych danych formularza, aby przeciwdziałać niespójności prezentowanych danych.



Rysunek 5.6. Strona profilu w aplikacji Spittr wyświetla informacje o użytkowniku zapisane w modelu w kontrolerze SpitterController

5.4.2. Walidujemy formularze

Jeśli użytkownik, zatwierdzając formularz, pozostawi pola `username` bądź `password` puste, może nastąpić utworzenie nowego obiektu klasy `Spitter`, którego wartością pola nazwy użytkownika albo hasła będzie pusty ciąg znaków. Jest to zachowanie co najmniej dziwne. Pozostawienie tego w tej postaci może prowadzić do powstania zagrożeń bezpieczeństwa i umożliwić każdemu użytkownikowi zalogowanie się do aplikacji bez wypełniania pól formularza.

Powinniśmy poczynić kroki chroniące użytkownika przed wysłaniem pustej wartości imienia i (lub) nazwiska na wypadek, gdyby zechciał on zachować jakiś poziom anonimowości. Dobrym pomysłem jest też prawdopodobnie ograniczenie długości tekstów podanych w tych polach do jakiejś rozsądnej wartości, co pozwoli uniknąć nadużyć w tym zakresie.

Jednym ze sposobów walidacji, choć naiwnym, jest dodanie kodu do metody `process` `↪Registration()` w celu sprawdzenia niepoprawnych wartości i przekierowania użytkownika z powrotem na formularz rejestracji, jeżeli walidacja poprawności danych wejściowych zwróci wynik negatywny. Metoda obsługi żądania jest bardzo krótka, więc dorzucenie kilku warunków `if` nas nie zaboli. Prawda?

Nie chcemy jednak zaśmiecać kodu metody obsługującej żądanie logiką walidacji. Zamiast tego możemy wykorzystać obsługę API Java Validation (JSR-303) dostępną w Springu. Poczynając od wersji 3.0 Springa, mamy możliwość użycia Java Validation

API w Spring MVC. Do jej obsługi nie jest potrzebna żadna dodatkowa konfiguracja. Musimy się tylko upewnić, że w ścieżce klas projektu dostępna jest implementacja Java API, taka jak Hibernate Validator.

Java Validation API definiuje kilka adnotacji, które możemy zastosować na właściwościach, co pozwala nałożyć ograniczenia na ich dozwolone wartości. Wszystkie te adnotacje pochodzą z pakietu `javax.validation.constraints` i znajdują się na liście w tabeli 5.1.

Tabela 5.1. Adnotacje walidacji dostarczane przez Java Validation API

Adnotacja	Opis
`@AssertFalse`	Oznaczony element musi być typu `Boolean` i przyjąć wartość `false`.
`@AssertTrue`	Oznaczony element musi być typu `Boolean` i przyjąć wartość `true`.
`@DecimalMax`	Oznaczony element musi być liczbą, której wartość jest mniejsza od danej wartości typu `BigDecimalString` lub jej równa.
`@DecimalMin`	Oznaczony element musi być liczbą, której wartość jest większa od danej wartości typu `BigDecimalString` lub jej równa.
`@Digits`	Oznaczony element musi być liczbą posiadającą określoną liczbę cyfr.
`@Future`	Wartością oznaczonego element musi być data z przyszłości.
`@Max`	Oznaczony element musi być liczbą, której wartość jest mniejsza od danej wartości lub jej równa.
`@Min`	Oznaczony element musi być liczbą, której wartość jest większa od danej wartości lub jej równa.
`@NotNull`	Wartością oznaczonego elementu nie może być `null`.
`@Null`	Wartością oznaczonego elementu musi być `null`.
`@Past`	Wartością oznaczonego elementu musi być data z przeszłości.
`@Pattern`	Wartość oznaczonego elementu musi spełniać warunek określony wyrażeniem regularnym.
`@Size`	Wartością oznaczonego elementu musi być ciąg znaków typu `String`, kolekcja lub tablica, której długość mieści się w podanym zakresie.

Poza adnotacjami wymienionymi w tabeli 5.1 różne implementacje Java Validation API mogą dostarczać swoje własne adnotacje. Możliwe jest również zdefiniowanie własnych ograniczeń. My jednak skupimy się na kilku kluczowych adnotacjach spośród tych wymienionych w tabeli.

Biorąc pod uwagę ograniczenia, które musimy nałożyć na pola w klasie `Spitter`, przydatne wydają się adnotacje `@NotNull` oraz `@Size`. Musimy jedynie ustawić te adnotacje na właściwościach, co zostało zrobione na listingu 5.18.

Listing 5.18. SpittleForm: zawiera jedynie pola wysłane w żądaniu SpittlePOST

```
package spittr;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import org.apache.commons.lang3.builder.EqualsBuilder;
import org.apache.commons.lang3.builder.HashCodeBuilder;

public class Spitter {
   private Long id;
```

```
    @NotNull
    @Size(min=5, max=16) ←—— Niepuste, pomiędzy 5 a 16 znakami
    private String username;
    @NotNull
    @Size(min=5, max=25) ←—— Niepuste, pomiędzy 5 a 25 znakami
    private String password;
    @NotNull
    @Size(min=2, max=30) ←—— Niepuste, pomiędzy 2 a 30 znakami
    private String firstName;
    @NotNull
    @Size(min=2, max=30) ←—— Niepuste, pomiędzy 2 a 30 znakami
    private String lastName;
    ...
}
```

Wszystkie właściwości klasy `Spitter` oznaczone są adnotacją `@NotNull`, aby upewnić się, że nie przyjmą wartości `null`. W podobny sposób zastosowana została adnotacja `@Size`, by określić ich minimalną i maksymalną dozwoloną długość. W związku z tymi zmianami użytkownicy aplikacji Spittr muszą wypełnić formularz rejestracji wartościami odpowiadającymi podanym ograniczeniom.

Po oznaczeniu adnotacjami klasy Spitter musimy zmodyfikować metodę `process➥Registration()`, żeby zastosować ustalone reguły walidacji. Nowa wersja metody `process➥Registration()` znajduje się na listingu 5.19.

Listing 5.19. processRegistration(): weryfikuje, czy przesłane dane są poprawne

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    @Valid Spitter spitter, ←—— Walidujemy dane wejściowe
    Errors errors) {
    if (errors.hasErrors()) { ←—— Powracamy do formularza po wystąpieniu błędów walidacji
      return "registerForm";
    }
    spitterRepository.save(spitter);
    return "redirect:/spitter/" + spitter.getUsername();
}
```

W stosunku do oryginalnej wersji metody `processRegistration()`, zamieszczonej na listingu 5.17, wiele się zmieniło. Parametr `Spitter` jest teraz opatrzony adnotacją `@Valid`, co sygnalizuje Springowi, że obiekt wspierający zawiera błędy naruszające ograniczenia walidacji.

Samo ustawienie ograniczeń na właściwościach klasy `Spitter` nie uchroni nas przed wysłaniem niepoprawnego formularza. Nawet jeśli użytkownik nie wypełni pola formularza lub wprowadzi wartość przekraczającą dozwoloną długość, metoda `processRegistra➥tion()` i tak zostanie wywołana. Daje nam to szansę na właściwą obsługę powstałych problemów z walidacją.

Wszystkie ewentualne błędy dostępne są w obiekcie `Errors`, który przyjmujemy teraz jako parametr metody `processRegistration()`. (Warto wiedzieć, że parametr `Errors` powinien wystąpić zaraz za parametrem opatrzonym adnotacją `@Valid`). Pierwszą czyn-

nością, jaką wykonuje metoda `processRegistration()`, jest wywołanie metody `Errors.hasErrors()`, która pozwala sprawdzić, czy formularz zawiera jakieś błędy.

Jeśli wystąpiły jakieś błędy, zwracamy ciąg `"registerForm"`, nazwę widoku formularza rejestracji. Spowoduje to przeniesienie użytkownika z powrotem na formularz, na którym będzie miał możliwość rozwiązania wszystkich problemów i wysłania formularza ponownie. Na początek wyświetlaliśmy jedynie pusty formularz, ale w następnym rozdziale poprawimy go tak, by zawierał pierwotnie wysłane wartości, i pokażemy użytkownikowi błędy walidacji.

Jeżeli formularz nie zawiera żadnych błędów, obiekt klasy `Spitter` jest zapisywany w repozytorium, a kontroler przekierowuje użytkownika na stronę profilu, tak jak to było wcześniej.

5.5. Podsumowanie

W tym rozdziale przygotowaliśmy podstawy pod internetową część aplikacji. Jak mieliśmy okazję zobaczyć, Spring dostarcza potężny i elastyczny framework internetowy. Adnotacje umożliwiają Spring MVC wykorzystanie modelu programowania opartego na obiektach POJO, czyniąc pracę z kontrolerami i obsługę żądań prostą oraz łatwą w testowaniu.

Spring MVC jest bardzo elastyczny w zakresie tworzenia metod kontrolera. Ogólna zasada jest taka, że jeśli metoda obsługi żądania potrzebuje jakiegoś obiektu, prosi o ten obiekt w postaci parametru. Analogicznie — jeżeli coś nie jest nam potrzebne, powinniśmy to z listy parametrów usunąć. Daje to nieograniczone możliwości obsługi żądań przy zachowaniu prostego modelu programowania.

Chociaż większa część tego rozdziału koncentrowała się na obsłudze żądań za pomocą kontrolerów, równie istotna jest kwestia generowania odpowiedzi. Zobaczyłeś pokrótce, jak tworzyć widoki dla kontrolerów z użyciem plików JSP. Spring MVC oferuje jednak w tym zakresie znacznie więcej niż tylko konstruowanie prostych szablonów JSP.

W rozdziale 6. dowiesz się więcej na temat widoków w Springu, w tym na temat wykorzystania bibliotek znaczników w szablonach JSP. Zobaczysz również, jak tworzyć układy stron w Apache Tiles i współdzielić je między widokami. Na koniec poznasz Thymeleaf, ekscytującą alternatywę dla JSP, która zawiera wbudowaną obsługę Springa.

6 Generowanie widoków

W tym rozdziale omówimy:

- Generowanie danych modelu w postaci plików HTML
- Wykorzystywanie widoków JSP
- Definiowanie układów stron za pomocą kafelków
- Pracę z widokami Thymeleaf

W poprzednim rozdziale skoncentrowaliśmy się na tworzeniu kontrolerów obsługujących żądania internetowe. Utworzyliśmy też kilka prostych widoków do generowania danych modelu wyprodukowanych przez te kontrolery, ale nie spędziliśmy zbyt wiele czasu na rozważaniach o widokach i o tym, co dzieje się pomiędzy zakończeniem pracy kontrolera a wyświetleniem wyników w oknie przeglądarki użytkownika. Ten etap pracy aplikacji jest właśnie tematem bieżącego rozdziału.

6.1. Poznajemy sposób produkowania widoków

Żadna z metod zdefiniowanych w kontrolerach, które utworzyliśmy w rozdziale 5., nie produkuje bezpośrednio kodu HTML wyświetlanego w przeglądarce. Zamiast tego wypełniają one model jakimiś danymi i przekazują go do widoku. Metody te zwracają wartości typu String, który stanowią nazwę widoku. Nazwa ta nie wiąże się jednak bezpośrednio z żadną konkretną implementacją widoku. Chociaż zdążyliśmy już napisać kilka widoków JSP (*JavaServer Pages*), kontrolery nie miały o nich najmniejszego pojęcia.

Ta separacja logiki obsługi żądań w kontrolerze od generowania widoku jest ważną funkcjonalnością Spring MVC. Jeśli metody kontrolera byłyby bezpośrednio odpowiedzialne za produkowanie kodu HTML, powstały kod byłby trudny w utrzymaniu, trudne

byłoby też aktualizowanie widoków bez grzebania w logice kontrolera. Metody kontrolera i implementacje widoków powinny co najwyżej ustalić między sobą zawartość modelu. Poza tym powinny się trzymać wzajemnie na dystans.

Jeśli jednak kontroler zna tylko logiczną nazwę widoku, skąd Spring wie, którą jego implementację ma wykorzystać przy generowania modelu? Jest to zadanie dla producentów widoków Springa.

W rozdziale 5. użyliśmy producenta widoków o nazwie `InternalResourceViewResolver`. Został on skonfigurowany tak, aby poprzedzać widok prefiksem `/WEB-INF/views` i uzupełniać sufiksem `.jsp`. Otrzymaliśmy w ten sposób fizyczną lokalizację pliku JSP, który posłuży do generowania modelu. Cofnijmy się teraz do etapu produkowania widoków i poznajmy niektórych producentów widoków dostępnych w Springu.

Spring MVC definiuje interfejs o nazwie `ViewResolver`, wyglądający podobnie jak przedstawiony poniżej:

```
public interface ViewResolver {
   View resolveViewName(String viewName, Locale locale) throws Exception;
}
```

Po podaniu nazwy widoku i ustawień regionalnych `Locale` metoda `resolveViewName()` zwraca instancję `View`. Jest to kolejny interfejs, zdefiniowany w przedstawiony poniżej sposób:

```
public interface View {
   String getContentType();
   void render(Map<String, ?> model,
      HttpServletRequest request,
      HttpServletResponse response) throws Exception;
}
```

Zadaniem interfejsu `View` jest pobranie modelu, jak również obiektów żądania i odpowiedzi, a następnie generowanie danych wyjściowych w odpowiedzi.

Nie wydaje się to skomplikowane. Wystarczy, że rozpoczniemy tworzenie implementacji widoku i producenta widoków, aby wygenerować odpowiedź i wyświetlić ją w przeglądarce użytkownika. Zgadza się?

Niekoniecznie. Choć mamy możliwość napisania własnych implementacji widoku oraz producenta widoków i choć istnieją pewne szczególne przypadki, w których jest to konieczne, z reguły nie musimy się martwić o implementację tych interfejsów. Wspomniałem o nich tylko po to, by pokrótce wyjaśnić, jak działa mechanizm produkowania widoków. Spring udostępnia kilka gotowych implementacji interfejsów `View` i `ViewResolver`, wymienionych w tabeli 6.1. Spełniają one dobrze swoje zadania w najczęściej spotykanych scenariuszach użycia.

Wszyscy producenci widoków zamieszczeni w tabeli 6.1 dostępni są zarówno w Springu 4, jak i Springu 3.2, i poza jednym (producentem `TilesViewResolver` w wersji dla Tiles 3.0) obsługiwani są również w Springu 3.1.

Książka ta nie jest wystarczająco obszerna, aby zamieścić opis wszystkich przedstawionych powyżej producentów. Nie jest to jednak wielka strata, bo większość aplikacji wykorzystuje tylko kilku z nich.

Tabela 6.1. Spring udostępnia 13 producentów widoków, zajmujących się tłumaczeniem logicznych nazw widoków na ich fizyczną implementację

Producent widoków	Opis
`BeanNameViewResolver`	Produkuje widoki jako komponenty w kontekście aplikacji Springa, których identyfikatorem jest nazwa widoku.
`ContentNegotiatingViewResolver`	Produkuje widoki, biorąc pod uwagę typ zawartości oczekiwanej przez klienta i przekazując zadanie do innego producenta widoków, który potrafi obsłużyć żądania tego typu.
`FreeMarkerViewResolver`	Produkuje widoki jako szablony FreeMarker.
`InternalResourceViewResolver`	Produkuje widoki jako zasoby zewnętrzne w stosunku do aplikacji internetowej (z reguły szablony JSP).
`JasperReportsViewResolver`	Produkuje widoki jako definicje JasperReports.
`ResourceBundleViewResolver`	Produkuje widoki z pakietu zasobów (z reguły pliku z właściwościami).
`TilesViewResolver`	Produkuje widoki w postaci definicji Apache Tile, w której identyfikatorem kafelka jest nazwa widoku. Zwróć uwagę, że istnieją dwie różne implementacje producenta `Tiles ↪ViewResolver` — jedna dla Tiles 2.0, a druga dla Tiles 3.0.
`UrlBasedViewResolver`	Produkuje widoki bezpośrednio w oparciu o ich nazwę; nazwa widoku odpowiada fizycznej definicji widoku.
`VelocityLayoutViewResolver`	Produkuje widoki jako szablony Velocity, komponując strony z różnych szablonów Velocity.
`VelocityViewResolver`	Produkuje widoki jako szablony Velocity.
`XmlViewResolver`	Produkuje widoki jako definicje komponentów z określonego pliku XML, podobnie jak w przypadku producenta `BeanNameViewResolver`.
`XsltViewResolver`	Produkuje widoki powstałe w wyniku transformacji XSLT.

Niemal każdy z producentów widoków wymienionych w tabeli 6.1 odnosi się do określonej technologii widoków dostępnej dla aplikacji Javy. `InternalResourceViewHandler` używany jest z reguły z szablonami JSP, `TilesViewResolver` z widokami Apache Tiles, a `FreemarkerViewResolver`, a `VelocityViewResolver` odpowiednio z szablonami Freemarker i Velocity.

W tym rozdziale skoncentrujemy się na tych technologiach widoków, które są najistotniejsze dla większości programistów Javy. Ponieważ większość aplikacji Javy korzysta z szablonów JSP, rozpoczniemy od związanego z nimi producenta `Internal ↪ResourceViewResolver`. Następnie wykorzystamy producenta `TilesViewResolver`, żeby pozyskać kontrolę nad układami stron w JSP.

W podsumowaniu rozdziału przyjrzymy się producentowi widoków, który nie znalazł się w tabeli 6.1. Thymeleaf jest popularną alternatywą dla plików JSP i dostarcza producenta widoków do pracy z naturalnymi szablonami Thymeleaf — szablonami mającymi więcej wspólnego z wytwarzanym kodem HTML niż z kodem Javy, który je napędza. Thymeleaf jest tak ekscytującą technologią, że wcale się nie obrażę, jeśli postanowisz przeskoczyć kilka stron tej książki, aby przejść do rozdziału 6.4 i dowiedzieć się, jak wykorzystać tę technologię w Springu.

Jeżeli jednak w dalszym ciągu czytasz tę stronę, to zapewne wiesz, że JSP było i wciąż jest najpopularniejszą technologią stosowaną przy tworzeniu widoków w Javie. Chyba każdy z nas pracował wcześniej z szablonami JSP w kilku projektach i niejednokrotnie jeszcze się z nimi zetknie. Zobaczmy więc, jak możemy je wykorzystać w Spring MVC.

6.2. Tworzymy widoki JSP

Trudno w to uwierzyć, ale JSP jest podstawową technologią stosowaną w widokach w aplikacjach javowych już od niemal 15 lat. Choć szablony JSP były początkowo brzydkim javowym klonem rozwiązań dostępnych dla innych platform (jak ASP Microsoftu), to na przestrzeni lat wyraźnie się rozwinęły, co zaowocowało wsparciem dla języka wyrażeń i niestandardowych bibliotek znaczników.

Spring wspiera widoki JSP na dwa sposoby:

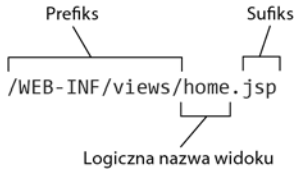
- Możemy wykorzystać producenta `InternalResourceViewResolver` do odwzorowywania nazw widoków na pliki JSP. Co więcej, jeśli w plikach JSP stosujemy znaczniki JSTL (*JavaServer Pages Standard Tag Library*), odwzorowanie to może wykorzystywać widok `JstlView` w celu użycia ustawień regionalnych i zmiennych zasobów JSTL w znacznikach formatowania oraz komunikatach.
- Spring dostarcza dwie biblioteki znaczników JSP — jedna służy do wiązania formularzy z modelem, a druga dostarcza narzędzia ogólnego przeznaczenia.

Niezależnie od tego, czy korzystamy z JSTL, czy chcemy użyć biblioteki znaczników JSP Springa, ważna jest konfiguracja producenta do odwzorowywania widoków na pliki JSP. Moglibyśmy wykorzystać kilku innych producentów widoków, ale `InternalResource`
`↪ViewResolver` jest najprostszym i najczęściej stosowanym rozwiązaniem w tego typu zadaniach. Z jego konfiguracją zetknęliśmy się już w rozdziale 5. Robiliśmy to jednak w pośpiechu, aby poćwiczyć pracę z kontrolerami w przeglądarce internetowej. Przyjrzyjmy się dokładniej `InternalResourceViewResolver` i zobaczmy, jak go skonfigurować zgodnie z naszymi wymaganiami.

6.2.1. Konfigurujemy producenta widoków gotowego do pracy z JSP

Niektórzy producenci widoków, jak `ResourceBundleViewResolver`, odwzorowują nazwę widoku bezpośrednio na konkretną implementację interfejsu `View`. `InternalResource`
`↪ViewResolver` jest jednak mniej bezpośredni i polega na konwencji, zgodnie z którą do nazwy widoku dołączane są prefiks i sufiks. W rezultacie otrzymujemy fizyczną ścieżkę do producenta widoków w tej samej aplikacji internetowej.

Rozważmy prosty przykład, w którym nazwą widoku jest `home`. Pliki JSP umieszcza się często w folderze *WEB-INF* aplikacji, by uniemożliwić bezpośredni dostęp do nich. Jeśli wszystkie pliki JSP umieścimy w folderze */WEB-INF/views/*, a nazwą pliku strony domowej jest *home.jsp*, to fizyczną ścieżkę do pliku uzyskujemy poprzez poprzedzenie nazwy `home` prefiksem */WEB-INF/views/* i dodanie sufiksu `.jsp`. Ilustruje to rysunek 6.1.



Rysunek 6.1. InternalResourceViewResolver produkuje widoki poprzez dodanie prefiksu i sufiksu do nazwy widoku

Zgodnie z tą konwencją konfigurację producenta `InternalResourceViewResolver` możemy wykonać za pomocą metody opatrzonej adnotacją `@Bean`:

```
@Bean
public ViewResolver viewResolver() {
   InternalResourceViewResolver resolver = new InternalResourceViewResolver();
   resolver.setPrefix("/WEB-INF/views/");
   resolver.setSuffix(".jsp");
   return resolver;
}
```

Opcjonalnie, jeżeli wolisz przechowywać konfigurację Springa w plikach XML, konfigurację producenta `InternalResourceViewResolver` można przeprowadzić w sposób podany niżej:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/views/"
      p:suffix=".jsp" />
```

Dzięki zastosowaniu tej konfiguracji możliwe będzie odwzorowywanie logicznych nazw widoków na pliki JSP, takie jak:

- `home` na */WEB-INF/views/home.jsp*;
- `productList` na */WEB-INF/views/productList.jsp*;
- `books/details` na */WEB-INF/views/books/detail.jsp*.

Przyjrzyjmy się bliżej ostatniemu przykładowi. Jeśli w nazwie widoku pojawia się ukośnik, jest on przenoszony do nazwy ścieżki. Nazwa widoku jest więc odwzorowywana na plik JSP zlokalizowany w podkatalogu katalogu określonego we właściwości `prefix`. Dzięki temu możemy wygodnie organizować szablony w hierarchii katalogów, zamiast wrzucać je wszystkie w jedno miejsce.

PRODUKUJEMY WIDOKI JSTL

Do tej pory skonfigurowaliśmy pojedynczego producenta `InternalResourceViewResolver`. W wyniku otrzymaliśmy odwzorowanie logicznych nazw widoków na instancje `Internal`
`↪ResourceView`, które odnoszą się do plików JSP. Jeżeli jednak te pliki JSP wykorzystują znaczniki JSTL do formatowania lub wyświetlania komunikatów, konieczna może być taka konfiguracja producenta widoków `InternalResourceViewResolver`, aby w wyniku odwzorowywania otrzymywać instancje `JstlView`.

Do właściwego formatowania wartości opartych na ustawieniach regionalnych, takich jak daty i waluty, znaczniki formatowania JSTL wymagają ustawień `Locale`. Znaczniki komunikatów mogą wykorzystać źródła komunikatów Springa i ustawienia `Locale` do

wstawienia właściwych komunikatów w kodzie HTML. W wyprodukowanych widokach `JstlView` do znaczników JSTL przekazywane są ustawienia `Locale` oraz źródła wiadomości skonfigurowane w Springu.

Aby producent `InternalResourceViewResolver` zamiast widoków `InternalResourceView` produkował widoki `JstlView`, wystarczy ustawić wartość właściwości `viewClass`:

```
@Bean
public ViewResolver viewResolver() {
   InternalResourceViewResolver resolver = new InternalResourceViewResolver();
   resolver.setPrefix("/WEB-INF/views/");
   resolver.setSuffix(".jsp");
   resolver.setViewClass(org.springframework.web.servlet.view.JstlView.class);
   return resolver;
}
```

Możemy to też oczywiście osiągnąć za pomocą konfiguracji XML:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/WEB-INF/views/" p:suffix=".jsp"
      p:viewClass="org.springframework.web.servlet.view.JstlView" />
```

Niezależnie od wybranego sposobu konfiguracji, dzięki powyższym ustawieniom do znaczników formatowania i komunikatów przekazane zostaną ustawienia `Locale` i źródła komunikatów skonfigurowane w Springu.

6.2.2. Korzystamy z bibliotek JSP Springa

Biblioteki znaczników są potężnym narzędziem. Pozwalają na wykorzystanie w szablonach JSP dodatkowych możliwości bez potrzeby umieszczania kodu Javy bezpośrednio w blokach scripletów. Spring udostępnia dwie biblioteki znaczników JSP, których możemy użyć przy tworzeniu widoków w Spring MVC. Jedna biblioteka służy do generowania znaczników HTML formularzy, powiązanych z atrybutem `model`. Druga biblioteka zawiera bogactwo różnorodnych znaczników użytkowych, które od czasu do czasu mogą się okazać przydatne.

W codziennej pracy częściej sięgniemy po znaczniki służące do wiązania formularzy. Dlatego poznawanie znaczników rozpoczniemy od omówienia tej właśnie biblioteki. Zobaczysz, jak powiązać formularz rejestracji aplikacji Spittr z modelem tak, aby po niepoprawnym wypełnieniu formularza ponownie wypełnić go przesłanymi danymi i wyświetlić powstałe błędy walidacji.

WIĄŻEMY FORMULARZE Z MODELEM

Biblioteka znaczników JSP wiązania formularzy zawiera 14 znaczników, z których większość generuje znaczniki formularzy HTML. Elementem, który je różni od zwykłych znaczników HTML, jest wiązanie ich wartości z obiektem w modelu i możliwość wypełnienia wartościami zapisanymi we właściwościach tego obiektu. Biblioteka znaczników zawiera też znacznik służący do przekazywania użytkownikowi komunikatów o błędach poprzez ich umieszczenie w wynikowym kodzie HTML.

Wykorzystanie biblioteki znaczników wiązania formularzy możliwe jest po jej zadeklarowaniu w nagłówku strony JSP:

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="sf" %>
```

Warto zauważyć, że jako prefiks ustawiłem ciąg `sf`; często można się też spotkać z użyciem prefiksu `form`. W praktyce możemy zastosować dowolny wybrany prefiks. Wybrałem `sf`, bo jest zwięzły, łatwo go wpisać i jest skrótem od „Spring forms" (formularze Springa). W tej książce za każdym razem, gdy będziemy korzystać z biblioteki znaczników wiązania formularzy, użyjemy tego właśnie prefiksu.

Po zadeklarowaniu biblioteki otrzymujemy do dyspozycji 14 znaczników. Zostały one wymienione w tabeli 6.2.

Tabela 6.2. Biblioteka znaczników wiązania formularzy w Springu zawiera znaczniki do wiązania obiektów zapisanych w modelu z wygenerowanymi polami formularzy HTML

Znacznik JSP	Opis
`<sf:checkbox>`	Generuje znacznik HTML `<input>` z wartością atrybutu `type` ustawioną na `checkbox`.
`<sf:checkboxes>`	Generuje wiele znaczników HTML `<input>` z wartościami atrybutu `type` ustawionymi na `checkbox`.
`<sf:errors>`	Generuje pola błędów zawarte w znaczniku HTML `<span>`.
`<sf:form>`	Generuje znacznik HTML `<form>` i wystawia ścieżki wiązania, aby umożliwić wiązanie danych z wewnętrznymi znacznikami.
`<sf:hidden>`	Generuje znacznik HTML `<input>` z wartością atrybutu `type` ustawioną na `hidden`.
`<sf:input>`	Generuje znacznik HTML `<input>` z wartością atrybutu `type` ustawioną na `text`.
`<sf:label>`	Generuje znacznik HTML `<label>`.
`<sf:option>`	Generuje znacznik HTML `<option>`. Wartość atrybutu `selected` uzależniona jest od wartości atrybutu `bound`.
`<sf:options>`	Generuje listę znaczników HTML `<option>` odpowiadającą powiązanej kolekcji, tablicy lub mapie.
`<sf:password>`	Generuje znacznik HTML `<input>` z wartością atrybutu `type` ustawioną na `password`.
`<sf:radiobutton>`	Generuje znacznik HTML `<input>` z wartością atrybutu `type` ustawioną na `radio`.
`<sf:radiobuttons>`	Generuje wiele znaczników HTML `<input>` z wartością atrybutu `type` ustawioną na `radio`.
`<sf:select>`	Generuje znacznik HTML `<select>`.
`<sf:textarea>`	Generuje znacznik HTML `<textarea>`.

Ciężko byłoby utworzyć przykład, który uwzględniałby wszystkie powyższe znaczniki. A już z pewnością byłby on mocno naciągany. W przykładzie Spittr wykorzystamy tylko te znaczniki, które przydadzą się przy budowie formularza rejestracji. W szczególności będą to znaczniki `<sf:form>`, `<sf:input>` oraz `<sf:password>`. Po ich zastosowaniu na stronie rejestracji JSP otrzymamy następujący kod formularza:

```
<sf:form method="POST" commandName="spitter">
   Imię: <sf:input path="firstName" /><br/>
   Nazwisko: <sf:input path="lastName" /><br/>
   Adres e-mail: <sf:input path="email" /><br/>
```

```
   Nazwa użytkownika: <sf:input path="username" /><br/>
   Hasło: <sf:password path="password" /><br/>
   <input type="submit" value="Zarejestruj" />
</sf:form>
```

Znacznik `<sf:form>` generuje znacznik HTML `<form>`. Tworzy też pewien kontekst wokół obiektu modelu przekazywany przez atrybut `commandName`. Właściwości obiektu modelu będą się odwoływać w innych wykorzystywanych znacznikach wiązania danych.

W poniższym kodzie wartość atrybutu `commandName` ustawiamy na `spitter`. W modelu musi więc istnieć obiekt, którego kluczem jest `spitter`, w przeciwnym wypadku nie uda się nam wygenerować formularza (i otrzymamy błąd JSP). Oznacza to, że musimy wykonać niewielką modyfikację kontrolera `SpitterController`, aby obiekt `Spitter` został zapisany w modelu pod kluczem `spitter`:

```
@RequestMapping(value="/register", method=GET)
public String showRegistrationForm(Model model) {
   model.addAttribute(new Spitter());
   return "registerForm";
}
```

Po poprawkach metoda `showRegistrationForm()` dodaje do modelu nową instancję klasy `Spitter`. Kluczem modelu na podstawie typu dodawanego obiektu będzie `spitter` — jest to dokładnie taka nazwa, jaką chcieliśmy uzyskać.

Wracając do formularza — jego pierwsze trzy pola zostały zamienione ze znaczników `<input>` na `<sf:input>`. Znacznik ten generuje kod znacznika HTML `<input>` z atrybutem `type` ustawionym na `text`. Jego atrybut `value` przyjmuje wartość właściwości obiektu modelu określonej w atrybucie `path`. Na przykład jeśli w obiekcie klasy `Spitter` wartością właściwości `firstName` jest `Jack`, to znacznik `<sf:input path="firstName"/>` zostanie wygenerowany jako znacznik `<input>` z wartością atrybutu `value="Jack"`.

Pole `password` nie wykorzystuje znacznika `<sf:input>`. Zamiast niego stosuje znacznik `<sf:password>`. Znacznik ten jest podobny do znacznika `<sf:input>`, ale generuje znacznik HTML `<input>`, którego atrybut `type` ma wartość `password`. Tekst wpisywany w tym polu jest więc ukryty.

Aby lepiej zwizualizować wynikowy kod HTML, załóżmy, że użytkownik przesłał już formularz zawierający nieprawidłowe wartości dla wszystkich pól. Po niepowodzeniu walidacji i przekierowaniu użytkownika na formularz rejestracji kod HTML wewnątrz znacznika `<form>` wygląda tak:

```
<form id="spitter" action="/spitter/spitter/register" method="POST">
   Imię:
      <input id="firstName" name="firstName" type="text" value="J"/><br/>
   Nazwisko:
      <input id="lastName" name="lastName" type="text" value="B"/><br/>
   Adres e-mail:
      <input id="email" name="email" type="text" value="jack"/><br/>
   Nazwa użytkownika:
      <input id="username" name="username" type="text" value="jack"/><br/>
   Hasło:
      <input id="password" name="password" type="password" value=""/><br/>
   <input type="submit" value="Zarejestruj" />
</form>
```

Warto zaznaczyć, że poczynając od Springa 3.1, znacznik `<sf:input>` umożliwia określenie wartości atrybutu `type`, pozwalającej na deklarację typów tekstowych specyficznych dla HTML 5, takich jak `data`, `range` czy `email`. Możemy na przykład zadeklarować pole e-mail w następujący sposób:

```
Adres e-mail: <sf:input path="email" type="email" /><br/>
```

Zostanie to wygenerowane w postaci kodu HTML:

```
Adres e-mail:
   <input id="email" name="email" type="email" value="jack"/><br/>
```

Znaczniki wiązania formularzy w Springu dają nam delikatną przewagę nad standardowym znacznikiem HTML — formularz wypełniony jest wprowadzonymi wcześniej niepoprawnymi danymi. W dalszym ciągu nie informuje nas jednak, dlaczego wpis jest nieprawidłowy. Żeby umożliwić użytkownikom naprawę ich błędów, musimy wykorzystać znacznik `<sf:errors>`.

WYŚWIETLAMY BŁĘDY

Po wystąpieniu błędów walidacji szczegóły ich dotyczące przechowywane są w żądaniu wraz z danymi zapisanymi w modelu. Musimy więc tylko „zanurkować" w modelu i wyciągnąć błędy, aby móc je pokazać użytkownikowi. Znacznik `<sf:errors>` znacznie upraszcza to zadanie.

Przyjrzyjmy się na przykład, jak znacznik `<sf:errors>` jest wykorzystywany w poniższym przykładzie w pliku *registerForm.jsp*:

```
<sf:form method="POST" commandName="spitter">
   Imię: <sf:input path="firstName" />
   <sf:errors path="firstName" /><br/> ...
</sf:form>
```

Pokazany jest tu tylko fragment użycia `<sf:errors>` w połączeniu z polem *Imię*, ale podobnie można zastosować ten znacznik dla pozostałych pól formularza. W przykładzie wartością atrybutu `path` jest `firstName`, nazwa właściwości obiektu modelu `Spitter`, dla którego mają być wyświetlone błędy. Jeśli dla tego pola nie ma żadnych błędów walidacji, znacznik `<sf:errors>` nic nie wyświetli. Jeżeli jednak wystąpią jakieś błędy, zostaną wygenerowane wewnątrz znacznika HTML `<span>`.

Na przykład jeśli użytkownik wpisze jako swoje imię wartość `J`, zostanie wygenerowany następujący kod HTML dla pola `firstName`:

```
Imię: <input id="firstName" name="firstName" type="text" value="J"/>
<span id="firstName.errors">size must be between 2 and 30</span>
```

Pokazujemy teraz użytkownikom ich błędy, dzięki czemu mogą je oni naprawić. Prezentację danych możemy jeszcze ulepszyć, modyfikując styl błędu, co sprawi, że będzie się bardziej wyróżniał. Możemy do tego wykorzystać atrybut `cssClass`:

```
<sf:form method="POST" commandName="spitter" >
   Imię: <sf:input path="firstName" />
   <sf:errors path="firstName" cssClass="error" /><br/>
   ...
</sf:form>
```

Ponownie, by zachować zwięzłość przykładu, ustawiliśmy atrybut `cssClass` tylko w tym znaczniku `<sf:errors>`, w którym atrybut `path` ma wartość `firstName`. Zmiany możemy też oczywiście zastosować dla pozostałych pól.

Teraz atrybut `class` znacznika `<span>` ma wartość error. Pozostaje nam już jedynie zdefiniowanie stylu CSS dla tej klasy. Poniżej znajduje się prosty styl CSS, dzięki któremu błędy mają kolor czerwony:

```
span.error {
   color: red;
}
```

Na rysunku 6.2 widać, jak w tym momencie wygląda formularz w oknie przeglądarki.

Rysunek 6.2. Wyświetlanie błędów walidacji obok pól formularza

Wyświetlanie błędów walidacji obok problematycznego pola jest dobrym sposobem na skierowanie uwagi użytkownika na pole wymagające naprawy. Może jednak sprawiać kłopot z układem strony. Alternatywnym rozwiązaniem jest wyświetlenie wszystkich błędów walidacji w jednym miejscu. W tym celu musimy usunąć znaczniki `<sf:errors>` z każdego pola i umieścić je u góry formularza w sposób przedstawiony poniżej:

```
<sf:form method="POST" commandName="spitter" >
   <sf:errors path="*" element="div" cssClass="errors" />
   ...
</sf:form>
```

Zmianą widoczną w tym przykładzie, dokonaną na znaczniku <sf:errors>, jest przypisanie do atrybutu path wartości *. Jest to symbol wieloznaczny, który informuje znacznik <sf:errors>, aby wygenerował komunikaty o błędach dla wszystkich właściwości.

Warto też zauważyć, że wartość atrybutu element ustawiliśmy na div. Przy ustawieniu domyślnym błędy generowane są wewnątrz znacznika HTML , co sprawdza się świetnie w sytuacji, gdy mamy do czynienia z jednym błędem. Kiedy jednak musimy wyświetlić komunikaty o błędach dotyczące wszystkich pól, widocznych może być równocześnie wiele błędów. W tej sytuacji znacznik (znacznik typu inline) nie jest najlepszym rozwiązaniem. Lepsze byłoby użycie znacznika blokowego, takiego jak . W tym celu definiujemy atrybut element jako div, a wszystkie błędy zostaną wypisane wewnątrz niego.

Tak jak poprzednio, atrybutowi cssClass ustawiliśmy wartość errors, żebyśmy mogli ustawić styl naszego elementu . Poniższy przykład stylu CSS powoduje otoczenie elementu czerwoną ramką i jasnoczerwonym tłem.

```
div.errors {
   background-color: #ffcccc;
   border: 2px solid red;
}
```

Teraz, po przeniesieniu wszystkich błędów do góry formularza, przygotowanie układu strony jest prostsze. Straciliśmy jednak możliwość podświetlania pól, których wartości wymagają poprawy. Pomoże nam w tym atrybut cssErrorClass, dostępny dla każdego pola. Poniżej znajduje się kod dla pola *Imię* ze wszystkimi naniesionymi zmianami:

```
<sf:form method="POST" commandName="spitter" >
   <sf:label path="firstName" cssErrorClass="error">Imię</sf:label>:
   <sf:input path="firstName" cssErrorClass="error" /><br/> ...
</sf:form>
```

Znacznik <sf:label>, podobnie jak inne znaczniki wiązania formularza, posiada atrybut path, wskazujący właściwość obiektu modelu, do którego należy. W tym przypadku wartością atrybutu jest firstName, następuje więc dowiązanie z właściwością firstName obiektu Spitter. Przy założeniu, że nie wystąpią żadne błędy walidacji, wygenerowany kod HTML będzie wyglądał następująco:

```
<label for="firstName">Imię</label>
```

Ustawienie atrybutu path w znaczniku <sf:label> samo w sobie nie robi zbyt dużo. Ustawiamy też jednak atrybut cssErrorClass. Jeśli powiązana właściwość zawiera jakieś błędy, atrybut class wygenerowanego znacznika <label> przyjmie wartość error:

```
<label for="firstName" class="error">Imię</label>
```

Podobnie atrybut cssErrorClass znacznika <sf:input> ma teraz wartość error. Jeżeli wystąpi błąd walidacji, to wartością atrybutu class wygenerowanego znacznika <input> staje się error. Teraz możemy przygotować style dla etykiety i pól, aby przyciągały uwagę użytkownika, gdy zajdzie potrzeba ich poprawienia. Przykładowo poniższy kod CSS wygeneruje etykietę w kolorze czerwonym i pole wpisywania o jasnoczerwonym tle:

```
label.error {
   color: red;
}
input.error {
   background-color: #ffcccc;
}
```

Mamy już ładny sposób prezentacji błędów użytkownika. To jednak nie wszystko, co możemy zrobić, by komunikaty błędów były przyjemne w odbiorze. W klasie `Spitter` ustawmy wartość atrybutu `message` w adnotacjach walidacji, żeby użyć czytelnych, zdefiniowanych w plikach właściwości komunikatów o błędach:

```
@NotNull
@Size(min=5, max=16, message="{username.size}")
private String username;
@NotNull
@Size(min=5, max=25, message="{password.size}")
private String password;
@NotNull
@Size(min=2, max=30, message="{firstName.size}")
private String firstName;
@NotNull
@Size(min=2, max=30, message="{lastName.size}")
private String lastName;
@NotNull
@Email(message="{email.valid}")
private String email;
```

Wartością atrybutu `message` adnotacji `@Size` każdego ze zdefiniowanych pól jest ciąg znaków otoczony przez parę nawiasów klamrowych. Gdybyśmy pominęli te nawiasy, użytkownik otrzymałby komunikat błędu o treści będącej tym właśnie ciągiem. Dzięki zastosowaniu nawiasów klamrowych treść komunikatu pobierana jest z pliku właściwości.

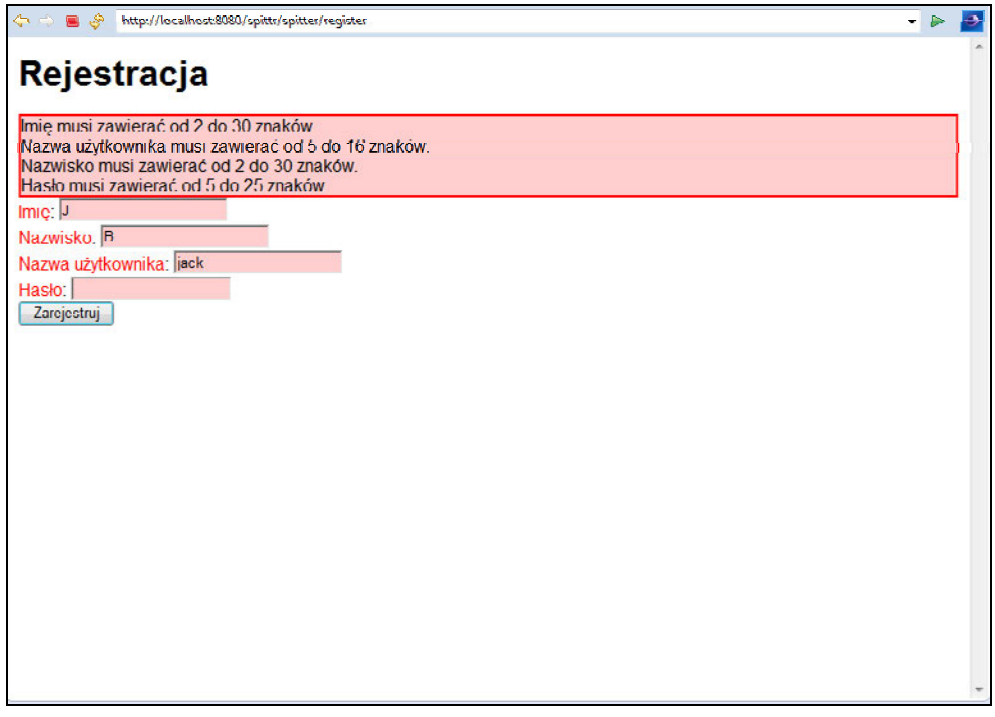
Pozostaje nam więc utworzenie tego pliku. Nadamy mu nazwę *ValidationMessages.properties* i umieścimy w głównym folderze ścieżki klas:

```
firstName.size=Imię musi zawierać od {min} do {max} znaków.
lastName.size=Nazwisko musi zawierać od {min} do {max} znaków.
username.size=Nazwa użytkownika musi zawierać od {min} do {max} znaków.
password.size=Hasło musi zawierać od {min} do {max} znaków.
email.valid=Adres e-mail musi być poprawny.
```

Zauważmy, że klucz każdego z komunikatów w pliku *ValidationMessages.properties* odpowiada wartości symboli zastępczych ustawionych w atrybutach `message`. Równocześnie w komunikatach w pliku *ValidationMessages.properties* nie wpisano na sztywno wartości minimalnej i maksymalnej długości ciągu znaków. Pojawiły się za to symbole zastępcze `{min}` i `{max}`, które odnoszą się do atrybutów `min` i `max` ustawionych w adnotacji `@Size`.

Po zatwierdzeniu formularza rejestracji i niepomyślnej walidacji użytkownik zobaczy w przeglądarce stronę przypominającą rysunek 6.3.

Użycie plików właściwości do przechowywania komunikatów jest pożyteczne, bo umożliwia wyświetlenie komunikatów w języku użytkownika oraz zastosowanie odpowiednich ustawień regionalnych dzięki przygotowaniu osobnych plików właściwości.



Rysunek 6.3. Błędy walidacji wyświetlone z użyciem przyjaznych komunikatów zapisanych w pliku właściwości

Na przykład aby wyświetlić komunikaty błędów w języku hiszpańskim, wtedy gdy przeglądarka użytkownika korzysta z tego właśnie języka, możemy utworzyć plik *ValidationErrors_es.properties* o następującej zawartości:

```
firstName.size=
   Nombre debe ser entre {min} y {max} caracteres largo.
lastName.size=
   El apellido debe ser entre {min} y {max} caracteres largo.
username.size=
   Nombre de usuario debe ser entre {min} y {max} caracteres largo.
password.size=
   Contraseña debe estar entre {min} y {max} caracteres largo.
email.valid=La dirección de email no es válida
```

Możemy stworzyć tak wiele wersji `ValidationMessages.properties` w różnych językach, ile musimy mieć, by zaspokoić wszystkie potrzeby w tym zakresie.

BIBLIOTEKA OGÓLNYCH ZNACZNIKÓW SPRINGA

Poza biblioteką znaczników wiązania formularza Spring oferuje też bardziej ogólną bibliotekę znaczników JSP. W praktyce to właśnie ta biblioteka znaczników JSP pojawiła się w Springu jako pierwsza. Przez wiele lat biblioteka ta zdążyła się rozrosnąć, ale w prostej postaci była dostępna już w najwcześniejszych wersjach Springa.

Aby skorzystać z możliwości biblioteki ogólnych znaczników Springa, musimy ją zadeklarować na stronie JSP:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
```

Tak jak w przypadku każdej innej biblioteki znaczników JSP, tak i w tym przypadku atrybut `prefix` może mieć dowolną wartość. W odniesieniu do tej biblioteki najczęściej stosuje się prefiks `spring`. Ja jednak preferuję użycie prefiksu `s`, bo jest dużo zwięźlejszy i łatwiejszy we wpisywaniu oraz odczytywaniu kodu.

Po zadeklarowaniu biblioteki znaczników możemy rozpocząć pracę z 10 znacznikami JSP wypisanymi w tabeli 6.3.

Tabela 6.3. Biblioteka innych znaczników JSP Springa udostępnia, poza paroma „przestarzałymi" już znacznikami wiązania danych, kilka użytecznych znaczników narzędziowych

Znacznik JSP	Opis
`<s:bind>`	Eksportuje stan powiązanej właściwości do właściwości `status` o zasięgu strony. Wykorzystywany obok znacznika `<s:path>` do pozyskania wartości powiązanej właściwości.
`<s:escapeBody>`	Kod HTML i (lub) JavaScript w ciele znacznika poddawany jest „eskejpowaniu".
`<s:hasBindErrors>`	Warunkowo generuje zawartość, jeśli wskazany obiekt modelu (w atrybucie żądania) zawiera błędy wiązania.
`<s:htmlEscape>`	Ustawia domyślną wartość HTML dla aktualnej strony.
`<s:message>`	Pobiera komunikat z podanym kodem i generuje go (domyślnie) lub przypisuje do zmiennej o zasięgu strony, żądania, sesji bądź aplikacji (przy zastosowaniu atrybutów `var` oraz `scope`).
`<s:nestedPath>`	Ustawia zagnieżdżoną ścieżkę poprzez użycie znacznika `<s:bind>`.
`<s:theme>`	Pobiera komunikat tematu z danym kodem i generuje go (domyślnie) lub przypisuje go do zmiennej o zasięgu strony, żądania, sesji albo aplikacji (przy zastosowaniu atrybutów `var` oraz `scope`).
`<s:transform>`	Przetwarza właściwości nieistniejące w obiekcie wspierającym za pomocą edytorów właściwości obiektu wspierającego.
`<s:url>`	Tworzy adresy URL uzależnione od kontekstu ze wsparciem zmiennych szablonów URI i „eskejpowania" kodu HTML/XML/JavaScript. Może generować adresy URL (domyślnie) lub przypisywać je do zmiennej o zasięgu strony, żądania, sesji bądź aplikacji (przy zastosowaniu atrybutów `var` oraz `scope`).
`<s:eval>`	Wylicza wartość wyrażenia SpEL (*Spring Expression Language*) i generuje wynik (domyślnie) lub przypisuje go do zmiennej o zasięgu strony, żądania, sesji albo aplikacji (przy zastosowaniu atrybutów `var` oraz `scope`).

Kilka z przedstawionych powyżej znaczników zostało po wprowadzeniu biblioteki znaczników wiązania formularzy Springa uznanych za „przestarzałe". Przykładowo znacznik `<s:bind>` był pierwotnie znacznikiem wiązania formularzy, a jego stosowanie było dużo bardziej skomplikowane niż stosowanie znaczników opisanych w poprzedniej sekcji. Prezentowana biblioteka znaczników jest używana dużo rzadziej niż biblioteka wiązania formularzy, nie będę więc omawiać szczegółowo poszczególnych jej znaczników. Opiszę tylko pokrótce kilka najczęściej stosowanych, a Tobie pozostawię dokładniejsze ich zbadanie. (Istnieje duże prawdopodobieństwo, że przydadzą Ci się one niezwykle rzadko, a może nawet nigdy).

WYŚWIETLANIE ZINTERNACJONALIZOWANYCH KOMUNIKATÓW

W tej chwili nasze szablony JSP zawierają dużo tekstu wpisanego na stałe. Nie ma w tym niby nic szczególnie złego, w ten sposób nie mamy jednak łatwego sposobu zmiany tego tekstu. Co więcej, nie ma możliwości umiędzynarodowienia tekstu, aby uzależnić go od ustawień językowych użytkownika.

Rozważmy na przykład komunikat powitalny na stronie domowej:

```
<h1>Witaj w aplikacji Spittr!</h1>
```

Jedyną metodą modyfikacji tego komunikatu jest otwarcie pliku *home.jsp* i wykonanie zmian. Nie jest to może jakiś wielki problem. Ale rozproszenie tekstu po plikach widoku oznacza konieczność modyfikacji wielu plików JSP w przypadku dużych zmian komunikatów.

Istotniejszym problemem jest to, że niezależnie od wyboru tekstu ekranu powitalnego wszyscy użytkownicy zobaczą ten sam komunikat. Internet jest siecią globalną, a tworzone aplikacje mogą być przeznaczone dla szerokiego grona użytkowników. Rozsądna jest więc komunikacja z użytkownikiem aplikacji w jego własnym języku, a nie zmuszanie go do wykorzystywania jakiegoś jednego konkretnego języka.

Znacznik `<s:message>` jest świetnym rozwiązaniem do generowania tekstu przeniesionego do jednego lub wielu plików właściwości. Dzięki temu znacznikowi możemy zastąpić nasz tekst powitalny zaszyty w szablonie strony:

```
<h1><s:message code="spittr.welcome" /></h1>
```

W tym przykładzie znacznik `<s:message>` wygeneruje tekst dostępny w źródle komunikatów pod kluczem `spittr.welcome`. Aby znacznik prawidłowo zadziałał, musimy więc skonfigurować takie źródło komunikatów.

Spring udostępnia kilka klas źródeł danych. Wszystkie one implementują interfejs `MessageSource`. Jedną z najczęściej stosowanych i najbardziej użytecznych implementacji jest klasa `ResourceBundleMessageSource`. Pobiera ona komunikaty z pliku właściwości o nazwie opartej na ustalonej nazwie podstawowej. Poniższa metoda komponentu konfiguruje źródło komunikatów `ResourceBundleMessageSource`:

```
@Bean
public MessageSource messageSource() {
   ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
   messageSource.setBasename("messages");
   return messageSource;
}
```

Kluczowym elementem tej deklaracji jest ustawienie właściwości `basename`. Można ją ustawić na dowolną pożądaną wartość, ale w tym miejscu zdecydowaliśmy się na użycie słowa `messages`. Dzięki tej deklaracji klasa `ResourceBundleMessageResolver` produkuje komunikaty z wykorzystaniem plików właściwości znajdujących się w głównym folderze ścieżki klas o nazwie wywodzącej się od nazwy podstawowej.

Możemy też wybrać klasę `ReloadableResourceBundleMessageSource`, która działa tak samo jak `ResourceBundleMessageSource`, lecz udostępnia możliwość ponownego wczytywania właściwości komunikatów bez rekompilacji lub restartu aplikacji. Poniżej znajduje się przykładowa konfiguracja klasy `ReloadableResourceBundleMessageSource`:

```
@Bean
public MessageSource messageSource() {
   ReloadableResourceBundleMessageSource messageSource = new
      ReloadableResourceBundleMessageSource();
   messageSource.setBasename("file:///etc/spittr/messages");
   messageSource.setCacheSeconds(10); return messageSource;
}
```

Kluczową różnicą jest tu takie ustawienie właściwości `basename`, aby nazwy plików szukane były poza aplikacją (a nie w ścieżce klas, jak to jest w przypadku klasy `ResourceBundleMessageSource`). Właściwość `basename` można też ustawić tak, by wyszukiwała komunikaty w ścieżce klas (za pomocą prefiksu `classpath`), w ścieżce do systemu plików (za pomocą prefiksu `file`) albo w głównym katalogu aplikacji (gdy nie użyjemy żadnego prefiksu). W powyższym przykładzie wyszukujemy komunikaty w plikach właściwości o nazwie podstawowej `messages` w katalogu */etc/spittr* w systemie plików serwera.

Utwórzmy teraz te pliki właściwości. Na początek stwórzmy domyślny plik właściwości o nazwie *messages.properties*. Zlokalizowany on będzie w głównym katalogu ścieżki klas (jeśli korzystamy z klasy `ResourceBundleMessageSource`) lub w ścieżce określonej we właściwości `basename` (jeśli korzystamy z klasy `ReloadableResourceBundleMessageSource`). Potrzebny nam jest następujący komunikat dla klucza `spitrr.welcome`:

```
spittr.welcome=Witaj w aplikacji Spittr!
```

Jeżeli nie utworzymy żadnych innych plików, to jedyną korzyścią z tej zmiany będzie wyodrębnienie do osobnego pliku komunikatu zaszytego wcześniej na sztywno w pliku JSP. Dostajemy możliwość edycji wszystkich komunikatów w jednym miejscu, ale niewiele więcej.

Tak czy inaczej, podstawowy krok w kierunku procesu internacjonalizacji został wykonany. Jeśli chcielibyśmy teraz pokazać komunikat powitalny w języku hiszpańskim wszystkim użytkownikom, których przeglądarka została ustawiona na obsługę tego języka, musielibyśmy utworzyć nowy plik właściwości o nazwie *messages_es.properties*:

```
spittr.welcome=Bienvenidos a Spittr!
```

To już duże osiągnięcie. Od międzynarodowego sukcesu dzieli nas tylko kilka znaczników `<s:message>` i plików właściwości zawierających komunikaty w różnych językach. Tobie zatem pozostawiam umiędzynarodowienie pozostałej części naszej aplikacji.

TWORZYMY ADRESY URL

Znacznik `<s:url>` jest niewielki i skromny. Jego głównym zadaniem jest tworzenie adresu URL i przypisywanie go do zmiennej lub generowanie w odpowiedzi. Zastępuje on znacznik JSTL `<c:url>`, posiada jednak kilka dodatkowych możliwości.

W najprostszej postaci `<s:url>` przyjmuje adres URL względny wobec kontekstu serwletu i przy generowaniu uzupełnia go prefiksem ścieżki kontekstu serwletu. Popatrzmy na najprostszy przykład użycia znacznika `<s:url>`:

```
<a href="<s:url href="/spitter/register" />">Zarejestruj</a>
```

Jeśli nazwą kontekstu serwletu aplikacji jest spittr, to w odpowiedzi wygenerowany zostanie następujący kod HTML:

```
<a href="/spittr/spitter/register">Zarejestruj</a>
```

Umożliwia to tworzenie adresów URL bez obawy o ustawienie ścieżki kontekstu serwletu. Zajmuje się tym za nas znacznik `<s:url>`.

Znacznika `<s:url>` możemy też użyć do konstruowania adresu URL i przypisania go do zmiennej, którą wykorzystamy w innym miejscu w szablonie:

```
<s:url href="/spitter/register" var="registerUrl" />
<a href="${registerUrl}">Zarejestruj</a>
```

W ustawieniu domyślnym zmienne URL tworzone są w zasięgu strony. Znacznik `<s:url>` pozwala nam też jednak na ich tworzenie w zasięgu aplikacji, sesji lub żądania dzięki wykorzystaniu atrybutu `scope`:

```
<s:url href="/spitter/register" var="registerUrl" scope="request" />
```

Znacznik `<s:param>` umożliwia nam dodanie parametrów do adresu URL. Na przykład poniżej przedstawiono użycie znacznika `<s:url>` z dwoma zagnieżdżonymi znacznikami `<s:param>`, służącymi do ustawienia parametrów `max` i `count` dla ścieżki `/spittles`:

```
<s:url href="/spittles" var="spittlesUrl">
   <s:param name="max" value="60" />
   <s:param name="count" value="20" />
</s:url>
```

Jak na razie nie przedstawiłem żadnych możliwości znacznika `<s:url>`, których nie można by osiągnąć za pomocą znacznika JSTL `<c:url>`. Załóżmy, że chcemy utworzyć adres URL zawierający parametr `path`. Jak stworzyć wartość `href` tak, aby można było ten parametr podmienić?

Przypuśćmy, że chcemy utworzyć adres URL dla konkretnej strony profilu użytkownika. Nie ma problemu. Pomoże nam w tym ponownie znacznik `<s:param>`:

```
<s:url href="/spitter/{username}" var="spitterUrl">
   <s:param name="username" value="jbauer" />
</s:url>
```

Gdy wartość `href` jest symbolem zastępczym, odpowiadającym parametrowi określonemu w znaczniku `<s:param>`, parametr ten jest wstawiany w miejsce symbolu zastępczego. Jeśli parametr `<s:param>` nie odpowiada żadnym symbolom zastępczym w parametrze `href`, parametr ten wykorzystywany jest jako parametr zapytania.

Znacznik `<s:url>` wykonuje też „eskejpowanie" na potrzeby tworzenia adresu URL. Przykładowo jeśli chcemy wygenerować adres URL jako zwykłą zawartość strony internetowej (nie w postaci łącza), możemy włączyć tryb „eskejpowania" HTML za pomocą opcji `htmlEscape` ustawionej na `true`. Poniższy przykład znacznika `<s:url>` generuje „wyeskejpowany" adres URL w trybie HTML:

```
<s:url value="/spittles" htmlEscape="true">
   <s:param name="max" value="60" />
   <s:param name="count" value="20" />
</s:url>
```

Spowoduje to wygenerowanie poniższego adresu URL:

```
/spitter/spittles?max=60&amp;count=20
```

Jeżeli z kolei chcielibyśmy użyć adresu URL w kodzie JavaScript, możemy zastosować atrybut `javaScriptEnable` z wartością `true`:

```
<s:url value="/spittles" var="spittlesJSUrl" javaScriptEscape="true">
   <s:param name="max" value="60" />
   <s:param name="count" value="20" />
</s:url>
<script>
   var spittlesUrl = "${spittlesJSUrl}"
</script>
```

Spowoduje to wygenerowanie następującej odpowiedzi:

```
<script>
   var spittlesUrl = "\/spitter\/spittles?max=60&count=20"
</script>
```

Jeśli chodzi o „eskejpowanie", to istnieje inny znacznik służący do „eskejpowania" zawartości niebędącej znacznikiem. Przyjrzyjmy mu się teraz bliżej.

„ESKEJPUJEMY" ZAWARTOŚĆ

Znacznik `<s:escapeBody>` to znacznik „eskejpowania" ogólnego przeznaczenia. Generuje zagnieżdżoną w jego ciele zawartość i w razie potrzeby poddaje ją „eskejpowaniu".

Przypuśćmy, że chcemy wyświetlić na stronie fragment kodu HTML. Aby był on prawidłowo wyświetlony, musimy zastąpić znaki < i > odpowiednio przez `<` oraz `>`. W przeciwnym wypadku przeglądarka zinterpretuje ten kod jak każdy inny kod HTML na stronie.

Oczywiście możemy sami wykonać ręcznie wszystkie zastąpienia, ale jest to uciążliwe, a otrzymany kod źle się czyta. Zamiast tego możemy wykorzystać znacznik `<s:escape` ➥`Body>` i pozwolić Springowi zająć się za nas tym problemem:

```
<s:escapeBody htmlEscape="true">
   <h1>Cześć</h1>
</s:escapeBody>
```

W odpowiedzi otrzymamy następujący kod:

```
&lt;h1&gt;Hello&lt;/h1&gt;
```

Oczywiście, chociaż w tej postaci wygląda to okropnie, przeglądarka wyświetli odpowiedź jako „niewyeskejpowany" kod HTML, który chcieliśmy otrzymać.

Znacznik `<s:escapeBody>` obsługuje też „eskejpowanie" dla JavaScriptu za pomocą atrybutu `javaScriptEscape`:

```
<s:escapeBody javaScriptEscape="true">
   <h1>Cześć</h1>
</s:escapeBody>
```

Znacznik `<s:escapeBody>` ma jedno zadanie i świetnie sobie z nim radzi. W przeciwieństwie do znacznika `<s:url>` umożliwia jedynie wygenerowanie zawartości strony i nie pozwala na przypisanie jej do zmiennej.

Teraz gdy dowiedziałeś się, jak wykorzystać pliki JSP do definiowania widoków w Springu, pomyślmy, co zrobić, żeby wyglądały one trochę ciekawiej. Wiele w tym

zakresie możemy osiągnąć poprzez umieszczenie na stronach różnych wspólnych elementów, takich jak nagłówki, logo, zastosowanie arkuszy stylów lub nawet informacji o prawach autorskich w stopce strony. Nie chcielibyśmy jednak dodawać tych elementów do wszystkich plików JSP w aplikacji Spittr. Zobaczmy więc, jak wykorzystać Apache Tiles do pracy z elementami wspólnymi dla wielu stron oraz z układami stron.

6.3. Definiujemy układ stron za pomocą widoków Apache Tiles

Jak na razie nie zrobiliśmy zbyt wiele w kwestii budowy układu strony w naszej aplikacji. Każda strona JSP w pełni odpowiada za definiowanie swojego własnego układu, ale niezbyt dużo w tym zakresie robi.

Przypuśćmy, że chcemy dodać wspólny nagłówek i stopkę do wszystkich stron w aplikacji. Można byłoby odwiedzić każdy szablon JSP po kolei i do każdego z nich dodać kod HTML nagłówka i stopki. Takie naiwne podejście nie sprawdza się jednak w utrzymaniu rozrastającej się aplikacji. Płacimy pewien wstępny koszt dodawania tych elementów do każdej strony i każda przyszła modyfikacja generuje podobny koszt.

Lepszym rozwiązaniem jest użycie silnika układów, takiego jak Apache Tiles, do zdefiniowania wspólnego układu strony i wykorzystanie go na wszystkich pozostałych stronach. Spring MVC zawiera wsparcie dla Apache Tiles w postaci producenta widoków, który umożliwia odwzorowywanie logicznych nazw widoków na definicje kafelków.

6.3.1. Konfigurujemy producenta widoków Tiles

Aby zintegrować Tiles ze Springiem, musimy skonfigurować kilka komponentów. Potrzebny nam jest komponent `TilesConfigurer`, którego zadaniem jest zlokalizowanie i wczytanie definicji kafelków, jak również ogólne zarządzanie działaniem Apache Tiles. Dodatkowo potrzebny nam też będzie komponent `TilesViewResolver` do mapowania nazw widoków na definicje kafelków.

Ta para wspomnianych komponentów dostępna jest w dwóch wydaniach: dla Apache Tiles 2 oraz dla Apache Tiles 3. Najistotniejszą różnicą pomiędzy oboma zestawami komponentów jest nazwa pakietów, w których się znajdują. Para komponentów `TilesConfigurer/TilesViewResolver` dla Apache Tiles 2 jest w pakiecie `org.springframework.web.servlet.view.tiles2`, a ich odpowiednik dla Apache Tiles 3 — w pakiecie `org.springframework.web.servlet.view.tiles3`. W naszym przykładzie wykorzystamy wersję Tiles 3.

Na początek dodajmy komponent TilesConfigurer, tak jak pokazano na listingu 6.1.

Listing 6.1. Konfigurujemy component TilesConfigurer do rozwiązywania definicji kafelków

```
@Bean
public TilesConfigurer tilesConfigurer() {
   TilesConfigurer tiles = new TilesConfigurer();
   tiles.setDefinitions(new String[] {          <-- Określamy lokalizację definicji kafelków
      "/WEB-INF/layout/tiles.xml"
   });
```

```
    tiles.setCheckRefresh(true);
    return tiles;
}
```

Włączamy odświeżanie

Przy konfiguracji ustawień komponentu `TilesConfigurer` najważniejszą właściwością jest `definitions`. Właściwość ta pobiera tablicę ciągów znaków, w której każdy wpis określa lokalizację plików XML zawierających definicję kafelków. W przypadku aplikacji Spittr będziemy szukać pliku *tiles.xml* w katalogu */WEB-INF/layout*.

Istnieje też możliwość określenia wielu plików z definicjami kafelków, a nawet wykorzystania symboli wieloznacznych w ścieżce lokalizacji. W naszym przykładzie nie skorzystaliśmy jednak z tych opcji. Przykładowo moglibyśmy za pomocą właściwości `definitions` poprosić komponent `TilesConfigurer` o wyszukanie wszystkich plików o nazwie *tiles.xml*, znajdujących się gdzieś w głębi katalogu */WEB-INF/*:

```
tiles.setDefinitions(
    new String[] {
        "/WEB-INF/**/tiles.xml"
});
```

W tym przypadku korzystamy z symboli wieloznacznych w stylu Anta (**), aby komponent `TilesConfigurer` przejrzał wszystkie podkatalogi */WEB-INF/* w poszukiwaniu definicji kafelków.

Następnie skonfigurujmy producenta `TilesViewResolver`. Jak widać, definicja komponentu jest raczej prosta i nie modyfikuje ustawień żadnych właściwości:

```
@Bean
public ViewResolver viewResolver() {
    return new TilesViewResolver();
}
```

Opcjonalnie komponenty `TilesConfigurer` i `TilesViewResolver` możesz skonfigurować za pomocą konfiguracji XML:

```
<bean id="tilesConfigurer"
      class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/layout/tiles.xml.xml</value>
            <value>/WEB-INF/views/**/tiles.xml</value>
        </list>
    </property>
</bean>
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.tiles3.TilesViewResolver" />
```

Komponent `TilesConfigurer` wczytuje definicje kafelków i zarządza pracą Apache Tiles, a `TilesViewResolver` odpowiada za odwzorowywanie logicznych nazw widoków na widoki odpowiadające definicjom kafelków. Robi to poprzez wyszukanie definicji kafelka o nazwie odpowiadającej logicznej nazwie widoku. Aby zobaczyć działanie kafelków, musimy przygotować kilka ich definicji.

DEFINIUJEMY KAFELKI

Apache Tiles udostępnia definicję DTD (*Document Type Definition*) umożliwiającą zdefiniowanie kafelków w pliku XML. Każda definicja składa się z elementu `<definition>`, zawierającego z reguły jeden lub więcej elementów `<put-attribute>`. Przykładowy dokument XML przedstawiony na listingu 6.2 definiuje kilka kafelków dla aplikacji Spittr.

Listing 6.2. Definiujemy kafelki dla aplikacji Spittr

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
   "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
   "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
   <tiles-definitions>
      <definition name="base" <-- Definiujemy bazowy kafelek
         template="/WEB-INF/layout/page.jsp">
         <put-attribute name="header" value="/WEB-INF/layout/header.jsp" />
         <put-attribute name="footer" value="/WEB-INF/layout/footer.jsp" /> <-- Ustawiamy atrybut
      </definition>
      <definition name="home" extends="base"> <-- Rozszerzamy bazowy kafelek
         <put-attribute name="body" value="/WEB-INF/views/home.jsp" />
      </definition>
      <definition name="registerForm" extends="base">
         <put-attribute name="body" value="/WEB-INF/views/registerForm.jsp" />
      </definition>
      <definition name="profile" extends="base">
         <put-attribute name="body" value="/WEB-INF/views/profile.jsp" />
      </definition>
      <definition name="spittles" extends="base">
         <put-attribute name="body" value="/WEB-INF/views/spittles.jsp" />
      </definition>
      <definition name="spittle" extends="base">
         <put-attribute name="body" value="/WEB-INF/views/spittle.jsp" />
      </definition>
   </tiles-definitions>
```

Każdy element `<definition>` zawiera definicję kafelka, który z kolei odwołuje się do szablonu JSP. W przypadku kafelka o nazwie `base` szablonem tym jest */WEB-INF/layout/page.jsp*. Kafelek może się też odwoływać do innych szablonów JSP, które mają zostać zagnieżdżone w szablonie głównym. W przypadku kafelka `base` następuje odwołanie do szablonów JSP `header` oraz `footer`.

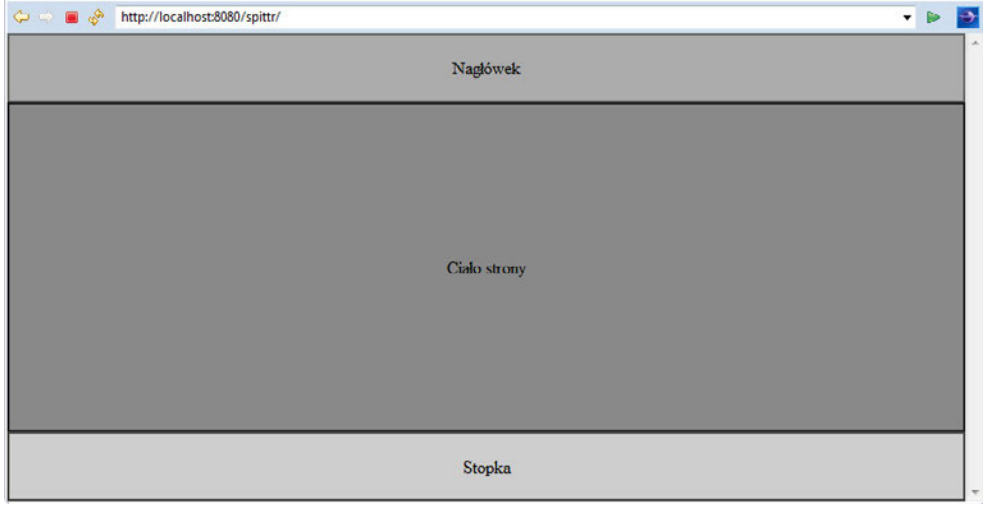
Kafelek `base` odwołuje się do szablonu *page.jsp*, którego kod znajduje się na listingu 6.3.

Listing 6.3. Główny szablon układu: zawiera odniesienia do innych szablonów, aby utworzyć widok

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="t" %>
<%@ page session="false" %>
<html>
   <head>
      <title>Spittr</title>
      <link rel="stylesheet"
            type="text/css"
```

```
            href="<s:url value="/resources/style.css" />" >
    </head>
    <body>
        <div id="header">
            <t:insertAttribute name="header" /> <-- Wstawiamy nagłówek
        </div>
        <div id="content">
            <t:insertAttribute name="body" /> <-- Wstawiamy ciało strony
        </div>
        <div id="footer">
            <t:insertAttribute name="footer" /> <-- Wstawiamy stopkę
        </div>
    </body>
</html>
```

Najciekawszym elementem listingu 6.3 jest sposób wykorzystania znacznika JSP `<t:insertAttribute>` z biblioteki znaczników Tiles do wstawiania innych szablonów. Używamy go do wstawienia atrybutów o nazwie `header` (nagłówek), `body` (ciało strony) oraz `footer` (stopka). W rezultacie otrzymujemy układ strony przypominający układ przedstawiony na rysunku 6.4.



Rysunek 6.4. Ogólny układ strony definiujący nagłówek, ciało strony oraz stopkę

Atrybuty `header` i `footer` ustawione w definicji podstawowego kafelka (`base`) wskazują odpowiednio na pliki */WEB-INF/layout/header.jsp* oraz */WEB-INF/layout/footer.jsp*. Co jednak z atrybutem body? Gdzie on jest ustawiony?

Kafelek `base` nie powinien być nigdy wykorzystywany bezpośrednio. Służy jako podstawowa definicja (stąd pochodzi jego nazwa), która jest następnie rozszerzana przez inne kafelki. Na listingu 6.2 widać wyraźnie, że wszystkie pozostałe kafelki rozszerzają kafelek `base`. Dziedziczą ustawienia atrybutów `header` oraz `footer` kafelka bazowego (choć w razie potrzeby mogą nadpisać dziedziczone wartości). Wszystkie kafelki definiują dodatkowo wartość atrybutu `body`, ustawiając jakiś konkretny szablon JSP.

Kiedy spojrzymy na definicję kafelka strony domowej `home`, zauważymy, że rozszerza on kafelek `base`. Dziedziczy w związku z tym szablon oraz wszystkie atrybuty kafelka podstawowego. Chociaż definicja kafelka `home` jest względnie prosta, w praktyce posiada następujące aktywne definicje:

```
<definition name="home" template="/WEB-INF/layout/page.jsp">
   <put-attribute name="header" value="/WEB-INF/layout/header.jsp" />
   <put-attribute name="footer" value="/WEB-INF/layout/footer.jsp" />
   <put-attribute name="body" value="/WEB-INF/views/home.jsp" />
</definition>
```

Poszczególne szablony wskazywane przez atrybuty są proste. Oto kod szablonu nagłówka *header.jsp*:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
<a href="<s:url value="/" />">
   <img src="<s:url value="/resources" />/images/spittr_logo_50.png"
        border="0"/>
</a>
```

Kod stopki jest jeszcze prostszy:

```
Copyright &copy; Craig Walls
```

Każdy kafelek dziedziczący z kafelka `base` definiuje swój własny szablon `body`, więc wszystkie kafelki różnią się od siebie. Oto kod szablonu *home.jsp* dla kafelka `home`:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<h1>Witaj w aplikacji Spittr</h1>
<a href="<c:url value="/spittles" />">Spittle</a> |
<a href="<c:url value="/spitter/register" />">Zarejestruj</a>
```

Najistotniejszą obserwacją jest to, że wszystkie elementy wspólne dla strony znajdują się w plikach *page.jsp*, *header.jsp* oraz *footer.jsp* i nie są dostępne w kodzie pozostałych szablonów kafelków. Umożliwia to ich wielokrotne wykorzystanie na różnych stronach i upraszcza utrzymanie tych elementów.

Aby zobaczyć, jak te elementy ze sobą współgrają, spójrzmy na rysunek 6 5. Strona zawiera podstawowe style i ilustracje, żeby wzmocnić wrażenia estetyczne towarzyszące użytkowaniu aplikacji. Szczegóły te nie mają większego znaczenia z punktu widzenia dyskusji dotyczącej tworzenia układów stron w Apache Tiles, postanowiłem więc nie omawiać ich w tym rozdziale. Tak czy inaczej, widzieliśmy, że różne komponenty strony zostały zebrane w całość z wykorzystaniem definicji kafelka generującego stronę domową aplikacji Spittr.

JSP jest od dawna najczęstszym wyborem, jeśli chodzi o systemy szablonów w aplikacjach internetowych utworzonych w Javie. Pojawił się jednak nowy pretendent do roli lidera, chętny do ubiegania się o palmę pierwszeństwa w tym obszarze. Jest nim Thymeleaf. W następnej sekcji dowiesz się, jak wykorzystać system szablonów Thymeleaf w aplikacjach Spring MVC.

Rysunek 6.5. Strona domowa aplikacji Spittr poukładana za pomocą Apache Tiles

6.4. Pracujemy z Thymeleaf

Chociaż szablony JSP są dostępne od dawna i są bardzo popularne w świecie internetowych aplikacji Java, to z ich wykorzystaniem wiąże się kilka przykrych problemów. Jednym z nich jest fakt, że choć pliki JSP przypominają swym wyglądem pliki HTML lub XML, to w praktyce nie są ani jednymi, ani drugimi. Większość szablonów JSP przybiera postać HTML i zawiera przy tym wiele dodatkowych znaczników z wielu różnych bibliotek JSP. I chociaż te biblioteki znaczników wnoszą do JSP możliwości dynamicznego generowania treści w zwięzłej postaci, to równocześnie niwelują szanse na otrzymanie poprawnie sformatowanego dokumentu. W skrajnym przypadku znacznik JSP można wykorzystać jako wartość parametru HTML:

```
<input type="text" value="<c:out value="${thing.name}"/>" />
```

Efektem ubocznym użycia bibliotek znaczników i braku dobrej postaci szablonów JSP jest to, że często jedynie z pozoru przypominają one produkowany kod HTML. W rzeczywistości przeglądanie niewygenerowanego szablonu JSP w przeglądarce internetowej lub edytorze HTML może prowadzić do mylnych i źle wyglądających rezultatów. Wyniki są nie tylko niekompletnie wygenerowane — są także wizualną katastrofą! Ponieważ JSP nie jest prawdziwym kodem HTML, wiele przeglądarek internetowych oraz edytorów nie potrafi choć w przybliżeniu wyświetlić wyglądu wygenerowanego szablonu.

Specyfikacja JSP jest też ściśle powiązana ze specyfikacją serwletu. Oznacza to, że z JSP można korzystać tylko w aplikacji internetowej opartej na mechanizmie serwletów. Szablony JSP nie nadają się do ogólniejszych zastosowań (takich jak formatowanie wiadomości e-mail) ani do aplikacji internetowych nieopartych na serwletach.

Podjęto już kilka prób przełamania hegemonii szablonów JSP jako głównej technologii wytwarzania widoków w aplikacjach Java. Najświeższym pretendentem do roli lidera jest Thymeleaf, który wydaje się rozwiązaniem bardzo obiecującym, dlatego warto poświęcić mu trochę uwagi. Szablony Thymeleaf są naturalne, a ich działanie nie opiera się na żadnej bibliotece znaczników. Możemy je edytować i generować wszędzie tam, gdzie mamy możliwość skorzystania ze zwykłego kodu HTML. Nie jesteśmy też

powiązani ze specyfikacją serwletów, możemy więc z nich korzystać wszędzie tam, gdzie wykorzystanie JSP nie jest dozwolone. Zobaczmy teraz, jak zastosować szablony Thymeleaf w aplikacji Spring MVC.

6.4.1. Konfigurujemy producenta widoków Thymeleaf

Włączenie obsługi szablonów Thymeleaf w Springu uzyskamy po skonfigurowaniu trzech komponentów:

- Producenta widoków `ThymeleafViewResolver`, który odwzorowuje nazwy widoków na szablony Thymeleaf.
- Silnika `SpringTemplateEngine`, który przetwarza szablony i generuje wyniki.
- Producenta `TemplateResolver`, który wczytuje szablony Thymeleaf.

Listing 6.4 przedstawia konfigurację Java zawierającą deklarację tych komponentów.

Listing 6.4. Włączamy obsługę Thymeleaf w Springu za pomocą konfiguracji Java

```
@Bean
public ViewResolver viewResolver(                          <-- Producent widoków Thymeleaf
   SpringTemplateEngine templateEngine) {
   ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
   viewResolver.setTemplateEngine(templateEngine);
   return viewResolver;
}
@Bean
public TemplateEngine templateEngine(                      <-- Silnik szablonów
   TemplateResolver templateResolver) {
   SpringTemplateEngine templateEngine = new SpringTemplateEngine();
   templateEngine.setTemplateResolver(templateResolver);
   return templateEngine;
}
@Bean
public TemplateResolver templateResolver() {               <-- Producent szablonów
   TemplateResolver templateResolver = new ServletContextTemplateResolver();
   templateResolver.setPrefix("/WEB-INF/templates/");
   templateResolver.setSuffix(".html");
   templateResolver.setTemplateMode("HTML5");
   return templateResolver;
}
```

Jeśli deklarację komponentów wolelibyśmy umieścić w plikach XML, możemy skorzystać z fragmentu kodu dostępnego na listingu 6.5.

Listing 6.5. Konfigurujemy obsługę Thymeleaf w Springu za pomocą konfiguracji XML

```
<bean id="viewResolver"                                    <-- Producent widoków Thymeleaf
      class="org.thymeleaf.spring3.view.ThymeleafViewResolver"
      p:templateEngine-ref="templateEngine" />
<bean id="templateEngine"                                  <-- Silnik szablonów
      class="org.thymeleaf.spring3.SpringTemplateEngine"
      p:templateResolver-ref="templateResolver" />
<bean id="templateResolver"                                <-- Producent szablonów
      class="org.thymeleaf.templateresolver.ServletContextTemplateResolver"
```

```
      p:prefix="/WEB-INF/templates/"
      p:suffix=".html"
      p:templateMode="HTML5" />
```

Niezależnie od wybranego sposobu konfiguracji, Thymeleaf zostanie przygotowany do generowania szablonów w odpowiedzi na żądania obsługiwane przez kontrolery Spring MVC.

Klasa `ThymeleafViewResolver` jest implementacją interfejsu `ViewResolver` Spring MVC. Jak każdy producent widoków, pobiera logiczną nazwę widoku i odwzorowuje na niego właściwy widok. W tym przypadku widokiem jest szablon Thymeleaf.

Zauważmy, że do komponentu `ThymeleafViewResolver` wstrzykiwana jest referencja do komponentu `SpringTemplateEngine`. `SpringTemplateEngine` jest silnikiem Thymeleaf z włączoną obsługą Springa, umożliwiającym parsowanie szablonów i generowanie wyników w oparciu o ich zawartość. Zależnością komponentu `SpringTemplateEngine` jest z kolei referencja do komponentu `TemplateResolver`.

Producent szablonów `TemplateResolver` jest tym elementem, który ostatecznie lokalizuje szablony. Jego konfiguracja przypomina konfigurację poznaną przy okazji producenta widoków `InternalResourceViewResolver` i wykorzystuje właściwości `prefix` oraz `suffix`. Prefiks i sufiks w połączeniu z nazwą widoku używane są do lokalizowania szablonu Thymeleaf. Ustawiliśmy też właściwość `templateMode` na wartość równą `HTML5`, co wskazuje, że oczekiwanym wynikiem generowania szablonu jest kod HTML5.

Po zakończeniu konfiguracji komponentów Thymeleaf nadszedł czas na utworzenie kilku szablonów.

6.4.2. Definiujemy szablony Thymeleaf

Szablony Thymeleaf są w zasadzie po prostu plikami HTML. W przeciwieństwie do szablonów JSP nie wykorzystują żadnych specjalnych znaczników ani bibliotek. Działanie szablonów Thymeleaf opiera się na zastosowaniu atrybutów dodawanych do standardowych znaczników HTML za pośrednictwem przestrzeni nazw Thymeleaf. Listing 6.6 zawiera kod szablonu strony głównej *home.html*, która wykorzystuje przestrzeń nazw Thymeleaf.

Listing 6.6. home.html: szablon strony domowej korzystający z przestrzeni nazw Thymeleaf

```
<html
   xmlns="http://www.w3.org/1999/xhtml"
   xmlns:th="http://www.thymeleaf.org"> <-- Deklarujemy przestrzeń nazw Thymeleaf
   <head>
      <title>Spittr</title>
      <link rel="stylesheet"
            type="text/css"
            th:href="@{/resources/style.css}"></link> <-- Łącze th:href do arkusza stylów
   </head>
   <body>
      <h1>Witamy w aplikacji Spittr</h1>
      <a th:href="@{/spittles}">Spittle</a> | <-- Łącza do stron th:href
```

```
      <a th:href="@{/spitter/register}">Zarejestruj</a>
   </body>
</html>
```

Szablon strony domowej jest względnie prosty i wykorzystuje jedynie atrybut `th:href`. Atrybut ten przypomina swój odpowiednik w HTML, atrybut `href`, i można go używać dokładnie w taki sam sposób. Wyjątkowość atrybutu `th:href` polega na tym, że może zawierać wyrażenia Thymeleaf umożliwiające dynamiczne wyliczanie wartości. Utworzona dynamiczna wartość wygenerowana zostanie w postaci standardowego atrybutu `href`. Podobnie działa wiele innych atrybutów dostępnych w przestrzeni nazw Thymeleaf — odzwierciedlają one standardowe atrybuty HTML, z którymi współdzielą nazwę i z których korzystają przy generowaniu wyliczanych wartości. W przedstawionym przykładzie wszystkie trzy użycia atrybutu `th:href` zawierają wyrażenie `@{}` służące do wyliczenia ścieżek URL świadomych kontekstu aplikacji (podobnie jak znacznik JSTL `<c:url>` lub znacznik Springa `<s:url>` na stronach JSP).

Chociaż plik *home.html* jest bardzo prostym przykładem szablonu Thymeleaf, widzimy wyraźnie, że jest to niemal czysty szablon HTML. Jedynym odstępstwem jest użycie atrybutu `th:href`. Poza tym jest to zwykły plik HTML.

Oznacza to, że szablony Thymeleaf, w przeciwieństwie do plików JSP, można w naturalny sposób edytować, a nawet generować bez użycia jakichkolwiek procesorów szablonów. Oczywiście Thymeleaf będzie niezbędny do przetworzenia szablonu i pełnego wygenerowania oczekiwanego wyniku. Ale nawet bez specjalnego przetwarzania możemy wczytać plik *home.html* w oknie przeglądarki, a to, co zobaczymy, będzie w dużym stopniu odpowiadało w pełni wygenerowanemu wynikowi. Jako przykład niech posłuży rysunek 6.6. Zobaczymy porównanie plików *home.jsp* (u góry) i *home.html* (na dole) po wyświetleniu obu plików w oknie przeglądarki.

Rysunek 6.6. Szablony Thymeleaf, w przeciwieństwie do plików JSP, są plikami HTML i możemy je generować oraz edytować jak zwykłe pliki HTML

Jak łatwo zauważyć, szablon JSP nie wygląda zbyt dobrze. Możemy wprawdzie zaobserwować znajome elementy, widoczna jest też jednak deklaracja biblioteki szablonów JSP. Widać także dziwny niedokończony kod JSP poprzedzający łącza, pozostałość po znaczniku `<s:url>` niewłaściwie zinterpretowanym przez przeglądarkę.

Szablon Thymeleaf generuje się w sposób niemal idealny. Jedynym odstępstwem są łącza. Przeglądarka nie traktuje atrybutu `th:href` jako atrybutu `href`, łącza zostały więc wygenerowane jako zwykły tekst. Poza tym niuansem szablon wygląda zgodnie z oczekiwaniami.

Prosty szablon, taki jak *home.html*, jest świetnym wprowadzeniem do systemu Thymeleaf. Siła JSP w aplikacjach Springa tkwiła jednak w wiązaniu formularzy. Czy rezygnując z szablonów JSP, musimy też porzucić ideę wiązania formularzy? Bez obaw. Thymeleaf ma coś do zaoferowania również w tym zakresie.

WIĄZANIE FORMULARZY W THYMELEAF

Wiązanie formularzy jest ważną funkcjonalnością w aplikacji Spring MVC. Umożliwia kontrolerom odebranie obiektów wspierających wypełnionych danymi przesłanymi z formularza, a przy wyświetlaniu formularza — jego wypełnienie wartościami zapisanymi w tych obiektach. Brak tej funkcjonalności skutkowałby potrzebą nazywania pól formularza w sposób odpowiadający nazwom właściwości obiektów wspierających. Również wyświetlanie formularza po nieudanej walidacji wymagałoby ręcznego ustawiania wartości pól formularza na podstawie danych zapisanych we właściwościach obiektu.

Mechanizm wiązania formularzy wykonuje tę pracę za nas. W celu przypomnienia sobie sposobu działania tego mechanizmu spójrzmy na pole *Imię* z pliku *registration.jsp*:

```
<sf:label path="firstName" cssErrorClass="error">
Imię </sf:label>:
<sf:input path="firstName" cssErrorClass="error" /><br/>
```

W tym miejscu znacznik `<sf:input>` z biblioteki znaczników wiązania formularzy Springa wywoływany jest dla wygenerowania znacznika HTML `<input>`, którego atrybut `value` przyjmie wartość właściwości `firstName` powiązanego obiektu. Wykorzystujemy też znacznik Springa `<sf:label>` oraz atrybut `cssErrorClass` do wygenerowania etykiety w kolorze czerwonym w sytuacji wystąpienia błędów walidacji.

W tej sekcji nie mówię jednak o JSP. Wprost przeciwnie — mówię o zastąpieniu JSP przez szablony Thymeleaf. Do wiązania nie wykorzystamy więc znaczników JSP, a możliwości dialektu Springa oferowane przez Thymeleaf.

Jako przykład rozważmy kawałek kodu szablonu Thymeleaf służącego do generowania pola Imię:

```
<label th:class="${#fields.hasErrors('firstName')}? 'error'">Imię</label>:
<input type="text"
       th:field="*{firstName}"
       th:class="${#fields.hasErrors('firstName')}? 'error'" /><br/>
```

W miejscu używanego w znacznikach JSP atrybutu `cssClassName` wykorzystujemy atrybut Thymeleaf `th:class` na standardowym znaczniku HTML. Atrybut `th:class` generuje atrybut `class` o wartości wyliczanej na podstawie podanego wyrażenia. W obu przypadkach użycia atrybutu `th:class` sprawdzamy bezpośrednio wystąpienie jakichś błędów walidacji dla pola `firstName`. Jeśli błędy wystąpią, atrybut `class` otrzymuje wartość `error`. W przypadku braku błędów dla tego pola atrybut `class` nie jest wcale generowany.

Znacznik `<input>` wykorzystuje atrybut `th:field` do wiązania pola z właściwością `firstName` obiektu wspierającego. Może to nie być do końca to, czego się spodziewasz.

W szablonach Thymeleaf niejednokrotnie spotykamy atrybuty, które odzwierciedlają standardowe atrybuty HTML. Mogłoby się więc wydawać, że do ustawienia wartości atrybutu `value` znacznika `<input>` służy atrybut `th:value`.

Ponieważ jednak wiążemy pole z właściwością `firstName` obiektu wspierającego, odwołanie do właściwości `firstName` realizujemy za pośrednictwem atrybutu `th:field`. Dzięki temu atrybut `value` przyjmuje wartość właściwości `firstName`, tę samą wartość przyjmuje również atrybut `name`.

Zobaczmy mechanizm wiązania danych Thymeleaf w akcji na przykładzie listingu 6.7, zawierającego kompletny szablon formularza rejestracji.

Listing 6.7. Strona rejestracji wykorzystująca Thymeleaf do wiązania formularza z obiektem wspierającym

```
<form method="POST" th:object="${spitter}">
  <div class="errors" th:if="${#fields.hasErrors('*')}"> <-- Błędy wyświetlania
    <ul>
        <li th:each="err : ${#fields.errors('*')}" th:text="${err}">
        Dane wejściowe są niepoprawne</li>
      </ul>
   </div>
 <label th:class="${#fields.hasErrors('firstName')}? 'error'"> <-- Imię
Imię
</label>:
<input type="text" th:field="*{firstName}"
➥th:class="${#fields.hasErrors('firstName')}? 'error'" /><br/>
 <label th:class="${#fields.hasErrors('lastName')}? 'error'">Nazwisko</label>: <-- Nazwisko
<input type="text" th:field="*{lastName}" th:class="${#fields.hasErrors('lastName')}?
'error'" /><br/>
<label th:class="${#fields.hasErrors('email')}? 'error'">Adres e-mail</label>:
                                                                    <-- Adres e-mail
<input type="text" th:field="*{email}" th:class="${#fields.hasErrors('email')}? 'error'"
/><br/>
<label th:class="${#fields.hasErrors('username')}? 'error'"> <-- Nazwa użytkownika
➥Nazwa użytkownika</label>:
<input type="text" th:field="*{username}" th:class="${#fields.hasErrors('username')}?
'error'" /><br/>
<label th:class="${#fields.hasErrors('password')}? 'error'">Hasło</label>: <-- Hasło
<input type="password" th:field="*{password}" th:class="${#fields.hasErrors('password')}?
'error'" /><br/>
<input type="submit" value="Zarejestruj" />
</form>
```

Na listingu 6.7 widać, że wszystkie pola formularza wykorzystują te same atrybuty Thymeleaf oraz wyrażenie `*{}` do wiązania z obiektem wspierającym. Jest to powtórzenie procedury, którą stosowaliśmy wcześniej dla pola *Imię*.

Z Thymeleaf skorzystaliśmy też u góry formularza do generowania wszystkich błędów. W elemencie `<div>` zastosowano atrybut `th:if` do sprawdzenia, czy wystąpiły jakieś błędy. W zależności od ich wystąpienia element zostanie lub nie zostanie wygenerowany.

W elemencie `<div>` znajduje się lista nieuporządkowana do wyświetlenia każdego błędu. Atrybut `th:each` znacznika `<li>` informuje Thymeleaf o tym, aby dla każdego błędu

wygenerowany został jeden element `<li>`, a w każdej iteracji do zmiennej `err` przypisany został aktualny błąd.

Znacznik `<li>` również posiada atrybut `th:text`. Dzięki temu atrybutowi Thymeleaf wie, że ma wyliczyć wyrażenie (w tym przypadku wartość zmiennej `err`) i wygenerować jego wartość jako ciało znacznika `<li>`. W rezultacie dla każdego błędu pojawi się jeden znacznik `<li>`, zawierający tekst błędu.

Możesz się zastanawiać, jaka jest różnica pomiędzy wyrażeniami otoczonymi symbolami `${}` a tymi otoczonymi symbolami `*{}`. Wyrażenia `${}` (jak `${spitter}`) są wyrażeniami zmiennych. Normalnie są to wyrażenia typu OGNL (*Object-Graph Navigation Language — http://commons.apache.org/proper/commons-ognl/*). W Springu są to jednak wyrażenia SpEL. Wynikiem wyrażenia `${spitter}` jest wartość klucza `spitter` właściwości modelu.

Wyrażenia `*{}` są wyrażeniami wyboru. Wyrażenia zmiennych wyliczane są w całym kontekście SpEL, natomiast wyrażenia wyboru są wyliczane na wybranym obiekcie. W formularzu obiekt wskazany jest za pomocą atrybutu `th:object` znacznika `<form>`: — w naszym przykładzie jest to pochodzący z modelu obiekt `Spitter`. Wyrażenie `*{firstName}` przyjmuje więc wartość właściwości `firstName` obiektu `Spitter`.

6.5. Podsumowanie

Działanie Spring MVC nie polega jedynie na przetwarzaniu żądań. Aby wyprodukowane w kontrolerze dane modelu były w ogóle widoczne, muszą zostać wygenerowane w widoku i wyświetlone na ekranie przeglądarki użytkownika. Spring jest w zakresie generowania widoków bardzo elastyczny i oferuje kilka opcji gotowych do użycia, wliczając w to tradycyjne strony JSP oraz popularny silnik układu stron Apache Tiles.

W tym rozdziale przyjrzeliśmy się pokrótce wszystkim widokom oferowanym przez Springa i sposobom ich generowania. Poznałeś też więcej szczegółów na temat pracy z szablonami JSP oraz Apache Tiles w aplikacji Spring MVC.

Dowiedziałeś się także, jak wykorzystać szablony Thymeleaf jako alternatywę dla JSP w warstwie widoku aplikacji Spring MVC. Thymeleaf jest ciekawą opcją, bo umożliwia tworzenie naturalnych szablonów, będących czystym kodem HTML, który można edytować i przeglądać jak każdy statyczny kod HTML. Równocześnie w trakcie działania aplikacji szablony te generują dane pochodzące z modelu. Co więcej, działanie szablonów Thymeleaf jest w dużej mierze oderwane od mechanizmu serwletów, dzięki czemu można z nich korzystać w miejscach, w których szablony JSP są niedostępne.

Po zdefiniowaniu warstwy widoku w aplikacji Spittr posiadamy małą, ale funkcjonalną aplikację Spring MVC. Możemy ją już uruchomić, ale wciąż pozostała nam praca nad takimi elementami aplikacji jak przechowywanie danych i kwestie bezpieczeństwa. Zajmiemy się nimi niebawem. Jednak już teraz aplikacja zaczyna nabierać właściwych kształtów.

Zanim zagłębimy się bardziej w rozwijanie naszej aplikacji, w następnym rozdziale będę kontynuował rozważania na temat Spring MVC, prezentując kilka użytecznych i zaawansowanych możliwości frameworka.

7 Zaawansowane możliwości Spring MVC

W tym rozdziale omówimy:

- Alternatywne opcje konfiguracji Spring MVC
- Wysyłanie plików na serwer
- Obsługę wyjątków w kontrolerach
- Pracę z atrybutami jednorazowymi

Ale zaczekaj! To nie wszystko!

Prawdopodobnie słyszałeś już wcześniej te słowa w reklamie telewizyjnej jakiegoś gadżetu. Kiedy pojawi się już całościowy opis produktu i jego możliwości, słyszymy: „Ale czekaj! To nie wszystko!", po czym następuje dalszy ciąg reklamy, mówiący o tym, jakie to jeszcze wspaniałe rzeczy reklamowany produkt potrafi robić.

Gdy poznajemy Spring MVC (a w istocie każdy element Springa), pod wieloma względami doznajemy takich właśnie uczuć. „To nie wszystko!". Kiedy już Ci się wydaje, że zgłębiłeś jego tajniki, dowiadujesz się, że oferuje on jeszcze więcej możliwości.

W rozdziale 5. poznałeś podstawy Spring MVC i nauczyłeś się obsługiwać różnego rodzaju żądania za pomocą tworzonych kontrolerów. Następnie w rozdziale 6. kontynuowaliśmy naukę, przygotowując widoki JSP oraz Thymeleaf, za pomocą których wyświetlaliśmy użytkownikowi dane pobrane z modelu. Może Ci się zdawać, że o Spring MVC wiesz już wszystko. Ale zaczekaj! To jednak nie wszystko!

W tym rozdziale pozostaniemy przy temacie Spring MVC. Omówię kilka funkcjonalności, które wykraczają poza podstawy przedstawione w rozdziałach 5. i 6. Zobaczysz,

jak tworzyć kontrolery przyjmujące przesyłane pliki, jak obsługiwać wyjątki wyrzucane w kontrolerach i jak przekazywać dane w modelu, aby ich nie utracić podczas przekierowania.

Na początek jednak chcę spełnić obietnicę złożoną w rozdziale 5. Zaprezentowałem wtedy pokrótce, jak wykorzystać do konfiguracji Spring MVC klasę `AbstractAnnotation➥ConfigDispatcherServletInitializer`, i obiecałem, że pokażę alternatywną opcję konfiguracji. Zanim więc spojrzymy na przesyłanie plików i obsługę wyjątków, poświęćmy chwilę na poznanie innych sposobów konfiguracji serwletu dyspozytora i serwletu nasłuchującego.

7.1. Alternatywna konfiguracja Spring MVC

W rozdziale 5. poznałeś podstawowy sposób konfiguracji Spring MVC poprzez rozszerzenie klasy `AbstractAnnotationConfigDispatcherServletInitializer`. Ta wygodna klasa bazowa zakłada, że chcesz wykorzystać podstawowe ustawienia serwletu dystrybutora `DispatcherServlet` oraz serwletu nasłuchującego `ContextLoaderListener`, a konfiguracja Springa zawarta jest w plikach Javy, a nie w plikach XML.

Ustawienia te mogą być odpowiednie dla wielu aplikacji Springa, ale w niektórych przypadkach konieczne jest wykonanie pewnych modyfikacji. Oprócz serwletu dyspozytora potrzebne mogą nam być serwlety i filtry. Możliwe też, że będziemy musieli zmodyfikować ustawienia samego serwletu dystrybutora lub, jeśli wdrażamy aplikację na kontener serwletów w wersji niższej niż 3.0, konieczne może być skonfigurowanie serwletu dystrybutora w tradycyjnym pliku *web.xml*.

Na szczęście jest kilka miejsc, w których Spring umożliwia nam modyfikację szerokiej gamy ustawień klasy `AbstractAnnotationConfigDispatcherServletInitializer`, gdy te modyfikacje są potrzebne. Przyjrzyjmy się teraz paru sposobom konfiguracji serwletu dystrybutora.

7.1.1. Dostosowujemy konfigurację serwletu dystrybutora

Kiedy patrzy się na klasę na listingu 7.1, nie jest to może od razu widoczne, ale klasa `AbstractAnnotationConfigDispatcherServletInitializer` oferuje więcej możliwości, niż się nam może początkowo wydawać. Trzy metody zdefiniowane przez nas w klasie `SpittrWebAppInitializer` były jedynymi metodami abstrakcyjnymi, które musieliśmy nadpisać. Jeśli chcemy zastosować dodatkowe zmiany konfiguracji, możemy też jednak nadpisać inne metody.

Jedną z tych metod jest `customizeRegistration()`. Kiedy `AbstractAnnotationConfig➥DispatcherServletInitializer` zarejestruje serwlet dystrybutora za pomocą kontenera serwletów, wywołuje metodę `customizeRegistration()` i przekazuje do niej instancję `ServletRegistration.Dynamic`, pozyskaną w wyniki rejestracji serwletu. Nadpisując implementację `customizeRegistration()`, możemy zastosować dodatkową konfigurację serwletu dystrybutora.

W sekcji 7.2 dowiesz się na przykład, jak obsłużyć żądania wieloczęściowe i przesyłanie plików w Spring MVC. Jeżeli chcemy skorzystać z możliwości specyfikacji Servlet 3.0 w zakresie konfiguracji żądań wieloczęściowych, musimy włączyć ich obsługę

przy rejestracji serwletu dystrybutora. W tym celu nadpisujemy metodę `customizeRegistration()`tak, aby ustawić konfigurację klasy `MutlipartConfigElement`:

```
@Override
protected void customizeRegistration(Dynamic registration) {
   registration.setMultipartConfig(
      new MultipartConfigElement("/tmp/spittr/uploads"));
}
```

Instancja `ServletRegistration.Dynamic` przekazana do metody `customizeRegistration()` umożliwia nam zrobienie kilku rzeczy, w tym ustawienie priorytetu ładowania przy starcie za pomocą metody `setLoadOnStartup()`, ustawienie parametru inicjalizacji za pomocą metody `setInitParameter()` oraz konfigurację obsługi żądań wieloczęściowych Servletu 3.0 za pomocą metody `setMultipartConfig()`. W poprzednim przykładzie ustawiliśmy przechowywanie przesłanych plików w żądaniach wieloczęściowych na */tmp/spittr/uploads*.

7.1.2. Dodajemy kolejne serwlety i filtry

Patrząc na definicję klasy `AbstractAnnotationConfigDispatcherServletInitializer`, widzimy, że tworzy ona serwlet dystrybutora oraz instancję serwletu nasłuchującego. Co jednak, jeśli chcemy zarejestrować dodatkowe serwlety, filtry lub listenery?

Jedną z ciekawszych opcji związanych z pracą z inicjalizatorami w klasach Javy jest to, że (w przeciwieństwie do pliku *web.xml*) możemy zdefiniować dowolną liczbę klas inicjalizatorów. Jeśli więc chcemy w kontenerze internetowym zarejestrować dodatkowy komponent, musimy tylko utworzyć nową klasę inicjalizatora. Najprostszym sposobem zdefiniowania tej klasy jest implementacja interfejsu Springa `WebApplicationInitializer`.

Na przykład listing 7.1 pokazuje sposób tworzenia implementacji interfejsu `Web`
`↪ApplicationInitializer` w celu zarejestrowania serwletu.

Listing 7.1. Implementujemy interfejs WebApplicationInitializer w celu zarejestrowania serwletu

```
package com.myapp.config;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration.Dynamic;
import org.springframework.web.WebApplicationInitializer;
import com.myapp.MyServlet;

public class MyServletInitializer implements WebApplicationInitializer {
   @Override
   public void onStartup(ServletContext servletContext)
         throws ServletException {
      Dynamic myServlet = <-- Rejestrujemy serwlet
         servletContext.addServlet("myServlet", MyServlet.class);
      myServlet.addMapping("/custom/**"); <-- Odwzorowujemy serwlet
   }
}
```

Listing 7.1 jest raczej prostym przykładem klasy inicjującej rejestrację serwletów. Rejestruje pojedynczy serwlet i odwzorowuje go na pojedynczą ścieżkę. Moglibyśmy

wykorzystać to podejście do ręcznej rejestracji serwletu dystrybutora. (Nie ma jednak takiej potrzeby, ponieważ klasa `AbstractAnnotationConfigDispatcherServletInitializer` dobrze sobie z tym radzi i nie wymaga pisania takiej ilości kodu).

Podobnie możemy rejestrować listenery i filtry, tworząc nową implementację interfejsu `WebApplicationInitializer`. Przykładowo listing 7.2 pokazuje sposób rejestracji filtra.

Listing 7.2. Interfejs WebApplicationInitializer może też służyć do rejestracji filtrów

```
@Override
public void onStartup(ServletContext servletContext)
      throws ServletException {
   javax.servlet.FilterRegistration.Dynamic filter =
      servletContext.addFilter("myFilter", MyFilter.class);
   filter.addMappingForUrlPatterns(null, false, "/custom/*");
}
```

Rejestrujemy filtr

Dodajemy odwzorowanie filtra

Interfejs `WebApplicationInitializer` jest przyjemną metodą rejestrowania serwletów, filtrów i listenerów w Javie przy wdrażaniu aplikacji na kontener zgodny ze specyfikacją Servlet 3.0. Jeśli jednak rejestrujemy filtr i chcemy go tylko odwzorować na serwlet dystrybutora, istnieje skrót w postaci klasy `AbstractAnnotationConfigDispatcher`
`↪ServletInitializer`.

Aby zarejestrować jeden lub więcej filtrów i odwzorować je na serwlet dystrybutora, musimy jedynie nadpisać metodę `getServletFilters()` klasy abstrakcyjnej `AbstractAnno`
`↪tationConfigDispatcherServletInitializer`. Przykładowo poniższa metoda `getServlet`
`↪Filters()` nadpisuje metodę klasy `AbstractAnnotationConfigDispatcherServletInitia`
`↪lizer` w celu rejestracji filtra:

```
@Override
protected Filter[] getServletFilters() {
   return new Filter[] { new MyFilter() };
}
```

Jak widzisz, metoda zwraca tablicę obiektów typu `javax.servlet.Filter`. W naszym przykładzie zwracamy tylko pojedynczy filtr, możemy jednak zwrócić dowolną ich liczbę. Nie musimy deklarować odwzorowania dla filtra — dowolny filtr zwrócony z metody `getServletFilters()` zostanie automatycznie odwzorowany na serwlet dystrybutora.

Przy wdrażaniu aplikacji na kontener zgodny ze specyfikacją Servlet 3.0 Spring oferuje kilka sposobów rejestracji serwletów (wliczając w to serwlet dystrybutora), filtry i listenery *bez* potrzeby tworzenia pliku *web.xml*. Nie musimy jednak korzystać z żadnej z tych metod, jeśli nie chcemy tego robić. Jeżeli nasza aplikacja nie jest wdrażana na kontener zgodny ze specyfikacją Servlet 3.0 (lub po prostu lubimy pracować z plikiem *web.xml*), nie ma żadnego powodu, abyśmy nie skorzystali ze starego sposobu konfiguracji Spring MVC z użyciem pliku *web.xml*. Zobaczmy, jak to zrobić.

7.1.3. Deklarujemy serwlet dystrybutora za pomocą pliku web.xml

W typowej aplikacji Spring MVC potrzebujemy serwletu dystrybutora `DispatcherServlet` i serwletu nasłuchującego `ContextLoaderListener`. Klasa `AbstractAnnotationConfigDis`
`↪patcherServletInitializer` zarejestruje za nas te serwlety, jeśli jednak zdecydujemy się na użycie pliku *web.xml*, to cała praca z tym związana spada na nas.

Listing 7.3 przedstawia podstawowy plik *web.xml* z typowymi ustawieniami serwletu dystrybutora i serwletu nasłuchującego.

Listing 7.3. Konfigurujemy Spring MVC za pomocą pliku web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" 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_2_5.xsd">
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
         /WEB-INF/spring/root-context.xml
      </param-value> <-- Ustawiamy lokalizację głównego kontekstu
   </context-param>
   <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener <-- Rejestrujemy serwlet nasłuchujący
      </listener-class>
   </listener>
   <servlet>
      <servlet-name>appServlet</servlet-name>
      <servlet-class>
         org.springframework.web.servlet.DispatcherServlet <-- Rejestrujemy serwlet dystrybutora
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>appServlet</servlet-name> <-- Odwzorowujemy serwlet dystrybutora na /
      <url-pattern>/</url-pattern>
   </servlet-mapping>
</web-app>
```

Jak już wspomniałem w rozdziale 5., zarówno serwlet nasłuchujący, jak i serwlet dystrybutora wczytują kontekst aplikacji Springa. Parametr kontekstu `contextConfig`
`↪Location` określa lokalizację pliku XML, zawierającego definicję głównego kontekstu aplikacji i wczytywanego przez serwlet nasłuchujący. Na listingu 7.3 główny kontekst wraz z definicjami komponentów wczytywany jest w pliku */WEB-INF/spring/root-context.xml*.

Serwlet dystrybutora wczytuje kontekst aplikacji wraz z komponentami zdefiniowanymi w pliku o nazwie bazującej na nazwie serwletu. W listingu 7.3 nazwą serwletu jest `appServlet`. W ten sposób serwlet dystrybutora wczytuje kontekst aplikacji z pliku XML */WEB-INF/appServlet-context.xml*.

Jeśli wolisz określić lokalizację pliku konfiguracji serwletu dystrybutora, możesz skonfigurować parametr inicjalizacji `contextConfigLocation` serwletu. Przykładowo poniższa konfiguracja powoduje, że serwlet dystrybutora wczytuje swoje komponenty z pliku */WEB-INF/spring/appServlet/servlet-context.xml*:

```
<servlet>
   <servlet-name>appServlet</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
         /WEB-INF/spring/appServlet/servlet-context.xml
      </param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
```

Oczywiście tak to wygląda, gdy chcemy, aby serwlet dystrybutora i serwlet nasłuchujący wczytywały swój kontekst aplikacji z pliku XML. W tej książce jednak preferujemy użycie konfiguracji Java. Chcemy więc skonfigurować Spring MVC do wczytywania konfiguracji z klas opatrzonych adnotacją `@Configuration`.

Żeby wykorzystać konfigurację opartą na klasach Javy w aplikacji Spring MVC, musimy powiedzieć serwletowi dystrybutora i serwletowi nasłuchującemu, by skorzystały z kontekstu `AnnotationConfigWebApplicationContext`, implementacji interfejsu `WebApplicationContext`, która wczytuje klasy konfiguracji Java zamiast plików XML. Możesz to osiągnąć poprzez ustawienie parametru kontekstu `contextClass` i inicjalizację parametru serwletu dystrybutora. Listing 7.4 pokazuje nowy plik *web.xml*, który przygotowuje Spring MVC do wykorzystania konfiguracji opartej na klasach Javy.

Listing 7.4. Ustawiamy plik web.xml do pracy z konfiguracją Java

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" 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_2_5.xsd">
   <context-param>
      <param-name>contextClass</param-name> <-- Korzystamy z konfiguracji Java
      <param-value> org.springframework.web.context.support.
            AnnotationConfigWebApplicationContext </param-value>
   </context-param>
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>com.habuma.spitter.config.RootConfig</param-value> <-- Określamy główną klasę konfiguracji
   </context-param>
   <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>
   <servlet>
```

```
      <servlet-name>appServlet</servlet-name>
      <servlet-class>
         org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
         <param-name>contextClass</param-name>
         <param-value>
            org.springframework.web.context.support.
                  AnnotationConfigWebApplicationContext
         </param-value>
      </init-param>
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>
            com.habuma.spitter.config.WebConfigConfig
         </param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>appServlet</servlet-name>
      <url-pattern>/</url-pattern>
   </servlet-mapping>
</web-app>
```

Korzystamy z konfiguracji Java

Określamy klasę konfiguracji serwletu dystrybutora

Teraz gdy poznałeś już kilka sposobów na konfigurację Spring MVC, zobacz, jak wykorzystać Spring MVC do przesyłania plików na serwer.

7.2. Przetwarzamy dane formularza wieloczęściowego

W wielu aplikacjach użytkownicy otrzymują możliwość przesłania danych na serwer. Na stronach takich jak Facebook lub Flickr użytkownicy często przesyłają swoje zdjęcia oraz filmy i dzielą się nimi ze swoją rodziną i znajomymi. Istnieje też kilka usług, które pozwalają użytkownikom na przesyłanie plików w celu „staromodnego" wydrukowania na papierze albo nadrukowania na koszulce czy kubku do kawy.

Aplikacja Spittr powinna udostępnić możliwość przesyłania plików w dwóch miejscach. Kiedy nowy użytkownik zarejestruje się w aplikacji, chcielibyśmy umożliwić mu dodanie zdjęcia do jego profilu. A kiedy użytkownik wyśle nową wiadomość `Spittle`, może chcieć dołączyć do niej jakieś zdjęcie.

Żądanie powstałe z wysłania standardowego formularza jest proste i przybiera postać par nazwa-wartość, rozdzielonych znakiem ampersand (&). Przykładowo gdy wysyłamy formularz rejestracji w aplikacji Spittr, żądanie może wyglądać następująco:

```
firstName=Charles&lastName=Xavier&email=profesorx%40xmen.org
➥&username=profesorx&password=wpuscmnie01
```

Ten schemat kodowania znaków jest prosty i w zupełności wystarczający do obsługi typowych tekstowych danych formularza. Nie jest jednak wystarczający do przesłania danych binarnych, takich jak obrazki. W odróżnieniu od zwykłych formularzy formularze wieloczęściowe rozbijane są na pojedyncze elementy, w których na każde pole przypada jedna część. Każda część może być innego rodzaju. Typowe pola formularza

zawierają w swoich częściach dane w postaci tekstowej. Kiedy jednak coś jest przesyłane na serwer, część może mieć postać binarną, tak jak na poniższym przykładzie treści żądania wieloczęściowego:

```
------WebKitFormBoundaryqgkaBn8IHJCuNmiW
Content-Disposition: form-data; name="firstName"
Charles
------WebKitFormBoundaryqgkaBn8IHJCuNmiW
Content-Disposition: form-data; name="lastName"
Xavier
------WebKitFormBoundaryqgkaBn8IHJCuNmiW Content-Disposition: form-data; name="email"
charles@xmen.com
------WebKitFormBoundaryqgkaBn8IHJCuNmiW Content-Disposition: form-data; name="username"
professorx
------WebKitFormBoundaryqgkaBn8IHJCuNmiW Content-Disposition: form-data; name="password"
letmein01
------WebKitFormBoundaryqgkaBn8IHJCuNmiW
Content-Disposition: form-data; name="profilePicture"; filename="me.jpg"
Content-Type: image/jpeg
[[ Tutaj wylądują dane obrazka ]]
------WebKitFormBoundaryqgkaBn8IHJCuNmiW--
```

W tym wieloczęściowym żądaniu część `profilePicture` różni się wyraźnie od pozostałych części. Zawiera między innymi swój własny nagłówek `Content-Type`, wskazujący, że zawarte w niej dane tworzą obraz JPEG. I chociaż może to nie być oczywiste, zawartość części `profilePicture` stanowią dane w postaci binarnej, a nie zwykły tekst.

Mimo że żądania wieloczęściowe wyglądają na skomplikowane, ich obsługa w kontrolerze Spring MVC jest prosta. Zanim utworzymy metody kontrolera do obsługi przesyłania plików, musimy skonfigurować rezolwer danych wieloczęściowych, aby serwlet dystrybutora wiedział, jak ma odczytywać żądania wieloczęściowe.

7.2.1. Konfigurujemy rezolwer danych wieloczęściowych

Serwlet dystrybutora nie implementuje żadnej logiki analizy danych w żądaniu wieloczęściowym. Zamiast tego oddelegowuje pracę do implementacji interfejsu `MultipartResolver`, żeby wydobyć zawartość tego żądania. Od wersji 3.1 Springa do wyboru otrzymujemy dwie wbudowane implementacje rezolwera `MultipartResolver`:

- `CommonsMultipartResolver` — wydobywa zawartość żądania z wykorzystaniem biblioteki Jakarta Commons FileUpload.
- `StandardServletMultipartResolver` — w żądaniach wieloczęściowych polega na obsłudze specyfikacji Servlet 3.0 (od wersji 3.1 Springa).

Mówiąc wprost, kiedy stoimy przed wyborem rezolwera, powinniśmy się skłonić ku użyciu tej drugiej opcji. `StandardServletMultipartResolver` wykorzystuje dostępne już w kontenerze serwletów mechanizmy i nie wymaga dołączania do projektu żadnych dodatkowych zależności. Z rezolwera `CommonsMultipartResolver` powinniśmy skorzystać wtedy, gdy wdrażamy aplikację na kontener serwletów w wersji wcześniejszej niż 3.0 lub nie korzystamy jeszcze ze Springa w wersji 3.1 bądź wyższej.

ANALIZUJEMY DANE Z ŻĄDAŃ WIELOCZĘŚCIOWYCH Z UŻYCIEM SERWLETÓW W WERSJI 3.0

Rezolwer `StandardServletMultipartResolver`, zgodny ze specyfikacją Servlet 3.0, nie posiada konstruktora przyjmującego argumenty ani nie pozwala na ustawienie żadnych właściwości. Dzięki temu jego deklaracja w postaci komponentu w pliku konfiguracyjnym Springa jest banalnie łatwa:

```
@Bean public MultipartResolver multipartResolver() throws IOException {
   return new StandardServletMultipartResolver();
}
```

Niewątpliwie wygląda to wyjątkowo prosto. Możemy się jednak zastanawiać, jak nałożyć ograniczenia na działanie tego rezolwera. Co zrobić, gdy zechcemy ograniczyć wielkość przesyłanego pliku? Albo jak określić lokalizację, w której następuje zapis plików tymczasowych w trakcie przesyłania danych na serwer? Obsługa jedynie kontrolera bezargumentowego i brak właściwości mogą nas skłaniać do myślenia, że możliwości rezolwera `StandardServletMultipartResolver` są bardzo ograniczone.

Jest wprost przeciwnie. Mamy możliwość konfiguracji ograniczeń, ale nie ustawiamy ich w konfiguracji rezolwera `StandardServletMultipartResolver`, tylko ustawiamy obsługę żądań wieloczęściowych w konfiguracji serwletu. Musimy określić przynajmniej lokalizację ścieżki dla plików tymczasowych do przechowywania danych przy przesyłaniu plików na serwer. Rezolwer `StandardServletMultipartResolver` nie zadziała, dopóki nie ustawimy tej minimalnej konfiguracji. Mówiąc konkretnie, musimy skonfigurować szczegóły obsługi żądań wieloczęściowych jako element konfiguracji serwletu dystrybutora w pliku *web.xml* lub klasie inicjalizacji serwletu.

Jeśli do konfiguracji serwletu dyspozytora użyjemy klasy inicjalizacji serwletu, która implementuje interfejs `WebApplicationInitializer`, to przy rejestracji serwletu możemy do ustawienia szczegółów obsługi żądań wieloczęściowych wykorzystać metodę `setMutlipartConfig()`, przekazując instancję `MultipartConfigElement`. Poniżej znajduje się minimalna konfiguracja potrzebna do obsługi żądań wieloczęściowych, która ustawia lokalizację ścieżki plików tymczasowych na */tmp/spittr/uploads*:

```
DispatcherServlet ds = new DispatcherServlet();
Dynamic registration = context.addServlet("appServlet", ds);
registration.addMapping("/");
registration.setMultipartConfig(
   new MultipartConfigElement("/tmp/spittr/uploads"));
```

Jeżeli jednak do konfiguracji serwletu dystrybutora wykorzystamy klasę inicjalizacji serwletu, która rozszerza klasę abstrakcyjną `AbstractAnnotationConfigDispatcherServlet ➥Initializer` bądź `AbstractDispatcherServletInitializer`, to nie tworzymy bezpośrednio instancji serwletu dystrybutora ani nie rejestrujemy go w kontekście serwletu. W rezultacie nie mamy wygodnego sposobu pracy z dynamiczną rejestracją serwletu z użyciem klasy `Dynamic`. Do konfiguracji szczegółów żądań wieloczęściowych możemy jednak nadpisać metodę `customizeRegistration()` (która otrzymuje obiekt `Dynamic` w postaci parametru):

```
@Override protected void customizeRegistration(Dynamic registration) {
   registration.setMultipartConfig(
      new MultipartConfigElement("/tmp/spittr/uploads"));
}
```

Korzystaliśmy do tej pory z pojedynczego argumentu konstruktora klasy `MultipartConfigElement`, który przyjmuje ścieżkę bezwzględną do katalogu w systemie plików, służącego do przechowywania tymczasowo przesyłanych plików. Ale istnieje jeszcze inny konstruktor, który pozwala na ustawienie ograniczeń na wielkość przesyłanych plików. Poza udostępnianiem ścieżki do plików tymczasowych konstruktor ten umożliwia ustawienie następujących opcji:

- Maksymalnego rozmiaru przesyłanych plików (w bajtach). Domyślnie nie są ustawione żadne ograniczenia.
- Maksymalnego rozmiaru całego żądania wieloczęściowego (w bajtach), niezależnie od tego, z ilu części się on składa i jak duża jest każda z tych części. Domyślnie nie są ustawione żadne ograniczenia.
- Maksymalnego rozmiaru pliku, który może zostać przesłany bez zapisu do tymczasowej lokalizacji (w bajtach). Domyślnym ustawieniem jest 0, co oznacza, że wszystkie przesyłane pliki są zapisywane na dysku.

Przypuśćmy, że chcemy ograniczyć wielkość przesyłanych plików do 2 MB, ograniczyć wielkość całego żądania do 4 MB i zapisywać wszystkie pliki na dysk. Poniższy kod wykorzystujący klasę `MultipartConfigElement` ustawia wszystkie nasze wymagania:

```
@Override
protected void customizeRegistration(Dynamic registration) {
   registration.setMultipartConfig(
      new MultipartConfigElement("/tmp/spittr/uploads",
         2097152, 4194304, 0));
}
```

Jeśli serwlet dystrybutora konfigurujemy w bardziej tradycyjny sposób w pliku *web.xml*, konfigurację możemy określić za pomocą elementu `<multipart-config>` wewnątrz elementu `<servlet>`, tak jak pokazano poniżej:

```
<servlet>
   <servlet-name>appServlet</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
   <multipart-config>
      <location>/tmp/spittr/uploads</location>
      <max-file-size>2097152</max-file-size>
      <max-request-size>4194304</max-request-size>
   </multipart-config>
</servlet>
```

Domyślne ustawienia elementu `<multipart-config>` są takie same jak w przypadku klasy `MutlipartConfigElement`. I tak jak w przypadku tej klasy, tu także musimy ustawić wartość elementu `<location>`.

KONFIGURUJEMY REZOLWER DANYCH WIELOCZĘŚCIOWYCH Z UŻYCIEM BIBLIOTEKI JAKARTA COMMONS FILEUPLOAD

Rezolwer `StandardServletMultipartResolver` jest z reguły najlepszym wyborem, jeżeli jednak aplikację wdrażamy na kontener serwletów w wersji wcześniejszej niż 3.0, potrzebna nam jest inna opcja. Możemy oczywiście stworzyć swoją własną implementację interfejsu `MutlipartResolver`, ale jeśli nie mamy jakichś specjalnych wymagań odnośnie do obsługi żądań wieloczęściowych, nie ma ku temu żadnych powodów. Spring dostarcza bowiem wbudowany we framework rezolwer `CommonsMultipartResolver`, alternatywną do `StandardServletMultipartResolver` implementację interfejsu `MutlipartResolver`.

Przedstawiony poniżej fragment kodu zawiera najprostszy sposób deklaracji rezolwera `CommonsMultipartResolver` w postaci komponentu springowego:

```
@Bean
public MultipartResolver multipartResolver() {
   return new CommonsMultipartResolver();
}
```

W przeciwieństwie do rezolwera `StandardSerlvetMultipartResolver`, w `CommonsMultipart`
`↪Resolver` nie ma potrzeby konfiguracji lokalizacji ścieżki plików tymczasowych. Domyślną lokalizacją jest w tym przypadku katalog tymczasowy kontenera serwletów. Mamy jednak możliwość wskazania innej lokalizacji za pośrednictwem właściwości `uploadTempDir`:

```
@Bean
public MultipartResolver multipartResolver() throws IOException {
   CommonsMultipartResolver multipartResolver =
      new CommonsMultipartResolver();
   multipartResolver.setUploadTempDir(
      new FileSystemResource("/tmp/spittr/uploads"));
   return multipartResolver;
}
```

W zasadzie w podobny sposób, bezpośrednio w konfiguracji Springa, możemy też określić inne ustawienia obsługi żądań wieloczęściowych. Przykładowo poniższa konfiguracja jest tak naprawdę odpowiednikiem utworzonej wcześniej konfiguracji rezolwera `StandardServletMultipartResolver` z użyciem elementu `MultipartConfigElement`:

```
@Bean
public MultipartResolver multipartResolver() throws IOException {
   CommonsMultipartResolver multipartResolver = new CommonsMultipartResolver();
   multipartResolver.setUploadTempDir(
      new FileSystemResource("/tmp/spittr/uploads"));
   multipartResolver.setMaxUploadSize(2097152);
   multipartResolver.setMaxInMemorySize(0);
   return multipartResolver;
}
```

Ustawiamy tutaj maksymalną wielkość pliku na 2 MB i limit przechowywania w pamięci na 0 bajtów. Te dwie właściwości odpowiadają drugiemu i czwartemu argumentowi konstruktora klasy `MultipartConfigElement`. Ustawienia te skutkują ustawieniem ograniczeń rozmiaru przesyłanych plików na 2 MB oraz tym, że wszystkie pliki zostaną

zapisane w pliku tymczasowym, niezależnie od ich wielkości. W przeciwieństwie do elementu `MutlipartConfigElement` tutaj nie mamy możliwości ograniczenia całkowitego rozmiaru żądania wieloczęściowego.

7.2.2. Obsługujemy żądania wieloczęściowe

Teraz gdy skonfigurowaliśmy już obsługę żądań wieloczęściowych w Springu (i być może w kontenerze serwletów), jesteśmy przygotowani na stworzenie metod kontrolera pozwalających przyjąć przesłane pliki. Najpopularniejszym sposobem jest dodanie adnotacji `@RequestPart` do parametru metody kontrolera.

Przypuśćmy, że chcemy udostępnić ludziom możliwość przesyłania obrazków w trakcie rejestracji użytkowników w aplikacji Spittr. Musimy zaktualizować formularz rejestracji, aby użytkownik mógł wybrać obrazek, który chce przesłać. Musimy również zmodyfikować metodę `processRegistration()` w kontrolerze `SpitterController`, żeby przyjęła przesłany plik. Poniższy fragment kodu z widoku Thymeleaf formularza rejestracji (*registrationForm.html*) wskazuje niezbędne zmiany w formularzu:

```
<form method="POST" th:object="${spitter}"
      enctype="multipart/form-data">
...
   <label>Profile Picture</label>:
   <input type="file"
          name="profilePicture"
          accept="image/jpeg,image/png,image/gif" /><br/>
...
</form>
```

Atrybut `enctype` znacznik `<form>` ma teraz wartość `multipart/form-data`. Jest to informacja dla przeglądarki, że formularz nie ma być przesłany w postaci danych formularza, a jako dane wieloczęściowe. Każde pole będzie posiadało swoją własną część w żądaniu wieloczęściowym.

Poza tym do wszystkich istniejących pól formularza rejestracji dodaliśmy nowe pole `<input>` typu `file`. Umożliwia to użytkownikowi wybór obrazka, który chce wysłać. Atrybut `accept` ustawiliśmy tak, aby ograniczyć akceptowane typy plików do obrazów w formatach JPEG, PNG i GIF. Zgodnie z atrybutem `name` dane obrazu przesłane zostaną w żądaniu wieloczęściowym, w części o nazwie `profilePicture`.

Teraz musimy już tylko zmodyfikować metodę `processRegistration()`, by zaakceptowała przesłany obraz. Jednym z rozwiązań jest dodanie parametru w postaci tablicy elementów typu `byte` i oznaczenie go adnotacją `@RequestPart`. Oto przykład:

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
      @RequestPart("profilePicture") byte[] profilePicture,
      @Valid Spitter spitter, Errors errors) {
   ...
}
```

Po zatwierdzeniu formularza atrybut `profilePicture` przyjmuje tablicę bajtów zawierającą dane z części żądania (wskazanej za pomocą atrybutu `@RequestPart`). Jeśli użytkownik wyśle formularz, nie wybierając pliku, tablica będzie pusta (ale nie przyjmie

wartości null). Po odebraniu danych pozostało nam tylko zlecić metodzie processRegi
➥stration() zapisanie pliku w jakimś miejscu.

Za chwilę pomyślimy, jak to zrobić. Najpierw jednak zastanówmy się, co wiemy o przesłanych danych obrazów lub, co chyba ważniejsze, czego o nich nie wiemy. Chociaż odebraliśmy dane obrazów w postaci tablicy bajtów i na tej podstawie jesteśmy w stanie wyliczyć rozmiar pliku obrazu, to poza tym nie wiemy w zasadzie nic więcej. Nie mamy pojęcia, jakiego typu jest ten plik ani nawet jaką nazwę miał pierwotnie. I do nas należy przekształcenie otrzymanej tablicy bajtów w plik, który możemy zapisać.

ODBIERAMY DANE W POSTACI MULTIPARTFILE

Praca z przesyłanymi plikami w postaci bajtów jest prosta, ale mocno ograniczona. Z tego powodu Spring udostępnia również interfejs MutlipartFile jako sposób na pozyskanie bardziej kompletnych informacji przy przetwarzaniu danych wieloczęściowych. Listing 7 5 pokazuje, jak wygląda ten interfejs.

Listing 7.5. Interfejs Springa MultipartFile umożliwia pracę z przesyłanymi plikami

```
package org.springframework.web.multipart;
import java.io.File; import java.io.IOException;
import java.io.InputStream;
public interface MultipartFile {
   String getName();
   String getOriginalFilename();
   String getContentType();
   boolean isEmpty();
   long getSize();
   byte[] getBytes() throws IOException;
   InputStream getInputStream() throws IOException;
   void transferTo(File dest) throws IOException;
}
```

Jak widzisz, interfejs MutlipartFile umożliwia pobranie bajtów danych przesłanego pliku. Oferuje też jednak znacznie więcej, włącznie z oryginalną nazwą pliku, jego rozmiarem oraz typem zawartości. Udostępnia również interfejs InputStream do strumieniowego odczytu danych pliku.

Co więcej, interfejs MultipartFile udostępnia wygodną metodę transferTo(), która pozwala na zapis przesłanego pliku w systemie plików. Przykładowo dodanie do metody processRegistration() następującej linii kodu pozwala zapisać przesłany plik w systemie plików:

```
profilePicture.transferTo(
   new File("/data/spittr/" + profilePicture.getOriginalFilename()));
```

Zapisanie pliku w lokalnym systemie plików w podany sposób jest proste, ale nakłada na nas odpowiedzialność za zarządzanie plikami. To my odpowiadamy za to, żeby na dysku było wystarczająco dużo miejsca. Odpowiadamy również za zapewnienie kopii bezpieczeństwa w przypadku awarii sprzętowej. I za synchronizację plików na różnych serwerach w klastrze.

ZAPISUJEMY PLIKI DO AMAZON S3

Inną opcją jest zrzucenie odpowiedzialności na kogoś innego. Za pomocą niewielkiej ilości kodu możemy zapisać obrazy w chmurze. Listing 7.6 pokazuje metodę `saveImage()`, którą możemy wywołać wewnątrz metody `processRegistration()`, aby zapisać przesłany plik w chmurze Amazon S3.

Listing 7.6. Zapisujemy dane typu MultipartFile w bazie Amazon S3

```
private void saveImage(MultipartFile image)
      throws ImageUploadException {
   try {
      AWSCredentials awsCredentials =
         new AWSCredentials(s3AccessKey, s2SecretKey);
      S3Service s3 = new RestS3Service(awsCredentials); <-- Przygotuj usługę S3

      S3Bucket bucket = s3.getBucket("spittrImages"); <-- Utwórz kubełek i obiekt S3
      S3Object imageObject = new S3Object(image.getOriginalFilename());

      imageObject.setDataInputStream( <-- Ustaw dane obrazu
            image.getInputStream());
      imageObject.setContentLength(image.getSize());
      imageObject.setContentType(image.getContentType());

      AccessControlList acl = new AccessControlList(); <-- Ustaw uprawnienia
      acl.setOwner(bucket.getOwner());
      acl.grantPermission(GroupGrantee.ALL_USERS,
            Permission.PERMISSION_READ);
      imageObject.setAcl(acl);

      s3.putObject(bucket, imageObject); <-- Zapisz obraz
   } catch (Exception e) {
      throw new ImageUploadException("Nie udało się zapisać obrazu ", e);
   }
}
```

Na początku metody `saveImage()` ustawiamy dane uwierzytelniające w AWS (*Amazon Web Service*). Potrzebne do tego będą klucz dostępu (*access key*) i tajny klucz dostępu (*secret access key*), które otrzymamy od Amazon przy rejestracji w usłudze S3. Zostaną one przekazane do kontrolera spittera poprzez wstrzyknięcie wartości.

Po ustawieniu danych uwierzytelniających `saveImage()` tworzy instancję dostarczanej przez JetS3t klasy `RestS3Service`, za pomocą której wykonywać będzie operacje na systemie plików S3. Następnie uzyskuje referencję do kubełka `spitterImages`, tworzy obiekt typu `S3Object` do przechowywania obrazu, po czym wypełnia go danymi obrazu.

Tuż przed wywołaniem metody `putObject()`, aby zapisać dane obrazu do S3, `save`
`↪Image()` ustawia uprawnienia dla obiektu `S3Object`, dzięki czemu użytkownik będzie go mógł zobaczyć. To istotne — bez tego obrazy nie byłyby widoczne dla użytkownika.

Tak jak w poprzedniej wersji `saveImage()`, w przypadku błędu zgłoszony zostanie wyjątek `ImageUploadException`.

ODBIERAMY WYSŁANY PLIK W POSTACI CZĘŚCI

Jeśli nasza aplikacja ma działać w kontenerze Servlet 3.0, otrzymujemy alternatywę dla interfejsu `MultipartFile`. Spring MVC jako parametr metody kontrolera akceptuje również obiekt typu `javax.servlet.http.Part`. Użycie interfejsu `Part` w miejscu `Multi➥partFile` sprawia, że sygnatura metody `processRegistration()` wygląda następująco:

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
      @RequestPart("profilePicture") Part profilePicture,
      @Valid Spitter spitter, Errors errors) {
   ...
}
```

Interfejs `Part` nie różni się zbytnio od interfejsu `MultipartFile`. Patrząc na listing 7.7, możemy zaobserwować, że interfejs `Part` posiada kilka metod odwzorowujących metody dostępne w interfejsie `MutlipartFile`.

Listing 7.7. Interfejs Part stanowi alternatywę dla interfejsu Springa MultipartFile

```
package javax.servlet.http;
import java.io.*;
import java.util.*;
public interface Part {
   public InputStream getInputStream() throws IOException;
   public String getContentType();
   public String getName();
   public String getSubmittedFileName();
   public long getSize();
   public void write(String fileName) throws IOException;
   public void delete() throws IOException;
   public String getHeader(String name);
   public Collection<String> getHeaders(String name);
   public Collection<String> getHeaderNames();
}
```

W wielu przypadkach nazwy metod interfejsu Part są takie same jak nazwy metod interfejsu `MultipartFile`. Kilka metod ma też nazwy podobne, choć jednak trochę inne. Przykładowo metoda `getSubmittedFileName()` odpowiada metodzie `getOriginalFilename()`, a metoda `write()` metodzie `transferTo()`. Zapis przesłanego pliku można więc zrealizować następująco:

```
profilePicture.write("/data/spittr/" +
      profilePicture.getOriginalFilename());
```

Warto zaznaczyć, że jeśli napiszemy metody kontrolera tak, aby przesłane pliki akceptowane były za pośrednictwem parametru typu `Part`, to nie musimy konfigurować komponentu `StandardServletMultipartResolver`. Tworzenie tego komponentu potrzebne jest jedynie w pracy z interfejsem `MultipartFile`.

7.3. Obsługujemy wyjątki

Do tego momentu zakładaliśmy, że w aplikacji Spittr wszystko będzie zawsze działać prawidłowo. Co się jednak stanie, jeżeli coś się nie uda? Co, jeśli przy obsłudze żądania wyrzucony zostanie wyjątek? Jaka odpowiedź zostanie wysłana do klienta?

Niezależnie od tego, czy wszystko się udaje, czy coś idzie nie tak, wynikiem zapytania serwletu jest zawsze odpowiedź serwletu. Jeżeli w trakcie działania aplikacji wystąpi jakiś wyjątek, rezultatem jest wciąż odpowiedź serwletu. Ale wyjątek musi jakoś zostać przetłumaczony na odpowiedź.

Spring udostępnia kilka użytecznych sposobów tłumaczenia wyjątków na odpowiedzi:

- Niektóre wyjątki Springa są automatycznie odwzorowywane na określone kody odpowiedzi HTTP.
- Wyjątek może zostać oznaczony adnotacją `@ResponseStatus` w celu odwzorowania na kod odpowiedzi HTTP.
- Metoda może zostać oznaczona adnotacją `@ExceptionHandler`, aby obsłużyć wskazany wyjątek.

Najprostszą metodą obsługi wyjątku jest jego odwzorowanie na kod odpowiedzi HTTP. Zobaczmy, jak to zrobić.

7.3.1. Mapujemy wyjątki na kody odpowiedzi HTTP

Standardowa dystrybucja Springa odwzorowuje automatycznie tuzin własnych wyjątków na odpowiednie kody odpowiedzi. Odwzorowania te przedstawia tabela 7.1.

Tabela 7.1. Niektóre wyjątki Springa odwzorowywane są na domyślne kody odpowiedzi HTTP

Wyjątek Springa	Kod odpowiedzi http
`BindException`	400 – Nieprawidłowe zapytanie
`ConversionNotSupportedException`	500 – Wewnętrzny błąd serwera
`HttpMediaTypeNotAcceptableException`	406 – Niedozwolone
`HttpMediaTypeNotSupportedException`	415 – Nieznany sposób żądania
`HttpMessageNotReadableException`	400 – Nieprawidłowe zapytanie
`HttpMessageNotWritableException`	500 – Wewnętrzny błąd serwera
`HttpRequestMethodNotSupportedException`	405 – Niedozwolona metoda
`MethodArgumentNotValidException`	400 – Nieprawidłowe zapytanie
`MissingServletRequestParameterException`	400 – Nieprawidłowe zapytanie
`MissingServletRequestPartException`	400 – Nieprawidłowe zapytanie
`NoSuchRequestHandlingMethodException`	404 – Nie znaleziono
`TypeMismatchException`	400 – Nieprawidłowe zapytanie

Wyjątki wymienione w tabeli 7.1 są z reguły wyrzucane przez samego Springa jako wynik nieprawidłowego działania w serwlecie dystrybutora lub podczas walidacji. Przykładowo jeśli serwlet dystrybutora nie może znaleźć metody kontrolera odpowiedzialnej za obsłużenie żądania, wyrzucany jest wyjątek `NoSuchRequestHandlingMethod` ➥`Exception`, co skutkuje odpowiedzią o kodzie HTTP 404 („Nie znaleziono”).

Chociaż te wbudowane mapowania są przydatne, nie pomagają w sytuacji, gdy wyrzucony zostanie jakiś wyjątek aplikacji. Na szczęście Spring udostępnia sposób odwzorowywania wyjątków na kody odpowiedzi HTTP za pośrednictwem adnotacji `@ResponseStatus`.

Aby zademonstrować działanie odwzorowywania wyjątków, posłużę się przykładem metody kontrolera `SpittleController`, której wynikiem mógłby być kod odpowiedzi HTTP 404 (ale nie jest):

```
@RequestMapping(value="/{spittleId}", method=RequestMethod.GET)
public String spittle(
      @PathVariable("spittleId") long spittleId, Model model) {
   Spittle spittle = spittleRepository.findOne(spittleId);
   if (spittle == null) {
      throw new SpittleNotFoundException();
   }
   model.addAttribute(spittle);
   return "spittle";
}
```

W powyższym przykładzie pobieramy z repozytorium `SpittleRepository` obiekt typu `Spittle` z użyciem jego identyfikatora. Jeśli metoda `findOne()` zwróci obiekt typu `Spittle`, zostanie on umieszczony w modelu, a zadanie wyświetlenia tego obiektu zostanie zlecone do widoku o nazwie `spittle`. Jeżeli jednak metoda `findOne()` zwróci wartość `null`, wyrzucony zostanie wyjątek `SpittleNotFoundException`. W tej chwili `SpittleNotFound`
`↪Exception` jest prostym niekontrolowanym wyjątkiem (typu runtime), który wygląda następująco:

```
package spittr.web;
public class SpittleNotFoundException extends RuntimeException {
}
```

Jeśli nastąpi wywołanie metody `spittle()` w celu obsługi żądania, a podany identyfikator okaże się pusty, wyjątek `SpittleNotFoundException` spowoduje (domyślnie) zwrócenie odpowiedzi o statusie HTTP 500 („Wewnętrzny błąd serwera"). W rzeczywistości każdy wyjątek, który nie został odwzorowany w inny sposób, skutkuje statusem odpowiedzi 500. Możemy to jednak zmienić, ustawiając odpowiednio odwzorowanie wyjątku `SpittleNotFoundException`.

Wyrzucenie wyjątku `SpittleNotFoundException` oznacza sytuację, w której nie znaleziono żądanego zasobu. Kod odpowiedzi 404 jest właśnie tym kodem, którego spodziewamy się w takiej sytuacji. Na listingu 7.8 użyjemy więc adnotacji `@ResponseStatus` do odwzorowania wyjątku `SpittleNotFoundException` na kod odpowiedzi 404.

Listing 7.8. Adnotacja @ResponseStatus odwzorowuje wyjątki na określone kody odpowiedzi

```
package spittr.web;
import org.springframework.http.HttpStatus;
Import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value=HttpStatus.NOT_FOUND, ← Odwzorowujemy wyjątek na kod odpowiedzi 404
      reason="Spittle nie został znaleziony ")
public class SpittleNotFoundException extends RuntimeException { }
```

Po wprowadzeniu adnotacji `@ResponseStatus` wyrzucenie wyjątku `SpittleNotFoundException` w metodzie kontrolera spowoduje zwrócenie odpowiedzi ze statusem 404, a jako ustawienie powodu takiej odpowiedzi na komunikat: „Spittle nie został znaleziony".

7.3.2. Tworzymy metody obsługi wyjątków

Odwzorowywanie wyjątków na kody odpowiedzi jest proste i w większości przypadków zupełnie wystarczające. Co jednak, jeśli chcemy zwrócić coś więcej niż tylko kod odpowiedzi reprezentujący błąd? Zamiast traktować wystąpienie wyjątku jako ogólny błąd, być może chcielibyśmy obsłużyć wyjątek tak samo, jak obsługujemy inne żądania.

Przypuśćmy, że w sytuacji, gdy użytkownik spróbuje utworzyć obiekt typu `Spittle` o identycznej treści jak napisana wcześniej, metoda repozytorium `SpittleRepository` wyrzuci wyjątek `DuplicateSpittleException`. Oznacza to, że metoda `saveSpittle()` kontrolera `SpittleController` musi obsłużyć ten wyjątek. Listing 7.9 pokazuje, że metoda `saveSpittle()` może bezpośrednio obsłużyć wyjątek `DuplicateSpittleException`.

Listing 7.9. Obsługujemy wyjątek bezpośrednio w metodzie obsługi żądania

```
@RequestMapping(method=RequestMethod.POST)
public String saveSpittle(SpittleForm form, Model model) {
   try {
      spittleRepository.save(
         new Spittle(null, form.getMessage(), new Date(),
            form.getLongitude(), form.getLatitude()));
      return "redirect:/spittles";
   } catch (DuplicateSpittleException e) {          <-- Łapiemy wyjątek
      return "error/duplicate";
   }
}
```

W kodzie przedstawionym na listingu 7.9 nie ma nic nadzwyczajnego. Jest to podstawowy przykład obsługi wyjątków w Javie. Nic więcej.

Metoda działa jak należy, ale jest trochę zbyt złożona. Istnieją dwie ścieżki wywołania metody, a każda z nich prowadzi do innego wyniku końcowego. Prościej by było, gdyby metoda `saveSpittle()` mogła się skoncentrować na pozytywnej ścieżce wywołania i pozostawić zadanie obsługi wyjątku innej metodzie.

Na początek wyodrębnijmy kod obsługi wyjątku poza metodę `saveSpittle()`:

```
@RequestMapping(method=RequestMethod.POST)
public String saveSpittle(SpittleForm form, Model model) {
   spittleRepository.save(
      new Spittle(null, form.getMessage(), new Date(),
         form.getLongitude(), form.getLatitude()));
   return "redirect:/spittles";
}
```

Jak widać, kod metody `saveSpittle()` jest teraz dużo prostszy. Ponieważ powstał z naciskiem na obsługę pozytywnego zapisu wiadomości `Spittle`, istnieje tylko jedna ścieżka wywołania i łatwo jest nią podążać (oraz ją testować).

Dodajmy teraz do kontrolera SpittleController nową metodę, która obsłuży przypadek, w którym wyrzucony zostanie wyjątek DuplicateSpittleException:

```
@ExceptionHandler(DuplicateSpittleException.class)
public String handleDuplicateSpittle() {
   return "error/duplicate";
}
```

Na metodę handleDuplicateSpittle() nałożona została adnotacja @ExceptionHandler, dzięki czemu Spring wie, że w przypadku wystąpienia wyjątku DuplicateSpittleException to właśnie ta metoda powinna zostać wywołana. Zwraca ona wartość typu String, określającą, podobnie jak w przypadku metod obsługi żądań, logiczną nazwę generowanego widoku, za pomocą którego poinformuje użytkownika, że próbował utworzyć zduplikowany wpis.

W metodach oznaczonych adnotacją @ExceptionHandler szczególnie interesujące jest to, że obsługują one wyjątki wyrzucone w *dowolnej* metodzie obsługi żądania zdefiniowanej w tym samym kontrolerze. Dlatego mimo że metodę handleDuplicateSpittle() utworzyliśmy na bazie kodu wyodrębnionego z metody saveSpittle(), obsłuży ona wszystkie przypadki wyrzucenia wyjątku DuplicateSpittleException w dowolnej metodzie kontrolera SpittleController. Nie musimy zatem powielać kodu obsługi wyjątku w każdej metodzie, która może go wyrzucić, bo utworzona przez nas metoda obsłuży wszystkie te metody.

Skoro metody oznaczone adnotacją @ExceptionHandler mogą obsługiwać wyjątki wyrzucane przez dowolne metody obsługi żądań zdefiniowane w ramach tego samego kontrolera, możesz się zastanawiać, czy istnieje sposób, żeby obsłużyć wyjątki wyrzucone w metodach zdefiniowanych w *dowolnym* innym kontrolerze. Od wersji 3.2 Springa jest to możliwe, ale tylko wtedy, gdy definicja metod obsługi wyjątków znajduje się w klasie porady.

Czym jest klasa porady? Cieszę się, że pytasz, bo jest to tematem następnej sekcji.

7.4. Doradzamy kontrolerom

Niektóre aspekty pracy kontrolerów byłyby bardziej poręczne, jeśli można by je było zastosować do wszystkich kontrolerów w danej aplikacji. Przykładowo metody opatrzone adnotacją @ExceptionHandler mogłyby być przydatne do obsługi wyjątków we wszystkich kontrolerach w aplikacji. Jeżeli określony wyjątek wyrzucany jest w kilku klasach kontrolera, mogłoby się okazać, że duplikujemy kod metody obsługi wyjątku we wszystkich tych klasach lub, aby uniknąć duplikacji kodu, utworzylibyśmy bazową klasę kontrolera, za pośrednictwem której wszystkie nasze kontrolery dziedziczyłyby wspólną metodę obsługi wyjątków.

Spring 3.2 udostępnia nam jeszcze inną opcję: porady kontrolerów. *Poradą kontrolera* nazywamy każdą klasę oznaczoną adnotacją @ControllerAdvice, posiadającą jedną bądź kilka metod:

- oznaczonych adnotacją @ExceptionHandler;
- oznaczonych adnotacją @InitBinder;
- oznaczonych adnotacją @ModelAttribute.

Powyższe metody klasy porad stosowane są do wszystkich metod oznaczonych adnotacją `@RequestMapping` we wszystkich kontrolerach w aplikacji.

Adnotacja `@ControllerAdvice` opatrzona jest adnotacją `@Component`. Dzięki temu oznaczona nią klasa wychwytywana jest przez mechanizm skanowania komponentów, tak samo jak klasa oznaczona adnotacją `@Controller`.

Jednym z najpraktyczniejszych zastosowań adnotacji `@ControllerAdvice` jest zgromadzenie wszystkich metod obsługi wyjątków w pojedynczej klasie, aby wyjątki występujące we wszystkich kontrolerach obsługiwane były w spójny sposób w jednym miejscu. Przypuśćmy, że chcemy wykorzystać metodę obsługi wyjątku `DuplicateSpittle` ➥`Exception` we wszystkich kontrolerach w aplikacji. Na listingu 7.10 przedstawiona została klasa `AppWideExceptionHandler` opatrzona adnotacją `@ControllerAdvice`, która wykonuje właśnie to zadanie.

Listing 7.10. Adnotację @ControllerAdvice wykorzystujemy do obsługi wyjątków we wszystkich kontrolerach

```
package spitter.web;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
@ControllerAdvice ←——— Deklarujemy poradę kontrolera
public class AppWideExceptionHandler {
   @ExceptionHandler(DuplicateSpittleException.class) ←— Definiujemy metodę obsługi wyjątku
   public String duplicateSpittleHandler() {
      return "error/duplicate";
   }
}
```

Teraz, jeśli wyjątek `DuplicateSpittleException` zostanie wyrzucony przez metodę *dowolnego* kontrolera, nastąpi wywołanie metody `duplicateSpittleHandler()` w celu obsługi tego wyjątku. Metoda oznaczona adnotacją `@ExceptionHandler` może być zapisana podobnie jak metoda oznaczona adnotacją `@RequestMapping`. Przykład umieszczony na listingu 7.10 zwraca logiczną nazwę widoku `error/duplicate`, by przedstawić użytkownikowi przyjazną stronę błędu.

7.5. Przenosimy dane między przekierowaniami

Jak wspominałem w sekcji „Tworzymy kontroler do obsługi formularza", po obsłużeniu żądania `POST` dobrą praktyką jest wykonanie przekierowania. Jedną z korzyści z tego rozwiązania jest ochrona użytkownika przed ponownym wysłaniem niebezpiecznego żądania `POST` w przypadku odświeżenia strony lub użycia przycisku *Wstecz* w oknie przeglądarki.

W rozdziale 5. skorzystaliśmy z możliwości prefiksu `redirect:` w nazwie widoku zwróconego z metod kontrolera. Gdy metoda kontrolera zwraca wartość typu `String`, którą rozpoczyna prefiks `redirect:`, wartość ta nie jest wykorzystywana do wyszukiwania widoku, ale służy jako ścieżka, na którą ma nastąpić przekierowanie w oknie przeglądarki. Jeśli wrócimy do listingu 5.19, zobaczymy, że ostatnia linia metody `process` ➥`Registration()` zwraca wartość typu `String` rozpoczynającą się prefiksem `redirect:`:

```
return "redirect:/spitter/" + spitter.getUsername();
```

Prefiks redirect: czyni pracę z przekierowaniami przejrzystą i prostą. Wydawałoby się, że Spring nie może już zrobić nic więcej, żeby jeszcze uprościć ten proces. Ale czekaj, to nie wszystko, co Spring ma do zaoferowania w tym zakresie.

W szczególności — jak metoda przekierowująca może przesłać dane do metody obsługującej przekierowanie? W praktyce, gdy metoda obsługująca żądanie się zakończy, wszystkie dane modelu w metodzie kopiowane są do żądania w postaci atrybutów, a żądanie przesyłane jest do widoku do wyświetlenia. Ponieważ zarówno metoda kontrolera, jak i widok obsługiwane są przez to samo żądanie, metody atrybutu nie znikają po przesłaniu.

Na rysunku 7.1 możemy jednak zaobserwować, że gdy metoda kontrolera skutkuje przekierowaniem, oryginalne żądanie się kończy i rozpoczyna się nowe żądanie HTTP `GET`. Wszystkie dane modelu przenoszone w oryginalnym żądaniu znikają wraz z końcem tego żądania. Nowe żądanie jest pozbawione wszystkich danych modelu w swoich atrybutach i musi samo je wypełnić.

Rysunek 7.1. Atrybuty modelu przenoszone są w żądaniu w postaci atrybutów, które nie przetrwają przekierowania

Z pewnością model nie pomoże nam przenosić danych pomiędzy przekierowaniami. Istnieje jednak kilka opcji przeniesienia danych z metody przekierowującej do metody obsługującej przekierowanie:

- przekazanie danych w postaci zmiennych ścieżki i (lub) parametrów zapytania za pomocą szablonów URL;
- wysłanie danych w atrybutach flash.

Na początek dowiesz się, jak Spring może Ci pomóc w wysłaniu danych w zmiennych ścieżki i (lub) parametrach zapytania.

7.5.1. Wykonujemy przekierowanie z użyciem szablonów URL

Przekazywanie danych w zmiennych ścieżki i parametrów zapytania wydaje się proste. Przykładowo na listingu 5.19 nazwa użytkownika nowo utworzonej wiadomości `Spitter` przekazywana jest w postaci zmiennej ścieżki. W obecnej postaci wartość `username` połączona jest ze ścieżką przekierowania. Ta metoda działa, ale nie jest „kuloodporna". Konkatenacja `Stringów` jest niebezpieczna przy tworzeniu adresów URL i zapytań SQL.

Zamiast w sposób ręczny łączyć ciągi w celu utworzenia adresu URL przekierowania, możemy skorzystać z oferowanej przez Springa możliwości użycia szablonów do ich zdefiniowania. Przykładowo ostatnia linia metody `processRegistration()` na listingu 5.19 mogłaby zostać zapisana tak:

```
return "redirect:/spitter/{username}";
```

Musimy jedynie ustawić wartość w modelu. W tym celu metoda `processRegistration()` musi zostać zapisana tak, aby przyjmowała jako parametr obiekt typu `Model` i wypełniała go nazwą użytkownika. Żeby tak się stało, możemy ustawić wartość nazwy użytkownika w modelu, by wypełniła symbol zastępczy w ścieżce przekierowania:

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(Spitter spitter, Model model) {
   spitterRepository.save(spitter);
   model.addAttribute("username", spitter.getUsername());
   return "redirect:/spitter/{username}";
}
```

Ponieważ parametr `username` nie jest sklejany z adresem przekierowania, ale wypełnia symbol zastępczy w szablonie URL, wszystkie niebezpieczne znaki są poddawane automatycznemu „eskejpowaniu". Jest to metoda bezpieczniejsza niż umożliwienie użytkownikowi wpisania dowolnego ciągu znaków i dopisywania go do ścieżki.

Co więcej, dowolne wartości typów prostych w modelu są również dodawane do adresu URL przekierowania w postaci parametrów zapytania. Na potrzeby przykładu przypuśćmy, że poza nazwą użytkownika model zawierał także właściwość `id` nowo utworzonego obiektu typu `Spitter`. Metodę `processRegistration()` moglibyśmy zapisać następująco:

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
      Spitter spitter, Model model) {
   spitterRepository.save(spitter);
   model.addAttribute("username", spitter.getUsername());
   model.addAttribute("spitterId", spitter.getId());
   return "redirect:/spitter/{username}";
}
```

Niewiele się zmieniło w zakresie zwracanego ciągu przekierowania. Ponieważ jednak atrybut modelu `spitterId` nie jest odwzorowany na żaden symbol zastępczy w przekierowaniu, zostanie dołączony do adresu URL automatycznie w postaci parametru zapytania.

Jeśli wartością atrybutu `username` jest `habuma`, a wartością atrybutu `spitterId` jest `42`, to ostateczna postać ścieżki przekierowania będzie taka: */spitter/habuma?spitterId=42*.

Przekazywanie danych pomiędzy przekierowaniami za pośrednictwem zmiennych ścieżki i parametrów zapytania jest łatwe, ale ma pewne ograniczenia. Nadaje się jedynie do przesyłania prostych wartości, takich jak `String` lub wartości numeryczne. Nie ma dobrego sposobu na przekazanie poprzez adres URL czegoś bardziej złożonego. W tym miejscu z pomocą przychodzą nam atrybuty jednorazowe.

7.5.2. Pracujemy z atrybutami jednorazowymi

Powiedzmy, że zamiast nazwy użytkownika lub identyfikatora chcielibyśmy przekazać w przekierowaniu cały obiekt typu `Spitter`. Jeśli prześlemy sam identyfikator, to metoda obsługująca przekierowanie będzie musiała wyszukać odpowiedni obiekt `Spitter` w bazie danych. Jednak przed wykonaniem przekierowania mamy już dostęp do tego obiektu. Dlaczego by go zatem nie przesłać do metody obsługującej przekierowanie?

Obiekt `Spitter` jest trochę bardziej złożony niż obiekt typu `String` czy `int`. Nie może więc być w prosty sposób wysłany w postaci zmiennej ścieżki albo parametru zapytania. Możemy go jednak ustawić w modelu w postaci atrybutu.

Jak już wiadomo, atrybuty modelu są kopiowane do żądania w postaci atrybutów żądania i znikają w momencie przekierowania. Musimy więc umieścić obiekt `Spitter` w takim miejscu, które przetrwa przekierowanie.

Jedną z opcji jest umieszczenie obiektu `Spitter` w sesji. Sesja jest długotrwała, a zapisane w niej dane są dostępne pomiędzy żądaniami. Możemy zatem umieścić obiekt `Spitter` w sesji przed wykonaniem przekierowania, a następnie pobrać z sesji po jego wykonaniu. Jesteśmy też oczywiście odpowiedzialni za wyczyszczenie danych z sesji po zakończeniu tej operacji.

Jak się okazuje, Spring zgadza się z tym, że umieszczanie danych w sesji jest doskonałym sposobem przekazywania informacji pomiędzy przekierowywaniami. Nie uważa jednak, że to Ty powinieneś odpowiadać za zarządzanie tymi danymi. Zamiast tego Spring oferuje możliwość wysyłania danych w postaci *atrybutów jednorazowych* (ang. *flash attributes*). Atrybuty jednorazowe, zgodnie z definicją, przenoszą dane do następnego żądania, po czym znikają.

Spring umożliwia ustawienie atrybutów jednorazowych za pośrednictwem dodanego w Springu 3.1 interfejsu `RedirectAttributes`, rozszerzającego interfejs `Model`. Dzięki temu oferuje wszystko to, co interfejs `Model`, a dodatkowo kilka metod służących do ustawiania atrybutów jednorazowych.

W szczególności interfejs `RedirectAttributes` dostarcza kilka metod `addFlashAttri`
`↪bute()`, które pozwalają na dodawanie atrybutów jednorazowych. Powróćmy ponownie do metody `processRegistration()` i wykorzystajmy metodę `addFlashAttribute()`, aby dodać do modelu obiekt `Spitter`:

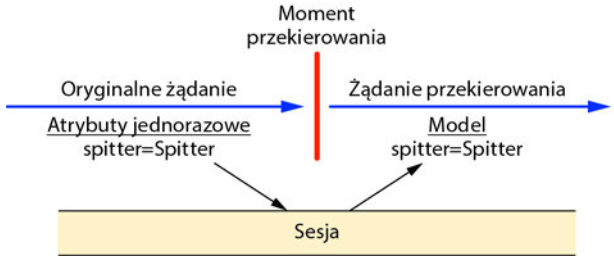
```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
      Spitter spitter, RedirectAttributes model) {
   spitterRepository.save(spitter);
   model.addAttribute("username", spitter.getUsername());
   model.addFlashAttribute("spitter", spitter);
   return "redirect:/spitter/{username}";
}
```

Wywołujemy tutaj metodę `addFlashAttribute()`, przekazując do niej pod kluczem `spitter` obiekt typu `Spitter`. Moglibyśmy również pominąć podanie klucza i pozwolić na ustawienie jego niejawnej domyślnej wartości na podstawie typu wartości:

```
model.addFlashAttribute(spitter);
```

Ponieważ do metody `addFlashAttribute()` przekazujemy obiekt typu `Spitter`, niejawnie wybranym kluczem jest `spitter`.

Zanim nastąpi przekierowanie, wszystkie atrybuty jednorazowe są kopiowane do sesji. Po przekierowaniu atrybuty jednorazowe zapisane w sesji są z niej wyciągane i przekazywane do modelu. Metoda obsługująca żądanie przekierowania może uzyskać dostęp do zapisanego w modelu obiektu `Spitter` dokładnie tak samo jak do każdego innego obiektu modelu. Rysunek 7.2 przedstawia działanie tego procesu.



Rysunek 7.2. Atrybuty jednorazowe są przechowywane w sesji, a następnie pobierane z modelu, co pozwala im przetrwać moment przekierowania

Aby zakończyć temat atrybutów jednorazowych, poniżej prezentuję odrobinę zaktualizowaną wersję metody `showSpitterProfile()`, która sprawdza obecność obiektu `Spitter` w modelu, zanim podejmie próbę jego pobrania z bazy danych:

```
@RequestMapping(value="/{username}", method=GET)
public String showSpitterProfile(
      @PathVariable String username, Model model) {
   if (!model.containsAttribute("spitter")) {
      model.addAttribute( spitterRepository.findByUsername(username));
   }
   return "profile";
}
```

Jak widać, pierwszą czynnością wykonaną przez metodę `showSpitterProfile()` jest sprawdzenie, czy istnieje atrybut modelu, którego kluczem jest `spitter`. Jeśli model zawiera ten atrybut, to nic już nie musimy robić. Obiekt `Spitter` zapisany w modelu zostanie przekazany do wyświetlenia w widoku. Jeżeli jednak model nie zawiera atrybutu `spitter`, metoda `showSpitterProfile()` pobierze obiekt w repozytorium i zapisze go w modelu.

7.6. Podsumowanie

Gdy zaczynamy temat Springa, to ciężko go skończyć, bo zawsze jest coś więcej: więcej funkcji, więcej wyborów i więcej sposobów na osiągnięcie naszych celów. Spring MVC ma wiele możliwości i wiele asów w rękawie.

Konfiguracja Spring MVC jest z pewnością jednym z obszarów, w którym jest mnóstwo opcji. W tym rozdziale rozpoczęliśmy od spojrzenia na różne sposoby konfiguracji serwletu dystrybutora i serwletu nasłuchującego Spring MVC. Dowiedziałeś się, jak dostosować rejestrację serwletu dystrybutora oraz zarejestrować dodatkowe serwlety i filtry. W przypadku wdrażania aplikacji na starsze serwery aplikacji spojrzeliśmy na sposób deklaracji serwletu dystrybutora oraz serwletu nasłuchującego w pliku *web.xml*.

Następnie sprawdziliśmy, jak obsłużyć wyjątki rzucane w kontrolerach Spring MVC. Chociaż metody oznaczone adnotacjami `@RequestMapping` są w stanie samodzielnie obsługiwać wyjątki, wyodrębnienie obsługi wyjątków do osobnej metody sprawia, że kod kontrolera staje się dużo bardziej przejrzysty.

Aby we wszystkich kontrolerach obsługiwać w sposób spójny takie zadania jak obsługa wyjątków, możemy skorzystać z adnotacji porad kontrolerów, wprowadzonych w Springu 3.2. Umożliwiają one tworzenie klas zbierających w jednym miejscu najczęstsze zachowania kontrolerów.

Na koniec przyjrzeliśmy się metodzie przenoszenia danych pomiędzy przekierowaniami w Springu, w tym również obsłudze atrybutów jednorazowych — atrybutów przypominających zwykły model, z wyjątkiem tego, że zapisane w nich dane nie znikają w wyniku przekierowania. Umożliwia to prawidłową odpowiedź na żądania `POST` z użyciem przekierowania przy zachowaniu danych pozyskanych w tym żądaniu i pozwoleniu na ich użycie oraz wyświetlenie po wykonaniu przekierowania.

W razie pytań — nie, to nie wszystko! Wciąż nie omówiłem wszystkich możliwości Spring MVC. Do tego tematu powrócimy w rozdziale 16., gdzie dowiesz się, jak skorzystać ze Spring MVC przy tworzeniu API REST-owego.

Jednak w tej chwili pozostawimy Spring MVC i przyjrzymy się bliżej Spring Web Flow, frameworkowi przepływów zbudowanemu na bazie Spring MVC. Framework ten umożliwia tworzenie aplikacji prowadzących użytkowników do celu poprzez zaplanowaną serię kroków.

8 Praca ze Spring Web Flow

W tym rozdziale omówimy:

- Tworzenie konwersacyjnych aplikacji sieciowych
- Definiowanie stanów przepływu i akcji
- Zabezpieczanie przepływów sieciowych

Jedną z wad (albo zalet) Internetu jest to, że tak łatwo można się w nim zgubić. Bogactwo zawartych w nim treści jest ogromne, ale prawdziwą potęgę daje mu hiperłącze. Z drugiej strony, wcale nie dziwi, że Internet jest nazywany *siecią*. Można go porównać do sieci utkanej przez pająka, z której, jeśli się już raz w niej znajdziesz, trudno się potem wyplątać.

Sam muszę się do czegoś przyznać: ta książka powstawała tak długo między innymi dlatego, że zagubiłem się raz w niekończącej się sekwencji odnośników Wikipedii.

Są sytuacje, w których aplikacja sieciowa musi przejąć stery i przeprowadzić użytkownika krok po kroku przez dany proces. Typowym przykładem takiej aplikacji jest finalizacja zamówienia w sklepie internetowym. Poczynając od koszyka, aplikacja przeprowadza Cię przez kolejne etapy: wprowadzenie danych adresowych, wprowadzenie danych związanych z płatnością i ostatecznie wyświetlenie potwierdzenia zamówienia.

Spring Web Flow jest frameworkiem sieciowym, umożliwiającym budowanie aplikacji z elementów, które składają się na sekwencję czynności zwaną inaczej przepływem sterowania (ang. *flow*). W tym rozdziale przyjrzymy się bliżej Spring Web Flow i jego działaniu na tle frameworka sieciowego Springa.

Aplikację z przepływami sterowania można zbudować za pomocą dowolnego frameworka sieciowego. Spotkałem się nawet z aplikacją Struts, która do pewnego stopnia

wykorzystywała przepływy. Ale bez możliwości oddzielenia przepływu od implementacji definicja przepływu będzie rozproszona po różnych jego elementach. Nie będzie jednego miejsca, które moglibyśmy odwiedzić, żeby w pełni zrozumieć przepływy.

Spring Web Flow jest rozszerzeniem Spring MVC, które umożliwia tworzenie aplikacji sieciowych opartych na przepływach. Dokonuje tego, oddzielając definicję przepływu aplikacji od klas i widoków, które implementują zachowanie przepływu.

Na czas poznawania Spring Web Flow zrobimy sobie przerwę od przykładowej aplikacji Spittr. W jej miejsce opracujemy nową aplikację sieciową, której zadaniem będzie przyjmowanie zamówień na pizzę. Do zdefiniowania całej procedury zamówienia użyjemy Spring Web Flow.

Pierwszym krokiem do pracy z frameworkiem Spring Web Flow będzie jego instalacja w ramach projektu. Zacznijmy więc od tego.

8.1. Konfiguracja Spring Web Flow

Spring Web Flow jest zbudowany na fundamencie Spring MVC. Oznacza to, że wszystkie żądania przepływu przechodzą najpierw przez serwlet dyspozytora Spring MVC. Aby obsługa żądania przepływu i kierowanie przepływem były możliwe, należy dokonać konfiguracji kilku specjalnych komponentów w kontekście aplikacji Springa.

W tej chwili nie ma możliwości konfiguracji Spring Web Flow za pomocą klas Javy, nie mamy więc innego wyjścia, jak skorzystać z konfiguracji zapisanej w formacie XML. Niektóre komponenty przepływów deklaruje się przy użyciu elementów konfiguracyjnej przestrzeni nazw XML Spring Web Flow. Musimy więc dodać deklarację przestrzeni nazw do pliku definicji kontekstu XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:flow="http://www.springframework.org/schema/webflow-config"
   xsi:schemaLocation=
   "http://www.springframework.org/schema/webflow-config
    http://www.springframework.org/schema/webflow-config/[CA] spring-webflow-config-2.3.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
```

Mając zadeklarowaną przestrzeń nazw, jesteśmy gotowi na dowiązanie komponentów przepływu, zaczynając od egzekutora przepływu.

8.1.1. Dowiązanie egzekutora przepływu

Jak sama nazwa wskazuje, **egzekutor przepływu** (ang. *flow executor*) kieruje wykonaniem przepływu. Kiedy użytkownik rozpoczyna przepływ, egzekutor przepływu tworzy i uruchamia instancję wykonania przepływu dla tego użytkownika. Jeśli przepływ zostanie wstrzymany (na przykład w momencie, gdy użytkownikowi wyświetlany jest widok), egzekutor przepływu wznawia przepływ (o ile użytkownik podejmie jakieś działanie).

Element `<flow:flow-executor>` tworzy egzekutor przepływu w Springu.

```
<flow:flow-executor id="flowExecutor" />
```

Egzekutor przepływu jest odpowiedzialny za utworzenie i wykonanie przepływu, ale ładowanie definicji przepływu nie należy do jego zadań. Odpowiada za nie rejestr przepływów, który utworzymy w następnej kolejności.

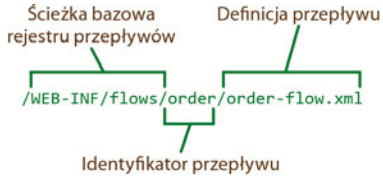
8.1.2. Konfiguracja rejestru przepływów

Zadaniem **rejestru przepływów** jest załadowanie definicji przepływu i udostępnienie ich egzekutorowi przepływu. Możemy umieścić rejestr przepływów w konfiguracji Springa za pomocą elementu `<flow:flow-registry>` w następujący sposób:

```
<flow:flow-registry id="flowRegistry" base-path="/WEB-INF/flows">
   <flow:flow-location-pattern value="*-flow.xml" />
</flow:flow-registry>
```

Zgodnie z powyższą deklaracją rejestr przepływów będzie szukać definicji w katalogu */WEB-INF/flows*, co określono w atrybucie `base-path`. Element `<flow:flow-location-pattern>` mówi nam natomiast, że za definicję przepływu uznawany będzie każdy plik XML, którego nazwa kończy się ciągiem *-flow.xml*.

Do wszystkich przepływów odnosimy się za pomocą identyfikatorów. Jeżeli użyjemy `<flow:flow-location-pattern>`, jak w przykładzie powyżej, identyfikatorem przepływu będzie nazwa podkatalogu ścieżki `base-path` lub jej części reprezentowanej przez dwa znaki gwiazdki, który zawiera definicję danego przepływu. Na rysunku 8.1 pokazano, jak prawidłowo odczytać identyfikator przepływu.



Rysunek 8.1. Jeżeli użyjemy do wyszukiwania przepływów flow-location-pattern, jako identyfikator przepływu zostanie użyta ścieżka do pliku definicji przepływu względem ścieżki bazowej base-path

Ewentualnie, można atrybut `base-path` pominąć i jawnie określić lokalizację pliku definicji przepływu:

```
<flow:flow-registry id="flowRegistry">
   <flow:flow-location path="/WEB-INF/flows/springpizza.xml" />
</flow:flow-registry>
```

Używamy tu elementu `<flow:flow-location>` zamiast `<flow:flow-location-pattern>`. Atrybut `path` wskazuje bezpośrednio na */WEB-INF/flows/springpizza.xml* jako plik definicji przepływu. Przy takiej konfiguracji identyfikator przepływu tworzony jest na podstawie nazwy pliku, w tym przypadku *springpizza*.

Jeśli chcesz określić identyfikator przepływu samodzielnie, możesz to zrobić przy pomocy atrybutu `id` elementu `<flow:flow-location>`. Przykładowo, aby nadać przepływowi identyfikator `pizza`, skonfiguruj `<flow:flow-location>` w następujący sposób:

```
<flow:flow-registry id="flowRegistry">
   <flow:flow-location id="pizza" path="/WEB-INF/flows/springpizza.xml" />
</flow:flow-registry>
```

8.1.3. Obsługa żądań przepływu

Jak dowiedzieliśmy się w poprzednim rozdziale, serwlet dyspozytora z reguły przesyła żądania do kontrolerów. Ale przy przepływach potrzebujemy odwzorowania obsługi `FlowHandlerMapping`, które poinformuje serwlet dyspozytora o konieczności przesyłania żądań do Spring Web Flow. `FlowHandlerMapping` konfiguruje się w kontekście aplikacji Springa w następujący sposób:

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
   <property name="flowRegistry" ref="flowRegistry" />
</bean>
```

Jak łatwo zauważyć, odwzorowanie `FlowHandlerMapping` jest powiązane z referencją do rejestru przepływów i dlatego wie, kiedy należy dokonać odwzorowania adresu URL żądania na przepływ. Jeśli, przykładowo, mamy przepływ o identyfikatorze `pizza`, `FlowHandlerMapping` dokona odwzorowania żądania na ten przepływ, gdy adres URL żądania (względem ścieżki kontekstu aplikacji) będzie mieć postać `/pizza`.

Podczas gdy `FlowHandlerMapping` zajmuje się kierowaniem żądań do Spring Web Flow, odpowiadanie na nie jest zadaniem adaptera obsługi `FlowHandlerAdapter`. `FlowHandler`
`↪Adapter` jest odpowiednikiem kontrolera Spring MVC, jako że obsługuje i przetwarza nadchodzące do przepływu żądania. `FlowHandlerAdapter` należy dowiązać jako komponent Springa następująco:

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
   <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

Adapter obsługi jest łącznikiem pomiędzy serwletem dyspozytora i Spring Web Flow. Obsługuje żądania przepływu i kieruje przepływem w zależności od tych żądań. Tutaj jest powiązany z referencją do egzekutora przepływu, aby kierować przepływem, dla którego żądania obsługuje.

Skonfigurowaliśmy już wszystkie komponenty potrzebne do pracy ze Spring Web Flow. Pozostała nam definicja samego przepływu. Dojdziemy do niej wkrótce. Najpierw jednak poznajmy elementy, które składają się na przepływ.

8.2. Składowe przepływu

Przepływ w Spring Web Flow to trzy podstawowe elementy: stany, przejścia i dane przepływu.

Stany (ang. *states*) to punkty przepływu, w których dzieje się coś interesującego. Jeżeli wyobrazisz sobie, że przepływ to wycieczka autokarowa, stany będą kolejnymi miastami i przystankami krajobrazowymi na trasie. Przystanek na trasie wycieczki może być traktowany jako czas na zakup paczki chipsów i gazowanego napoju, natomiast w czasie „postoju" przepływu wykonywane są pewne obliczenia, podejmowane są pewne decyzje lub wyświetlane są pewne strony.

O ile stany przepływu są jak punkty na mapie, w których można się zatrzymać, o tyle **przejścia** (ang. *transitions*) są niczym drogi łączące te punkty. W przepływie z jednego punktu do drugiego dostajemy się za pomocą przejścia.

Podróżując z miasta do miasta, gromadzisz pamiątki, wspomnienia i puste paczki po chipsach. Na tej samej zasadzie również w trakcie przepływu zbierane są pewne dane: o aktualnej kondycji przepływu. Chciałoby się tu użyć słowa *stan*, ale nie zapominajmy, że nadaliśmy mu już inne znaczenie w kontekście dyskusji o przepływach.

Zobaczmy, jak te trzy elementy są definiowane w Spring Web Flow.

8.2.1. Stany

Spring Web Flow definiuje pięć różnych rodzajów stanów, które wymieniono w tabeli 8.1.

Tabela 8.1. Stany w Spring Web Flow

Typ stanu	Zastosowanie
Akcja	W stanach akcji przetwarzane są dane, zgodnie z logiką przepływu.
Decyzja	Stany decyzyjne rozgałęziają przepływ w dwóch kierunkach, uzależniając dalszy ciąg przepływu od konkretnych wartości danych przepływu.
Koniec	Stan końcowy przepływu, jak sama nazwa wskazuje, jest ostatnim jego przystankiem. Po dojściu do stanu końcowego przepływ jest zamykany.
Podprzepływ	Stan podprzepływu rozpoczyna nowy przepływ w ramach przepływu, który już trwa.
Widok	Stan widoku wstrzymuje przepływ i zaprasza użytkownika do partycypacji w przepływie.

Oferowany wybór stanów pozwala nam na wbudowanie praktycznie dowolnych funkcji w naszą konwersacyjną aplikację sieciową. Chociaż nie wszystkie stany z opisanych w tabeli 8.1 będą potrzebne w każdym przepływie, prawdopodobnie prędzej czy później użyjesz większości z nich.

Za chwilę zobaczymy, jak połączyć różne rodzaje stanów w jeden kompletny przepływ. Najpierw jednak pokażemy definicję tych elementów przepływu w Spring Web Flow.

STANY WIDOKÓW

Stany widoków służą do wyświetlenia użytkownikowi informacji i dają mu szansę odegrania aktywnej roli w przepływie. Rzeczywistą implementacją może być dowolny widok obsługiwany przez Spring MVC, ale najczęściej jest on zaimplementowany w JSP.

W pliku XML definicji przepływu do zdefiniowania stanu widoku używamy elementu `<view-state>`:

```
<view-state id="welcome" />
```

W tym prostym przykładzie atrybut `id` spełnia dwie role. Po pierwsze, identyfikuje stan w przepływie. Po drugie, ponieważ nie określiliśmy w inny sposób widoku, definiuje `welcome` jako logiczną nazwę widoku, który zostanie wyświetlony, gdy przepływ dojdzie do tego stanu.

Jeżeli wolisz jawnie zidentyfikować inną nazwę widoku, możesz tego dokonać za pomocą atrybutu `view`:

```
<view-state id="welcome" view="greeting" />
```

Jeśli widok ma za zadanie wyświetlenie użytkownikowi formularza, możesz zechcieć określić obiekt, pod który formularz zostanie podpięty. Wykorzystasz do tego atrybut `model`:

```
<view-state id="takePayment" model="flowScope.paymentDetails"/>
```

Określamy tutaj, że formularz z widoku `takePayment` będzie podpięty pod obiekt `payment` ↪`Details` o zasięgu `flowScope` (więcej o zasięgu i danych przepływu powiemy sobie za chwilę).

STANY AKCJI

Podczas gdy stany widoków angażują w przepływ użytkownika aplikacji, stany akcji łączą się z pracą samej aplikacji. Najczęściej stany akcji wywołują metodę zarządzanego przez Springa komponentu, po czym dokonują przejścia do innego stanu, w zależności od wyniku jej wywołania.

W pliku XML definicji przepływu stany akcji wyrażane są przy pomocy atrybutu `<action-state>`. Oto przykład:

```
<action-state id="saveOrder">
   <evaluate expression="pizzaFlowActions.saveOrder(order)" />
   <transition to="thankYou" />
</action-state>
```

Choć nie jest to bezwzględny wymóg, elementy `<action-state>` z reguły zawierają element potomny `<evaluate>`. Jest on dla stanu akcji kluczowy. W atrybucie `expression` podane zostało wyrażenie, którego wartość jest szacowana w momencie wejścia w stan. W naszym przykładzie w atrybucie `expression` podaliśmy wyrażenie SpEL, które wskazuje, że powinna zostać wywołana metoda `saveOrder()` komponentu o identyfikatorze `pizzaFlowActions`.

> **Spring Web Flow i języki wyrażeń**
>
> Języki wyrażeń używane przez Spring Web Flow ulegały modyfikacjom na przestrzeni czasu. W wersji 1.0 Spring Web Flow korzystał z języka OGNL (Object-Graph Navigation Language). W wersji 2.0 język ten został zastąpiony przez język Unified EL (Unified Expression Language). Teraz, w wersji 2.1, Spring Web Flow wykorzystuje język SpEL.
>
> I choć możliwa jest konfiguracja Spring Web Flow z użyciem każdego z tych języków wyrażeń, to domyślną i zalecaną opcją jest język SpEL. Dlatego przy omawianiu definicji przepływów skoncentruję się na języku SpEL i pominę pozostałe opcje.

STANY DECYZYJNE

Teoretycznie przepływ może być całkowicie liniowy, dokonywać przejść od jednego stanu do drugiego bez alternatywnych ścieżek. Najczęściej jednak przepływ rozgałęzia się w pewnych punktach i dalsza jego część zależy od aktualnych okoliczności.

Stany decyzyjne umożliwiają dwukierunkowe rozgałęzienie w wykonaniu przepływu. Stan decyzyjny oblicza wartość wyrażenia logicznego i wybiera jedno z przejść w zależności od tego, czy obliczona wartość to `true` czy `false`. W pliku XML definicji przepływu stany decyzyjne definiujemy za pomocą elementu `<decision-state>`. Typowy przykład stanu decyzyjnego mógłby wyglądać następująco:

```
<decision-state id="checkDeliveryArea">
   <if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"
       then="addCustomer"
       else="deliveryWarning" />
</decision-state>
```

Jak widać, element <decision-state> współpracuje z innymi elementami. Najważniejszym elementem stanu decyzyjnego jest element <if>. To tu obliczana jest wartość wyrażenia. Jeżeli obliczona wartość wynosi true, przepływ przechodzi do stanu wskazanego w atrybucie then. Gdy zaś jest równa false, przepływ przechodzi do stanu wskazanego w atrybucie else.

STANY PODPRZEPŁYWÓW

Jako doświadczony programista, prawdopodobnie nie umieszczasz całej logiki swojej aplikacji w pojedynczej metodzie. Zamiast tego rozbijasz ją na wiele klas, metod i innych struktur.

Rozbicie przepływu na kilka odrębnych części jest równie dobrym pomysłem. Element <subflow-state> pozwala wywołać inny przepływ w ramach aktualnie wykonywanego przepływu. Jest to działanie analogiczne do wywoływania metody z poziomu innej metody.

Elementu <subflow-state> można użyć do deklaracji stanu podprzepływu w następujący sposób:

```
<subflow-state id="order" subflow="pizza/order">
   <input name="order" value="order"/>
   <transition on="orderCreated" to="payment" />
</subflow-state>
```

Używamy tu elementu <input> do przekazania obiektu zamówienia do podprzepływu. I jeżeli podprzepływ zakończy się stanem <end-state> o identyfikatorze orderCreated, przepływ dokona przejścia do stanu o identyfikatorze payment.

Ale wybiegamy trochę za daleko. Nie omówiliśmy jeszcze ani elementu <end-state>, ani przejść. Przejściami zajmiemy się w punkcie 8.2.2, stanami końcowymi natomiast już teraz.

STANY KOŃCOWE

Każdy przepływ musi się kiedyś zakończyć. Dzieje się to właśnie przy przejściu do stanu końcowego. Element <end-state> wyznacza koniec przepływu i może zostać zdefiniowany w następujący sposób:

```
<end-state id="customerReady" />
```

Po osiągnięciu stanu <end-state> przepływ kończy się. To, co zdarzy się dalej, zależy od kilku czynników:

- Jeżeli kończącym się przepływem jest podprzepływ, przepływ wywołujący zostanie wznowiony od stanu <subflow-state>. Identyfikator stanu <end-state> zostanie użyty jako zdarzenie wywołujące przejście ze stanu <subflow-state>.

- Jeżeli `<end-state>` posiada atrybut `view`, zostanie wyświetlony określony widok. Widok może być relatywną do przepływu ścieżką do szablonu, posiadać prefiks `externalRedirect:` celem przekierowania do strony zewnętrznej względem przepływu lub też posiadać prefiks `flowRedirect:`, pozwalający na przekierowanie do innego przepływu.
- Jeżeli kończący się przepływ nie jest podprzepływem i nie został określony widok (za pomocą atrybutu `view`), przepływ się kończy. Przeglądarka kieruje użytkownika do bazowego adresu URL przepływu i, z racji braku aktywnego przepływu, rozpoczyna się nowy.

Trzeba mieć świadomość, że przepływ może mieć więcej niż jeden stan końcowy. Skoro identyfikator stanu końcowego determinuje zdarzenie generowane z podprzepływu, możesz zechcieć zakończyć przepływ kilkoma stanami końcowymi, aby wywołać różne zdarzenia w przepływie nadrzędnym. Nawet przepływy niebędące podprzepływami mogą mieć kilka różnych stron docelowych wyświetlanych po zakończeniu przepływu, w zależności od jego przebiegu.

Teraz, kiedy znamy już poszczególne stany przepływu, powinniśmy poświęcić chwilę na zrozumienie, jak przepływ porusza się pomiędzy tymi stanami. Zobaczmy zatem, jak możemy utorować dla przepływu drogę.

8.2.2. Przejścia

Jak już powiedzieliśmy wcześniej, przejścia łączą stany przepływu. Każdy stan w przepływie, z wyjątkiem stanów końcowych, powinien mieć co najmniej jedno przejście, tak aby przepływ wiedział, co zrobić w momencie zakończenia tego stanu. Stan może posiadać wiele przejść, każde reprezentujące inną ścieżkę do obrania po zakończeniu stanu.

Przejście definiujemy za pomocą `<transition>` — elementu potomnego elementów poszczególnych stanów (`<action-state>`, `<view-state>` i `<subflow-state>`). W swojej najprostszej formie element `<transition>` identyfikuje następny stan w przepływie:

```
<transition to="customerReady" />
```

Atrybut `to` służy do określenia następnego stanu w przepływie. Jeżeli `<transition>` jest zadeklarowane tylko z atrybutem `to`, przejście jest przejściem domyślnym dla danego stanu i zostanie dokonane w razie braku innych przejść.

Częściej zdarza się, że przejścia są definiowane w sposób powodujący ich dokonanie w momencie wystąpienia konkretnego zdarzenia. W stanie widoku zdarzeniem jest najczęściej jakaś czynność użytkownika. W stanie akcji zdarzeniem jest wynik oszacowania wartości wyrażenia. W przypadku podprzepływu zdarzenie jest wyznaczane przez identyfikator stanu końcowego podprzepływu. Możesz również samodzielnie wyznaczyć zdarzenie, które powinno wywołać przejście w atrybucie `on`:

```
<transition on="phoneEntered" to="lookupCustomer"/>
```

W tym przykładzie przepływ dokona przejścia do stanu z identyfikatorem o wartości `lookupCustomer` po wystąpieniu zdarzenia `phoneEntered`.

Przepływ może też dokonać przejścia do innego stanu w odpowiedzi na zgłoszony wyjątek. Jeżeli na przykład dane klienta nie zostaną odnalezione, możesz zechcieć dokonać przejścia do stanu widoku, który wyświetli formularz rejestracyjny. Poniższy fragment kodu pokazuje ten rodzaj przejścia:

```
<transition
   on-exception="com.springinaction.pizza.service.CustomerNotFoundException"
   to="registrationForm" />
```

Atrybut `on-exception` różni się od atrybutu `on` tylko tym, że zamiast zdarzenia określa wyjątek, który powinien wywołać przejście. W powyższym przykładzie wyjątek `Custo` ➥`merNotFoundException` spowoduje przejście przepływu do stanu `registrationForm`.

PRZEJŚCIA GLOBALNE

Po utworzeniu przepływu może się okazać, że kilka stanów posiada pewne wspólne przejścia. Nie byłoby na przykład zaskoczeniem, gdyby okazało się, że poniższe przejście powtarza się kilka razy w przekroju całego przepływu:

```
<transition on="cancel" to="endState" />
```

Zamiast powtarzać przejścia w wielu stanach, możesz je zdefiniować jako przejścia globalne, umieszczając element `<transition>` w specjalnym elemencie `<global-transitions>`. Na przykład:

```
<global-transitions>
   <transition on="cancel" to="endState" />
</global-transitions>
```

Przy tak zdefiniowanym przejściu globalnym wszystkie stany przepływu będą posiadały przejście `cancel`.

Omówiliśmy do tej pory stany i przejścia. Zanim jednak zaczniemy tworzyć przepływy, przyjrzymy się jeszcze danym przepływu, ostatniemu z trzech elementów przepływu sieciowego.

8.2.3. Dane przepływu

Jeśli kiedykolwiek grałeś w starszą grę przygodową z komunikatami tekstowymi, wiesz, że wraz z przemieszczaniem się z miejsca na miejsce od czasu do czasu znajdujesz w nich przedmioty, które możesz zabrać ze sobą. Czasami możesz taki przedmiot wykorzystać od razu. Innym razem nosisz go przez całą grę, nie wiedząc, do czego służy, aż wreszcie rozgryziesz tę zagadkę.

Pod wieloma względami przepływy są jak te gry przygodowe. Przechodząc ze stanu do stanu, przepływ gromadzi dane. Czasami dane te potrzebne są tylko na chwilę (na przykład do wyświetlenia użytkownikowi strony). Kiedy indziej mogą być przekazywane dalej i użyte dopiero przy zakończeniu przepływu.

DEKLARACJA ZMIENNYCH

Dane przepływu są przechowywane w zmiennych, do których można się odnieść na każdym etapie przepływu. Mogą one być tworzone i gromadzone na szereg sposobów. Najprostszą metodą utworzenia zmiennej w przepływie jest użycie elementu `<var>`:

```
<var name="customer" class="com.springinaction.pizza.domain.Customer"/>
```

W tym przykładzie nowa instancja obiektu `Customer` jest tworzona, a następnie umieszczana w zmiennej o nazwie `customer`. Zmienna jest dostępna dla wszystkich stanów przepływu.

Jako część stanu akcji lub przy wejściu w stan widoku możesz również użyć elementu `<evaluate>` do utworzenia zmiennej. Na przykład:

```
<evaluate result="viewScope.toppingsList"
   expression="T(com.springinaction.pizza.domain.Topping).asList()" />
```

W tym przypadku element `<evaluate>` oblicza wartość wyrażenia SpEL, po czym umieszcza wynik w zmiennej `toppingsList` w zasięgu widoku (na temat zasięgu dowiemy się więcej już wkrótce).

Wartość zmiennej możemy również ustawić za pomocą elementu `<set>`:

```
<set name="flowScope.pizza"
   value="new com.springinaction.pizza.domain.Pizza()" />
```

Element `<set>` działa bardzo podobnie do elementu `<evaluate>`, podstawiając do zmiennej wynik obliczenia wartości wyrażenia. Tutaj podstawiamy nową instancję obiektu `Pizza` do zmiennej `pizza` o zasięgu przepływu.

Więcej szczegółów na temat tych elementów poznasz w podrozdziale 8.3, kiedy to rozpoczniemy budowę rzeczywistego przepływu sieciowego. Najpierw jednak wyjaśnijmy, co to znaczy, że zmienna ma zasięg przepływu, zasięg widoku lub inny rodzaj zasięgu.

ZASIĘG DANYCH PRZEPŁYWU

Dane przekazywane na poszczególnych etapach przepływu będą miały różne okresy życia i widoczności, w zależności od zasięgu, w którym przechowywana jest dana zmienna. Spring Web Flow definiuje pięć zasięgów; opisano je w tabeli 8.2.

Tabela 8.2. Zasięgi w Spring Web Flow

Zasięg	Okres życia i widoczność zmiennej
Konwersacji (ang. *conversation*)	Tworzona w momencie startu przepływu najwyższego poziomu i niszczona na jego końcu. Widoczna dla przepływu najwyższego poziomu i wszystkich jego podprzepływów.
Przepływu (ang. *flow*)	Tworzona w momencie startu przepływu i niszczona na jego końcu. Widoczna tylko dla przepływu, przez który została utworzona.
Żądania (ang. *request*)	Tworzona w momencie przekształcenia żądania w przepływ i niszczona po zakończeniu przepływu.
Chwilowy	Tworzona w momencie startu przepływu i niszczona na jego końcu. Czyszczona po wyświetleniu stanu widoku.
Widoku (ang. *view*)	Tworzona przy wejściu w stan widoku i niszczona przy wyjściu z tego stanu. Widoczna tylko w obrębie stanu widoku.

Zmienna zadeklarowana za pomocą elementu `<var>` ma zawsze zasięg na poziomie przepływu, w ramach przepływu, który ją definiuje. W przypadku elementów `<set>` i `<evaluate>` zasięg jest określany przez prefiks atrybutów `name` lub `result`. Poniżej pokazano, jak możemy przypisać wartość zmiennej `theAnswer` o zasięgu na poziomie przepływu:

```
<set name="flowScope.theAnswer" value="42"/>
```

Mamy już wszystkie potrzebne materiały, czas więc ich użyć i rozpocząć budowę kompletnego i w pełni funkcjonalnego przepływu. Zwróć uwagę na przykłady użycia zmiennych o różnych zasięgach w jej trakcie.

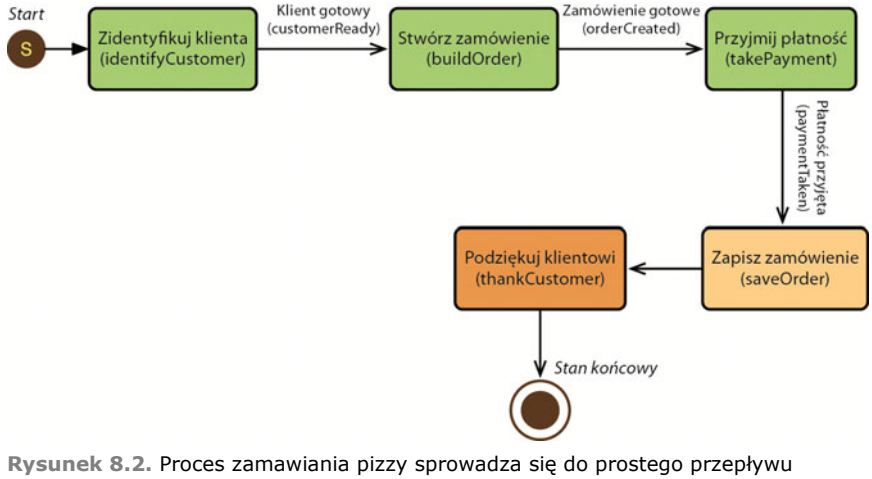
8.3. Łączymy wszystko w całość: zamówienie pizzy

Jak już wspomniałem wcześniej, w tym rozdziale robimy sobie przerwę od aplikacji Spittr. Zamiast tego zostaliśmy poproszeni o stworzenie aplikacji umożliwiającej zamawianie pizzy online, która pozwoli głodnym internautom zamówić tę włoską potrawę w swojej ulubionej wersji.

Okazuje się, że proces zamawiania pizzy nadaje się doskonale na przepływ. Zaczniemy od zbudowania nadrzędnego przepływu, który zdefiniuje ogólny proces zamawiania pizzy. Potem rozbijemy go na podprzepływy, definiujące szczegóły na niższym poziomie.

8.3.1. Definiowanie bazowego przepływu

Sieć pizzerii Spizza zdecydowała się odciążyć telefony w swoich lokalach, dając klientom możliwość składania zamówień online. Każdy klient odwiedzający stronę Spizza podaje dane identyfikacyjne, wybiera przynajmniej jedną pizzę, dostarcza informacji związanych z płatnością, a następnie wysyła zamówienie, po czym pozostaje mu już tylko czekać na dostawę świeżej i gorącej pizzy. Rysunek 8.2 jest ilustracją tego przepływu.



Rysunek 8.2. Proces zamawiania pizzy sprowadza się do prostego przepływu

Prostokąty na schemacie reprezentują stany, strzałki natomiast reprezentują przejścia. Jak łatwo zauważyć, ogólny przepływ związany z zamawianiem pizzy jest prosty i liniowy. Wyrażenie tego przepływu w Spring Web Flow nie powinno nastręczyć trudności. Warto jedynie zwrócić uwagę, że pierwsze trzy stany będą dużo bardziej zaangażowane w przepływ, niż wynika to z rysunku.

Listing 8.1 przedstawia nadrzędny przepływ zamówienia pizzy, zdefiniowany za pomocą definicji przepływów XML w Spring Web Flow.

Listing 8.1. Zamówienie pizzy jako przepływ Spring Web Flow

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/webflow
   http://www.springframework.org/schema/webflow/spring-webflow-2.3.xsd">

   <var name="order" class="com.springinaction.pizza.domain.Order"/>

   <subflow-state id="identifyCustomer" subflow="pizza/customer">
      <output name="customer" value="order.customer"/>
      <transition on="customerReady" to="buildOrder" />
   </subflow-state>

   <subflow-state id="buildOrder" subflow="pizza/order">
      <input name="order" value="order"/>
      <transition on="orderCreated" to="takePayment" />
   </subflow-state>

   <subflow-state id="takePayment" subflow="pizza/payment">
      <input name="order" value="order"/>
      <transition on="paymentTaken" to="saveOrder"/>
   </subflow-state>

   <action-state id="saveOrder">
      <evaluate expression="pizzaFlowActions.saveOrder(order)" />
      <transition to="thankCustomer" />
   </action-state>

   <view-state id="thankCustomer">
      <transition to="endState" />
   </view-state>

   <end-state id="endState" />

   <global-transitions>
      <transition on="cancel" to="endState" />
   </global-transitions>
</flow>
```

Wywołaj podprzepływ klienta (customer)

Wywołaj podprzepływ zamówienia (order)

Wywołaj podprzepływ płatności (payment)

Zapisz zamówienie

Podziękuj klientowi

Globalne przejście rezygnacji z zamówienia

Pierwszą rzeczą w powyższej definicji przepływu jest deklaracja zmiennej `order`. Za każdym razem, gdy rozpoczyna się nowy przepływ, tworzona jest instancja klasy `Order` (`listing 8.2`). Klasa ta posiada wszystkie właściwości potrzebne do przetwarzania zamówienia, włączając w to dane klienta, listę zamówionych pizz oraz szczegóły płatności.

Listing 8.2. Instancja klasy Order zawiera wszystkie dane odnośnie zamówienia

```
package com.springinaction.pizza.domain;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
```

```
public class Order implements Serializable {
   private static final long serialVersionUID = 1L;

   private Customer customer;
   private List<Pizza> pizzas;
   private Payment payment;

   public Order() {
      pizzas = new ArrayList<Pizza>();
      customer = new Customer();
   }

   public Customer getCustomer() {
      return customer;
   }

   public void setCustomer(Customer customer) {
      this.customer = customer;
   }

   public List<Pizza> getPizzas() {
      return pizzas;
   }

   public void setPizzas(List<Pizza> pizzas) {
      this.pizzas = pizzas;
   }

   public void addPizza(Pizza pizza) {
      pizzas.add(pizza);
   }

   public float getTotal() {
      return 0.0f;
   }

   public Payment getPayment() {
      return payment;
   }

   public void setPayment(Payment payment) {
      this.payment = payment;
   }
}
```

Zasadniczą część definicji przepływu tworzą jego stany. Domyślnie, pierwszy stan w definicji przepływu jest również pierwszym odwiedzanym stanem. W naszym przykładzie jest to stan `identifyCustomer` (stan podprzepływu). Ale jeśli chcesz, za pomocą atrybutu `start-state` elementu `<flow>` możesz samodzielnie określić, który stan ma być stanem początkowym.

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/webflow
```

```
    http://www.springframework.org/schema/webflow/spring-webflow-2.3.xsd"
    start-state="identifyCustomer">
...
</flow>
```

Identyfikacja klienta, budowa zamówienia i przyjmowanie płatności są zbyt skomplikowanymi czynnościami, aby umieszczać je wszystkie w jednym stanie. Dlatego zdefiniujemy je później jako samodzielne przepływy. Na potrzeby nadrzędnego przepływu `pizza` czynności te zostały ujęte jako elementy `<subflow-state>`.

Zmienna `order` będzie wypełniana wartościami podczas pierwszych trzech stanów, a zapisana w czwartym. Stan podprzepływu `identifyCustomer` używa elementu `<output>` do nadania wartości właściwości `customer` zmiennej `order`. Wartość tę uzyskuje, wywołując podprzepływ klienta. Stany `buildOrder` i `takePayment` stosują nieco inne podejście, używając elementu `<input>` do przekazania zmiennej `order` w sposób umożliwiający jej wewnętrzne wypełnienie wartościami przez podprzepływy.

Kiedy zamówienie dysponuje już informacjami o kliencie, liczbie i rodzajach zamawianych pizz oraz szczegółami płatności, trzeba je zapisać. Zadanie to wykonuje stan akcji `saveOrder`. Wykorzystuje element `<evaluate>` do wywołania metody `saveOrder()` komponentu o identyfikatorze `pizzaFlowActions`, przekazując zamówienie jako parametr. Po zapisaniu zamówienia przechodzi do stanu `thankCustomer`.

`thankCustomer` jest prostym stanem widoku, wspieranym przez plik JSP nazwany */WEB-INF/flows/pizza/thankCustomer.jsp*, który pokazano na listingu 8.3.

Listing 8.3. Widok JSP dziękujący klientom za zamówienie

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
   <jsp:output omit-xml-declaration="yes"/>
   <jsp:directive.page contentType="text/html;charset=UTF-8" />

   <head><title>Spizza</title></head>

   <body>
      <h2>Dziękujemy za zamówienie!</h2>

      <![CDATA[
      <a href='${flowExecutionUrl}&_eventId=finished'>Zakończ</a>
      ]]>
   </body>
</html>
```

Wygeneruj zdarzenie zakończenia przepływu

Strona dziękuje klientowi za zamówienie i dostarcza odnośnik, za pomocą którego może on zakończyć przepływ. Odnośnik ten jest najbardziej interesującym elementem tej strony, z racji tego, że pokazuje jeden ze sposobów interakcji użytkownika z przepływem.

Spring Web Flow posiada zmienną `flowExecutionUrl`, która zawiera adres URL przepływu, do używania w widoku. Odnośnik Zakończ dołącza do tego adresu parametr `_eventId`, aby wygenerować zdarzenie `finished` z powrotem do przepływu. Zdarzenie to kieruje przepływ do stanu końcowego.

Stan końcowy powoduje zakończenie przepływu. Ponieważ w momencie zakończenia przepływu kończą się wytyczne co do dalszych przejść, przepływ zaczyna się od nowa stanem `identifyCustomer` i jest gotowy przyjąć nowe zamówienie na pizzę.

Wyczerpaliśmy tym samym temat ogólnego przepływu zamówienia pizzy. Kod z listingu 8.1 to jednak nie wszystko, jeśli chodzi o przepływy w naszej aplikacji. Nadal musimy zdefiniować podprzepływy dla stanów `indentifyCustomer`, `buildOrder` i `takePayment`. Przejdźmy zatem do ich budowy, zaczynając od przepływu identyfikującego klienta.

8.3.2. Zbieranie informacji o kliencie

Jeśli zamawiałeś wcześniej pizzę, procedura jest Ci zapewne znana. Na początku pytany jesteś o swój numer telefonu. Oprócz tego, że ułatwi on pracę doręczycielowi, gdy będzie miał kłopoty ze znalezieniem adresu, numer telefonu służy też jako Twój identyfikator w pizzerii. Jeżeli jesteś powracającym klientem, pizzeria może użyć tego numeru telefonu do sprawdzenia Twojego adresu, dzięki czemu będzie wiadomo, gdzie dostarczyć zamówienie.

W przypadku nowego klienta wyszukiwanie na podstawie numeru telefonu nie zwróci żadnych wyników. Tak więc kolejną informacją, o którą zostaniesz poproszony, będzie adres zamieszkania. Mając te dane, pizzeria wie już, kim jesteś i gdzie ma dostarczyć pizzę. Zanim jednak zostaniesz poproszony o wybór pizzy, pizzeria musi się upewnić, że Twój adres mieści się w obsługiwanym przez nią rejonie. W przeciwnym razie pizzę będziesz musiał odebrać osobiście.

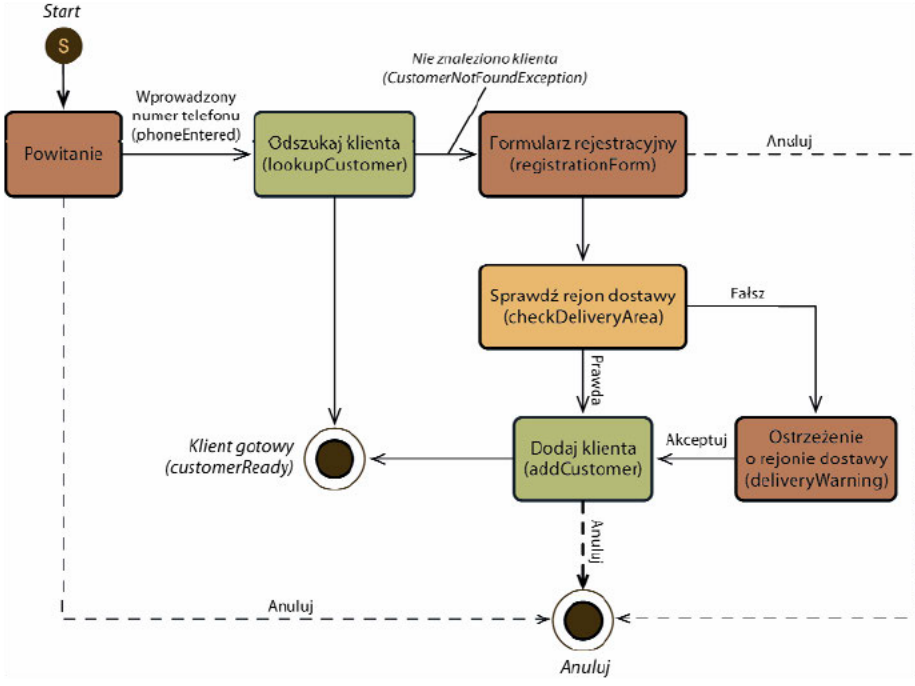
Etap zbierania informacji, który poprzedza każde zamówienie pizzy, został zilustrowany na rysunku 8.3.

Ten przepływ jest dużo ciekawszy od nadrzędnego przepływu `pizza`. Nie jest liniowy i rozgałęzia się w kilku miejscach, w zależności od warunków. Na przykład, po próbie odszukania klienta przepływ mógł albo się zakończyć (jeśli klient został odnaleziony), albo przejść do formularza rejestracyjnego (jeśli klient nie został znaleziony). Podobnie w stanie `checkDeliveryArea` klient mógł zostać ostrzeżony lub nie o tym, że jego adres leży poza rejonem dostawy.

Na listingu 8.4 pokazano definicję przepływu identyfikacji klienta.

Listing 8.4. Identyfikacja głodnego amatora pizzy przy pomocy przepływu sieciowego

```
<?xml version="1.0" encoding="UTF-8"?>
   <flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
      http://www.springframework.org/schema/webflow/spring-webflow-2.3.xsd">

      <var name="customer" class="com.springinaction.pizza.domain.Customer"/>

      <view-state id="welcome"> <-- Przywitaj klienta
         <transition on="phoneEntered" to="lookupCustomer"/>
      </view-state>

      <action-state id="lookupCustomer"> <-- Odszukaj klienta
```



Rysunek 8.3. Przepływ identyfikujący klienta jest nieco bardziej skomplikowany od przepływu pizza

```
    <evaluate result="customer" expression=
      "pizzaFlowActions.lookupCustomer(requestParameters.phoneNumber)" />
    <transition to="registrationForm" on-exception=
      "com.springinaction.pizza.service.CustomerNotFoundException" />
    <transition to="customerReady" />
</action-state>

<view-state id="registrationForm" model="customer">          <-- Zarejestruj nowego klienta
    <on-entry>
      <evaluate expression=
        "customer.phoneNumber = requestParameters.phoneNumber" />
    </on-entry>
    <transition on="submit" to="checkDeliveryArea" />
</view-state>

<decision-state id="checkDeliveryArea">                      <-- Sprawdź rejon dostawy
    <if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"
      then="addCustomer"
      else="deliveryWarning"/>
</decision-state>

<view-state id="deliveryWarning">                            <-- Pokaż ostrzeżenie o rejonie dostawy
    <transition on="accept" to="addCustomer" />
</view-state>

<action-state id="addCustomer">                              <-- Dodaj klienta
    <evaluate expression="pizzaFlowActions.addCustomer(customer)" />
    <transition to="customerReady" />
```

```
    </action-state>

    <end-state id="cancel" />
    <end-state id="customerReady">
       <output name="customer" />
    </end-state>

    <global-transitions>
       <transition on="cancel" to="cancel" />
    </global-transitions>
  </flow>
```

Powyższy przepływ wprowadza kilka nowych elementów; po raz pierwszy użyliśmy na przykład elementu `<decision-state>`. Ponadto, ponieważ mamy do czynienia z podprzepływem przepływu `pizza`, oczekuje on obiektu `Order` dostarczanego na wejściu.

Tak jak poprzednio, rozbijemy tę definicję przepływu na stany i przeanalizujemy je kolejno, zaczynając od stanu `welcome`.

PYTANIE O NUMER TELEFONU

Stan `welcome` jest w miarę prostym stanem widoku, który wita klienta na stronie Spizza i prosi go o wprowadzenie numeru telefonu. Stan sam w sobie nie jest szczególnie interesujący. Posiada on dwa przejścia: jedno, po wystąpieniu zdarzenia `phoneEntered` w widoku, kieruje przepływ do stanu `lookupCustomer`, drugie to globalne przejście rezygnacji z zamówienia, które reaguje na zdarzenie `cancel`.

Najciekawszą częścią stanu `welcome` jest sam widok. Jest on zdefiniowany w pliku */WEB-INF/flows/pizza/customer/welcome.jspx* w sposób pokazany na listingu 8.5.

Listing 8.5. Przywitanie klienta i prośba o podanie numeru telefonu

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:form="http://www.springframework.org/tags/form">
   <jsp:output omit-xml-declaration="yes"/>
   <jsp:directive.page contentType="text/html;charset=UTF-8" />

   <head><title>Spizza</title></head>

   <body>
      <h2>Witaj na stronie Spizza!!!</h2>

      <form:form>
         <input type="hidden" name="_flowExecutionKey"
         ↪value="${flowExecutionKey}"/>
         <input type="text" name="phoneNumber"/><br/>
         <input type="submit" name="_eventId_phoneEntered"
         ↪value="Odszukaj klienta" />
      </form:form>
   </body>
</html>
```

Klucz wykonania przepływu

Wygeneruj zdarzenie phoneEntered

To prosty formularz, proszący użytkownika o wprowadzenie numeru telefonu. Posiada jednak dwa interesujące elementy, które pomagają kierować przepływem.

Pierwszym z nich jest ukryte pole klucza wykonania przepływu `_flowExecutionKey`. Po wejściu w stan widoku przepływ zostaje wstrzymany i czeka na działanie użytkownika. Klucz wykonania przepływu dostarczany widokowi jest traktowany przez przepływ jako swego rodzaju „kwitek odbioru". Kiedy użytkownik zatwierdza wprowadzone dane, pole `_flowExecutionKey` zawierające klucz wykonania przepływu jest wysyłane razem z formularzem i przepływ kontynuuje swoje działanie od miejsca, w którym zostało przerwane.

Zwróć też szczególną uwagę na nazwę przycisku. Zawarty w niej ciąg *_eventId_* informuje Spring Web Flow, że dalsza jej część jest zdarzeniem, które powinno zostać wygenerowane. Kiedy użytkownik wyśle formularz, klikając ten przycisk, wygenerowane zostanie zdarzenie `phoneEntered`, co spowoduje przejście przepływu do stanu `lookupCustomer`.

ODSZUKANIE KLIENTA

Po wysłaniu formularza powitalnego numer telefonu klienta znajduje się wśród parametrów żądania i może zostać użyty do odszukania klienta. Dzieje się to w elemencie `<evaluate>` stanu `lookupCustomer`. Wydobywa on numer telefonu z parametrów żądania i przekazuje go do metody `lookupCustomer()` komponentu `pizzaFlowActions`.

Implementacja metody `lookupCustomer()` nie jest w tej chwili istotna. Wystarczy wiedzieć, że zwróci ona obiekt klienta `Customer` albo zgłosi wyjątek `CustomerNotFound`
`↪Exception`, jeżeli klient nie zostanie znaleziony.

W pierwszym przypadku obiekt `Customer` zostanie przypisany do zmiennej `customer` (poprzez atrybut `result`), a domyślne przejście skieruje przepływ do stanu `customer`
`↪Ready`. Jeśli natomiast dane klienta nie zostaną odnalezione, zgłoszony zostanie wyjątek `CustomerNotFoundException`, a przepływ przejdzie do stanu `registrationForm`.

REJESTRACJA NOWEGO KLIENTA

W stanie `registrationForm` klient proszony jest o podanie adresu dostawy. Podobnie jak wcześniejsze widoki, również ten generowany jest za pomocą JSP. Odpowiedni plik pokazano na listingu 8.6.

Listing 8.6. Rejestracja nowego klienta

```
<html xmlns:c="http://java.sun.com/jsp/jstl/core"
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:spring="http://www.springframework.org/tags"
  xmlns:form="http://www.springframework.org/tags/form">

   <jsp:output omit-xml-declaration="yes"/>
   <jsp:directive.page contentType="text/html;charset=UTF-8" />

   <head><title>Spizza</title></head>

   <body>
      <h2>Rejestracja klienta</h2>

      <form:form commandName="customer">
         <input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey}"/>
```

```
        <b>Numer telefonu: </b><form:input path="phoneNumber"/><br/>
        <b>Imię i nazwisko: </b><form:input path="name"/><br/>
        <b>Addres: </b><form:input path="address"/><br/>
        <b>Miasto: </b><form:input path="city"/><br/>
        <b>Województwo: </b><form:input path="state"/><br/>
        <b>Kod pocztowy: </b><form:input path="zipCode"/><br/>
        <input type="submit" name="_eventId_submit" value="Wyślij" />
        <input type="submit" name="_eventId_cancel" value="Anuluj" />
      </form:form>
   </body>
</html>
```

Nie jest to pierwszy formularz, z jakim spotkaliśmy się w naszym przepływie. Stan widoku `welcome` także wyświetla klientowi formularz. Jest on w miarę prosty, bo składa się z tylko jednego pola. Wydobycie wartości tego pola z parametrów żądania nie stanowi więc dużego problemu. Formularz rejestracyjny wymaga nieco więcej wysiłku.

Zamiast wydobywać wartości pojedynczych pól z parametrów żądania, lepiej podpiąć formularz pod obiekt `Customer` i pozwolić wykonać całą żmudną pracę frameworkowi.

SPRAWDZENIE REJONU DOSTAWY

Kiedy mamy już adres klienta, musimy się upewnić, że mieści się on w naszym rejonie dostawy. Jeżeli leży poza regionem, powinniśmy poinformować klienta o konieczności osobistego odbioru.

Do podjęcia powyższej decyzji użyjemy stanu decyzyjnego. Stan decyzyjny `checkDeliveryArea` zawiera element `<if>`, który przekazuje kod pocztowy klienta do metody `checkDeliveryArea()` komponentu `pizzaFlowActions`. Metoda ta zwraca wartość logiczną: `true`, jeżeli klient znajduje się w rejonie dostawy, i `false` w przeciwnym razie.

Jeżeli klient znajduje się w rejonie dostawy, przepływ przechodzi do stanu `addCustomer`. Jeżeli nie, klient przenoszony jest do stanu widoku `deliveryWarning`, wyświetlającego ostrzeżenie o rejonie dostawy. Widok kryjący się za `deliveryWarning` i zlokalizowany w */WEBINF/flows/pizza/customer/deliveryWarning.jspx* pokazano na listingu 8.7.

Listing 8.7. Ostrzeżenie klienta, że pizza nie może być dostarczona na jego adres

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
   <jsp:output omit-xml-declaration="yes"/>
   <jsp:directive.page contentType="text/html;charset=UTF-8" />

   <head><title>Spizza</title></head>

   <body>
      <h2>Dostawa niemożliwa</h2>

      <p>Adres znajduje się poza naszym rejonem dostawy. Nadal możesz złożyć zamówienie,
        ale do odbioru konieczna będzie osobista wizyta w naszym lokalu.</p>
      <![CDATA[
      <a href="${flowExecutionUrl}&_eventId=accept">Dalej, odbiorę osobiście</a> |
      <a href="${flowExecutionUrl}&_eventId=cancel">Nie, dziękuję</a>
      ]]>
   </body>
</html>
```

Kluczowymi z punktu widzenia przepływu elementami w pliku *deliveryWarning.jspx* są dwa odnośniki, dające klientowi możliwość kontynuacji zamówienia lub rezygnacji. Za pomocą tej samej zmiennej `flowExecutionUrl`, której używamy przy stanie `welcome`, odnośniki te wywołają zdarzenia `accept` lub `cancel` w przepływie. Jeżeli użytkownik zdecyduje się na kontynuację zamówienia i wysłane zostanie zdarzenie `accept`, przepływ przejdzie do stanu `addCustomer`. W innym przypadku nastąpi globalne przejście rezygnacji z zamówienia i podprzepływ przejdzie do stanu końcowego `cancel`.

O stanach końcowych powiemy więcej już wkrótce. Wcześniej spójrzmy przez chwilę na stan `addCustomer`.

ZAPISANIE DANYCH KLIENTA

Zanim przepływ dotarł do stanu `addCustomer`, klient wprowadził już swój adres. Aby można się było odnosić do niego w przyszłości, powinien zostać zapisany (najprawdopodobniej w bazie danych). Stan `addCustomer` posiada element `<evaluate>`, który wywołuje metodę `addCustomer()` komponentu `pizzaFlowActions`, przekazując jej zmienną przepływu `customer`.

Po zakończeniu tej metody przepływ dokona domyślnego przejścia do stanu końcowego o identyfikatorze `customerReady`.

KOŃCZENIE PRZEPŁYWU

Z reguły koniec przepływu nie jest specjalnie interesujący. W tym przepływie mamy jednak nie jeden, ale dwa stany końcowe. Podprzepływ generuje w momencie swojego zakończenia zdarzenie przepływu o nazwie identycznej z identyfikatorem stanu końcowego podprzepływu. Jeżeli przepływ ma tylko jeden stan końcowy, zawsze generowane jest to samo zdarzenie. Ale jeżeli stanów końcowych jest więcej, przepływ może nadać bieg przepływowi nadrzędnemu.

Jeżeli przepływ `customer` obierze jedną ze standardowych ścieżek, powinien zakończyć się stanem o identyfikatorze `customerReady`. Kiedy przepływ nadrzędny `pizza` zostaje wznowiony, otrzymuje zdarzenie `customerReady`, co powoduje przejście do stanu `buildOrder`.

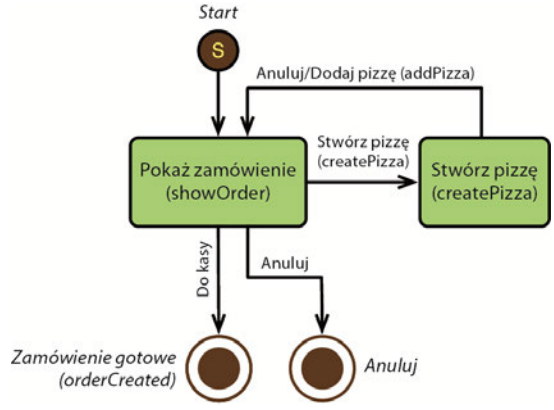
Zwróć uwagę, że stan końcowy `customerReady` zawiera element `<output>`. Element ten jest przepływowym ekwiwalentem wyrażenia `return` w Javie. Przekazuje dane z podprzepływu z powrotem do przepływu nadrzędnego. W tym przypadku `<output>` zwraca zmienną `customer`, aby stan `identifyCustomer` podprzepływu w przepływie `pizza` mógł przypisać ją do zamówienia.

Z drugiej strony, jeśli na jakimkolwiek etapie przepływu `customer` wygenerowane zostaje zdarzenie `cancel`, kończy ono ten przepływ stanem końcowym o identyfikatorze `cancel`. To z kolei wywołuje zdarzenie `cancel` w przepływie `pizza` i powoduje przejście (globalne) do stanu końcowego przepływu `pizza`.

8.3.3. Budowa zamówienia

Kolejnym — po identyfikacji klienta — krokiem głównego przepływu jest uzyskanie informacji na temat zamawianej pizzy. To w podprzepływie zamówienia (`order`), co pokazano na rysunku 8.4, klient proszony jest o wybór pizzy i dodanie wybranych pozycji do zamówienia.

Jak łatwo zauważyć, stan showOrder jest centralnym punktem podprzepływu order. Jest to pierwszy stan pojawiający się po rozpoczęciu przepływu oraz stan, do którego użytkownik jest kierowany po dodaniu nowej pizzy do zamówienia. Wyświetla on bieżący stan zamówienia i daje użytkownikowi możliwość dodania kolejnej pizzy.



Rysunek 8.4. Dodawanie pizzy przy użyciu podprzepływu order

Po decyzji użytkownika o dodaniu pizzy do zamówienia przepływ przechodzi do stanu createPizza. Jest to kolejny stan, który daje użytkownikowi możliwość wyboru dodatków do pizzy i jej rozmiaru. Z tego miejsca użytkownik może albo dodać pizzę, albo anulować. W obu przypadkach przepływ dokonuje przejścia z powrotem do stanu showOrder.

Z poziomu stanu showOrder użytkownik może albo wysłać zamówienie, albo je anulować. Cokolwiek wybierze, powoduje to zakończenie podprzepływu order i wyznacza dalszą ścieżkę przepływu głównego.

Listing 8.8 pokazuje próbę przetłumaczenia schematu na definicję Spring Web Flow.

Listing 8.8. Stany widoków podprzepływu order wyświetlające zamówienie i tworzące pizzę

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.3.xsd">

   <input name="order" required="true" /> <-- Przyjmujemy zamówienie jako dane wejściowe

   <view-state id="showOrder"> <-- Stan pokazywania zamówienia
      <transition on="createPizza" to="createPizza" />
      <transition on="checkout" to="orderCreated" />
      <transition on="cancel" to="cancel" />
   </view-state>
   <view-state id="createPizza" model="flowScope.pizza"> <-- Stan tworzenia pizzy
      <on-entry>
         <set name="flowScope.pizza"
           value="new com.springinaction.pizza.domain.Pizza()" />
         <evaluate result="viewScope.toppingsList" expression=
           "T(com.springinaction.pizza.domain.Topping).asList()" />
      </on-entry>
      <transition on="addPizza" to="showOrder">
         <evaluate expression="order.addPizza(flowScope.pizza)" />
      </transition>
      <transition on="cancel" to="showOrder" />
   </view-state>
```

```
    <end-state id="cancel" />
    <end-state id="orderCreated" />
</flow>
```

Stan końcowy anulowania zamówienia

Stan końcowy wysłania zamówienia

W zasadzie podprzepływ operuje na obiekcie `Order` utworzonym w przepływie głównym. Dlatego też potrzebujemy sposobu na przekazanie `Order` z przepływu głównego do podprzepływu. Przypomnij sobie listing 8.1. Użyliśmy tam elementu `<input>` w celu przekazania obiektu `Order` do przepływu. Tutaj używamy go do akceptacji tego obiektu. Jeżeli pomyślisz o przepływie jako o analogii metody w Javie, element `<input>` w tym użyciu definiuje sygnaturę podprzepływu. Ten przepływ wymaga pojedynczego parametru o nazwie `order`.

Dalej na listingu 8.1 widzimy definicję stanu `showOrder`, prosty stan widoku z trzema różnymi przejściami: tworzącym pizzę, wysyłającym formularz oraz rezygnującym z zamówienia.

Ciekawszym przypadkiem jest stan `createPizza`. Zawiera on widok formularza, który wysyła nowy, dodawany do zamówienia obiekt `Pizza`. Element `<on-entry>` dodaje nowy obiekt `Pizza` do zasięgu przepływu. Obiekt ten zostanie wypełniony wartościami po wysłaniu formularza. Zauważ, że atrybut `model` tego stanu widoku odnosi się do tego samego obiektu `Pizza` o zasięgu przepływu. Pod ten obiekt `Pizza` podpięty jest formularz pokazany na listingu 8.9:

Listing 8.9. Dodawanie pizzy za pomocą formularza HTML podpiętego pod obiekt o zasięgu przepływu

```
<div xmlns:form="http://www.springframework.org/tags/form"
  xmlns:jsp="http://java.sun.com/JSP/Page">

   <jsp:output omit-xml-declaration="yes"/>
   <jsp:directive.page contentType="text/html;charset=UTF-8" />

   <h2>Stwórz pizzę</h2>
   <form:form commandName="pizza">
      <input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey}"/>

      <b>Rozmiar: </b><br/>
      <form:radiobutton path="size" label="Mała (20 cm)" value="SMALL"/><br/>
      <form:radiobutton path="size" label="Średnia (30 cm)" value="MEDIUM"/><br/>
      <form:radiobutton path="size" label="Duża (40 cm)" value="LARGE"/><br/>
      <form:radiobutton path="size" label="Ogromna (50 cm)" value="GINORMOUS"/>
      <br/>
      <br/>
      <b>Dodatki: </b><br/>
      <form:checkboxes path="toppings" items="${toppingsList}"
        delimiter="&lt;br/&gt;"/><br/><br/>

      <input type="submit" class="button" name="_eventId_addPizza" value="Kontynuuj"/>
      <input type="submit" class="button" name="_eventId_cancel" value="Anuluj"/>
   </form:form>
</div>
```

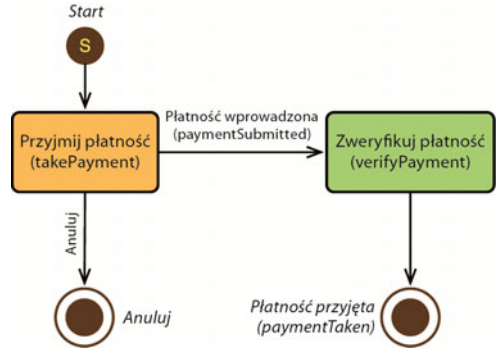
Przy wysyłaniu formularza za pomocą przycisku *Kontynuuj* rozmiar pizzy i wybór dodatków podpinane są pod obiekt `Pizza`, po czym następuje przejście `addPizza`. Skoja-

rzony z tym przejściem element `<evaluate>` wskazuje, że obiekt `Pizza` o zasięgu przepływu powinien zostać przekazany do metody `addPizza()` zamówienia przed przejściem do stanu `showOrder`.

Przepływ można zakończyć na dwa sposoby. Użytkownik może kliknąć przycisk *Anuluj* w widoku `showOrder` albo przycisk *Do kasy*. W obu przypadkach przepływ przechodzi do stanu `<end-state>`. Ale `id` wybranego stanu końcowego determinuje zdarzenie generowane przy zakończeniu przepływu i tym samym następny krok w przepływie głównym. Przepływ główny przechodzi albo do `cancel`, albo do `orderCreated`. W pierwszym przypadku zewnętrzny przepływ się kończy, w drugim przepływ przechodzi do podprzepływu `takePayment`, któremu to przyjrzymy się teraz bliżej.

8.3.4. Przyjmowanie płatności

Nikt nie dostanie pizzy za darmo, pizzeria Spizza szybko wypadłaby z rynku, pozwalając klientom składać zamówienia bez płatności w jakiejkolwiek formie. W końcowym stadium przepływu `pizza` ostatni podprzepływ prosi użytkownika o wprowadzenie szczegółów płatności. Na rysunku 8.5 zaprezentowano schemat tego prostego przepływu.



Rysunek 8.5. Ostatnim etapem procesu zamawiania pizzy jest przyjęcie płatności od klienta za pomocą podprzepływu płatności

Podobnie jak podprzepływ zamówienia, podprzepływ płatności również akceptuje obiekt `Order` przekazywany za pomocą elementu `<input>`.

Jak łatwo zauważyć, po wkroczeniu do przepływu płatności użytkownik kierowany jest do stanu `takePayment`. Jest to stan widoku, w którym użytkownik wskazuje, czy zapłaci kartą kredytową, czekiem czy gotówką. Po wysłaniu informacji o płatności użytkownik jest kierowany do stanu akcji `verifyPayment`, gdzie sprawdzana jest poprawność wprowadzonych danych.

Definicję podprzepływu płatności w XML pokazano na listingu 8.10.

Listing 8.10. Podprzepływ płatności z jednym stanem widoku i jednym stanem akcji

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.3.xsd">

   <input name="order" required="true"/>

   <view-state id="takePayment" model="flowScope.paymentDetails">
      <on-entry>
         <set name="flowScope.paymentDetails"
           value="new com.springinaction.pizza.domain.PaymentDetails()" />
         <evaluate result="viewScope.paymentTypeList" expression=
           "T(com.springinaction.pizza.domain.PaymentType).asList()" />
```

```
      </on-entry>
      <transition on="paymentSubmitted" to="verifyPayment" />
      <transition on="cancel" to="cancel" />
   </view-state>

   <action-state id="verifyPayment">
      <evaluate result="order.payment" expression=
        "pizzaFlowActions.verifyPayment(flowScope.paymentDetails)" />
      <transition to="paymentTaken" />
   </action-state>
   <end-state id="cancel" />
   <end-state id="paymentTaken" />
</flow>
```

Kiedy przepływ wchodzi w stan widoku `takePayment`, element `<on-entry>` przygotowuje płatność, używając najpierw wyrażenia SpEL do utworzenia nowej instancji `Payment ↪Details` w zasięgu przepływu. Będzie to obiekt wspomagający dla formularza. Ustawia też wartość zmiennej `paymentTypeList` w zasięgu widoku jako listę wartości wyliczenia `PaymentType` (pokazanego na listingu 8.11). Wykorzystywany jest tu operator `T()` języka SpEL do wywołania statycznej metody `toList()` klasy `PaymentType`.

Listing 8.11. Wyliczenie PaymentType definiuje opcje płatności klienta

```
package com.springinaction.pizza.domain;

import static org.apache.commons.lang.WordUtils.*;

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

public enum PaymentType {
   CASH, CHECK, CREDIT_CARD;

   public static List<PaymentType> asList() {
      PaymentType[] all = PaymentType.values();
      return Arrays.asList(all);
   }

   @Override
   public String toString() {
      return capitalizeFully(name().replace('_', ' '));
   }
}
```

Będąc na stronie formularza, klient może albo wysłać szczegóły płatności, albo anulować płatność. W zależności od dokonanego wyboru podprzepływ płatności kończy się stanem końcowym `paymentTaken` albo stanem końcowym `cancel`. Tak jak przy innych podprzepływach, każdy z tych dwóch stanów końcowych kończy podprzepływ i zwraca kontrolę do przepływu głównego. Atrybut `id` elementu `<end-state>` wyznacza z kolei następne przejście w przepływie głównym.

Prześledziliśmy już cały przepływ `pizza` oraz wszystkie jego podprzepływy. Poznaliśmy dużo możliwości Spring Web Flow. Zanim zakończymy ten temat, zatrzymajmy się jeszcze na chwilę przy sposobach zabezpieczenia dostępu do przepływu i jego stanów.

8.4. Zabezpieczanie przepływu

W następnym rozdziale dowiemy się, jak zabezpieczać aplikacje Springa za pomocą Spring Security. Skoro jesteśmy już jednak przy temacie przepływów sieciowych, spójrzmy na obsługę zabezpieczeń na poziomie przepływu, którą Spring Web Flow zapewnia w połączeniu ze Spring Security.

Stany, przejścia i całe przepływy mogą zostać zabezpieczone w Spring Web Flow za pomocą elementu `<secured>` zagnieżdżanego w tych elementach. Do zabezpieczenia dostępu do stanu widoku można użyć elementu `<secured>` na przykład w taki sposób:

```
<view-state id="restricted">
   <secured attributes="ROLE_ADMIN" match="all"/>
</view-state>
```

Zgodnie z powyższą konfiguracją dostęp do stanu widoku zostanie ograniczony do użytkowników z uprawnieniami `ROLE_ADMIN` (co określono w atrybucie `attributes`). Atrybut `attributes` przyjmuje jako wartość oddzieloną przecinkami listę uprawnień, które użytkownik musi posiadać, aby uzyskać dostęp do danego stanu, przejścia lub przepływu. Atrybut `match` może przyjmować wartość `any` lub `all`. Jeżeli ma wartość `any`, użytkownik powinien mieć przynajmniej jedno upoważnienie z listy uprawnień zawartej w atrybucie `attributes`. Jeśli natomiast ma wartość `all`, wymagane są wszystkie uprawnienia na liście.

Być może zastanawiasz się, w jaki sposób użytkownikowi przyznawane są uprawnienia, które sprawdza element `<secured>`, i w jaki sposób użytkownik się w ogóle loguje do aplikacji. Odpowiedzi na te pytania udzielimy w następnym rozdziale.

8.5. Podsumowanie

Nie we wszystkich aplikacjach sieciowych nawigacja zależy w całości od użytkownika. Czasami użytkownik musi zostać poprowadzony, muszą mu zostać zadane pytania i musi zostać pokierowany na konkretne strony w zależności od odpowiedzi. W takich sytuacjach nawigacja bardziej przypomina konwersację między aplikacją a użytkownikiem niż tradycyjne menu dostępnych opcji.

W tym rozdziale poznaliśmy możliwości Spring Web Flow, frameworka umożliwiającego tworzenie aplikacji konwersacyjnych. W jego trakcie zbudowaliśmy opartą na przepływach sterowania aplikację obsługi klienta przez pizzerię. Najpierw zdefiniowaliśmy ogólną ścieżkę zamówienia, zaczynając od zbierania informacji o kliencie, a na zapisie zamówienia w systemie kończąc.

Przepływ składa się z szeregu stanów i przejść, które określają, w jaki sposób ma się odbywać konwersacja pomiędzy stanami. Same stany natomiast różnią się między sobą. Stany akcji przetwarzają dane i wykonują obliczenia, stany widoków angażują użytkownika w przepływ, stany decyzyjne dynamicznie kierują przepływem, a stany końcowe powodują jego zakończenie. Istnieją też stany podprzepływów, także definiowane przez przepływ.

Na końcu dowiedzieliśmy się, jak dostęp do przepływu, stanu czy przejścia może zostać ograniczony do użytkowników posiadających określone uprawnienia. Kwestie samego przyznawania uprawnień oraz uwierzytelniania użytkownika w aplikacji pozostały jednak nierozstrzygnięte. Wymagają one wprowadzenia Spring Security, którym to właśnie zajmiemy się w następnym rozdziale.

9 Zabezpieczanie Springa

W tym rozdziale omówimy:

- Podstawy Spring Security
- Zabezpieczanie aplikacji sieciowych za pomocą filtrów serwletów
- Uwierzytelnianie w bazach danych oraz katalogach LDAP

Czy zwróciłeś kiedyś uwagę, że większość ludzi w amerykańskich serialach komediowych nie zamyka drzwi? W *Kronikach Seinfelda* Kramer często zapuszcza się do apartamentu Jerry'ego, wyjadając mu co lepsze rzeczy z lodówki. W *Przyjaciołach* różni bohaterowie wchodzą sobie wzajemnie do mieszkań bez ostrzeżenia i chwili zawahania. W jednym z odcinków Ross wpada nawet do pokoju hotelowego Chandlera, prawie nakrywając go w niedwuznacznej sytuacji z siostrą Rossa.

Kiedyś może nikogo by to nie dziwiło. Ale w dzisiejszych czasach, kiedy prywatność i bezpieczeństwo są tak często na pierwszym planie, dostępność domostw telewizyjnych postaci wydaje się wręcz niesamowita.

Informacja jest obecnie najcenniejszym dobrem, nie powinno zatem dziwić, że część oszustów poluje na nasze dane, wkradając się do niezabezpieczonych aplikacji.

Jako programiści powinniśmy podjąć kroki w celu ochrony informacji przechowywanej przez naszą aplikację. Niezależnie od tego, czy jest to konto poczty elektronicznej chronione parą nazwa użytkownika-hasło, czy konto w domu maklerskim chronione numerem PIN, bezpieczeństwo jest kluczowym *aspektem* większości aplikacji.

Użycie słowa „aspekt" w kontekście bezpieczeństwa aplikacji nie było przypadkowe. Zagadnienie bezpieczeństwa wykracza poza ramy poszczególnych funkcji aplikacji. Generalnie, rola aplikacji przy zabezpieczaniu samej siebie powinna zostać ograniczona

do minimum. Chociaż część zabezpieczeń można umieścić bezpośrednio w kodzie aplikacji (i nie jest to rzadka praktyka), oddzielenie zagadnień bezpieczeństwa od zagadnień aplikacji jest lepszym rozwiązaniem.

Jeżeli zaczynasz podejrzewać, że bezpieczeństwo aplikacji będziemy osiągać przy użyciu technik programowania aspektowego, masz rację. W tym rozdziale poznamy sposoby zabezpieczenia aplikacji za pomocą aspektów. Samodzielne tworzenie tych aspektów nie będzie jednak konieczne. Powiemy o Spring Security, frameworku bezpieczeństwa zaimplementowanym za pomocą Spring AOP i filtrów serwletów.

9.1. Rozpoczynamy pracę ze Spring Security

Spring Security jest frameworkiem, który zapewnia deklaratywne bezpieczeństwo aplikacji bazujących na Springu. Dostarcza szeroko zakrojonych rozwiązań w zakresie bezpieczeństwa, zajmując się uwierzytelnieniem i autoryzacją zarówno na poziomie żądania sieciowego, jak i na poziomie wywołania metody. Spring Security bazuje na frameworku Spring, co pozwala mu w pełni wykorzystywać wstrzykiwanie zależności (ang. *dependency injection*) i techniki aspektowe.

Spring Security nazywał się początkowo Acegi Security. Acegi był frameworkiem bezpieczeństwa o dużych możliwościach, miał jednak jedną zasadniczą wadę: wymagał *bardzo obszernej* konfiguracji XML. Nie będę się tu zagłębiał w szczegóły takiej konfiguracji. Wystarczy jednak powiedzieć, że typowa konfiguracja Acegi często potrafiła urosnąć do kilkuset wierszy kodu XML.

Wraz z wersją 2.0 Acegi Security został przemianowany na Spring Security. Ta wersja przyniosła jednak dużo więcej niż tylko powierzchowną zmianę nazwy. Spring Security 2.0 wprowadził nową przestrzeń nazw XML, pozwalającą na konfigurację bezpieczeństwa w Springu. Nowa przestrzeń nazw wraz z adnotacjami i rozsądnymi wartościami domyślnymi odchudziły typową konfigurację bezpieczeństwa z setek wierszy kodu do zaledwie kilkunastu. Spring Security 3.0, dorzuciła jeszcze do tego język SpEL, upraszczając konfigurację jeszcze bardziej.

W wersji 3.2, Spring Security zabezpiecza aplikację na dwa sposoby. Do zabezpieczenia żądań sieciowych i ograniczenia dostępu na poziomie adresów URL wykorzystuje filtry serwletów. Potrafi także zabezpieczyć wywołania metod za pomocą Spring AOP — tworząc obiekty pośredniczące i stosując radę, która pilnuje, żeby użytkownik miał wystarczające uprawnienia do wywołania zabezpieczonych metod.

W tym rozdziale skupimy się na zabezpieczaniu warstwy internetowej za pomocą Spring Security. W rozdziale 14. powrócimy do tematu Spring Security i omówione zostanie zabezpieczanie wywołań metod.

9.1.1. Poznajemy moduły Spring Security

Bez względu na to, jaką aplikację zabezpiecza się za pomocą Spring Security, należy zacząć od dodania modułów Spring Security do ścieżki do klas aplikacji. Spring Security 3.2 podzielony został na jedenaście modułów, które opisano w tabeli 9.1.

Tabela 9.1. Spring Security jest podzielony na osiem modułów

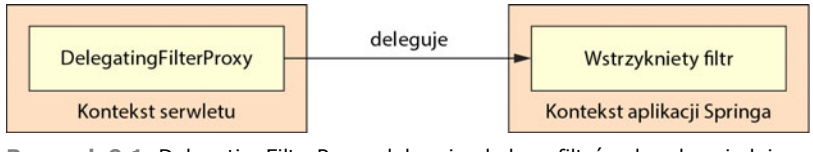
Moduł	Opis
ACL	Zapewnia obsługę zabezpieczeń obiektu dziedziny poprzez listy kontroli dostępu (ang. ACL — *access control list*).
Aspects	Niewielki moduł, który zapewnia wsparcie dla aspektów AspectJ, zastępując standardowe aspekty Spring AOP, przy wykorzystaniu adnotacji Spring Security.
CAS Client	Zapewnia wsparcie dla pojedynczego punktu logowania z użyciem Centralnego Systemu Uwierzytelniania (ang. CAS — *Central Authentication Service*) projektu Jasig.
Configuration	Zawiera wsparcie dla konfiguracji Spring Security za pomocą XML i Javy. Wsparcie dla konfiguracji Javy w Spring Security pojawiło się w wersji 3.2.
Core	Dostarcza podstawową bibliotekę Spring Security.
Cryptography	Dostarcza wsparcie dla szyfrowania i kodowania hasła.
LDAP	Zapewnia obsługę uwierzytelniania za pomocą protokołu LDAP (*Lightweight Directory Access Protocol*).
OpenID	Zapewnia wsparcie dla scentralizowanego uwierzytalniania z wykorzystaniem standardu OpenID.
Remoting	Zapewnia integrację z projektem Spring Remoting.
Tag Library	Biblioteka znaczników JSP Spring Security.
Web	Zapewnia obsługę bezpieczeństwa sieciowego opartą na filtrach.

Moduły Core i Configuration to minimum niezbędne do zabezpieczenia aplikacji. Jeżeli mamy do czynienia z aplikacją internetową, a taką z pewnością jest aplikacja Spittr, trzeba też dodać moduł Web do ścieżki do klas. Jako że będziemy korzystać również z biblioteki znaczników Spring Security, dodamy jeszcze moduł Tag Library.

9.1.2. Filtrujemy żądania internetowe

Spring Security wykorzystuje szereg filtrów odpowiadających za różne aspekty bezpieczeństwa. Może Ci się wydawać, że wiąże się to z konfiguracją kilku filtrów w pliku *web.xml* lub klasie `WebApplicationInitializer`, jednak dzięki odrobinie magii Springa potrzebujemy konfiguracji tylko jednego z tych filtrów.

`DelegatingFilterProxy` jest specjalnym filtrem serwletów. Jego rola ogranicza się do pośredniczenia w dostępie do implementacji interfejsu `javax.servlet.Filter`, która zarejestrowana jest jako komponent w kontekście aplikacji Springa, co zilustrowano na rysunku 9.1.



Rysunek 9.1. DelegatingFilterProxy deleguje obsługę filtrów do odpowiedniego komponentu filtra w kontekście aplikacji Springa

Do deklaracji serwletów i filtrów w tradycyjnym pliku *web.xml* możemy wykorzystać element `<filter>`:

```
<filter>
   <filter-name>springSecurityFilterChain</filter-name>
   <filter-class>
```

```
      org.springframework.web.filter.DelegatingFilterProxy
   </filter-class>
</filter>
```

Najistotniejsze jest tu ustawienie wartości elementu `<filter-name>` na `springSecurity➥FilterChain`. Będziemy bowiem wkrótce konfigurować bezpieczeństwo internetowe Spring Security i korzystać z filtra `springSecurityFilterChain`, do którego pracę oddeleguje filtr `DelegatingFilterProxy`.

Jeśli do konfiguracji filtra `DelegatingFilterProxy` chcemy użyć klasy Javy `WebAppli➥cationInitializer`, potrzebujemy tylko stworzenia nowej klasy rozszerzającej klasę `SecurityWebApplicationInitializer`:

```
package spitter.config;
import org.springframework.security.web.context.
   AbstractSecurityWebApplicationInitializer;
public class SecurityWebInitializer extends
   AbstractSecurityWebApplicationInitializer {}
```

Klasa `AbstractSecurityWebInitializer` implementuje interfejs `WebApplicationInitializer`, zostanie więc wyszukana przez Springa i wykorzystana do rejestracji filtra `Delegating➥FilterProxy` w kontenerze. Chociaż do rejestrowania wybranych przez siebie filtrów możemy nadpisywać metody `appendFilters()` oraz `insertFilters()`, nie musimy tego robić, rejestrując filtr `DelegatingFilterProxy`.

Niezależnie od tego, czy filtr ten rejestrujemy w pliku *web.xml*, czy w podklasie klasy abstrakcyjnej `AbstractSecurityWebApplicationInitializer`, wszystkie żądania przychodzące do aplikacji zostaną przechwycone i oddelegowane do komponentu o identyfikatorze `springSecurityFilterChain`.

Sam filtr `springSecurityFilterChain` jest innym specjalnym filtrem typu `FilterChain➥Proxy`. Ten pojedynczy filtr łączy ze sobą jeden lub więcej dodatkowych filtrów. Działanie Spring Security opiera się na kilku filtrach serwletów, które dostarczają różne funkcje bezpieczeństwa. W praktyce bardzo rzadko musimy znać te szczegóły, bo nie mamy potrzeby jawnej deklaracji komponentu `springSecurityFilterChain` ani żadnego z łączonych przez niego filtrów. Filtry te zostaną utworzone po włączeniu ustawień bezpieczeństwa.

Rozpocznijmy naszą przygodę ze Spring Security od stworzenia najprostszej możliwej konfiguracji bezpieczeństwa.

9.1.3. Tworzymy prostą konfigurację bezpieczeństwa

We wczesnych wersjach Spring Security (gdy projekt ten nosił jeszcze nazwę Acegi Security) włączenie zabezpieczeń wymagało napisania setek linii konfiguracji w pliku XML. Spring Security 2.0 uprościł konfigurację poprzez wprowadzenie specjalnej przestrzeni nazw XML.

W Springu 3.2 pojawiła się nowa opcja konfiguracji z wykorzystaniem klas Javy, która pozwala wyeliminować całkowicie potrzebę konfiguracji XML. Poniższy listing 9.1 prezentuje najprostszą możliwą konfigurację Spring Security z użyciem konfiguracji JavaConfig.

Listing 9.1. Najprostsza klasa konfiguracji włączająca ustawienia bezpieczeństwa internetowego w Spring MVC

```
package spitter.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.
   configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.
   configuration.WebSecurityConfigurerAdapter;
@Configuration aplikacji internetowej
@EnableWebSecurity ← Włączamy ustawienia bezpieczeństwa
public class SecurityConfig extends WebSecurityConfigurerAdapter { }
```

Jak sugeruje sama nazwa adnotacji, `@EnableWebSecurity` włącza ustawienia bezpieczeństwa aplikacji internetowej. Sama w sobie jest jednak bezużyteczna. Konfiguracja Spring Security musi się znaleźć w komponencie implementującym interfejs `WebSecurity`
`↪Configurer` lub (dla wygody) rozszerzającym klasę `WebSecurityConfigurerAdapter`. W konfiguracji Spring Security może uczestniczyć każdy komponent dostępny w kontekście aplikacji Springa, najwygodniejsze jest jednak z reguły rozszerzenie klasy `Web`
`↪SecurityConfigurerAdapter`, co zaprezentowano na listingu 9.1.

Adnotacja `@EnableWebSecurity` może się przydać do włączenia zabezpieczeń w każdej aplikacji internetowej. Przy tworzeniu aplikacji z wykorzystaniem Spring MVC warto pomyśleć nad jej zastąpieniem adnotacją `@EnableWebMvcSecurity`, jak pokazano na listingu 9.2.

Listing 9.2. Najprostsza klasa konfiguracji pozwalająca włączyć mechanizmy bezpieczeństwa aplikacji Spring MVC

```
package spitter.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.
   configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.annotation.web.servlet.
   configuration.EnableWebMvcSecurity;
@Configuration
@EnableWebMvcSecurity ← Włączamy ustawienia bezpieczeństwa aplikacji Spring MVC
public class SecurityConfig extends WebSecurityConfigurerAdapter { }
```

Jednym z zadań adnotacji `@EnableWebMvcSecurity` jest konfiguracja rozwiązywania argumentów Spring MVC, aby metoda obsługi żądania uzyskała dostęp do danych uwierzytelniania użytkownika (bądź jego nazwy) za pośrednictwem parametrów opatrzonych adnotacją `@AuthenticationPrincipal`. Konfiguruje też komponenty, które automatycznie dodają do formularzy ukryte pola tokenów CSRF (*cross-site request forgery*) z wykorzystaniem biblioteki znaczników wiązania formularzy Springa.

Może się wydawać, że nie jest to wiele, ale klasa konfiguracji bezpieczeństwa pokazana na listingach 9.1 oraz 9.2 może nam dać nieźle w kość. Obie wersje skutecznie blokują aplikację przed jakąkolwiek próbą dostępu!

Nie jest to wymagane, jednak prawdopodobnie zechcesz dostosować ustawienia bezpieczeństwa, nadpisując jedną lub kilka metod klasy `WebSecurityConfigurerAdapter`.

Konfigurację bezpieczeństwa aplikacji internetowej umożliwia nadpisanie trzech metod `configure` klasy `WebSecurityConfigurerAdapeter` i ustawienie odpowiednich zachowań za pomocą przekazanych parametrów. Tabela 9.2 opisuje te trzy metody.

Tabela 9.2. Nadpisywanie metod configure() klasy WebSecurityConfigurerAdapter

Metoda	Opis
`configure(WebSecurity)`	Jej nadpisanie umożliwia konfigurację łańcucha filtrów Spring Security.
`configure(HttpSecurity)`	Jej nadpisanie pozwala na konfigurację sposobu zabezpieczenia żądań za pomocą interceptorów.
`configure(AuthenticationManagerBuilder)`	Jej nadpisanie umożliwia konfigurację usług szczegółów użytkownika.

Wracając do listingu 9.2 — możemy zauważyć, że nie nadpisuje on żadnej z trzech metod `configure()`, co tłumaczy, dlaczego aplikacja została całkowicie zablokowana. Domyślne ustawienia łańcucha filtrów powinny spełnić nasze oczekiwania, ale domyślna implementacja metody `configure(HttpSecurity)` w praktyce wygląda następująco:

```
protected void configure(HttpSecurity http) throws Exception {
   http
      .authorizeRequests()
         .anyRequest().authenticated()
         .and()
      .formLogin().and()
      .httpBasic();
}
```

Ta prosta domyślna konfiguracja określa sposób zabezpieczania żądań i dostępne opcje uwierzytelniania użytkownika. Wywołanie metod `authorizeRequests()` i `anyRequest().authenticated()` ustawia konieczność uwierzytelniania wszystkich żądań HTTP przychodzących do aplikacji. Włącza również mechanizm Spring Security uwierzytelniania za pomocą formularza logowania (z wykorzystaniem gotowej strony logowania) oraz uwierzytelnianie HTTP Basic.

Równocześnie nie nadpisaliśmy metody `configure(AuthenticationManagerBuilder)`, w systemie uwierzytelniania nie istnieje więc żaden użytkownik. W praktyce oznacza to, że wszystkie żądania wymagają uwierzytelniania, lecz jednocześnie nikt nie może się zalogować.

Aby aplikacja spełniała nasze wymagania, musimy trochę rozbudować konfigurację. W szczególności musimy:

- Skonfigurować bazę użytkowników.
- Określić, które żądania powinny, a które nie powinny wymagać uwierzytelniania, jak również to, jakie uprawnienia muszą mieć odwiedzający stronę użytkownicy.
- Dostarczyć własną stronę logowania, by podmienić ekran domyślny.

Dodatkowo, w zależności od ustawień bezpieczeństwa, możemy wygenerować wybrane fragmenty widoków.

Rozpocznijmy jednak od konfiguracji usługi użytkowników, żeby uzyskać dostęp do ich danych w czasie uwierzytelniania.

9.2. Wybieramy usługi szczegółów użytkownika

Przypuśćmy, że planujesz wyjście na wykwintną kolację w ekskluzywnej restauracji. Rezerwację wykonałeś z kilkutygodniowym wyprzedzeniem, aby mieć pewność, że znajdziesz wolny stolik. Po wejściu do restauracji podajesz gospodarzowi swoje nazwisko. Dowiadujesz się niestety, że nie ma żadnego zapisu o Twojej rezerwacji. Twoje wieczorne plany są poważnie zagrożone. Nie poddajesz się jednak zbyt łatwo i prosisz gospodarza o ponowne sprawdzenie listy rezerwacji. W tym momencie sprawy przybierają nieoczekiwany obrót.

Gospodarz odpowiada, że nie istnieje żadna lista rezerwacji. Twoje nazwisko nie znajduje się na liście — tak jak i żadne inne — bo po prostu nie ma żadnej listy. Jest to pewne wytłumaczenie faktu, że nie możesz wejść do lokalu, mimo że jest pusty. Wyjaśnia to również, dlaczego kilka tygodni później restauracja zostaje zamknięta, a w jej miejscu powstaje meksykańska knajpka.

Podobny scenariusz realizuje się obecnie w naszej aplikacji. Nie ma możliwości zalogowania się do aplikacji. Użytkownikom może się wydawać, że powinni być wpuszczeni do aplikacji, ale nie ma żadnego zapisu na temat ich uprawnień. Przez brak bazy użytkowników aplikacja stała się tak ekskluzywna, że aż kompletnie bezużyteczna.

Potrzebna nam jest właśnie baza użytkowników — miejsce, w którym mogą być przechowywane nazwy użytkowników, hasła i inne dane. Dane te są następnie pobierane przy podejmowaniu decyzji o uwierzytelnianiu użytkowników.

Spring Security jest na szczęście pod tym względem bardzo elastyczny i umożliwia uwierzytelnianie użytkowników w oparciu o niemal dowolne źródło danych. Obsługa najczęściej spotykanych scenariuszy, takich jak baza danych w pamięci, baza relacyjna, serwer LDAP, jest już dostępna w standardzie. Mamy też jednak możliwość utworzenia i podpięcia własnej implementacji bazy użytkowników.

Konfiguracja Spring Security za pomocą klas Javy ułatwia ustawienie jednej lub większej liczby baz użytkowników. Rozpoczniemy od najprostszej opcji: przechowywania danych o użytkownikach w pamięci.

9.2.1. Pracujemy z bazą użytkowników zapisaną w pamięci

Klasa konfiguracji zabezpieczeń rozszerza klasę `WebSecurityConfigurerAdapter`, najprostszym sposobem konfiguracji bazy użytkowników jest więc nadpisanie metody `confi➥gure()` przyjmującej jako parametr obiekt klasy `AuthenticationManagerBuilder`. Klasa `AuthenticationManagerBuilder` udostępnia kilka metod umożliwiających konfigurację mechanizmów Spring Security. Metoda `inMemoryAuthentication()` pozwala włączyć i skonfigurować bazę użytkowników zapisaną w pamięci, a w razie potrzeby wypełnić ją przykładowymi danymi.

Na listingu 9.3 widzimy klasę `SecurityConfig`, która nadpisuje metodę `configure()` w celu ustawienia w pamięci bazy wypełnionej dwoma użytkownikami.

Listing 9.3. Konfigurujemy Spring Security do obsługi bazy użytkowników zapisanych w pamięci

```
package spitter.config;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.security.config.annotation.
   authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.
   configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.annotation.web.servlet.
   configuration.EnableWebMvcSecurity;
@Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

   @Override protected void configure(AuthenticationManagerBuilder auth)
      throws Exception {                                          Włączamy bazę
      auth.                                                       użytkowników
         inMemoryAuthentication()                                 zapisaną w pamięci
            .withUser("user").password("password").roles("USER").and()
     .withUser("admin").password("password").roles("USER", "ADMIN");
}
    }
```

Jak widać na powyższym listingu, klasa `AuthenticationManagerBuilder` przekazana do metody `configure()` korzysta z interfejsu budowania konfiguracji uwierzytelniania. Wywołanie metody `inMemoryAuthentication()` włącza mechanizm przechowywania bazy użytkowników w pamięci. Musimy też utworzyć jakichś użytkowników, bo pusta baza jest równie bezużyteczna jak całkowity jej brak.

Użytkowników tworzymy poprzez wywołanie metody `withUser()`. Metoda przyjmuje jako parametr nazwę użytkownika, a zwraca obiekt typu `UserDetailsManagerConfigurer.UserDetailsBuilder`. Obiekt ten udostępnia kilka metod umożliwiających dalszą konfigurację użytkownika, takich jak metoda `password()`, służąca do określania hasła, i `roles()`, służąca do przydzielania uprawnień.

Na listingu 9.3 dodaliśmy dwóch użytkowników, "`user`" oraz "`admin`", i obu ustawiliśmy hasło "`password`". Użytkownikowi "`user`" nadaliśmy uprawnienie `USER`, a użytkownikowi "`admin`" zarówno `USER`, jak i `ADMIN`. Konfiguracje ustawień poszczególnych użytkowników rozdzielone są za pomocą metody `and()`.

Poza metodami `password()`, `roles()` oraz `and()` dostępnych jest kilka innych metod konfiguracji szczegółów użytkownika dla bazy zapisanej w pamięci. W tabeli 9.3 znajdziesz wszystkie metody dostępne w klasie `UserDetailsManagerConfigurer.UserDetailsBuilder`.

Zauważ, że metoda `roles()` jest skróconą wersją metody `authorities()`. Dowolne wartości przekazane do metody `roles()` poprzedzane są prefiksem `ROLE_` i przypisywane w postaci uprawnień użytkownika. W rezultacie poniższa konfiguracja jest równoważna konfiguracji podanej na listingu 9.3:

```
auth
   .inMemoryAuthentication()
   .withUser("user").password("password")
   .authorities("ROLE_USER").and()
   .withUser("admin").password("password")
   .authorities("ROLE_USER", "ROLE_ADMIN");
```

Tabela 9.3. Metody konfiguracji szczegółów użytkownika

Metoda	Opis
`accountExpired(boolean)`	Definiuje, czy konto wygasło, czy nie.
`accountLocked(boolean)`	Definiuje, czy konto zostało zablokowane, czy nie.
`and()`	Używa się jej do łączenia konfiguracji.
`authorities(GrantedAuthority…)`	Określa jedno lub więcej uprawnień przyznawanych użytkownikowi.
`authorities(List<? extends GrantedAuthority>)`	Określa jedno lub więcej uprawnień przyznawanych użytkownikowi.
`authorities(String…)`	Określa jedno lub więcej uprawnień przyznawanych użytkownikowi.
`credentialsExpired(boolean)`	Definiuje, czy dane logowania wygasły, czy nie.
`disabled(boolean)`	Definiuje, czy konto jest wyłączone, czy nie.
`password(String)`	Określa hasło użytkownika.
`roles(String…)`	Określa jedną lub więcej ról przypisanych użytkownikowi.

Choć baza użytkowników przechowywana w pamięci może być użyteczna w debugowaniu kodu lub w trakcie jego tworzenia, to nie jest najlepszym rozwiązaniem w aplikacji produkcyjnej. W tym przypadku lepsze jest z reguły przechowywanie danych o użytkowniku w jakiejś bazie danych.

9.2.2. Uwierzytelnianie w oparciu o tabele danych

Dane użytkowników przechowywane są często w relacyjnej bazie danych, podłączonej za pośrednictwem JDBC. W Spring Security do konfiguracji uwierzytelniania z wykorzystaniem bazy użytkowników opartej na JDBC możemy zastosować metodę `jdbcAuthen➥tication()`. Minimalna konfiguracja, która jest do tego potrzebna, wygląda następująco:

```
@Autowired
DataSource dataSource;

@Override protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
      .jdbcAuthentication()
      .dataSource(dataSource);
}
```

Jedynym ustawieniem wymagającym konfiguracji jest źródło danych typu `DataSource`, które umożliwia dostęp do relacyjnej bazy danych. W podanym przykładzie jest ono przekazywane z wykorzystaniem magii mechanizmu autowiązania.

NADPISYWANIE DOMYŚLNYCH ZAPYTAŃ UŻYTKOWNIKA

Choć podana powyżej minimalistyczna konfiguracja działa, przyjmuje pewne założenia odnośnie do schematu wykorzystywanej bazy danych. Oczekuje istnienia pewnych tabel, w których mają być przechowywane dane użytkowników. W szczególności poniższy fragment kodu źródłowego Spring Security zawiera zapytania, które zostaną uruchomione podczas sprawdzania danych użytkowników:

```
public static final String DEF_USERS_BY_USERNAME_QUERY =
   "select username,password,enabled " +
   "from users " +
   "where username = ?";
public static final String DEF_AUTHORITIES_BY_USERNAME_QUERY =
   "select username,authority " +
   "from authorities " +
   "where username = ?";
public static final String DEF_GROUP_AUTHORITIES_BY_USERNAME_QUERY =
   "select g.id, g.group_name, ga.authority " +
   "from groups g, group_members gm, group_authorities ga " +
   "where gm.username = ? " +
   "and g.id = ga.group_id " +
   "and g.id = gm.group_id";
```

Pierwsze zapytanie pobiera nazwę i hasło użytkownika oraz informację o tym, czy jego konto jest włączone. Informacje te wykorzystywane są do uwierzytelniania użytkownika. Kolejne zapytanie sprawdza dane użytkownika w celu przeprowadzenia uwierzytelniania, a ostatnie pobiera uprawnienia przyznane użytkownikowi jako członkowi grupy.

Jeśli definiowanie tabel i wypełnianie ich danymi zgodnie z wymaganiami tych zapytań nie jest dla nas problemem, nie musimy już nic więcej robić. Często się jednak zdarza, że schemat naszej bazy wcale tak nie wygląda i potrzebna nam jest kontrola nad strukturą tych zapytań. W takim przypadku możemy zdefiniować własne zapytania w następujący sposób:

```
@Override protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
      .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
         "select username, password, true " +
         "from Spitter where username=?")
      .authoritiesByUsernameQuery(
         "select username, 'ROLE_USER' from Spitter where username=?");
}
```

W takiej sytuacji nadpisujemy tylko zapytania uwierzytelniania i autoryzacji. Mamy też jednak możliwość nadpisania zapytań uprawnień grupy poprzez wywołanie metody `groupAuthoritiesByUsername()` i przekazanie do niej własnego zapytania w postaci parametru.

Przy nadpisaniu domyślnych zapytań SQL musimy pamiętać o przestrzeganiu podstawowego kontraktu zapytań. Wszystkie one przyjmują nazwę użytkownika jako jedyny parametr. Zapytanie uwierzytelniające pobiera nazwę użytkownika, hasło oraz status o aktywności konta. Zapytanie o uprawnienia pobiera zero lub więcej rekordów, z których każdy zawiera identyfikator grupy, nazwę grupy i nazwę przydzielonego uprawnienia. Zapytanie o uprawnienia grupy pobiera zero bądź więcej rekordów, z których każdy zawiera identyfikator grupy, nazwę grupy oraz nazwę uprawnienia.

PRACUJEMY Z SZYFROWANYMI HASŁAMI

Gdy spojrzymy na zapytanie uwierzytelniające, zauważymy, że w bazie musi być zapisane hasło użytkownika. Jedynym problemem, jaki się z tym wiąże, jest to, że hasła te przechowywane są w postaci niezaszyfrowanego ciągu znaków, co grozi ich podejrzeniem przez wścibskie oko hakera. Z kolei jeśli zaszyfrujemy hasło w bazie, uwierzytelnianie nie powiedzie się, bo zapisane hasło nie będzie odpowiadać niezaszyfrowanemu hasłu przesłanemu przez użytkownika.

Aby poradzić sobie z tym problemem, musimy określić sposób kodowania hasła za pomocą metody `passwordEncoder()`:

```
@Override protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
      .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
         "select username, password, true " +
         "from Spitter where username=?")
      .authoritiesByUsernameQuery(
         "select username, 'ROLE_USER' from Spitter where username=?")
      .passwordEncoder(new StandardPasswordEncoder("53Kr3t"));
}
```

Metoda `passwordEncoder` przyjmuje jako parametr dowolną implementację dostępnego w Spring Security interfejsu `PasswordEncoder`. Moduł kryptografii Spring Security udostępnia trzy takie implementacje: `BCryptPasswordEncoder`, `NoOpPasswordEncoder` oraz `StandardPasswordEncoder`.

W poniższym kodzie wykorzystujemy implementację `StandardPasswordEncoder`. Możemy jednak w każdej chwili przygotować naszą własną implementację, jeśli żadna z dostarczonych przez framework nie spełnia naszych oczekiwań. Interfejs `Password` `↪Encoder` jest raczej prosty:

```
public interface PasswordEncoder {
   String encode(CharSequence rawPassword);
   boolean matches(CharSequence rawPassword, String encodedPassword);
}
```

Niezależnie od wybranego sposobu szyfrowania musimy wiedzieć, że hasła zapisane w bazie nie są nigdy odszyfrowywane. Szyfrowane jest natomiast tym samym algorytmem hasło wprowadzone przez użytkownika, a po zaszyfrowaniu porównywane jest ono z hasłem zapisanym w bazie. Do porównywania haseł służy metoda `matches()` interfejsu.

Bazy relacyjne są zaledwie jedną z opcji zapisywania danych użytkownika. Inną powszechnie wykorzystywaną możliwością jest przechowywanie danych w repozytorium LDAP.

9.2.3. Uwierzytelniamy użytkownika w oparciu o usługę LDAP

Do konfiguracji Spring Security w oparciu o usługę LDAP służy metoda `ldapAuthenti` `↪cation()`. Jest ona odpowiednikiem metody `jdbcAuthentication()` dla uwierzytelnianiu LDAP. Poniższa metoda `configure()` wskazuje prostą konfigurację uwierzytelniania LDAP:

```
@Override protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
      .ldapAuthentication()
         .userSearchFilter("(uid={0})")
         .groupSearchFilter("member={0}");
}
```

Metody `userSearchFilter()` oraz `groupSearchFilter()` umożliwiają wykorzystanie filtrów dla podstaw zapytań LDAP do wyszukiwania użytkowników i grup. Domyślnie podstawa zapytania dla użytkowników i grup jest pusta, co wskazuje, że wyszukiwanie zostanie przeprowadzone z katalogu głównego hierarchii LDAP. Możemy to jednak zmienić poprzez określenie podstawy zapytania:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
      .ldapAuthentication()
         .userSearchBase("ou=people")
         .userSearchFilter("(uid={0})")
         .groupSearchBase("ou=groups")
         .groupSearchFilter("member={0}");
}
```

Metoda `userSearchBase()` dostarcza podstawę zapytania wyszukiwania użytkowników. Analogicznie metoda `groupSearchBase()` określa podstawę zapytania wyszukiwania grup. Wyszukiwania nie musimy więc rozpoczynać od korzenia drzewa. W naszym przykładzie szukamy użytkowników w gałęzi, w której wartością jednostki organizacyjnej jest `people`. Grupy przeszukiwane są w jednostce organizacyjnej `groups`.

KONFIGURUJEMY PORÓWNYWANIE HASEŁ

Domyślną strategią uwierzytelniania za pośrednictwem usługi LDAP jest przeprowadzenie operacji wiązania, czyli uwierzytelnianie użytkownika bezpośrednio na serwerze LDAP. Inną dostępną opcją jest wykonanie operacji porównania. Hasło wprowadzone przez użytkownika przesyłane jest do katalogu LDAP z prośbą o porównanie z atrybutem `password` użytkownika. Porównanie przeprowadzane jest w ramach serwera LDAP, hasło pozostaje więc tajne.

Możliwość wykorzystania opcji porównywania haseł otrzymujemy dzięki deklaracji metody `passwordCompare()`:

```
@Override protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
      .ldapAuthentication()
         .userSearchBase("ou=people")
         .userSearchFilter("(uid={0})")
         .groupSearchBase("ou=groups")
         .groupSearchFilter("member={0}")
         .passwordCompare();
}
```

Domyślnie hasło wprowadzone na formularzu logowania porównywane jest z wartością atrybutu `userPassword` dostępnego we wpisie użytkownika w katalogu LDAP. Jeśli hasło przechowywane jest w innym atrybucie, możemy ustawić jego nazwę za pomocą metody `passwordAttribute()`:

```
@Override protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
      .ldapAuthentication()
         .userSearchBase("ou=people")
         .userSearchFilter("(uid={0})")
         .groupSearchBase("ou=groups")
         .groupSearchFilter("member={0}")
         .passwordCompare()
         .passwordEncoder(new Md5PasswordEncoder())
         .passwordAttribute("passcode");
}
```

W powyższym przykładzie wskazaliśmy, że podane przez użytkownika hasło ma zostać porównane z atrybutem "`passcode`" zapisanym w katalogu LDAP. Mamy również możliwość określenia sposobu szyfrowania hasła. Dobre jest to, że przy porównywaniu haseł po stronie serwera są one przechowywane w wersji zaszyfrowanej. Hasło wysłane do porównania przesyłane jest jednak w sieci i może zostać przechwycone przez hakera. Możemy temu zapobiec, określając strategię szyfrowania za pomocą metody `password` `↪Encoder()`.

W przykładzie wykorzystaliśmy szyfrowanie z użyciem algorytmu MD5. Zakładamy, że algorytm ten jest stosowany do szyfrowania haseł na serwerze LDAP.

ŁĄCZYMY SIĘ ZE ZDALNYM SERWEREM LDAP

Kwestią, którą do tej pory pomijałem, jest lokalizacja serwera LDAP i miejsca przechowywania danych. Konfigurowaliśmy Springa do uwierzytelniania za pomocą serwera LDAP, ale gdzie ten serwer się znajduje?

Przy ustawieniu domyślnym Spring Security zakłada, że serwer LDAP nasłuchuje na porcie 33389 na serwerze lokalnym. Jeśli serwer LDAP znajduje się na innej maszynie, metoda `contextSource()` umożliwia nam skonfigurowanie jego lokalizacji:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
   .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .contextSource()
         .url("ldap://habuma.com:389/dc=habuma,dc=com");
}
```

Metoda `contextSource()` zwraca obiekt typu `ContextSourceBuilder`, który udostępnia różne metody. Jest pośród nich metoda `url()`, pozwalającą na określenie lokalizacji serwera LDAP.

KONFIGURUJEMY WBUDOWANY SERWER LDAP

Jeżeli nie mamy do dyspozycji żadnego serwera LDAP, możemy skorzystać z wbudowanego serwera LDAP dostarczanego przez Spring Security. Nie musimy określać adresu URL zdalnego serwera, a zamiast tego możemy za pomocą metody `root()` określić główny sufiks wbudowanego serwera:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
   .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .contextSource()
         .root("dc=habuma,dc=com");
}
```

Serwer LDAP po uruchomieniu próbuje wczytać dane ze wszystkich plików LDIF, które odnajdzie w ścieżce klas. Pliki LDIF (*LDAP Data Interchange Format*) są standardowym sposobem przedstawiania danych LDAP w zwykłych plikach tekstowych. Każdy rekord składa się z jednej lub większej liczby linii, złożonych z par nazwa-wartość. Rekordy oddzielone są od siebie za pomocą pustych linii.

Jeśli nie chcemy, żeby Spring przegrzebywał się przez całą ścieżkę klas w poszukiwaniu plików LDIF, możemy przy użyciu metody `ldif()` w bardziej konkretny sposób wskazać mu lokalizację pożądanego pliku LDIF:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
   .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .contextSource()
         .root("dc=habuma,dc=com")
         .ldif("classpath:users.ldif");
}
```

Poprosiliśmy serwer LDAP o wczytanie zawartości pliku *users.ldiff*, zapisanego w głównym katalogu ścieżki plików. Do wczytania danych użytkownika do serwera LDAP możemy wykorzystać wbudowany serwer LDAP:

```
dn: ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups
dn: ou=people,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people
```

```
dn: uid=habuma,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Craig Walls
sn: Walls
uid: habuma
userPassword: password
dn: uid=jsmith,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: John Smith
sn: Smith
uid: jsmith
userPassword: password
dn: cn=spittr,ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: groupOfNames
cn: spittr
member: uid=habuma,ou=people,dc=habuma,dc=com
```

Wbudowane w Spring Security bazy użytkowników są wygodne i obejmują najczęstsze przypadki użycia. Jeżeli jednak potrzebujemy przeprowadzenia niestandardowego procesu uwierzytelniania, musimy utworzyć i skonfigurować własną usługę obsługi danych użytkowników.

9.2.4. Tworzymy własną usługę użytkowników

Załóżmy, że chcemy uwierzytelnić użytkowników za pomocą nierelacyjnej bazy danych, takiej jak Mongo lub Neo4j. W takim przypadku musimy utworzyć własną implementację interfejsu `UserDetailsService`.

Interfejs `UserDetailsService` jest raczej prosty:

```
public interface UserDetailsService {
   UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException;
}
```

Musimy jedynie zaimplementować metodę `loadUserByUsername()` do wyszukiwania użytkownika w oparciu o jego nazwę. Metoda `loadUserByUsername()` zwraca obiekt `User`
`↪Details`, reprezentujący wybranego użytkownika. Listing 9.4 zawiera implementację usługi `UserDetailsService`, która wyszukuje użytkownika za pośrednictwem przekazanej implementacji repozytorium `SpitterRepository`.

Listing 9.4. Pobieramy obiekt typu UserDetails z repozytorium SpitterRepository

```
package spittr.security;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.
   SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
```

```
import org.springframework.security.core.userdetails.
   UserDetailsService;
import org.springframework.security.core.userdetails.
   UsernameNotFoundException;
import spittr.Spitter;
import spittr.data.SpitterRepository;

public class SpitterUserService implements UserDetailsService {
   private final SpitterRepository spitterRepository;

   public SpitterUserService(
      SpitterRepository spitterRepository) {
      this.spitterRepository = spitterRepository;
   }

   @Override
   public UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException {
      Spitter spitter =
         spitterRepository.findByUsername(username);
      if (spitter != null) {
         List<GrantedAuthority> authorities =
            new  ArrayList<GrantedAuthority>();
         authorities.add(new SimpleGrantedAuthority("ROLE_SPITTER"));
         return new User(
            spitter.getUsername(),
            spitter.getPassword(),
            authorities);
      }
      throw new UsernameNotFoundException(
         "Nie znaleziono użytkownika '" + username + "'.");
   }
}
```

Wstrzykujemy SpitterRepository

Wyszukujemy Spitter

Tworzymy listę uprawnień

Zwracamy użytkownika

W klasie `SpitterUserService` najbardziej interesuje nas sposób przechowywania danych. Przekazywane do usługi repozytorium `SpitterRepository` może pobierać dane z bazy relacyjnej, dokumentowej lub grafowej, może je również utworzyć w locie. Usługa `SpitterUserService` nie ma najmniejszego pojęcia, z jakiej bazy korzysta, i wcale nie musi tego wiedzieć. Pobiera po prostu obiekt typu `Spitter` i tworzy obiekt `User`. (`User` jest konkretną implementacją interfejsu `UserDetails`).

Aby do uwierzytelniania użytkowników zastosować serwis `SpitterUserService`, konfigurujemy ustawienia zabezpieczeń za pomocą metody `userDetailService()`:

```
@Autowired
SpitterRepository spitterRepository;

@Override
protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
   auth
      .userDetailsService(new SpitterUserService(spitterRepository));
}
```

Metoda `userDetailsService()` (podobnie jak metody `jdbcAuthentication()`, `ldapAuthen`
`↪tication()` oraz `inMemoryAuthentication()`) służy do konfiguracji bazy. Nie korzysta jednak z opcji dostarczanych przez Springa, ale przyjmuje dowolną implementację interfejsu `UserDetailsService`.

Moglibyśmy też zmienić implementację klasy `Spitter` tak, by implementowała interfejs `UserDetailsService`. W ten sposób moglibyśmy zwracać obiekt `Spitter` bezpośrednio z metody `loadUserByUsername()` bez kopiowania wartości do obiektu `User`.

9.3. Przechwytywanie żądań

W sekcji 9.1.3 widzieliśmy bardzo prosty przykład konfiguracji Spring Security; dowiedziałeś się, że domyślnym ustawieniem jest uwierzytelnianie wszystkich żądań. Ktoś mógłby powiedzieć, że nadmiar bezpieczeństwa jest lepszy niż jego niedobór. Trzeba jednak też coś wspomnieć o wyważeniu potrzeb w tym zakresie.

Nie ma praktycznie dwóch aplikacji, które zabezpieczamy dokładnie w taki sam sposób. Niektóre żądania wymagają uwierzytelniania, a inne nie. Część żądań może być dostępna dla użytkowników o określonych uprawnieniach i niedostępna dla użytkowników bez tych uprawnień.

Rozważmy dla przykładu żądania obsługiwane przez aplikację Spittr. Strona główna jest publiczna i nie musimy jej zabezpieczać. Również wszystkie obiekty typu `Spittle` są publiczne, dlatego strony je wyświetlające także nie wymagają zabezpieczeń. Ale działania służące do tworzenia spittle'ów powinny być przeprowadzane przez uwierzytelnionego użytkownika. Podobnie, chociaż strony profili użytkownika są publiczne i nie wymagają uwierzytelniania, do obsłużenia żądania do zasobu `/spitters/me` tak, żeby wyświetlał profil aktualnego użytkownika, uwierzytelnianie jest już wymagane.

Kluczem do ustawienia zabezpieczeń dla poszczególnych żądań jest nadpisanie metody `configure(HttpSecurity)`. Poniższy przykład pokazuje, jak nadpisać metodę `configure(HttpSecurity)`, aby ustawić wybrane reguły zabezpieczeń dla różnych adresów URL.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .authorizeRequests()
         .antMatchers("/spitters/me").authenticated()
         .antMatchers(HttpMethod.POST, "/spittles").authenticated()
         .anyRequest().permitAll();
}
```

Obiekt `HttpSecurity` przekazany do metody `configure()` można wykorzystać do konfiguracji kilku aspektów bezpieczeństwa HTTP. W przykładzie wywołujemy metodę `authorizeRequests()`, a następnie metody zwracanego obiektu, by skonfigurować szczegóły zabezpieczeń na poziomie żądania.

Pierwsze wywołanie metody `antMatchers()` wskazuje, że żądanie, którego ścieżką jest `/spitters/me`, należy uwierzytelnić. Drugie wywołanie metody `antMatchers()` jest jeszcze bardziej konkretne i wskazuje, że uwierzytelnione mają być wszystkie żądania typu `POST` do zasobu `/spittles`.

Na koniec wywołanie metody `anyRequests()` informuje, że wszystkie pozostałe żądania powinny być dozwolone bez potrzeby uwierzytelniania i jakichkolwiek uprawnień.

Ścieżka przekazana do metody `antMatchers()` wspiera użycie symboli zastępczych w stylu Anta. Choć w podanym przykładzie z nich nie korzystamy, moglibyśmy określić ścieżkę następująco:

```
.antMatchers("/spitters/**").authenticated();
```

W pojedynczym wywołaniu metody `antMatchers()` możemy określić wiele ścieżek:

```
.antMatchers("/spitters/**", "/spittles/mine").authenticated();
```

Metoda `antMatchers()` umożliwia zdefiniowanie ścieżek z użyciem symboli zastępczych w stylu Anta. Równocześnie istnieje też metoda `regexMatchers()`, która do definiowania ścieżek żądań wykorzystuje wyrażenia regularne. Przykładowo poniższy fragment kodu definiuje wyrażenie regularne odpowiadające wyrażeniu `/spitters/**` (w stylu Anta):

```
.regexMatchers("/spitters/.*").authenticated();
```

Poza zdefiniowaniem ścieżki za pomocą metod `authenticated()` i `permitAll()` wybraliśmy też sposób jej zabezpieczenia. Metoda `authenticated()` wymaga, aby żądanie przeprowadził zalogowany użytkownik. Jeśli użytkownik nie jest zalogowany, filtry Spring Security przechwytują żądanie i przekierowują użytkownika na stronę logowania aplikacji. Metoda `permitAll()` pozwala z kolei na wywoływanie żądań bez potrzeby uwierzytelniania.

Poza wspomnianymi metodami `authenticated()` i `permitAll()` istnieją też inne metody, które można wykorzystać do definiowania sposobu obsługi żądań. W tabeli 9.4 opisano wszystkie dostępne opcje.

Metody wypisane w tabeli 9.4 pozwalają tak skonfigurować ustawienia bezpieczeństwa, żeby wymagane było coś więcej niż tylko bycie uwierzytelnionym użytkownikiem. Moglibyśmy na przykład zmienić poprzednią metodę `configure()` tak, by oprócz samego uwierzytelniania wymagane też było uprawnienie `ROLE_SPITTER`:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .authorizeRequests()
         .antMatchers("/spitters/me").hasAuthority("ROLE_SPITTER")
         .antMatchers(HttpMethod.POST, "/spittles")
            .hasAuthority("ROLE_SPITTER")
         .anyRequest().permitAll();
}
```

Możemy również użyć metody `hasRole()`, aby prefiks `ROLE_` został dodany automatycznie:

```
@Override protected void configure(HttpSecurity http) throws Exception {
   http
      .authorizeRequests()
         .antMatchers("/spitter/me").hasRole("SPITTER")
         .antMatchers(HttpMethod.POST, "/spittles").hasRole("SPITTER")
         .anyRequest().permitAll();
}
```

Tabela 9.4. Metody konfiguracji do definiowania sposobu zabezpieczania ścieżek

Metoda	Do czego służy
`access(String)`	Umożliwia dostęp, jeśli wartością wyrażenia SpEL jest `true`.
`anonymous()`	Umożliwia dostęp anonimowym użytkownikom.
`authenticated()`	Umożliwia dostęp uwierzytelnionym użytkownikom.
`denyAll()`	Bezwarunkowo odmawia dostępu.
`fullyAuthenticated()`	Umożliwia dostęp w pełni uwierzytelnionym użytkownikom (niezapamiętanym).
`hasAnyAuthority(String…)`	Umożliwia dostęp użytkownikom, którzy posiadają przynajmniej jedno z podanych uprawnień.
`hasAnyRole(String…)`	Umożliwia dostęp użytkownikom, którzy posiadają przynajmniej jedną z podanych ról.
`hasAuthority(String)`	Umożliwia dostęp użytkownikowi, jeśli posiada wybrane uprawnienie.
`hasIpAddress(String)`	Umożliwia dostęp, jeżeli żądanie pochodzi z podanego adresu IP.
`hasRole(String)`	Umożliwia dostęp użytkownikowi, jeśli posiada podaną rolę.
`not()`	Neguje efekt wszystkich powyższych metod dostępu.
`permitAll()`	Bezwarunkowo przyznaje dostęp.
`rememberMe()`	Umożliwia dostęp użytkownikom, którzy są uwierzytelnieni z użyciem opcji zapamiętania (remember-me).

Możemy łączyć ze sobą dowolną liczbę wywołań metod `antMatchers()`, `regexMatchers()` oraz `anyRequest()`, by w pełni ustalić reguły bezpieczeństwa aplikacji. Musimy jednak pamiętać, że zostaną one zastosowane w zadanej kolejności. Z tego powodu jako pierwsze powinniśmy zdefiniować najbardziej konkretne wzorce ścieżek, a te najbardziej ogólne (jak `anyRequest()`) — na samym końcu. W przeciwnym wypadku te ogólne wzorce „przechwycą" te bardziej konkretne.

9.3.1. Zabezpieczanie za pomocą wyrażeń Springa

Większość metod zdefiniowanych w tabeli 9.4 jest jednowymiarowa. Oznacza to, że możemy użyć metody `hasRole()` do sprawdzenia pojedynczej roli, ale dla tej samej ścieżki nie możemy równocześnie wykorzystać metody `hasIpAddress()` do weryfikacji adresu IP.

Co więcej, nie możemy pracować z warunkami niezdefiniowanymi przez metody wskazane w tabeli 9.4. A co, jeśli chcemy ograniczyć dostęp dla pewnej roli jedynie w czwartek?

W rozdziale 3. nauczyłeś się korzystać z języka SpEL (*Spring Expression Language*) jako zaawansowanej techniki wiązania właściwości komponentów. Język ten możemy też zastosować do deklaracji wymagań dostępu, używając metody `access()`. Na przykład aby za pomocą wyrażenia SpEL ustawić wymagane uprawnienia `ROLE_SPITTER` dla wzorca URL `/spitter/me`, możemy wykorzystać poniższy fragment kodu:

```
.antMatchers("/spitter/me").access("hasRole('ROLE_SPITTER')")
```

Ograniczenie nałożone na URL `/spitter/me` jest równoważne temu, od którego zaczęliśmy, ale tym razem do wyrażenia reguł bezpieczeństwa użyliśmy wyrażenia SpEL.

Wyrażenie `hasRole()` zwraca wartość `true`, jeśli aktualnemu użytkownikowi przyznano wskazane uprawnienie.

Wyrażenia SpEL oferują jednak większe możliwości, bo `hasRole()` jest tylko jednym z dostępnych wyrażeń specyficznych dla bezpieczeństwa. Tabela 9.5 zawiera listę wszystkich wyrażeń SpEL dostępnych w Spring Security.

Tabela 9.5. Spring Security rozszerza język Spring Expression Language o szereg wyrażeń związanych z bezpieczeństwem

Wyrażenie bezpieczeństwa	Przyjmowane wartości
`authentication`	Obiekt `Authentication` użytkownika.
`denyAll`	Zawsze `false`.
`hasAnyRole(lista ról)`	`true`, jeżeli użytkownik ma przydzieloną przynajmniej jedną z listy określonych ról.
`hasRole(rola)`	`true`, jeżeli użytkownik ma przydzieloną określoną rolę.
`hasIpAddress(adres IP)`	`true`, jeżeli adres IP użytkownika pasuje do przekazanego adresu (tylko w przypadku aplikacji internetowych).
`isAnonymous()`	`true`, jeżeli bieżący użytkownik jest anonimowy.
`isAuthenticated()`	`true`, jeżeli bieżący użytkownik nie jest anonimowy.
`isFullyAuthenticated()`	`true`, jeżeli bieżący użytkownik nie jest ani anonimowym, ani pamiętanym użytkownikiem.
`isRememberMe()`	`true`, jeżeli użytkownik został automatycznie uwierzytelniony dzięki opcji „zapamiętaj mnie”.
`permitAll`	Zawsze `true`.
`principal`	Obiekt `Principal` użytkownika.

Mając do dyspozycji takie narzędzie jak wyrażenia SpEL, możemy zrobić coś więcej, niż ograniczyć dostęp na podstawie przyznanych użytkownikowi uprawnień. Jeśli chcielibyśmy na przykład, aby adres URL zgodny ze wzorcem `/spitter/me` nie tylko wymagał roli `ROLE_ADMIN`, ale również był dostępny wyłącznie z określonego adresu IP, wywołanie metody `access()` mogłoby wyglądać w następujący sposób:

```
.antMatchers("/spitter/me")
   .access("hasRole('ROLE_SPITTER') and hasIpAddress('192.168.1.2')")
```

Możliwości w zakresie restrykcji bezpieczeństwa są dzięki SpEL praktycznie nieograniczone. Jestem pewien, że masz już kilka pomysłów restrykcji bazujących na SpEL.

Teraz jednak spójrzmy na inny sposób przechwytywania żądań przez Spring Security w celu wymuszenia bezpieczeństwa kanału komunikacji.

9.3.2. Wymuszamy bezpieczeństwo kanału komunikacji

Wysyłanie danych za pomocą HTTP to ryzykowna sprawa. Być może nie ma to aż tak dużego znaczenia przy wiadomości spittle, ale przesyłanie informacji poufnych, jak na przykład hasła czy numery kart kredytowych, jest proszeniem się o kłopoty. Wymiana danych za pomocą HTTP następuje w postaci niezaszyfrowanej, co umożliwia hakerowi

przechwycenie żądania i podejrzenie informacji, do której nie powinien mieć dostępu. Tego rodzaju informacje powinny być przekazywane w postaci zaszyfrowanej poprzez HTTPS.

Praca z HTTPS wydaje się dosyć prosta. Właściwie wystarczy tylko dodać `s` do `http` w adresie URL. Ale czy to na pewno wszystko?

Powyższe stwierdzenie jest prawdziwe, ale warto też wspomnieć o odpowiedzialności za użycie kanału HTTP w złym miejscu. Tak jak łatwo jest zabezpieczyć połączenie poprzez dodanie s w adresie URL, tak samo łatwo można o tym s zapomnieć. Kiedy mamy kilka odnośników, które powinny korzystać z HTTPS, jest bardzo prawdopodobne, że w jednym lub dwóch miejscach o tym zapomnimy.

Możesz też przesadzić z korektami i użyć HTTPS w miejscach, gdzie nie było to potrzebne.

Obiekt `HttpSecurity` przekazany do metody `configure()` poza metodą `authorize` ↪`Requests()` posiada także metodę `requiresChannel()`, która umożliwia deklarację wymagań dotyczących wykorzystywanego kanału komunikacji dla różnych wzorców URL.

Rozważmy przykładowo formularz rejestracyjny aplikacji Spittr. Mimo że Spittr nie wymaga podawania numerów kart kredytowych, numeru PESEL ani żadnych innych specjalnie poufnych danych, użytkownicy mogą nie chcieć upubliczniać wysyłanych informacji. Aby formularz rejestracyjny był zawsze przesyłany kanałem HTTPS, możemy w konfiguracji wywołać metodę `requiresChannel()`, jak na listingu 9.5.

Listing 9.5. Metoda requiresChannel() wymusza komunikację HTTPS dla wybranych adresów URL

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .authorizeRequests()
         .antMatchers("/spitter/me").hasRole("SPITTER")
         .antMatchers(HttpMethod.POST, "/spittles").hasRole("SPITTER")
         .anyRequest().permitAll();
      .and()
      .requiresChannel()
         .antMatchers("/spitter/form").requiresSecure(); ← Wymagamy użycia HTTPS
}
```

Każde żądanie `/spitter/form` zostanie zidentyfikowane przez Spring Security jako wymagające kanału *https* (dzięki użyciu metody `requiresSecure()`), a następnie automatycznie odpowiednio przekierowane.

Równocześnie niektóre strony nie wymagają zastosowania kanału HTTPS. Strona domowa na przykład nie przekazuje żadnych poufnych informacji i powinna być przesyłana kanałem HTTP. Możemy zatem zadeklarować, że jej żądania przesyłane będą zawsze przez HTTP przy wykorzystaniu metody `requiresInsecure()` zamiast metody `requireSecure()`:

```
.antMatchers("/").requiresInecure();
```

Jeśli do strony / przyjdzie żądanie kanałem HTTPS, Spring Security automatycznie przekieruje je na niezabezpieczony kanał HTTP.

Zauważmy, że opcje wyboru ścieżki dla kanału HTTP są dokładnie takie same jak w przypadku metody `authorizeRequests()`. Na listingu 9.5 zastosowaliśmy metodę `antMatchers()`, ale moglibyśmy też wykorzystać metodę `regexMatchers()` i ustalić wzorzec ścieżki za pomocą wyrażenia regularnego.

9.3.3. Ochrona przed atakami CSRF

Jak zapewne pamiętasz, kontroler `SpittleController` tworzy nowy obiekt typu `Spittle`, kiedy użytkownik wyśle na adres `/spittles` żądanie `POST`. Co się jednak stanie, gdy żądanie `POST` zostanie wysłane z jakiejś innej strony? Co się stanie, jeśli jest wynikiem wysłania danych z następującego formularza, umieszczonego na innej stronie?

```
<form method="POST" action="http://www.spittr.com/spittles">
   <input type="hidden" name="message" value="Ale ze mnie głupek!" />
   <input type="submit" value="Kliknij tutaj, aby wygrać nowy samochód!" />
</form>
```

Załóżmy, że skusimy się na tę wspaniałą ofertę, klikniemy przycisk i wyślemy formularz na adres *http://www.spittr.com/spittles*. Jeżeli byliśmy wcześniej zalogowani w serwisie *http://spittr.com/*, wyślemy nową wiadomość, jasno wskazującą na to, że nie była to dobra decyzja.

Jest to prosty przykład ataku typu CSRF (*cross-site request forgery*). W skrócie atak ten polega na nakłonieniu użytkownika do przesłania żądania na inną stronę, co z reguły kończy się niemiłym dla zaatakowanej osoby wynikiem. I chociaż wysłanie komunikatu: „Ale ze mnie głupek!" na stronę mikroblogu nie jest najgorszą z możliwych konsekwencji tego ataku, łatwo można sobie wyobrazić poważniejsze zagrożenia, jak chociażby wykonanie niepożądanej operacji na naszym koncie bankowym.

Począwszy od Spring Security w wersji 3.2, zabezpieczenie przed atakami typu CSRF jest domyślnie włączone. W praktyce, jeśli nie wykonamy żadnych czynności, aby przystosować aplikację do pracy z tą ochroną lub ją wyłączyć, będziemy mieć problem z pomyślnym wysłaniem żądania przez formularz w naszej aplikacji.

Spring Security realizuje ochronę CSRF za pomocą synchronizowanych tokenów. Żądania zmieniające stan (na przykład wszystkie żądania poza `GET`, `HEAD`, `OPTIONS` i `TRACE`) zostaną przechwycone i sprawdzone pod kątem istnienia i poprawności tokena CSRF. Jeśli żądanie nie zawiera takiego tokena lub nie odpowiada on tokenowi znajdującemu się na serwerze, żądanie nie powiedzie się i spowoduje wystąpienie wyjątku `CsrfException`.

Oznacza to, że wszystkie formularze dostępne w aplikacji muszą przesyłać token w polu `_csrf`. Token ten musi mieć taką samą wartość jak ten policzony i przechowywany przez serwer, aby wartości te zgadzały się po wysłaniu formularza.

Na szczęście Spring Security ułatwia to zadanie, przechowując token w atrybutach żądania. Jeżeli na stronie wykorzystujemy szablony Thymeleaf, ukryte pole `_csrf` zostanie dodane automatycznie. Musimy jedynie atrybut `action` w znaczniku `<form>` poprzedzić prefiksem pochodzącym z przestrzeni nazw Thymeleaf:

```
<form method="POST" th:action="@{/spittles}">
   ...
</form>
```

Jeśli stosujemy system szablonów JSP, podobny efekt możemy osiągnąć za pomocą poniższego kodu:

```
<input type="hidden"
       name="${_csrf.parameterName}"
       value="${_csrf.token}" />
```

Możemy to wykonać jeszcze prościej, przy użyciu biblioteki znaczników dla formularzy Springa i znacznika `<sf:form>`. Wtedy ukryte pole dla tokena CSRF zostanie dodane automatycznie.

Innym sposobem radzenia sobie z tokenami CSRF jest ich całkowite wyłączenie. Możemy to zrobić poprzez wywołanie metody `csrf().disable()` w konfiguracji Spring Security, jak pokazano na listingu 9.6.

Listing 9.6. Możemy wyłączyć zabezpieczenia Spring Security przed atakami CSRF

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      ...
      .csrf()
         .disable(); <-- Wyłączamy zabezpieczenia CSRF
}
```

Pamiętaj jednak, że z reguły nie jest to najlepsze rozwiązanie. Aplikacja pozostanie otwarta na ataki CSRF. Z opcji wyłączania zabezpieczeń CSRF należy więc korzystać po głębokim namyśle.

Teraz gdy ustawiliśmy już bazę użytkowników oraz skonfigurowaliśmy Spring Security do przechwytywania żądań, możemy rozpocząć odpytywanie użytkownika o jego dane logowania.

9.4. Uwierzytelnianie użytkowników

Po skorzystaniu z bardzo prostej konfiguracji Spring Security, przedstawionej na listingu 9.1, otrzymaliśmy „za darmo" domyślną stronę logowania. Do momentu nadpisania metody `configure(HttpSecurity)` mieliśmy dostęp do prostej, ale w pełni funkcjonalnej strony logowania. Po nadpisaniu metody `configure(HttpSecurity)` utraciliśmy możliwość jej wykorzystania.

Na szczęście łatwo możemy to naprawić. Musimy tylko w metodzie `configure(Http` ↪`Security)` wywołać metodę `formLogin()` w sposób pokazany na listingu 9.7.

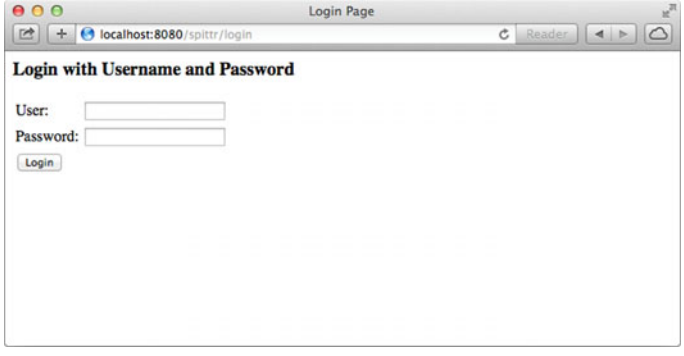
Listing 9.7. Metoda formLogin() udostępnia prostą stronę logowania

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
     .formLogin() <-- Włączamy domyślną stronę logowania
     .and()
     .authorizeRequests()
        .antMatchers("/spitter/me").hasRole("SPITTER")
        .antMatchers(HttpMethod.POST, "/spittles").hasRole("SPITTER")
```

```
            .anyRequest().permitAll();
        .and()
        .requiresChannel()
            .antMatchers("/spitter/form").requiresSecure();
}
```

Zauważ, że podobnie jak wcześniej, do łączenia ze sobą różnych instrukcji konfiguracji zastosowaliśmy metodę `and()`.

Jeśli użytkownik kliknie odnośnik do strony `/login` lub przejdzie na stronę wymagającą uwierzytelnienia, w oknie przeglądarki pojawi się strona logowania. Możemy ją zobaczyć na rysunku 9.2. Strona nie jest zbyt atrakcyjna wizualnie, ale dobrze wykonuje swoją pracę.



Rysunek 9.2. Domyślny ekran logowania jest prosty pod względem wizualnym, ale w pełni funkcjonalny

Założę się jednak, że wolisz, aby ekran logowania aplikacji wyglądał lepiej od tego domyślnego. Szkoda by było, żeby ten prosty ekran logowania psuł wrażenia estetyczne na Twojej dopracowanej wizualnie stronie. Ale nie ma żadnego problemu — w następnej sekcji dowiesz się, jak dodać własną stronę logowania do aplikacji.

9.4.1. Dodajemy własną stronę logowania

Pierwszym krokiem do utworzenia własnej strony logowania jest zorientowanie się, co jest potrzebne do jej działania. Wystarczy, że spojrzymy na źródło HTML domyślnej strony logowania:

```
<html>
<head><title>Login Page</title></head>
<body onload='document.f.username.focus();'>
<h3>Login with Username and Password</h3>
<form name='f' action='/spittr/login' method='POST'>
   <table>
      <tr><td>User:</td><td>
         <input type='text' name='username' value=''></td></tr>
      <tr><td>Password:</td>
         <td><input type='password' name='password'/></td></tr>
      <tr><td colspan='2'>
         <input name="submit" type="submit" value="Login"/></td></tr>
      <input name="_csrf" type="hidden" value="6829b1ae-0a14-4920-aac4-5abbd7eeb9ee" />
   </table>
```

```
</form>
</body>
</html>
```

Warto zwrócić uwagę na adres, na który przesyłany jest formularz, oraz na pola zawierające nazwę i hasło użytkownika. Tworzona przez nas strona też musi zawierać takie pola. Na koniec załóżmy, że nie wyłączyliśmy obsługi tokenów CSRF, musimy więc dodać ukryte pole `_csrf` zawierające wartość tego tokena.

Na listingu 9.8 znajduje się szablon Thymeleaf, przygotowany specjalnie na potrzeby aplikacji Spittr.

Listing 9.8. Własna strona logowania dla aplikacji Spittr (w postaci szablonu Thymeleaf)

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
   <title>Spitter</title>
   <link rel="stylesheet"
         type="text/css"
         th:href="@{/resources/style.css}"></link>
   </head>
   <body onload='document.f.username.focus();'>
      <div id="header" th:include="page :: header"></div>
   <div id="content">
   <form name='f' th:action='@{/login}' method='POST'> <-------- Wysyłamy dane na /login
      <table>
      <tr><td>User:</td><td>
         <input type='text' name='username' value='' /></td></tr>
      <tr><td>Password:</td>
         <td><input type='password' name='password'/></td></tr>
      <tr><td colspan='2'>
         <input name="submit" type="submit" value="Zaloguj"/></td></tr>
      </table>
   </form>
   </div>
   <div id="footer" th:include="page :: copy"></div>
   </body>
</html>
```

Szablon Thymeleaf zawiera pola `username` oraz `password`, tak samo jak domyślna strona logowania. Dane logowania przesyłane są również na stronę `/login`, uzależnioną od kontekstu aplikacji. Ponieważ korzystamy z szablonu Thymeleaf, ukryte pole `_csrf` jest automatycznie dołączane do formularza.

9.4.2. Włączamy uwierzytelnianie HTTP Basic

Uwierzytelnianie za pomocą formularza jest świetnym rozwiązaniem, gdy użytkownikami naszej aplikacji są ludzie. W rozdziale 16. dowiesz się, jak przekształcić naszą aplikację internetową w REST-owe API. Kiedy użytkownikiem aplikacji jest inna aplikacja, prośba o zalogowanie za pomocą formularza po prostu nie zadziała.

Jednym ze sposobów uwierzytelniania użytkowników bezpośrednio za pomocą żądania HTTP jest uwierzytelnianie HTTP Basic. Być może spotkałeś się już z tego

typu uwierzytelnianiem wcześniej. Gdy w oknie przeglądarki wejdziemy na tak zabezpieczoną stronę, pojawi się proste okno dialogowe z prośbą o podanie nazwy użytkownika i hasła.

Takie zachowanie jest tylko sposobem obsługi tego żądania przez przeglądarkę. W rzeczywistości otrzymujemy odpowiedź HTTP 401, co sygnalizuje, że wraz z żądaniem pojawia się wymaganie podania nazwy użytkownika i hasła. W związku z tym ten rodzaj zabezpieczenia jest odpowiedni dla klientów REST-owych, którzy mogą się w ten sposób uwierzytelnić w serwisie.

Włączenie uwierzytelniania HTTP Basic jest bardzo proste. Wystarczy wywołać metodę `httpBasic()` na obiekcie `HttpSecurity` przekazanym do metody `configure()`. Mamy też możliwość określenia domeny (ang. *realm*), na co pozwala nam metoda `realm➥Name()`. Poniżej znajduje się prosty przykład konfiguracji Spring Security włączający uwierzytelnianie HTTP Basic:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .formLogin()
         .loginPage("/login")
      .and()
      .httpBasic()
         .realmName("Spittr")
      .and()
   ...
}
```

Ponownie wykorzystujemy metodę `and()` do łączenia różnych dyrektyw konfiguracji w metodzie `configure()`.

Nie ma zbyt wielu opcji konfiguracji uwierzytelniania HTTP Basic, a nawet nie są one potrzebne. Uwierzytelnianie HTTP Basic może być po prostu włączone lub wyłączone. Nie będziemy więc drążyć tego tematu, a przejdziemy do automatycznego uwierzytelniania użytkowników za pomocą funkcji „pamiętaj mnie” (ang. *remember-me*).

9.4.3. Włączenie funkcji „pamiętaj mnie”

Z punktu widzenia aplikacji uwierzytelnienie użytkowników jest ważne. Sami użytkownicy woleliby jednak ominąć konieczność logowania przy każdej wizycie w aplikacji. Dlatego właśnie wiele stron internetowych oferuje funkcję „pamiętaj mnie”, dzięki której użytkownik loguje się tylko odwiedzając aplikację po raz pierwszy.

Spring Security pozwala na łatwe dodanie funkcji „pamiętaj mnie” do aplikacji. Aby ją włączyć, musimy jedynie dodać wywołanie metody `rememberMe()` na obiekcie `Http➥Security` przekazanym do metody `configure()`:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .formLogin()
         .loginPage("/login")
      .and()
      .rememberMe()
         .tokenValiditySeconds(2419200)
```

```
        .key("spittrKey")
   ...
}
```

Włączyliśmy tu funkcję „pamiętaj mnie" i dokonaliśmy drobnej konfiguracji. W ustawieniu domyślnym użytkownik zostanie zapamiętany za pomocą klucza przechowywanego w ciasteczku ważnym przez dwa tygodnie. My jednak ustaliliśmy długość ważności klucza na cztery tygodnie (2 419 200 sekund).

Klucz przechowywany w ciasteczku składa się z nazwy użytkownika, hasła, daty wygaśnięcia i klucza prywatnego. Przed zapisem do ciasteczka całość kodowana jest za pomocą algorytmu MD5. Klucz prywatny ma domyślnie wartość `SpringSecured`, ale na potrzeby aplikacji Spitter zmieniliśmy ją na `spitterKey`.

To dosyć proste. Gdy funkcja „pamiętaj mnie" jest już włączona, musimy umożliwić użytkownikom wskazanie, że chcą być zapamiętani przez aplikację. Potrzebujemy do tego parametru `remember-me` w żądaniu logowania. Proste pole wyboru w formularzu logowania powinno wystarczyć:

```
<input id="remember_me" name="remember-me" type="checkbox"/>
<label for="remember_me" class="inline">Pamiętaj mnie</label>
```

Równie istotna jak możliwość logowania do aplikacji jest możliwość wylogowania. Jest ona szczególnie ważna, gdy zdecydujemy się skorzystać z opcji „pamiętaj mnie". W przeciwnym wypadku bylibyśmy już na zawsze zalogowani do aplikacji. Zobaczmy więc, jak wzbogacić naszą aplikację o możliwość wylogowania.

9.4.4. Wylogowujemy się

Okazuje się, że możliwość wylogowania jest już włączona w konfiguracji bez potrzeby jakiejkolwiek interwencji z naszej strony. Musimy tylko dodać łącze, które pozwala z tej opcji skorzystać.

Wylogowanie zaimplementowane jest jako filtr serwletu, który (domyślnie) przechwytuje żądania przychodzące na adres `/logout`. W ten sposób dodanie możliwości wylogowania sprowadza się do dodania następującego łącza (przykład dla szablonu Thymeleaf):

```
<a th:href="@{/logout}">Wyloguj się</a>
```

Po kliknięciu odnośnika żądanie do adresu `/logout` zostanie przechwycone przez filtr Spring Security `LogoutFilter`. Użytkownik zostanie wylogowany, a klucze typu „pamiętaj mnie" zostaną usunięte. Po wylogowaniu przeglądarka przekieruje użytkownika na stronę `/login?logout`, aby dać mu możliwość ponownego zalogowania.

Możemy też przekierować użytkownika na jakąś inną stronę. Przykładowo niech to będzie strona domowa. Konfigurację strony logowania możemy wykonać za pomocą metody `configure()` w podany niżej sposób:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
      .formLogin()
         .loginPage("/login")
      .and()
```

```
        .logout()
          .logoutSuccessUrl("/")
    ...
}
```

Tak jak poprzednio, korzystamy z metody `and()`, by dołączyć wywołanie metody `logout()`. Metoda `logout()` udostępnia metody służące do konfiguracji zachowania w trakcie wylogowywania. W podanym przykładzie wywoływana metoda `logoutSuccessUrl()` wskazuje, że po pomyślnym wylogowaniu przeglądarka powinna nas przekierowywać na adres `/`.

Możemy też nadpisać domyślną wartość ścieżki, dla której filtr `LogoutFilter` przechwytuje żądania. Służy do tego metoda `logoutUrl()`:

```
.logout()
   .logoutSuccessUrl("/")
   .logoutUrl("/signout")
```

Dotychczas zajmowaliśmy się bezpieczeństwem aplikacji sieciowych tylko podczas napływających żądań. Założeniem było uniemożliwienie użytkownikowi dostępu do adresów URL, których nie powinien używać. Dobrą praktyką jest również niepokazywanie użytkownikowi odnośników do tych adresów. Zobaczmy, co Spring Security oferuje w zakresie bezpieczeństwa, jeśli chodzi o widoki.

9.5. Zabezpieczanie elementów na poziomie widoku

Gdy generujemy stronę HTML w przeglądarce, chcemy mieć możliwość odzwierciedlenia ograniczeń i informacji związanych z zabezpieczeniami aplikacji. Prostym przykładem jest generowanie nazwy uwierzytelnionego użytkownika (na przykład „Jesteś zalogowany jako…"). Możemy też warunkowo generować pewne fragmenty widoku w zależności od uprawnień posiadanych przez użytkownika.

W rozdziale 6. poznałeś dwie opcje generowania widoków w aplikacji Spring MVC: szablony JSP oraz Thymeleaf. Niezależnie od dokonanego wyboru otrzymujemy narzędzia do pracy z mechanizmami zabezpieczeń w widoku. Spring Security dostarcza bibliotekę znaczników dla szablonów JSP, a Thymeleaf oferuje integrację ze Spring Security za pośrednictwem specjalnego języka.

Zobaczmy więc, jak pracować ze Spring Security w widokach, a rozpoczniemy od biblioteki znaczników JSP.

9.5.1. Korzystamy z biblioteki znaczników JSP w Spring Security

Spring Security posiada niewielką bibliotekę znaczników JSP. Zawiera ona tylko trzy znaczniki, które opisano w tabeli 9.6.

Aby skorzystać z biblioteki znaczników JSP, musimy ją zadeklarować w plikach JSP, w których będzie używana:

```
<%@ taglib prefix="security" uri="http://www.springframework.org/security/tags" %>
```

Po zadeklarowaniu biblioteka jest już gotowa do użycia. Pokażemy teraz działanie trzech dostarczanych przez Spring Security znaczników JSP.

Tabela 9.3. Spring Security zapewnia bezpieczeństwo na poziomie widoku za pomocą biblioteki znaczników JSP

Znacznik JSP	Co robi
`<security:accesscontrollist>`	Pozwala na wyświetlenie zawartości znacznika, o ile bieżący uwierzytelniony użytkownik posiada jedno z wymienionych praw na liście dostępu.
`<security:authentication>`	Zapewnia dostęp do właściwości obiektu uwierzytelnienia bieżącego użytkownika.
`<security:authorize>`	Pozwala na wyświetlenie zawartości znacznika, o ile użytkownik posiada wskazane uprawnienia lub spełniony jest określony warunek bezpieczeństwa.

DOSTĘP DO INFORMACJI UWIERZYTELNIAJĄCYCH

Jedną z najprostszych rzeczy, które możemy osiągnąć dzięki dostarczanej przez Spring Security bibliotece znaczników JSP, jest wygodny dostęp do informacji uwierzytelniających użytkownika. Na przykład umieszczanie wiadomości powitalnej, identyfikującej użytkownika po jego nazwie, w nagłówku strony internetowej jest często stosowaną praktyką. Właśnie do tego służy element `<security:authentication>`. Przykładowo:

```
Witaj <security:authentication property="principal.username" />!
```

Atrybut `property` identyfikuje właściwość obiektu uwierzytelniającego użytkownika. Dostępność właściwości może się różnić, w zależności od sposobu uwierzytelnienia. Niektóre z nich są jednak dostępne w każdej sytuacji. Wymieniono je w tabeli 9.7.

Tabela 9.7. Poszczególne informacje uwierzytelniające użytkownika można uzyskać dzięki znacznikowi JSP <security:authentication>

Właściwość uwierzytelniania	Opis
`authorities`	Kolekcja obiektów typu `GrantedAuthority`, reprezentujących uprawnienia przydzielone użytkownikowi.
`credentials`	Informacje uwierzytelniające użyte do weryfikacji obiektu `principal` (najczęściej hasło użytkownika).
`details`	Dodatkowe informacje uwierzytelniające (adres IP, certyfikat i numer seryjny, identyfikator sesji i tak dalej).
`principal`	Obiekt `principal` użytkownika.

W naszym przykładzie generowaną właściwością jest zagnieżdżona własność `username` właściwości `principal`.

Powyższe użycie `<security:authentication>` spowoduje wyświetlenie wartości właściwości w widoku. Jeżeli jednak wolisz przypisać ją do zmiennej, wystarczy, że określisz nazwę zmiennej w atrybucie `var`. Poniższy fragment kodu przypisuje ją do właściwości loginId:

```
<security:authentication property="principal.username" var="loginId"/>
```

Tworzona zmienna ma domyślnie zasięg strony. Aby nadać jej inny zasięg, na przykład żądania lub sesji (lub jakikolwiek zasięg zdefiniowany w `javax.servlet.jsp.PageContext`), skorzystaj z atrybutu `scope`. Poniżej pokazano przykład użycia `<security:authentication>` do nadania zmiennej zasięgu żądania.

```
<security:authentication property="principal.username" var="loginId" scope="request" />
```

Znacznik `<security:authentication>` często się przydaje, ale jest on tylko namiastką możliwości biblioteki znaczników JSP Spring Security. Pokażemy teraz, jak wyświetlać warunkowo zawartość, w zależności od uprawnień użytkownika.

WYŚWIETLANIE Z UPRAWNIENIAMI

Zdarza się, że pewne partie widoku powinny zostać wyświetlone lub nie, w zależności od praw danego użytkownika. Wyświetlanie formularza logowania zalogowanemu użytkownikowi nie ma sensu, podobnie jak wyświetlanie spersonalizowanego powitania anonimowej osobie.

Znacznik JSP `<security:authorize>` Spring Security warunkowo wyświetla dany fragment widoku, w zależności od udzielonych użytkownikowi uprawnień. W aplikacji Spittr na przykład nie chcemy pokazywać formularza umożliwiającego dodanie nowego spittle'a, jeżeli użytkownik nie posiada uprawnienia `ROLE_SPITTER`. Listing 9.9 pokazuje użycie znacznika `<security:authorize>` do wyświetlenia formularza dodawania spittle'a, jeśli użytkownik posiada uprawnienie `ROLE_SPITTER`.

Listing 9.9. Warunkowe wyświetlanie za pomocą znacznika <security:authorize> i języka SpEL

```
<sec:authorize access="hasRole('ROLE_SPITTER')"> ←—— Tylko z uprawnieniem OLE_SPITTER

   <s:url value="/spittles" var="spittle_url" />
      <sf:form modelAttribute="spittle" action="${spittle_url}">
      <sf:label path="text"><s:message code="label.spittle"
        text="Napisz spittle:"/></sf:label>
      <sf:textarea path="text" rows="2" cols="40" />
      <sf:errors path="text" />
      <br/>
      <div class="spitItSubmitIt">
         <input type="submit" value="Wyślij!" class="status-btn round-btn disabled" />
      </div>
   </sf:form>
</sec:authorize>
```

W atrybucie `access` podano wyrażenie SpEL, którego wynik decyduje o tym, czy zawartość `<security:authorize>` zostanie wyświetlona. W tym przypadku użyte zostało wyrażenie `hasRole('ROLE_SPITTER')`, które sprawdza, czy użytkownik posiada rolę `ROLE_SPITTER`. Przy ustawianiu wartości atrybutu `access` do dyspozycji mamy jednak cały język SpEL, w tym również wyrażenia dostarczane przez Spring Security z tabeli 9 5.

Wyrażenia te dają nam dużo większe pole manewru przy tworzeniu ograniczeń bezpieczeństwa. Wyobraźmy sobie na przykład, że aplikacja ma pewne funkcje administracyjne, dostępne tylko dla użytkownika o nazwie „habuma". Możemy wtedy użyć wyrażeń `isAuthenticated()` i `principal` w następujący sposób:

```
<security:authorize access="isAuthenticated() and principal.username=='habuma'">
   <a href="/admin">Administracja</a>
</security:authorize>
```

Jestem pewien, że wymyślisz jeszcze ciekawsze wyrażenia. Bogactwo możliwości języka SpEL sprawia, że ogranicza Cię praktycznie tylko Twoja wyobraźnia.

Jest jednak jeden problem z powyższym przykładem. Wydawać by się mogło, że ograniczenie funkcji administracyjnych do użytkownika „habuma" za pomocą języka SpEL nie jest najlepszym pomysłem. Owszem, odpowiedni odnośnik nie zostanie wyświetlony w widoku. Ale nic nie stoi na przeszkodzie, aby dowolny użytkownik wprowadził URL ręcznie, wpisując `/admin` w pasku adresu przeglądarki.

Bazując na wiedzy zdobytej w tym rozdziale, powinniśmy sobie z tym łatwo poradzić. Dodanie nowego wywołania metody `antMatchers()` w konfiguracji bezpieczeństwa pozwoli nałożyć ograniczenia na adres URL `/admin`:

```
.antMatchers("/admin")
   .access("isAuthenticated() and principal.username=='habuma'");
```

Funkcja administratora jest teraz dostatecznie zabezpieczona. Dostęp do adresu URL mają tylko użytkownicy z odpowiednimi uprawnieniami, a odnośniki do tego adresu nie będą wyświetlane użytkownikom nieuprawnionym. Aby to osiągnąć, musieliśmy jednak zadeklarować wyrażenie SpEL w dwóch miejscach — w konfiguracji bezpieczeństwa oraz w atrybucie `access` znacznika `<security:authorize>`. Czy istnieje sposób na wyeliminowanie duplikacji kodu i pokazywanie odnośnika tylko wtedy, gdy warunki są spełnione?

Atrybut `url` znacznika `<security:authorize>` służy właśnie do tego. W przeciwieństwie do atrybutu `access`, gdzie ograniczenie bezpieczeństwa jest deklarowane jawnie, atrybut `url` odwołuje się pośrednio do ograniczeń bezpieczeństwa dla danego wzorca URL. A ponieważ zadeklarowaliśmy już ograniczenia dla adresu `/admin` w konfiguracji Spring Security, możemy użyć atrybutu `url` następująco:

```
<security:authorize url="/admin/**">
   <spring:url value="/admin" var="admin_url" />
   <br/><a href="${admin_url}">Administrator</a>
</security:authorize>
```

Ponieważ dostęp do adresu URL `/admin` jest ograniczony do uwierzytelnionych użytkowników posiadających nazwę użytkownika „habuma", zawartość znacznika `<security:` ↪`authorize>` zostanie wyświetlona tylko w przypadku spełnienia powyższych warunków. Wyrażenie zostało skonfigurowane w jednym miejscu (w konfiguracji bezpieczeństwa), ale wykorzystane w dwóch.

Biblioteka znaczników JSP dla Spring Security jest bardzo przydatna, szczególnie wtedy, gdy chcemy wyświetlać elementy widoku wyłącznie tym użytkownikom, którzy mają do nich odpowiednie uprawnienia. Jeśli zdecydujesz się jednak na użycie szablonów Thymeleaf, też nie zostaniesz bez wsparcia. Widzieliśmy już, jak dialekt Springa udostępniany przez Thymeleaf dodaje w sposób automatyczny ukryte pola CSRF do formularzy. Poznajmy teraz wsparcie Thymeleaf dla Spring Security.

9.5.2. Pracujemy z dialektem Spring Security w Thymeleaf

Dialekt Thymeleaf do obsługi zabezpieczeń, podobnie jak biblioteka znaczników JSP dla Spring Security, oferuje warunkowe generowanie treści i możliwość wypisania informacji uwierzytelniających. Tabela 9.8 zawiera listę atrybutów udostępnianą przez dialekt bezpieczeństwa.

Tabela 9.8. Dialekt bezpieczeństwa Thymeleaf udostępnia atrybuty, które odwzorowują większość elementów biblioteki znaczników Spring Security

Atrybut	Do czego służy
`sec:authentication`	Generuje właściwości obiektu uwierzytelniania. Podobny do znacznika JSP Spring Security `<sec:authentication/>`.
`sec:authorize`	Warunkowo generuje zawartość w oparciu o wartość wyrażenia. Podobny do znacznika JSP Spring Security `<sec:authorize/>`.
`sec:authorize-acl`	Warunkowo generuje zawartość w oparciu o wartość wyrażenia. Podobny do znacznika JSP Spring Security `<sec:accesscontrollist/>`.
`sec:authorize-expr`	Alias dla atrybutu `sec:authorize`.
`sec:authorize-url`	Warunkowo generuje zawartość w oparciu o reguły zabezpieczeń powiązane z daną ścieżką URL. Podobny do znacznika JSP Spring Security `<sec:authorize/>` przy wykorzystaniu atrybutu url.

Możliwość wykorzystania tego dialektu bezpieczeństwa uzależniona jest od obecności modułu Thymeleaf Extras Spring Security w ścieżce klas aplikacji. Następnie musimy zarejestrować dialekt `SpringSecurityDialect` za pomocą silnika `SpringTemplateEngine` w konfiguracji. Listing 9.10 pokazuje metodę deklarującą komponent typu `SpringTem`
`↪plateEngine` oraz dialekt `SpringSecurityDialect`.

Listing 9.10. Rejestrujemy dialekt Thymeleaf Spring Security

```
@Bean
public SpringTemplateEngine templateEngine(
      TemplateResolver templateResolver) {
   SpringTemplateEngine templateEngine = new SpringTemplateEngine();
   templateEngine.setTemplateResolver(templateResolver);
   templateEngine.addDialect(new SpringSecurityDialect());
   return templateEngine;
}
```

Rejestrujemy dialekt bezpieczeństwa

Po włączeniu dialektu jesteśmy już niemal gotowi do pracy z jego atrybutami w szablonach Thymeleaf. Na początek deklarujemy przestrzeń nazw bezpieczeństwa w szablonach, w których chcemy z tych atrybutów korzystać:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org"
      xmlns:sec=
         "http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
   ...
</html>
```

W powyższym przykładzie standardowy dialekt Thymeleaf przypisany jest, tak jak poprzednio, do prefiksu `th`, a dialekt bezpieczeństwa do prefiksu `sec`.

Teraz już możemy w dowolny sposób używać atrybutów Thymeleaf. Przypuśćmy, że chcemy przywitać zalogowanego użytkownika tekstem „Witaj". Poniższy fragment kodu przedstawia rozwiązanie tego zadania:

```
<div sec:authorize="isAuthenticated()">
   Witaj <span sec:authentication="name">ktosiu</span>
</div>
```

Atrybut `sec:authorize` przyjmuje jako wartość wyrażenie SpEL. Jeśli wyrażenie zwróci `true`, ciało elementu zostanie wygenerowane. W naszym przypadku wyrażeniem jest metoda `isAuthenticated()`. Ciało znacznika `<div>` zostanie więc wygenerowane jedynie wtedy, gdy użytkownik zostanie uwierzytelniony. W ciele znacznika znajduje się powitanie „Witaj", skierowane do osoby o imieniu zapisanym we właściwości `name` w danych uwierzytelniających.

Jak zapewne pamiętasz, znacznik JSP Spring Security `<sec:authorize>` posiada atrybut `url`. Ciało znacznika generowane jest warunkowo w oparciu o dane uwierzytelniania skojarzone z podaną ścieżką URL. Thymeleaf umożliwia nam osiągnięcie tego samego rezultatu dzięki wykorzystaniu atrybutu `sec:authorize-url`. Poniższy fragment kodu Thymeleaf daje ten sam efekt, który uzyskany został wcześniej dla znacznika JSP `<sec:authorize>` i atrybutu `url`:

```
<span sec:authorize-url="/admin">
   <br/><a th:href="@{/admin}">Administrator</a>
</span>
```

Zakładając, że użytkownik posiada uprawnienia dostępu do zasobu `/admin`, na stronie pojawi się łącze do tego adresu, natomiast w przeciwnym wypadku nie będzie ono widoczne.

9.6. Podsumowanie

Bezpieczeństwo jest kluczowym aspektem wielu aplikacji. Spring Security udostępnia mechanizm zabezpieczania aplikacji, który jest prosty, elastyczny i potężny.

Filtry serwletów umożliwiają Spring Security kontrolowanie dostępu do zasobów internetowych, włączając w to kontrolery Spring MVC. Dzięki modelowi konfiguracji z użyciem plików Javy nie musimy w sposób bezpośredni modyfikować tych filtrów, a zabezpieczenia aplikacji internetowej mogą zostać zapisane zwięźle.

Spring Security oferuje kilka opcji uwierzytelniania użytkowników. Dowiedziałeś się, jak konfigurować uwierzytelnianie za pomocą bazy zapisanej w pamięci, relacyjnej bazy danych oraz usług katalogowych LDAP. Jeśli żadna z tych opcji nie odpowiada Twoim potrzebom w zakresie uwierzytelniania, wiesz już, jak utworzyć i skonfigurować własną usługę informacji uwierzytelniającej.

W kilku ostatnich rozdziałach dowiedziałeś się, jak Spring współpracuje z frontendem aplikacji. W następnej części skierujemy się bardziej w stronę backendu. Naszą przygodę w kolejnym rozdziale rozpoczniemy od spojrzenia na warstwę abstrakcji JDBC w Springu.

Część III

Spring po stronie serwera

Chociaż strony internetowe są jedynym elementem aplikacji, który użytkownicy kiedykolwiek zobaczą, to prawdziwa praca odbywa się po drugiej stronie, czyli tam, gdzie dane są przetwarzane i utrwalane. W części III dowiesz się, jak korzystanie ze Springa może Ci pomóc w pracy po stronie serwera.

Relacyjne bazy danych napędzają aplikacje klasy enterprise już od kilkudziesięciu lat. W rozdziale 10., „Korzystanie z bazy danych z użyciem Springa i JDBC", zobaczysz, jak użyć warstwy abstrakcji JDBC Springa do odpytywania relacyjnej bazy danych w znacznie prostszy sposób niż poprzez natywne JDBC.

Jeśli nie chcesz korzystać z JDBC, bardziej może Ci odpowiadać praca z frameworkami ORM (*Object-relational Mapping*). Z rozdziału 11., „Zapisywanie danych z użyciem mechanizmów ORM", dowiesz się, jak Spring integruje się z frameworkami ORM, takimi jak Hibernate, oraz innymi implementacjami JPA (*Java Persistent API*). Dodatkowo dowiesz się, jak wykorzystać możliwości Spring Data JPA do automatycznego generowania implementacji repozytoriów po uruchomieniu aplikacji.

Relacyjne bazy danych nie zawsze są najlepszym rozwiązaniem. Dlatego w rozdziale 12., „Pracujemy z bazami NoSQL", przyjrzymy się innym projektom Spring Data umożliwiającym utrwalanie danych z użyciem różnych nierelacyjnych baz danych, wliczając w to bazy MongoDB, Neo4j oraz Redis.

W rozdziale 13., „Cachowanie danych”, uzupełnimy naszą wiedzę dotyczącą utrwalania danych o mechanizmy cachowania, które pomagają poprawić wydajność aplikacji dzięki wyeliminowaniu zbędnych zapytań do bazy danych w sytuacji, gdy potrzebne dane są już dostępne.

Bezpieczeństwo jest ważnym aspektem aplikacji zarówno po stronie serwera, jak i przeglądarki. Z rozdziału 14., „Zabezpieczanie metod”, dowiesz się, jak zastosować mechanizmy zabezpieczeń Spring Security po stronie serwera, przechwytując wywołania metod i upewniając się, że osobie je wywołującej zostały przydzielone właściwe uprawnienia.

10 Korzystanie z bazy danych z użyciem Springa i JDBC

W tym rozdziale omówimy:

- Definiowanie obsługi dostępu do danych Springa
- Konfigurację zasobów baz danych
- Pracę z szablonami JDBC Springa

Znasz już kluczowe elementy kontenera Springa, czas zatem zastosować tę wiedzę w praktyce. Doskonale nada się do tego utrwalanie danych, bez którego niewiele aplikacji korporacyjnych potrafi się dziś obejść. Każdy z nas spotkał się kiedyś prawdopodobnie z zagadnieniem dostępu do bazy danych przez aplikację. Znamy również liczne zagrożenia wiążące się z dostępem do danych. Uruchomienie mechanizmu dostępu do danych, otwarcie połączeń, obsługa rozmaitych wyjątków i zamknięcie połączeń to tylko niektóre czynności, podczas których istnieje potencjalne ryzyko usunięcia lub uszkodzenia ważnych dla firmy danych. Błąd przy obsłudze dostępu do danych może okazać się bardzo kosztowny.

Właśnie dlatego, że nie lubimy ponosić niepotrzebnych kosztów, zajmujemy się tu Springiem. Spring oferuje całą rodzinę mechanizmów dostępu do danych, które integrują się z różnymi technologiami dostępu. Bez względu na to, czy utrwalasz dane za pomocą dostępu bezpośredniego JDBC, czy za pomocą mechanizmów odwzorowań obiektowo-relacyjnych (ang. *objects-relational mapping* — ORM), takich jak Hibernate, Spring pozwala Ci zaoszczędzić dużo niskopoziomowej pracy, upraszczając w znacznym

stopniu kod dostępu do danych, dzięki czemu możesz się skoncentrować na zarządzaniu danymi swojej aplikacji.

Podczas tworzenia warstwy trwałości musimy dokonać kilku wyborów. Możemy użyć JDBC, Hibernate, Java Persistence API (JPA) lub dowolnego innego frameworka utrwalania danych. Możemy też zdecydować się na wykorzystanie którejś z popularnych ostatnio baz danych NoSQL (które wolę nazywać bazami bez ustalonego schematu).

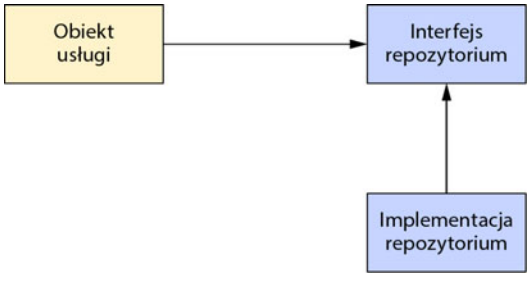
Niezależnie od dokonanego wyboru warto wiedzieć, że najprawdopodobniej istnieje już wsparcie dla tego rozwiązania ze strony Springa. W tym rozdziale skupimy się na wsparciu Springa dla JDBC. Ale na początek, w ramach przygotowania, zapoznajmy się z filozofią trwałości Springa.

10.1. Filozofia dostępu do danych Springa

W poprzednich rozdziałach dowiedziałeś się, że jednym z założeń Springa jest umożliwienie tworzenia aplikacji zgodnie z dobrą praktyką programowania obiektowego — na bazie interfejsów. Moduł dostępu do danych Springa nie jest od tego wyjątkiem.

Tak jak większość aplikacji, aplikacja Spittr musi mieć możliwość odczytu i zapisu danych do jakiejś bazy. Aby uniknąć rozproszenia logiki trwałości po wszystkich komponentach aplikacji, dobrą praktyką jest wydzielenie kodu odpowiedzialnego za dostęp do bazy do jednego lub kilku komponentów przeznaczonych wyłącznie do tego celu. Komponenty te nazywane są często obiektami dostępu do danych (ang. *data access objects* — DAO) albo repozytoriami.

Żeby uniknąć wiązania aplikacji z jakąś konkretną strategią dostępu do danych, prawidłowo zaimplementowane repozytoria powinny udostępniać oferowaną funkcjonalność za pośrednictwem interfejsów. Na rysunku 10.1 pokazano modelowe podejście do projektowania warstwy dostępu do danych.



Rysunek 10.1. Obiekty usług nie zajmują się dostępem do danych. Delegują to zadanie do repozytoriów. Dzięki interfejsowi repozytorium zależność między obiektem usługi a samym repozytorium jest dość luźna

Jak łatwo zauważyć, dostęp obiektów usług do repozytoriów odbywa się poprzez interfejsy. Takie rozwiązanie ma kilka zalet. Po pierwsze, ułatwia testowanie obiektów usług, jako że te ostatnie nie są powiązane z konkretną implementacją dostępu do danych. Co więcej, można nawet stworzyć pozorne implementacje tych interfejsów dostępu do danych. Dzięki temu istnieje możliwość przetestowania obiektów usług bez konieczności łączenia się z bazą danych, co znacząco przyspiesza testy jednostkowe i pozwala wyeliminować ryzyko niepowodzenia testu z powodu niespójności danych.

Ponadto, dostęp do warstwy dostępu do danych odbywa się w sposób niezależny od technologii utrwalania. Wybrane rozwiązanie trwałości jest izolowane na poziomie obiektu DAO i tylko odpowiednie metody dostępu do danych są widoczne poprzez interfejs. Otrzymany w ten sposób projekt aplikacji jest elastyczny i pozwala na wymianę mechanizmu utrwalania bez konsekwencji dla reszty aplikacji. Jeśli szczegóły implementacji warstwy abstrakcji wydostawałyby się do poszczególnych części aplikacji, cała aplikacja byłaby mocno uzależniona od warstwy dostępu do danych. Taki projekt aplikacji nie jest elastyczny.

> **INTERFEJSY A SPRING** Jeżeli po przeczytaniu kilku ostatnich akapitów masz wrażenie, że reprezentuję zwolenników ukrywania warstwy trwałości za interfejsami, masz całkowitą rację. Uważam, że interfejsy są kluczem do tworzenia luźno powiązanych ze sobą porcji kodu i powinny być używane na wszystkich warstwach aplikacji, nie tylko na warstwie dostępu do danych. W tym miejscu muszę zaznaczyć, że choć używanie interfejsów w Springu jest wysoce zalecane, nie ma takiego wymogu — możesz powiązać komponent (obiekt repozytorium lub inny) bezpośrednio z właściwością innego komponentu, bez umieszczania pomiędzy nimi interfejsu.

W izolacji warstwy dostępu do danych od reszty aplikacji bardzo pomocna jest spójna hierarchia wyjątków, używana we wszystkich obsługiwanych mechanizmach trwałości.

10.1.1. Hierarchia wyjątków związanych z dostępem do danych w Springu

Jest pewien stary dowcip o spadochroniarzu, który zniesiony wiatrem z kursu, ląduje na drzewie, zawisając nad ziemią. Po jakimś czasie podchodzi mężczyzna i spadochroniarz pyta, gdzie jest.

Przechodzień odpowiada: „Jest Pan około 8 metrów nad ziemią".

Na co spadochroniarz: „Pan musi być analitykiem oprogramowania".

„To prawda. Skąd Pan wiedział?" — pyta przechodzień.

„Ponieważ to, co mi Pan powiedział, jest w 100 procentach dokładne, ale kompletnie bezwartościowe".

Dowcip był wielokrotnie powtarzany w różnych wersjach, każdorazowo z innym zawodem lub narodowością przechodnia. Ta historia kojarzy mi się z `SQLException` JDBC. Jeśli miałeś kiedykolwiek okazję pisać kod JDBC (bez Springa), prawdopodobnie doskonale wiesz, jak trudno zrobić cokolwiek z JDBC bez przechwytywania wyjątku `SQLE`
`↪xception`. Wyjątek `SQLException` informuje o błędzie przy dostępie do bazy danych. Nie dostarcza jednak zbyt wielu informacji odnośnie samego błędu i tego, co można zrobić, żeby się go pozbyć.

Oto niektóre z częstych przyczyn pojawienia się wyjątku `SQLException`:

- Aplikacja nie jest w stanie połączyć się z bazą danych.
- Są błędy składniowe w zapytaniu do bazy.
- Tabele i (lub) kolumny, do których odnosi się zapytanie, nie istnieją.
- Nastąpiła próba wstawienia bądź edycji danych, która narusza ograniczenia bazy.

Wielkim znakiem zapytania, jeśli chodzi o wyjątek `SQLException`, jest jego obsługa, gdy zostanie już przechwycony. Jak się okazuje, z wieloma problemami, które powodują zgłoszenie `SQLException`, nie jesteśmy sobie w stanie poradzić w ramach bloku `catch`. Większość wyjątków `SQLException` sygnalizuje stan krytyczny. Jeżeli aplikacja nie może się połączyć z bazą danych, jej dalsze działanie, nawet jeśli możliwe, jest z reguły bezcelowe. Podobnie, jeśli w zapytaniu są błędy, niewiele da się z tym zrobić podczas działania aplikacji.

Dlaczego zatem jesteśmy zmuszani do przechwytywania wyjątku `SQLException`, skoro nie możemy podjąć żadnych działań naprawczych?

Nawet jeśli zamierzasz zająć się obsługą wyjątków `SQLException`, musisz przechwycić wyjątek i przeszukać jego właściwości w poszukiwaniu istoty problemu. Przyczyną jest uniwersalny charakter wyjątku `SQLException`, który swoim zakresem obejmuje wszystkie problemy związane z dostępem do danych. Zamiast różnych typów wyjątków dla poszczególnych problemów zgłaszany jest zawsze `SQLException`.

Niektóre frameworki utrwalania danych oferują bogatszą hierarchię wyjątków. Hibernate, na przykład, dostarcza prawie dwa tuziny różnych wyjątków, każdy nakierowany na specyficzny problem przy dostępie do danych. Dzięki temu można utworzyć blok `catch` dla każdego typu wyjątku.

Wyjątki Hibernate są jednak specyficzne dla Hibernate. Zgodnie z wcześniejszą deklaracją, naszym celem jest izolacja mechanizmu utrwalania na poziomie warstwy dostępu do danych. Jeżeli zgłaszane są wyjątki Hibernate, pozostałe części aplikacji „wiedzą", że używane jest Hibernate. Częściowym, aczkolwiek dalekim od optymalnego, rozwiązaniem problemu jest przechwycenie wyjątku platformy utrwalania i ponowne zgłoszenie go w niezależnej od platformy formie.

Z jednej strony, hierarchia wyjątków JDBC jest zbyt uniwersalna — właściwie trudno ją nawet nazwać hierarchią. Z drugiej, hierarchia Hibernate jest specyficzna dla Hibernate. Potrzebujemy hierarchii wyjątków dostępu do danych, które będą wystarczająco opisowe, ale niepowiązane bezpośrednio z jakimkolwiek mechanizmem utrwalania danych.

NIEZALEŻNE OD PLATFORMY UTRWALANIA WYJĄTKI W SPRINGU

JDBC Springa dostarcza hierarchię wyjątków dostępu do danych, która rozwiązuje oba problemy. W przeciwieństwie do JDBC, Spring oferuje kilka rodzajów wyjątków dostępu do danych, każdy opisujący na swój sposób powód zgłoszenia wyjątku. W tabeli 10.1 pokazano wybrane wyjątki dostępu do danych Springa na tle wyjątków dostępnych w JDBC.

Jak widać, wyjątki Springa obejmują swoim zakresem każdą sytuację, która może być źródłem potencjalnych problemów przy zapisie i odczycie z bazy danych. A pełna lista wyjątków dostępu do danych Springa jest jeszcze okazalsza od tej w tabeli 10.1 (zamieściłbym ją całą, ale nie chciałem wpędzać JDBC w kompleks niższości).

Chociaż hierarchia wyjątków Springa jest dużo bogatsza od oferowanego nam przez JDBC prostego `SQLException`, nie jest ona skojarzona z żadnym konkretnym rozwiąza-

niem trwałości. Możemy zatem oczekiwać od Springa stałego zbioru zgłaszanych wyjątków, niezależnie od wyboru dostawcy mechanizmu utrwalania. Wybór ten ma konsekwencje tylko dla warstwy dostępu do danych.

O RETY! GDZIE SIĘ PODZIAŁ BLOK CATCH?

Tabela 10.1 nie pokazuje jednej ważnej rzeczy. Wszystkie zawarte w niej wyjątki są potomkami `DataAccessException`. `DataAccessException` jest wyjątkiem niekontrolowanym.

Tabela 10.1. Hierarchia wyjątków JDBC kontra wyjątki dostępu do danych Springa

Wyjątki JDBC	Wyjątki dostępu do danych Springa

`BatchUpdateException`
`DataTruncation`
`SQLException`
`SQLWarning`
	`BadSqlGrammarException`
	`CannotAcquireLockException`
	`CannotSerializeTransactionException`
	`CannotGetJdbcConnectionException`
	`CleanupFailureDataAccessException`
	`ConcurrencyFailureException`
	`DataAccessException`
	`DataAccessResourceFailureException`
	`DataIntegrityViolationException`
	`DataRetrievalFailureException`
	`DataSourceLookupApiUsageException`
	`DeadlockLoserDataAccessException`
	`DuplicateKeyException`
	`EmptyResultDataAccessException`
	`IncorrectResultSizeDataAccessException`
	`IncorrectUpdateSemanticsDataAccessException`
	`InvalidDataAccessApiUsageException`
	`InvalidDataAccessResourceUsageException`
	`InvalidResultSetAccessException`
	`JdbcUpdateAffectedIncorrectNumberOfRowsException`
	`LobRetrievalFailureException`
	`NonTransientDataAccessResourceException`
	`OptimisticLockingFailureException`
	`PermissionDeniedDataAccessException`
	`PessimisticLockingFailureException`
	`QueryTimeoutException`
	`RecoverableDataAccessException`
	`SQLWarningException`
	`SqlXmlFeatureNotImplementedException`
	`TransientDataAccessException`
	`TransientDataAccessResourceException`
	`TypeMismatchDataAccessException`
	`UncategorizedDataAccessException`
	`UncategorizedSQLException`

Innymi słowy, przechwytywanie jakichkolwiek wyjątków dostępu do danych zgłaszanych w Springu nie jest konieczne (choć nie jest również zabronione, jeśli odczuwasz taką potrzebę).

Wyjątek `DataAccessExeption` jest tylko jednym z przejawów ogólnej filozofii Springa w zakresie kontrolowanych i niekontrolowanych wyjątków. Spring przyjmuje, że duża część wyjątków jest skutkiem problemów, które nie mogą być rozwiązane w bloku `catch`.

Zamiast zmuszać programistów do tworzenia bloków `catch` (które i tak zwykle pozostają puste), Spring promuje stosowanie wyjątków niekontrolowanych. Decyzja o przechwyceniu wyjątku pozostaje w gestii programisty.

Aby skorzystać z wyjątków dostępu do danych Springa, musimy użyć jednego z obsługiwanych przezeń szablonów dostępu do danych. Spójrzmy, jak bardzo szablony Springa mogą uprościć dostęp do danych.

10.1.2. Szablony dostępu do danych

Zapewne zdarzyło Ci się podróżować samolotem. Jeśli tak, na pewno zgodzisz się, że jednym z najistotniejszych elementów podróżowania jest przemieszczenie bagażu z punktu A do punktu B. Proces ten można podzielić na wiele etapów. Po dotarciu na terminal najpierw musisz przejść odprawę bagażową. Następnie, w celu zapewnienia bezpieczeństwa lotu, bagaż zostaje przeskanowany przez pracowników ochrony. W kolejnym kroku jest umieszczany na taśmociągu bagażowym, którym trafia do samolotu. Jeżeli lecisz z przesiadką, Twój bagaż także musi się przesiąść. Po dotarciu do celu bagaż musi zostać wyładowany z samolotu i za pomocą taśmociągu przemieszczony do miejsca, w którym go odbierzesz.

Mimo że cały proces składa się z wielu etapów, aktywnie uczestniczysz tylko w kilku z nich. Za sprawne przeprowadzenie całej procedury odpowiedzialny jest przewoźnik. Twój udział ogranicza się do koniecznego minimum, resztą zajmują się odpowiedni ludzie. Z analogiczną sytuacją mamy do czynienia w przypadku potężnego wzorca projektowego, jakim jest wzorzec metody szablonowej (ang. *template method*).

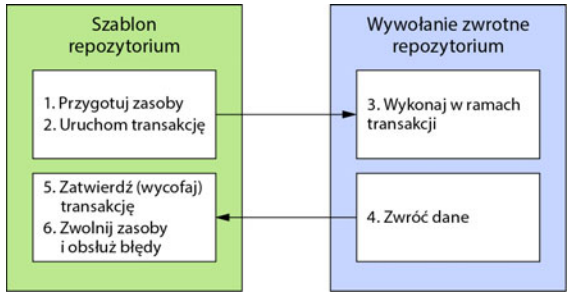
Metoda szablonowa określa szkielet procesu. W naszym przykładzie procesem jest przemieszczenie bagażu z miasta odlotu do miasta przylotu. Proces ten jest stały, nigdy się nie zmienia. Cała sekwencja zdarzeń jest za każdym razem identyczna: odprawa bagażowa, załadunek bagażu do samolotu i tak dalej. Niektóre etapy procesu są również stałe — ich przebieg jest każdorazowo taki sam. Kiedy samolot ląduje na lotnisku docelowym, walizki są wyładowywane jedna po drugiej na taśmociąg, którym trafiają na miejsce odbioru bagażu.

W pewnych punktach proces deleguje zadania do podklas, które zajmują się szczegółami specyficznymi dla implementacji. Jest to zmienna część procesu. Na przykład transport bagażu zaczyna się odprawą bagażową. Ta część procesu następuje zawsze na początku, jej miejsce w sekwencji procesu jest stałe. Ponieważ odprawa każdego pasażera ma inny przebieg, implementacja tej części procesu zależy od pasażera. Przekładając to na oprogramowanie, metoda szablonowa deleguje specyficzne dla implementacji części procesu interfejsowi. Różne implementacje tego interfejsu określają specyficzne implementacje tej części procesu.

Ten sam wzorzec jest używany przez Springa przy dostępie do danych. Bez względu na to, jakiej technologii używamy, pewne etapy dostępu do danych są zawsze konieczne. Na przykład, zawsze potrzebujemy połączenia z magazynem danych, a po zakończeniu musimy zwolnić zasoby. Zarówno połączenie z magazynem, jak i zwolnienie zasobów są stałymi etapami w procesie dostępu do danych. Natomiast każda stworzona przez

nas metoda dostępu do danych będzie nieco inna. Nasze zapytania dotyczą różnych obiektów, a dane uaktualniamy na różne sposoby. To właśnie są zmienne etapy procesu dostępu do danych.

Spring oddziela stałe i zmienne części procesu dostępu do danych za pomocą dwóch klas: **szablonów** (ang. *templates*) i **wywołań zwrotnych** (ang. *callbacks*). Szablony zajmują się stałą częścią procesu, natomiast wywołania zwrotne odpowiedzialne są za jego zmienną część. Na rysunku 10.2 pokazano zadania obu klas.



Rysunek 10.2. Klasy szablonowe repozytorium Springa wykonują zadania typowe dla dostępu do danych. Do zadań specyficznych dla aplikacji używają zaś wywołań zwrotnych

Patrząc na rysunek 10.2, widzimy, że klasy szablonowe Springa zajmują się stałymi etapami dostępu do danych — kontrolując transakcje, zarządzając zasobami i obsługując wyjątki. Natomiast za szczegółowe zadania związane z dostępem do danych aplikacji — tworzenie zapytań, podpinanie (ang. *binding*) parametrów, przetwarzanie wyników — odpowiada implementacja wywołań zwrotnych. Jest to eleganckie rozwiązanie, dzięki któremu możesz zająć się wyłącznie logiką dostępu do danych.

W Springu możemy wybrać jeden z kilku szablonów, w zależności od używanej przez nas platformy utrwalania. Jeżeli używasz zwykłego JDBC, wybierzesz `JdbcTemplate`. Jeśli natomiast preferujesz jeden z mechanizmów odwzorowań obiektowo-relacyjnych, lepszym wyborem będzie `HibernateTemplate` lub `JpaTemplate`. W tabeli 10.2 znajdziesz listę szablonów dostępu do danych Springa wraz z ich przeznaczeniem.

Tabela 10.2. Spring oferuje szereg szablonów dla poszczególnych mechanizmów utrwalania

Klasa szablonowa (org.springframework.*)	Używana z ...
`jca.cci.core.CciTemplate`	połączeniami JCA CCI
`jdbc.core.JdbcTemplate`	połączeniami JDBC
`jdbc.core.namedparam.NamedParameterJdbcTemplate`	połączeniami JDBC z obsługą parametrów nazwanych
`jdbc.core.simple.SimpleJdbcTemplate`	połączeniami JDBC, uproszczona dzięki konstrukcjom Javy 5 (uznana za przestarzałą w Springu 3.1)
`orm.hibernate3.HibernateTemplate`	sesjami Hibernate 3.x
`orm.ibatis.SqlMapClientTemplate`	klientami SqlMap iBATIS
`orm.jdo.JdoTemplate`	implementacjami Java Data Object
`orm.jpa.JpaTemplate`	menedżerami encji Java Persistence API

Spring obsługuje wiele mechanizmów utrwalania, omówienie ich wszystkich zajęłoby zbyt dużo miejsca. Dlatego skupimy się na, moim zdaniem, tych najbardziej przydatnych i tych najczęściej używanych.

W tym rozdziale zaczniemy od prostego dostępu JDBC, jako że jest to najbardziej podstawowa metoda odczytu i zapisu danych w bazie. W dalszej kolejności, w rozdziale 11., przyjrzymy się Hibernate i JPA, dwóm najpopularniejszym rozwiązaniom ORM opartym na obiektach POJO. Naszą przygodę z warstwą trwałości zakończymy w rozdziale 12., poznając możliwości projektu Spring Data w zakresie wykorzystania baz NoSQL w Springu.

Większość dotyczących utrwalania danych opcji Springa potrzebuje źródła danych. Dlatego na samym początku, przed przystąpieniem do deklaracji szablonów i obiektów DAO, musimy poinformować Springa o źródle danych, dzięki czemu obiekty repozytoriów uzyskają możliwość połączenia z bazą danych.

10.2. Konfiguracja źródła danych

Niezależnie od używanych klas bazowych repozytoriów musisz wskazać źródło danych. Spring oferuje szereg opcji w zakresie konfiguracji komponentów źródeł danych w aplikacji Springa, w tym:

- źródła danych zdefiniowane przez sterownik JDBC,
- źródła danych odszukiwane przez JNDI,
- źródła danych z pulą połączeń.

Jeśli chodzi o aplikacje produkcyjne, zalecam używanie źródła danych korzystającego z puli połączeń. Kiedy to tylko możliwe, staram się uzyskiwać źródło danych z puli serwera aplikacji poprzez JNDI. Mając na uwadze powyższe, zacznijmy od tego, jak skonfigurować Springa, aby uzyskać źródło danych z JNDI.

10.2.1. Źródła danych JNDI

Wdrożone aplikacje Springa często będą przeznaczone do uruchamiania na serwerach aplikacji dla platformy Java EE, takich jak WebSphere, JBoss, czy nawet kontenerach aplikacji sieciowych, jak Tomcat. Serwery te pozwalają na konfigurację źródła danych uzyskiwanego przez JNDI. Niewątpliwą zaletą tego sposobu konfiguracji źródeł danych jest to, że w ten sposób mogą być zarządzane całkowicie z zewnątrz aplikacji, umożliwiając aplikacji zażądanie źródła danych, gdy jest na to gotowa. Poza tym, źródła danych zarządzane w ramach serwera aplikacji są często w puli i mogą być przełączane przez administratorów podczas pracy serwera (w trybie „hot-swap").

Spring daje nam możliwość konfiguracji przechowywanego w JNDI źródła danych i powiązania go z klasami, które go potrzebują, podobnie jak w przypadku komponentów Springa. Element `<jee:jndi-lookup>` z przestrzeni nazw `jee` Springa pozwala na uzyskanie dowolnego obiektu, w tym źródeł danych, z JNDI i udostępnienie go jako komponentu Springa. Na przykład, jeśli źródło danych naszej aplikacji jest skonfigurowane w JNDI, możemy użyć `<jee:jndi-lookup>` do dowiązania go w następujący sposób:

```
<jee:jndi-lookup id="dataSource"
   jndi-name="/jdbc/SpitterDS"
   resource-ref="true" />
```

Atrybut `jndi-name` wskazuje nazwę zasobu w JNDI. Jeśli tylko właściwość `jndi-name` jest określona, źródło danych zostanie odszukane przy użyciu podanej nazwy. Jeśli jednak aplikacja została uruchomiona na serwerze aplikacji Javy, właściwości `resource-ref` należy nadać wartość `true`, co spowoduje dodanie na początku wartości `jndi-name` ciągu `java:comp/env/`.

Jeśli korzystamy z konfiguracji w klasach Javy, źródło danych możemy pobrać z JNDI za pomocą komponentu `JndiObjectFactoryBean`:

```
@Bean
public JndiObjectFactoryBean dataSource() {
   JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();
   jndiObjectFB.setJndiName("jdbc/SpittrDS");
   jndiObjectFB.setResourceRef(true);
   jndiObjectFB.setProxyInterface(javax.sql.DataSource.class);
   return jndiObjectFB;
}
```

Nietrudno zauważyć, że konfiguracja ta jest bardziej skomplikowana. Konfiguracja JavaConfig jest często prostsza od konfiguracji XML, ale w tym konkretnym przypadku musimy się jednak więcej napracować. Mimo to widać, że odzwierciedla swój odpowiednik w pliku XML. I nie jest aż tak dużo bardziej rozwlekła.

10.2.2. Źródła danych z pulą

Jeśli nie jesteśmy w stanie uzyskać źródła danych z JNDI, pozostaje nam konfiguracja źródła danych bezpośrednio w Springu. Chociaż Spring nie oferuje źródła danych z pulą, istnieje szereg zewnętrznych rozwiązań, jak choćby poniższe rozwiązania o otwartym źródle:

- Apache Commons DBCP (*http://jakarta.apache.org/commons/dbcp*);
- c3p0 (*http://sourceforge.net/projects/c3p0/*);
- BoneCP (*http://jolbox.com/*).

Większość z tych rozwiązań można skonfigurować jako źródło danych w Springu w sposób zbliżony do zastosowanego w przypadku źródła `DriverManagerDataSource` lub `SingleConnectionDataSource` Springa (które omówimy w dalszej kolejności). Poniższy przykład ilustruje metodę konfiguracji źródła `BasicDataSource` DBPC:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
   p:driverClassName="org.h2.Driver"
   p:url="jdbc:h2:tcp://localhost/~/spitter" p:username="sa"
   p:password="" p:initialSize="5" p:maxActive="10" />
```

Jeśli preferujemy konfigurację Java, deklaracja komponentu `DataSource` prezentuje się następująco:

```
@Bean
public BasicDataSource dataSource() {
   BasicDataSource ds = new BasicDataSource();
```

```
    ds.setDriverClassName("org.h2.Driver");
    ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
    ds.setUsername("sa");
    ds.setPassword("");
    ds.setInitialSize(5);
    ds.setMaxActive(10);
    return ds;
}
```

Pierwsze cztery właściwości są kluczowe dla konfiguracji komponentu `BasicDataSource`. Właściwość `driverClassName` określa jednoznacznie nazwę klasy sterownika JDBC. Tutaj skonfigurowaliśmy ją jako sterownik JDBC dla bazy danych H2. We właściwości `url` ustawiamy kompletny adres URL sterownika JDBC dla bazy danych. Na końcu używamy właściwości `username` i `password` do uwierzytelnienia przy połączeniu z bazą danych.

Te cztery podstawowe właściwości definiują wartości konfiguracji połączenia dla `BasicDataSource`. Szereg dodatkowych właściwości może zostać użyty do konfiguracji puli źródła danych. W tabeli 10.3 umieszczono listę niektórych najbardziej przydatnych właściwości konfiguracji puli komponentu `BasicDataSource` DBCP.

W naszym przykładzie ustawiliśmy liczbę początkową połączeń w puli na `5`. Jeśli liczba ta okaże się zbyt mała, źródło `BasicDataSource` ma możliwość ich tworzenia aż do maksymalnej liczby 10 połączeń.

Tabela 10.3. Właściwości konfiguracji puli komponentu BasicDataSource

Właściwość konfiguracji puli	Określa
`initialSize`	Liczbę połączeń tworzonych przy starcie puli.
`maxActive`	Maksymalną liczbę jednoczesnych połączeń, które mogą zostać przydzielone z puli. Jeśli jest równa `0`, nie ma limitu.
`maxIdle`	Maksymalną liczbę bezczynnych połączeń w puli bez zwalniania niepotrzebnych. Jeśli jest równa `0`, nie ma limitu.
`maxOpenPreparedStatements`	Maksymalną liczbę gotowych zapytań, które mogą zostać przydzielone z puli zapytań w tym samym czasie. Jeśli jest równa `0`, nie ma limitu.
`maxWait`	Długość czasu oczekiwania puli na zwrócenie połączenia do puli (gdy nie ma dostępnych połączeń), zanim zgłoszony zostanie wyjątek. Jeśli jest równa `–1`, pula czeka w nieskończoność.
`minEvictableIdleTimeMillis`	Długość czasu bezczynności połączenia w puli, po upływie którego może zostać usunięte.
`minIdle`	Minimalną liczbę połączeń, które mogą pozostawać bezczynne w puli bez tworzenia nowych połączeń.
`poolPreparedStatements`	Czy buforować gotowe zapytania (wartość logiczna).

10.2.3. Źródła danych oparte na sterowniku JDBC

Najprostsze do konfiguracji źródło danych w Springu to źródło zdefiniowane za pomocą sterownika JDBC. Spring oferuje trzy klasy źródeł danych do wyboru (wszystkie z pakietu `org.springframework.jdbc.datasource`):

- `DriverManagerDataSource` — zwraca nowe połączenie za każdym razem, gdy połączenie jest wymagane. W przeciwieństwie do klasy DBCP `BasicDataSurce`, `DriverManagerDataSource` nie oferuje puli połączeń.
- `SimpleDriverDataSource` — działa niemal dokładnie tak samo jak `DriverManagerDataSource`, ale korzysta bezpośrednio ze sterownika JDBC w celu uniknięcia problemów z wczytywaniem klas, które mogą wystąpić w niektórych środowiskach, jak na przykład w kontenerze OSGi.
- `SingleConnectionDataSource` — zwraca to samo połączenie za każdym razem, gdy połączenie jest wymagane. Choć klasa `SingleConnectionDataSource` nie jest źródłem danych z pulą sensu stricto, można ją traktować jako źródło danych z pulą składającą się z dokładnie jednego połączenia.

Konfiguracja tych źródeł danych jest podobna do konfiguracji `BasicDataSource` DBCP. Przykładowo konfiguracja komponentu `DriverManagerDataSource` może wyglądać następująco:

```
@Bean
public DataSource dataSource() {
   DriverManagerDataSource ds = new DriverManagerDataSource();
   ds.setDriverClassName("org.h2.Driver");
   ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
   ds.setUsername("sa");
   ds.setPassword("");
   return ds;
}
```

Analogiczna konfiguracja w pliku XML wyglądałaby tak:

```
<bean id="dataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource"
   p:driverClassName="org.h2.Driver"
   p:url="jdbc:h2:tcp://localhost/~/spitter"
   p:username="sa" p:password="" />
```

Jedyną wyraźną różnicą pomiędzy komponentami źródeł danych niewykorzystującymi pul połączeń a komponentami je wykorzystującymi jest brak właściwości konfiguracji puli.

Chociaż źródła danych sprawdzają się świetnie przy małych aplikacjach i podczas tworzenia oprogramowania, należałoby się poważnie zastanowić nad konsekwencjami zastosowania któregoś z nich w środowisku produkcyjnym. Ponieważ `SingleConnection` ↪`DataSource` operuje tylko i wyłącznie na jednym połączeniu z bazą danych, nie spisuje się najlepiej w aplikacjach wielowątkowych, a dużo lepiej wypada w testach. Z drugiej strony, o ile `DriverManagerDataSource` i `SimpleDriverDataSource` radzą sobie z obsługą wielu wątków, dzieje się to przy obniżonej wydajności, co jest spowodowane tworzeniem nowego połączenia za każdym razem, gdy połączenie jest wymagane. Z racji tych ograniczeń zalecam używanie źródeł danych z pulą.

10.2.4. Korzystamy z wbudowanego źródła danych

Istnieje jeszcze jedno źródło danych, o którym chcę Ci opowiedzieć: wbudowane źródło danych. Źródło to działa jako element aplikacji, a nie osobny serwer bazy danych, do którego aplikacja się łączy. Takie rozwiązanie nie jest zbyt użyteczne w środowisku produkcyjnym, świetnie się jednak nadaje do celów deweloperskich i testowych. Zawdzięczamy to w dużej mierze możliwości wypełnienia bazy danymi testowymi, usuwanymi przy każdym restarcie aplikacji.

Przestrzeń nazw `jdbc` Springa znacznie upraszcza konfigurację wbudowanej bazy danych. Przykładowo listing 10.1 pokazuje konfigurację przestrzeni nazw `jdbc` na potrzeby wbudowanej bazy H2, wypełnionej wstępnie danymi testowymi.

Listing 10.1. Konfigurujemy wbudowaną bazę danych za pomocą przestrzeni nazw jdbc

```
<?xml version="1.0" encoding="UTF-8"?>
   <beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jdbc="http://www.springframework.org/schema/jdbc"
      xmlns:c="http://www.springframework.org/schema/c"
      xsi:schemaLocation="http://www.springframework.org/schema/jdbc
         http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">
   ...
   <jdbc:embeddeddatabase id="dataSource" type="H2">
      <jdbc:script location="com/habum a/spitter/db/jdbc/schema.sql"/>
      <jdbc:script location="com/habuma/sp itter/db/jdbc/test-data.sql"/>
   </jdbc:embedded-database>
   ...
</beans>
```

Właściwość `type` elementu `<jdbc:embedded-database>` ustawiona jest na `H2`, co wskazuje, że chcemy skorzystać z wbudowanej bazy danych H2. (Pamiętaj, że baza H2 musi się znajdować w ścieżce klas aplikacji). Możemy też użyć bazy Apache Derby, ustawiając wartość właściwości `type` na `DERBY`.

Element `<jdbc:embedded-database>` umożliwia konfigurację bazy danych z wykorzystaniem zera lub większej liczby elementów `<jdbc:script>`. Listing 10.1 zawiera dwa elementy `<jdbc:script>`: pierwszy z nich odwołuje się do pliku *schema.sql*, zawierającego polecenia SQL służące do tworzenia tabel w bazie danych; drugi odnosi się do pliku *test-data.sql*, pozwalającego na wypełnienie bazy danymi testowymi.

Poza możliwością konfiguracji wbudowanej bazy danych element `<jdbc:embedded-database>` udostępnia również źródło danych, które można wykorzystać tak jak każdy inny poznany przez nas do tej pory komponent źródła danych. Atrybut `id` ma wartość `dataSource` i służy jako identyfikator wystawionego komponentu źródła danych. Dzięki temu za każdym razem, gdy jest nam potrzebny obiekt typu `javax.sql.DataSource`, możemy wstrzyknąć komponent `dataSource`.

Jeśli używamy konfiguracji w klasach Javy, nie mamy możliwości skorzystania z dobrodziejstwa przestrzeni nazw `jdbc`. Do utworzenia źródła `DataSource` możemy jednak zastosować klasę `EmbeddedDatabaseBuilder`:

```
@Bean public DataSource dataSource() {
   return new EmbeddedDatabaseBuilder()
      .setType(EmbeddedDatabaseType.H2)
      .addScript("classpath:schema.sql")
      .addScript("classpath:test-data.sql")
      .build();
}
```

Łatwo zauważyć, że metoda `setType()` jest odpowiednikiem atrybutu `type` elementu `<jdbc:embeddeddatabase>`. W pliku XML polecenia SQL inicjujące dane wskazywaliśmy za pomocą elementów `<jdbc:script>`. W klasie Javy służy do tego metoda `addScript()`.

10.2.5. Korzystamy z profili do wyboru źródła danych

Widziałeś już kilka różnych sposobów konfiguracji źródeł danych w Springu i założę się, że znajdziesz wśród nich odpowiednie rozwiązanie dla swojej aplikacji. Co więcej, zapragniesz zapewne skorzystać z różnych źródeł danych w różnych środowiskach.

Na przykład element `<jdbc:embedded-database>` jest świetnym rozwiązaniem na czas tworzenia aplikacji. W środowisku testowym bardziej odpowiednie może być użycie źródła DBCP `BasicDataSource`. W środowisku produkcyjnym najlepszym wyborem może być z kolei zastosowanie elementu `<jee:jndi-lookup>`.

Doskonałym rozwiązaniem tego problemu jest funkcjonalność profili komponentów Springa, które omówiłem w rozdziale 3. Musimy jedynie skonfigurować wszystkie źródła danych w oddzielnych profilach, jak pokazano na listingu 10.2.

Listing 10.2. Profile Springa umożliwiają wybór źródła danych w trakcie działania aplikacji

```
package com.habuma.spittr.config;
import org.apache.commons.dbcp.BasicDataSource;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import
org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration
public class DataSourceConfiguration {
   @Profile("development") <-- Źródło danych w środowisku deweloperskim
   @Bean
   public DataSource embeddedDataSource() {
      return new EmbeddedDatabaseBuilder()
         .setType(EmbeddedDatabaseType.H2)
         .addScript("classpath:schema.sql")
         .addScript("classpath:test-data.sql")
         .build();
   }

   @Profile("qa") <-- Źródło danych w środowisku testowym
   @Bean
   public DataSource Data() {
```

```
        BasicDataSource ds = new BasicDataSource();
        ds.setDriverClassName("org.h2.Driver");
        ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
        ds.setUsername("sa");
        ds.setPassword("");
        ds.setInitialSize(5);
        ds.setMaxActive(10);
        return ds;
    }

    @Profile("production") <-- Źródło danych w środowisku produkcyjnym
    @Bean
    public DataSource dataSource() {
        JndiObjectFactoryBean jndiObjectFactoryBean
            = new JndiObjectFactoryBean();
        jndiObjectFactoryBean.setJndiName("jdbc/SpittrDS");
        jndiObjectFactoryBean.setResourceRef(true);
        jndiObjectFactoryBean.setProxyInterface(
            javax.sql.DataSource.class);
        return (DataSource) jndiObjectFactoryBean.getObject();
    }
}
```

Dzięki użyciu profili źródło danych wybierane jest po uruchomieniu aplikacji w oparciu o aktywny profil. Wbudowane źródło danych tworzone jest tylko wtedy, gdy aktywny jest profil `development`. Podobnie źródło DBCP `BasicDataSource` tworzone jest wtedy i tylko wtedy, kiedy aktywny jest profil `qa` (ang. *quality assurance*). Źródło danych pobierane jest z JNDI jedynie wtedy, gdy aktywny jest profil `production`.

Na listingu 10.3 można też zobaczyć, jak przygotować te same ustawienia konfiguracyjne z użyciem pliku XML.

Listing 10.3. Konfiguracja źródeł danych w pliku XML w oparciu o wybrany profil

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <beans profile="development"> <-- Źródło danych w środowisku deweloperskim
<jdbc:embeddeddatabase id="dataSource" type="H2">
            <jdbc:script location="com/hab uma/spitter/db/jdbc/schema.sql"/>
            <jdbc:script location="com/habum a/spitter/db/jdbc/test-data.sql"/>
        </jdbc:embeddeddatabase>
    </beans>
    <beans profile="qa"> <-- Źródło danych w środowisku testowym
        <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
p:driverClassName="org.h2.Driver"
p:url="jdbc:h2:tcp://localhost/~/spitter"
```

```
p:username="sa" p:password="" p:initialSize="5" p:maxActive="10"/>
   </beans>
   <beans profile="production"> <-------- Źródło danych w środowisku produkcyjnym
      <jee:jndi-lookup id="dataSource"
jndi-name="/jdbc/SpitterDS" resource-ref="true" />
   </beans>
</beans>
```

Teraz, gdy już ustanowiliśmy połączenie z bazą danych za pomocą źródła danych, możemy zacząć komunikację z bazą. Jak już wspominałem, Spring ma dla nas szereg opcji w zakresie pracy z relacyjnymi bazami danych, takich jak JDBC, Hibernate i Java Persistence API (JPA). W następnym podrozdziale zobaczymy, jak zbudować warstwę trwałości aplikacji Springa, korzystając z dobrodziejstw oferowanych nam w zakresie obsługi JDBC przez Springa. Jeżeli wolisz rozwiązania oparte na Hibernate i JPA, możesz przejść od razu do następnego rozdziału, opisującego te rozwiązania.

10.3. Używanie JDBC w Springu

Istnieje wiele technologii utrwalania. Hibernate, iBATIS i JPA to tylko niektóre z nich. Pomimo to całkiem duża liczba aplikacji zapisuje obiekty Javy do bazy danych za pomocą starego dobrego JDBC.

I dlaczego nie? JDBC nie wymaga opanowywania języka zapytań innego frameworka. Jest zbudowane ponad SQL-em, który jest językiem dostępu do danych. Używając JDBC, możesz również bardziej precyzyjnie niż w przypadku praktycznie każdej innej technologii dostroić wydajność dostępu do danych. JDBC pozwala wreszcie na korzystanie z wbudowanych funkcji Twojej bazy danych, podczas gdy inne mechanizmy mogą takie działania odradzać lub nawet ich zabraniać.

Co więcej, JDBC umożliwia Ci pracę na znacznie niższym poziomie niż frameworki utrwalania. Oferuje pełną kontrolę nad sposobem zapisu i odczytu danych, pozwalając na dostęp do poszczególnych kolumn w bazie i na operacje na nich. Takie precyzyjne podejście do kwestii dostępu do danych jest wygodne w przypadku niektórych aplikacji (na przykład aplikacji generujących raporty), gdzie organizacja danych w obiekty tylko po to, żeby przekształcić je z powrotem w surowe dane, mija się z celem.

Moc, elastyczność i inne niewątpliwe zalety JDBC to tylko jedna strona medalu. JDBC ma też swoje niewątpliwe wady.

10.3.1. Kod JDBC a obsługa wyjątków

Pomimo że JDBC daje Ci pracujące bezpośrednio z bazą danych API, odpowiedzialność za obsługę wszystkiego, co jest związane z dostępem do bazy, spada na Ciebie. Do tych obowiązków należą, między innymi, zarządzanie zasobami bazy danych oraz obsługa wyjątków.

Jeżeli zdarzyło Ci się używać JDBC do wstawienia danych do bazy, poniższy fragment kodu nie powinien wyglądać zbyt obco (listing 10.4).

Listing 10.4. Wstawienie wiersza do bazy danych za pomocą JDBC

```
private static final String SQL_INSERT_SPITTER =
  "insert into spitter (username, password, fullname) values (?, ?, ?)";
private DataSource dataSource;
public void addSpitter(Spitter spitter) {
   Connectionconn = null;
   PreparedStatement stmt = null;
   try {
      conn = dataSource.getConnection();          <-- Uzyskujemy połączenie

      stmt = conn.prepareStatement(SQL_INSERT_SPITTER);   <-- Tworzymy zapytanie

      stmt.setString(1, spitter.getUsername());   <-- Podpinamy parametry
      stmt.setString(2, spitter.getPassword());
      stmt.setString(3, spitter.getFullName());

      stmt.execute();                             <-- Wykonujemy zapytanie

   } catch (SQLException e) {
      // Zrób coś tutaj... ale co?                <-- Obsługujemy wyjątki (w jakiś sposób)

   } finally {
      try {
         if (stmt != null) {                      <-- Sprzątamy
            stmt.close();
         }
         if (conn != null) {
            conn.close();
         }
      } catch(SQLException e) {
         // A co zrobić tutaj? Mam jeszcze więcej wątpliwości.
      }
   }
}
```

To całkiem pokaźny kawałek kodu! Więcej niż 20 wierszy tylko po to, żeby wstawić obiekt do bazy danych. A o bardziej elementarną operację JDBC już naprawdę trudno. Dlaczego zatem coś tak prostego wymaga aż tyle pisania? W rzeczywistości nie do końca tak jest. Sama operacja wstawienia to zaledwie kilka wierszy. JDBC ma jednak pewne dodatkowe wymagania odnośnie prawidłowej obsługi połączenia i zapytania, ale przede wszystkim obsługi wyjątku `SQLException`, który może zostać zgłoszony.

Zwróć uwagę, że nie tylko nie do końca wiadomo, jak obsłużyć wyjątek `SQLException` (ponieważ nie wiemy, co go spowodowało), ale — co gorsza — jesteśmy też zmuszeni przechwycić go aż dwa razy! Musimy to zrobić zarówno przy wstawianiu rekordu, jak i przy zamykaniu zapytania oraz połączenia. Czy to nie zbyt wiele pracy jak na coś, z czym z reguły i tak trudno jest sobie poradzić programowo?

Spójrz teraz na listing 10 5. Uaktualniamy na nim wiersz tabeli `Spitter` w bazie danych za pomocą tradycyjnego JDBC.

Listing 10.5. Użycie JDBC do uaktualnienia wiersza w bazie danych

```
private static final String SQL_UPDATE_SPITTER =
  "update spitter set username = ?, password = ?, fullname = ? " + "where id = ?";
public void saveSpitter(Spitter spitter) {
   Connection conn = null;
   PreparedStatement stmt = null;
   try {
      conn = dataSource.getConnection();          <-- Uzyskujemy połączenie

      stmt = conn.prepareStatement(SQL_UPDATE_SPITTER);   <-- Tworzymy zapytanie

      stmt.setString(1, spitter.getUsername());   <-- Podpinamy parametry
      stmt.setString(2, spitter.getPassword());
      stmt.setString(3, spitter.getFullName());
      stmt.setLong(4, spitter.getId());

      stmt.execute();                             <-- Wykonujemy zapytanie
   } catch(SQLException e) {
      // Nadal nie jestem przekonany, co mam zrobić tutaj   <-- Obsługujemy wyjątki (w jakiś sposób)
   } finally {
      try {
         if (stmt != null) {                      <-- Sprzątamy
            stmt.close();
         }
         if (conn != null) {
            conn.close();
         }
      } catch(SQLException e) {
         // A tym bardziej tutaj
      }
   }
}
```

Na pierwszy rzut oka listingi 10.5 i 10.4 mogą się wydawać identyczne. W rzeczywistości jest między nimi tylko jedna różnica — łańcuch zapytania SQL. Po raz kolejny zwraca uwagę ilość kodu potrzebna do wykonania tak prostej operacji, jak uaktualnienie pojedynczego wiersza w bazie danych. Co więcej, w tym momencie kod zaczyna się już powtarzać. Najchętniej wprowadzalibyśmy wyłącznie te fragmenty, które są niezbędne do wykonania aktualnego zadania. W końcu to tylko one odróżniają od siebie listingi 10 5 i 10.4. Cała reszta stanowi powielony kod.

Na zakończenie naszego przeglądu standardowego JDBC zobaczmy, jak wygląda pobieranie danych z bazy. Analizując listing 10.6, dojdziemy do wniosku, że także nie najładniej.

Listing 10.6. Użycie JDBC do pobrania wiersza z bazy danych

```
private static final String SQL_SELECT_SPITTER =
   "select id, username, fullname from spitter where id = ?";
public Spitter findOne(long id) {
   Connection conn = null;
   PreparedStatement stmt = null;
   ResultSet rs = null;
   try {
```

```
      conn = dataSource.getConnection();                    ← Uzyskujemy połączenie
      stmt = conn.prepareStatement(SQL_SELECT_SPITTER);     ← Tworzymy zapytanie
      stmt.setLong(1, id);                                  ← Podpinamy parametr
      rs = stmt.executeQuery();                             ← Wykonujemy zapytanie
      Spitter spitter = null;
      if (rs.next()) {                                      ← Przetwarzamy wyniki
         spitter = newSpitter();
         spitter.setId(rs.getLong("id"));
         spitter.setUsername(rs.getString("username"));
         spitter.setPassword(rs.getString("password"));
         spitter.setFullName(rs.getString("fullname"));
      }
      return spitter;
   } catch(SQLException e) {                                ← Obsługujemy wyjątki (w jakiś sposób)

   } finally {
      if (rs != null) {
         try {
            rs.close();
         } catch(SQLException e) {}
      }

      if (stmt != null) {
         try {
            stmt.close();                                   ← Sprzątamy
         } catch(SQLException e) {}
      }

      if (conn != null) {
         try {
            conn.close();
         } catch(SQLException e) {}
      }
   }
   return null;
}
```

Powyższy przykład jest tak samo rozwlekły jak wcześniejsze przykłady wstawiania i uaktualniania, o ile nie bardziej. Jest jak zasada Pareto postawiona na głowie: tylko 20 procent kodu bierze udział w zapytaniu, podczas gdy pozostałe 80 procent to „stały”, zduplikowany kod.

Myślę, że jest już wystarczająco jasne, jak duża część kodu JDBC jest powielana przy tworzeniu połączeń, zapytań i obsłudze wyjątków. Zakończę już zatem moje tortury i oszczędzę Ci kolejnych przykładów tego nieprzyjemnego kodu.

Duplikacja kodu nie jest nam potrzebna. Ale to nie oznacza, że mamy zrezygnować ze zwalniania zasobów czy obsługi błędów. Pozostawienie nieobsłużonych błędów i otwartych zasobów samym sobie naraziłoby nas na ryzyko nieprzewidywalności kodu i wycieku zasobów. Nie tylko zatem potrzebujemy tego kodu, ale musimy również mieć pewność, że jest on poprawny. Tym mniejsze powinny być nasze opory przed powierzeniem zduplikowanego kodu frameworkowi. Dzięki temu możemy być pewni, że kod wystąpi tylko w jednym miejscu i będzie wolny od błędów.

10.3.2. Praca z szablonami JDBC

Framework JDBC Springa oczyści Twój kod JDBC, biorąc na siebie ciężar zarządzania zasobami i obsługi wyjątków. Dzięki temu zyskujesz swobodę wprowadzania wyłącznie kodu potrzebnego do pobrania danych lub do ich zapisu w bazie.

Jak już wspomniałem w poprzedniej sekcji, Spring nakłada na powielany kod dostępu do danych warstwę abstrakcji w postaci klas szablonowych. Spring oferuje do wyboru trzy takie klasy:

- `JdbcTemplate` — Najbardziej podstawowy z szablonów JDBC Springa. Klasa ta zapewnia prosty dostęp do bazy danych poprzez JDBC i zapytania z indeksowanymi parametrami.
- `NamedParameterJdbcTemplate` — Ta klasa szablonowa JDBC pozwala na wykonywanie zapytań z podpinaniem wartości pod parametry nazwane w SQL.
- `SimpleJdbcTemplate` — Ta wersja szablonu JDBC wykorzystuje możliwości Javy 5, takie jak automatyczne opakowywanie (ang. *autoboxing*), typy sparametryzowane czy zmienna lista parametrów, upraszczając użycie szablonu JDBC.

Dawniej decyzja o wyborze szablonu JDBC wymagała starannego rozważenia wszystkich za i przeciw. Od wersji 3.1 Springa ta decyzja jest znacznie prostsza. Szablon `SimpleJdbcTemplate` został uznany za przestarzały, a jego możliwości związane z pojawieniem się Javy 5 zostały przeniesione do `JdbcTemplate`. Co więcej, szablon `NamedPara`
`↪meterJdbcTemplate` potrzebny jest nam tylko wtedy, gdy chcemy w zapytaniach wykorzystać parametry nazwane. Dzięki temu w zdecydowanej większości przypadków najlepszym wyborem do pracy z JDBC jest użycie starego, dobrego szablonu `Jdbc`
`↪Template`. I to właśnie na nim skoncentrujemy się w tym podrozdziale.

UMIESZCZANIE DANYCH W BAZIE Z UŻYCIEM JDBCTEMPLATE

Jedyną rzeczą potrzebną do działania `JdbcTemplate` jest źródło `DataSource`. Konfiguracja komponentu `JdbcTemplate` w Springu jest zatem dosyć prosta i można jej dokonać za pomocą poniższego kodu Java:

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
   return new JdbcTemplate(dataSource);
}
```

Instancja typu `DataSource` wstrzykiwana jest przez konstruktor. Komponent, do którego odnosi się właściwość `dataSource`, może być dowolną implementacją interfejsu `javax.sql.`
`↪DataSource`, na przykład jedną ze stworzonych przez nas w podrozdziale 10.2.

Teraz możemy dowiązać komponent `jdbcTemplate` do naszej klasy repozytorium i użyć go do dostępu do bazy danych. Załóżmy na przykład, że klasa repozytorium Spittera wykorzysta szablon `JdbcTemplate`:

```
@Repository
public class JdbcSpitterRepository implements SpitterRepository {

   private JdbcOperations jdbcOperations;

   @Inject
```

```
   public JdbcSpitterRepository(JdbcOperations jdbcOperations) {
      this.jdbcOperations = jdbcOperations;
   }
   ...
}
```

Klasa `JdbcSpitterRepository` opatrzona została adnotacją `@Repository`, dzięki czemu poddawana jest procesowi autoskanowania i tworzenia komponentów. Jej konstruktor oznaczony jest adnotacją `@Inject`, a więc przy tworzeniu komponentu przyjmuje instancję obiektu `JdbcOperations`. `JdbcOperations` jest interfejsem definiującym operacje zaimplementowane przez klasę `JdbcTemplate`. Dzięki wykorzystaniu interfejsu `JdbcOperations` zamiast konkretnej implementacji `JdbcTemplate`, repozytorium `JdbcSpitterRepository` jest tylko luźno powiązane z klasą `JdbcTemplate` poprzez ten interfejs.

Alternatywą dla skanowania komponentów i autowiązania jest jawna deklaracja komponentu `JdbcSpitterRepository` w Springu w podany niżej sposób:

```
@Bean
public SpitterRepository spitterRepository(JdbcTemplate jdbcTemplate) {
   return new JdbcSpitterRepository(jdbcTemplate);
}
```

Dzięki `JdbcTemplate` w naszym DAO możemy znacznie uprościć metodę `addSpitter()` z listingu 10.4. Poniżej, na listingu 10.7, pokazano jej wersję opartą na szablonie `Jdbc`
`↪Template`.

Listing 10.7. Wersja metody addSpitter() oparta na szablonie JdbcTemplate

```
public void addSpitter(Spitter spitter) {
   jdbcTemplate.update(INSERT_SPITTER, ←— Dodaj Spittera
      spitter.getUsername(),
      spitter.getPassword(),
      spitter.getFullName(),
      spitter.getEmail(),
      spitter.isUpdateByEmail());
}
```

Nietrudno zauważyć, że ta wersja metody `addSpitter()` jest o wiele prostsza. Udało się wyeliminować kod związany z połączeniem, z tworzeniem zapytania i — co najważniejsze — z obsługą wyjątków. Zostało tylko minimum niezbędne do wykonania operacji wstawienia danych.

To, że nie widać tego kodu, nie oznacza wcale, że go nie ma. Został on sprytnie ukryty w klasie szablonowej JDBC. Po wywołaniu metody `update()`, `JdbcTemplate` uzyskuje połączenie, tworzy zapytanie i wykonuje operację wstawienia w SQL.

Drugą rzeczą, której brak rzuca się w oczy, jest obsługa wyjątku `SQLException`. `SimpleJdbcTemplate` przechwytuje wewnętrznie każdy zgłoszony wyjątek `SQLException`, a następnie tłumaczy go na jeden z bardziej szczegółowych wyjątków dostępu do danych z tabeli 10.1, po czym zgłasza go ponownie. Ponieważ wszystkie wyjątki dostępu do danych Springa są wyjątkami czasu wykonania (ang. *runtime*), nazywanymi też wyjątkami niekontrolowanymi, w metodzie `addSpitter()` nie ma potrzeby ich przechwytywania.

ODCZYTUJEMY DANE ZA POMOCĄ SZABLONU JDBCTEMPLATE

JdbcTemplate ułatwia również odczyt danych. Poniżej, na listingu 10.8, zaprezentowano nową wersję metody findById(), która używa wywołań zwrotnych JdbcTemplate do odwzorowania zbioru wynikowego na obiekty dziedziny.

Listing 10.8. Uzyskiwanie z bazy obiektu Spitter z wykorzystaniem JdbcTemplate

```
public Spitter findById(long id) {
   return jdbcOperations.queryForObject(                    <-- Zapytanie o obiekt Spittera
      SELECT_SPITTER_BY_ID,                                 <-- Odwzorowanie wyników na obiekty
      new SpitterRowMapper(), id);
}
…
private static final class SpitterRowMapper
      implements RowMapper<Spitter> {
   public Spitter mapRow(ResultSet rs, int rowNum)
         throws SQLException {
      return new Spitter(                                   <-- Podpięcie parametrów
         rs.getLong("id"),
         rs.getString("username"),
         rs.getString("password"),
         rs.getString("fullName"),
         rs.getString("email"),
         rs.getBoolean("updateByEmail"));
   }
}
```

Metoda findById() używa metody queryForObject() szablonu JdbcTemplate, która wysyła do bazy zapytanie o obiekt Spittera. Metoda queryForObject() ma trzy parametry:

- Łańcuch znaków zawierających kod SQL potrzebny do pobrania danych z bazy.
- Obiekt typu RowMapper, który wydobywa wartości z obiektu ResultSet i konstruuje obiekt dziedziny (w tym przypadku obiekt Spitter).
- Zmienną listę argumentów zawierającą wartości, które zostaną podpięte pod indeksowane parametry zapytania.

Najciekawsze rzeczy dzieją się w obiekcie SpitterRowMapper, który implementuje interfejs RowMapper. Dla każdego zwróconego w wyniku zapytania wiersza JdbcTemplate wywołuje metodę mapRow() interfejsu RowMapper, przekazując jako parametr obiekt ResultSet i liczbę całkowitą przechowującą numer wiersza. Metoda mapRow() obiektu SpitterRow
↪Mapper zawiera kod tworzący obiekt Spitter i wypełniający go wartościami z obiektu ResultSet.

Podobnie jak w przypadku metody addSpitter(), metoda findById() została również uwolniona od zbędnego kodu. W przeciwieństwie do tradycyjnego JDBC nie zawiera kodu odpowiedzialnego za zarządzanie zasobami czy obsługę wyjątków. Metody używające JdbcTemplate są ściśle zorientowane na uzyskanie obiektu Spitter z bazy danych.

WYKORZYSTUJEMY LAMBDY JAVY 8 DO PRACY Z SZABLONAMI JDBCTEMPLATE

Interfejs `RowMapper` deklaruje tylko jedną metodę `addRow()`, jest więc interfejsem funkcyjnym. Oznacza to, że jeśli nasza aplikacja korzysta z Javy w wersji 8., implementację interfejsu `RowMapper` możemy przedstawić nie w postaci konkretnej implementacji klasy, a w postaci lambdy.

Na przykład metodę `findOne()` pokazaną na listingu 10.8 możemy przepisać z użyciem wyrażeń lambda Javy 8:

```
public Spitter findOne(long id) {
   return jdbcOperations.queryForObject(
      SELECT_SPITTER_BY_ID,
      (rs, rowNum) -> {
         return new Spitter(
            rs.getLong("id"), rs.getString("username"),
            rs.getString("password"), rs.getString("fullName"),
            rs.getString("email"), rs.getBoolean("updateByEmail"));
   }, id);
}
```

Jak nietrudno zauważyć, zapis z użyciem lambdy jest dużo przyjemniejszy dla oka niż pełna implementacja interfejsu RowMapper, a równocześnie tak samo efektywna. Java potrafi wykorzystać przekazaną lambdę jako parametr typu RowMapper.

Możemy też użyć funkcjonalności Javy 8 jako odwołania do metody i zdefiniować mapowanie w osobnej metodzie:

```
public Spitter findOne(long id) {
   return jdbcOperations.queryForObject(
      SELECT_SPITTER_BY_ID, this::mapSpitter, id);
}

private Spitter mapSpitter(ResultSet rs, int row)
      throws SQLException {
   return new Spitter(
      rs.getLong("id"), rs.getString("username"),
      rs.getString("password"), rs.getString("fullName"),
      rs.getString("email"), rs.getBoolean("updateByEmail"));
}
```

W obu przypadkach nie musimy implementować interfejsu `RowMapper` w sposób jawny. Dostarczamy lambdę lub metodę przyjmującą takie same parametry i zwracającą ten sam typ co metoda, którą zdefiniowalibyśmy w implementacji `RowMapper`.

PARAMETRY NAZWANE

Metoda `addSpitter()` z listingu 10.7 używa parametrów indeksowanych. Oznacza to, że musimy mieć świadomość kolejności, w jakiej parametry występują w zapytaniu, i zachować ją przy przekazywaniu wartości do metody `update()`. Jeśli kiedykolwiek dokonamy w SQL zmian, które spowodują inną kolejność parametrów, będziemy musieli zmodyfikować również kolejność wartości.

Rozwiązaniem alternatywnym jest użycie parametrów nazwanych. Parametry nazwane pozwalają na nadanie każdemu parametrowi w kodzie SQL jednoznacznej nazwy, do której można się odnieść przy podpinaniu wartości do zapytania. Załóżmy na przykład, że pod łańcuchem `SQL_INSERT_SPITTER` kryje się następująca definicja:

```
private static final String SQL_INSERT_SPITTER =
  "insert into spitter (username, password, fullname) " +
  "values (:username, :password, :fullname)";
```

W przypadku parametrów nazwanych kolejność podpinanych wartości nie jest istotna. Wartości są podpinane z wykorzystaniem nazw. Jeżeli kolejność parametrów w zapytaniu się zmieni, nie będzie potrzeby modyfikacji kodu.

Klasa `NamedParameterJdbcTemplate` jest specjalną klasą szablonową JDBC umożliwiającą pracę z nazwanymi parametrami. `NamedParameterJdbcTemplate` może być zadeklarowana w Springu w taki sam sposób jak zwykła klasa `JdbcTemplate`:

```
@Bean
public NamedParameterJdbcTemplate jdbcTemplate(DataSource dataSource) {
   return new NamedParameterJdbcTemplate(dataSource);
}
```

Jeśli wstrzyknęlibyśmy interfejs `NamedParameterJdbcOperations` (interfejs, który implementuje klasa `NamedParameterJdbcTemplate`), metoda `addSpitter()` mogłaby wyglądać tak jak na listingu 10.9.

Listing 10.9. Wykorzystanie parametrów nazwanych w szablonach JDBC Springa

```
private static final String INSERT_SPITTER =
   "insert into Spitter " +
      " (username, password, fullname, email, updateByEmail) " +
   "values " +
      " (:username, :password, :fullname, :email, :updateByEmail)";

public void addSpitter(Spitter spitter) {
   Map<String, Object> paramMap = new HashMap<String, Object>();
   paramMap.put("username", spitter.getUsername());   <-- Podpięcie parametrów
   paramMap.put("password", spitter.getPassword());
   paramMap.put("fullname", spitter.getFullName());
   paramMap.put("email", spitter.getEmail());
   paramMap.put("updateByEmail", spitter.isUpdateByEmail());

   jdbcOperations.update(INSERT_SPITTER, paramMap);   <-- Wykonanie operacji wstawienia
}
```

Pierwszą rzucającą się w oczy rzeczą w tej wersji metody `addSpitter()` jest większa ilość kodu. Spowodowane jest to użyciem interfejsu `java.util.Map` do podpięcia parametrów. Tym niemniej każdy wiersz kodu pozostaje skoncentrowany na zadaniu, jakim jest wstawienie obiektu `Spitter` do bazy danych. Metoda jest w dalszym ciągu pozbawiona niepotrzebnego kodu odpowiadającego za zarządzanie zasobami i obsługę wyjątków.

10.4. Podsumowanie

Dane są siłą napędową aplikacji. Co bardziej ortodoksyjni z nas mogą nawet twierdzić, że dane SĄ aplikacją. Przy tak dużym znaczeniu danych jest niezwykle istotne, aby związana z dostępem do nich część naszej aplikacji była tworzona w wydajny, prosty i przejrzysty sposób.

JDBC jest najbardziej podstawowym sposobem pracy z danymi relacyjnymi w Javie. Specyfikacja ta sprawia jednak, że czasem potrafi być on dość nieporęczny. Spring niweluje wiele problemów związanych z pracą z JDBC, eliminując nadmiar kodu i upraszczając obsługę wyjątków JDBC. Dzięki temu na naszych barkach pozostaje niewiele więcej niż tylko utworzenie zapytań SQL.

W tym rozdziale poznałeś możliwości Springa w zakresie utrwalania danych, a także szablony stanowiące warstwę abstrakcji Springa wokół JDBC, co w znacznym stopniu upraszcza pracę z tą specyfikacją.

W następnym rozdziale będziemy dalej poznawać warstwę trwałości Springa, obserwując mechanizmy jego pracy z Java Persistence API.

11 Zapisywanie danych z użyciem mechanizmów ORM

W tym rozdziale omówimy:

- Pracę ze Springiem i Hibernate
- Tworzenie repozytoriów niezależnych od Springa z obsługą sesji kontekstowych
- Wykorzystanie JPA w Springu
- Automatyczne repozytoria JPA z użyciem Spring Data

Kiedy byłem małym chłopcem, nic nie sprawiało mi takiej frajdy jak jazda na rowerze. Razem z kolegami jechaliśmy rano do szkoły. Po lekcjach, także na dwóch kółkach, odwiedzaliśmy przyjaciół. A gdy robiło się już późno i rodzice krzyczeli na nas za zostawanie na dworze po zmroku, pedałowaliśmy ile sił do domu. To były czasy!

Dziś ja i moi koledzy potrzebujemy czegoś więcej niż rower. Wielu z nas musi pokonać każdego dnia długą drogę do pracy. Raz na jakiś czas trzeba też zrobić większe zakupy, a nasze dzieci do szkoły i na trening nie jeżdżą już na rowerach. No i klimatyzacja… przydaje się — zwłaszcza jeśli mieszkasz w Teksasie. Wyrośliśmy z rowerów, a raczej nasze potrzeby z nich wyrosły.

JDBC jest rowerem w świecie utrwalania danych. Cieszy się sympatią i jest w sam raz do wielu zastosowań. Ale wraz ze wzrostem złożoności aplikacji wzrastają również potrzeby w zakresie utrwalania danych. Po jakimś czasie chcemy mieć możliwość

odwzorowania właściwości obiektu na kolumny w bazie danych i oczekujemy wsparcia przy tworzeniu zapytań (żeby nie musieć samodzielnie wpisywać niekończących się ciągów znaków zapytania). Zaczynamy także potrzebować tych bardziej skomplikowanych usług:

- **Leniwe ładowanie** (ang. *lazy loading*) — Przy bardziej złożonych grafach obiektów nie zawsze potrzebne nam są wszystkie zależności od razu. Załóżmy na przykład, że mamy kolekcję obiektów `Zamowienie`, a każdy z tych obiektów zawiera z kolei kolekcję obiektów `Pozycja`. Skoro jesteśmy zainteresowani wyłącznie atrybutami zamówienia, nie ma sensu pobierać danych o pozycjach. To mogłoby być kosztowne. Leniwe ładowanie pozwala na uzyskanie danych tylko wtedy, kiedy są potrzebne.
- **Chciwe pobieranie** (ang. *eager fetching*) — Jest przeciwieństwem leniwego ładowania. Chciwe pobieranie pozwala na uzyskanie całego grafu obiektu jednym zapytaniem. W sytuacjach, kiedy wiemy, że potrzebujemy obiektu `Zamowienie` wraz z przynależnymi obiektami `Pozycja`, chciwe pobieranie pozwoli na pobranie wszystkich danych w ramach jednej operacji, zmniejszając liczbę kosztownych cykli komunikacyjnych z bazą.
- **Kaskadowość** (ang. *cascading*) — Czasem zmiany w jednej tabeli bazy powinny wywołać zmiany w innych tabelach. Wracając do naszego przykładu zakupu, jeżeli `Zamowienie` zostanie usunięte, chcemy również usunąć z bazy przynależne obiekty `Pozycja`.

Istnieje szereg mechanizmów, w których znajdziemy powyższe usługi. Usługi te określamy ogólnym terminem **odwzorowań obiektowo-relacyjnych** (ang. *object-relational mapping* — ORM). Zastosowanie narzędzia ORM do warstwy trwałości pozwoli Ci zaoszczędzić dosłownie tysięcy wierszy kodu i długich godzin pracy. Dzięki niemu zamiast operować na podatnym na błędy SQL, skupisz się na potrzebach swojej aplikacji.

Spring pozwala na pracę z licznymi mechanizmami utrwalania, między innymi z Hibernate, iBATIS, Java Data Objects (JDO) i Java Persistence API (JPA).

Podobnie jak w przypadku JDBC, również dla mechanizmów ORM Spring zapewnia punkty integracyjne z tymi mechanizmami, a także dodatkowe usługi:

- wbudowaną obsługę transakcji deklaratywnych Springa,
- przejrzystą obsługę wyjątków,
- bezpieczne dla wielowątkowości (ang. *thread-safe*), lekkie klasy szablonowe,
- klasy bazowe DAO,
- zarządzanie zasobami.

Nie mamy wystarczająco dużo miejsca, aby omówić w tym rozdziale wszystkie obsługiwane przez Springa mechanizmy ORM. Nie jest to jednak wielki problem, ponieważ poszczególne rozwiązania ORM są obsługiwane w bardzo podobny sposób. Jeśli potrafisz korzystać z jednego mechanizmu ORM w Springu, przejście na inny nie powinno Ci sprawić większego kłopotu.

W tym rozdziale dowiesz się, jak Spring integruje się z dwoma najczęściej wykorzystywanymi rozwiązaniami ORM: Hibernate oraz JPA. Spotkamy się też po raz pierwszy

z projektem Spring Data JPA. Zobaczysz, w jaki sposób projekt ten niweluje potrzebę tworzenia nadmiarowego kodu w repozytoriach JPA, a dodatkowo zdobędziesz podstawową wiedzę, na której bazować będziemy w następnym rozdziale przy okazji pracy ze Spring Data w połączeniu z bazami bez ustalonego schematu danych.

Zacznijmy więc od obsługi Hibernate z wykorzystaniem Springa.

11.1. Integrujemy Hibernate ze Springiem

Hibernate jest rozwijaną na zasadzie open source platformą utrwalania, która zyskała sporą popularność w środowisku programistów. Umożliwia nie tylko elementarne odwzorowania obiektowo-relacyjne, ale również zaawansowane usługi oferowane przez profesjonalne narzędzia ORM, takie jak pamięć podręczna, leniwe ładowanie, chciwe pobieranie czy rozproszona pamięć podręczna.

W tym podrozdziale skoncentrujemy się na integracji Springa z Hibernate, nie zagłębiając się zbytnio w szczegóły pracy z samym Hibernate. Jeżeli chcesz poznać bliżej Hibernate, polecam książkę *Hibernate w akcji* (Helion, 2007) lub stronę domową projektu Hibernate: *http://www.hibernate.org*.

11.1.1. Deklarowanie fabryki sesji Hibernate

Podstawowym interfejsem w pracy z Hibernate jest `org.hibernate.Session`. Interfejs `Session` umożliwia wykonanie elementarnych czynności w zakresie dostępu do bazy danych, takich jak zapisywanie, uaktualnianie, usuwanie i ładowanie obiektów z bazy. Zaspokaja on wszystkie potrzeby obiektu repozytorium w dziedzinie utrwalania danych.

Referencję do obiektu `Session` frameworka Hibernate standardowo uzyskuje się poprzez implementację interfejsu `SessionFactory` frameworka Hibernate. Implementacja `SessionFactory` jest odpowiedzialna za otwieranie i zamykanie sesji Hibernate oraz zarządzanie nimi.

Fabrykę sesji Hibernate można uzyskać w Springu poprzez jeden z komponentów fabryki sesji Hibernate. Spring w wersji 3.1 oferuje do wyboru trzy komponenty fabryki sesji:

- `org.springframework.orm.hibernate3.LocalSessionFactoryBean`,
- `org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean`,
- `org.springframework.orm.hibernate4.LocalSessionFactoryBean`.

Komponenty te są implementacjami interfejsu `FactoryBean` Springa, które powiązane z jakąkolwiek właściwością typu `SessionFactory`, tworzą fabrykę sesji `SessionFactory`. Umożliwia to konfigurację fabryki sesji Hibernate razem z innymi komponentami w kontekście Springa Twojej aplikacji.

Wybór fabryki sesji uzależniony jest od wykorzystywanej wersji Hibernate i decyzji, czy definicje mapowań obiektów na wpisy bazy chcemy konfigurować w plikach XML, czy z użyciem adnotacji. Jeśli korzystamy z Hibernate w wersji 3.2 lub wyższej (ale niższej niż 4.0) i definicje mapowań chcemy zapisać w pliku XML, musimy skonfigurować komponent `LocalSessionFactoryBean`, pochodzący z pakietu Springa `org.spring`
`↪framework.orm.hibernate3`:

```
@Bean
public
LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
   LocalSessionFactoryBean sfb = new LocalSessionFactoryBean();
   sfb.setDataSource(dataSource);
   sfb.setMappingResources(new String[] { "Spitter.hbm.xml" });
   Properties props = new Properties();
   props.setProperty("dialect", "org.hibernate.dialect.H2Dialect");
   sfb.setHibernateProperties(props); return sfb;
}
```

`LocalSessionFactoryBean` jest tu skonfigurowany za pomocą trzech właściwości. Właściwość `dataSource` jest powiązana referencją z komponentem `DataSource`. Właściwość `mappingResources` definiuje listę jednego lub więcej plików odwzorowań, które określają strategię utrwalania danych dla aplikacji. Ostatnia właściwość, `hibernateProperties`, zawiera szczegółowe wytyczne dla Hibernate. W naszym przykładzie informujemy Hibernate, że będzie współpracować z bazą danych H2, w związku z czym do konstrukcji wyrażeń SQL powinno używać dialektu `H2Dialect`.

Jeśli preferujesz utrwalanie danych przy użyciu adnotacji, ale nie korzystasz jeszcze z Hibernate 4, zamiast `LocalSessionFactoryBean` możesz zastosować `AnnotationSession` ➥`FactoryBean`:

```
@Bean
public AnnotationSessionFactoryBean sessionFactory(DataSource ds) {
   AnnotationSessionFactoryBean sfb = new AnnotationSessionFactoryBean();
   sfb.setDataSource(ds);
   sfb.setPackagesToScan(new String[] { "com.habuma.spittr.domain" });
   Properties props = new Properties();
   props.setProperty("dialect", "org.hibernate.dialect.H2Dialect");
   sfb.setHibernateProperties(props);
   return sfb;
}
```

Jeżeli korzystasz z Hibernate 4, powinieneś użyć komponentu `LocalSessionFactoryBean` z pakietu `org.springframework.orm.hibernate4`. Nazwa ta jest identyczna jak w odpowiedniku dla Hibernate 3, jednak ten nowy komponent fabryki sesji, dodany w Springu 3.1, stanowi połączenie cech komponentu `LocalSessionFactoryBean` z wersji Hibernate 3 oraz `AnnotationSessionFactoryBean`. Posiada on wiele takich samych właściwości i udostępnia możliwość konfiguracji mapowań z użyciem zarówno pliku XML, jak i adnotacji. Poniższy przykład przedstawia konfigurację mapowania w oparciu o adnotacje:

```
@Bean
public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
   LocalSessionFactoryBean sfb = new LocalSessionFactoryBean();
   sfb.setDataSource(dataSource);
   sfb.setPackagesToScan(new String[] { "com.habuma.spittr.domain" });
   Properties props = new Properties();
   props.setProperty("dialect", "org.hibernate.dialect.H2Dialect");
   sfb.setHibernateProperties(props);
   return sfb;
}
```

W obu przypadkach właściwości `dataSource` oraz `hibernateProperties` wskazują nam, gdzie znaleźć połączenie z bazą danych i z jakim rodzajem bazy będziemy mieć do czynienia.

Ale zamiast listy plików odwzorowań Hibernate możemy skorzystać z właściwości `packagesToScan`, nakazując Springowi przeszukanie jednego lub więcej pakietów w poszukiwaniu klas dziedziny oznaczonych (za pomocą adnotacji) jako utrwalane przez Hibernate, włączając w to klasy z adnotacjami JPA `@Entity` i `@MappedSuperClass` oraz własną adnotacją Hibernate `@Entity`.

Jeśli wolisz, możesz też zdefiniować wyczerpującą listę trwałych klas swojej aplikacji. We właściwości `annotatedClasses` umieść listę pełnych nazw klas zgodnie ze wzorem:

```
sfb.setAnnotatedClasses(
   new Class<?>[] { Spitter.class, Spittle.class }
);
```

Właściwość `annotatedClass` spełnia swoją rolę przy wyborze pewnej niewielkiej liczby klas dziedziny. Jeśli jest ich dużo, bardziej odpowiednie będzie użycie właściwości `pack`
`↪agesToScan`. Zniknie wtedy potrzeba powracania każdorazowo do konfiguracji Springa przy usuwaniu i dodawaniu nowych klas.

Zadeklarowaliśmy już komponent fabryki sesji Hibernate w kontekście aplikacji Springa, możemy zatem przejść do tworzenia naszych klas repozytoriów.

11.1.2. Hibernate bez Springa

We wczesnych wersjach Springa i Hibernate tworzenie klasy repozytorium wymagało pracy z szablonem Springa `HibernateTemplate`, który brał na siebie zapewnienie jednej sesji na transakcję. Minusem tego rozwiązania jest jednak to, że implementacja repozytorium staje się bezpośrednio powiązana ze Springiem.

Najlepszym sposobem jest wykorzystanie sesji kontekstowych Hibernate i niestosowanie wcale szablonu `HibernateTemplate`. Jest to możliwe dzięki powiązaniu fabryki sesji Hibernate bezpośrednio do repozytorium i użyciu jej do utworzenia sesji, jak to pokazano na listingu 11.1.

Listing 11.1. Repozytoria Hibernate bez śladów użycia Springa dzięki wykorzystaniu sesji Hibernate

```
public HibernateSpitterRepository(SessionFactory sessionFactory) {
   this.sessionFactory = sessionFactory;
}
private Session currentSession() {
   return sessionFactory.getCurrentSession();
}
public long count() {
   return findAll().size();
}
public Spitter save(Spitter spitter) {
   Serializable id = currentSession().save(spitter);
   return new Spitter((Long) id,
      spitter.getUsername(), spitter.getPassword(),
      spitter.getFullName(), spitter.getEmail(),
      spitter.isUpdateByEmail());
```

Wstrzykujemy fabrykę sesji

Pobieramy aktualną sesję z fabryki sesji

Używamy bieżącej sesji

```
}
public Spitter findOne(long id) {
   return (Spitter) currentSession().get(Spitter.class, id);
}
public Spitter findByUsername(String username) {
   return (Spitter) currentSession()
      .createCriteria(Spitter.class)
      .add(Restrictions.eq("username", username))
      .list().get(0);
}
public List<Spitter> findAll() {
   return (List<Spitter>) currentSession()
      .createCriteria(Spitter.class).list();
}
```

Warto zwrócić uwagę na kilka rzeczy na listingu 11.1. Najpierw, dzięki adnotacji `@Inject`, Spring automatycznie wstrzykuje implementację interfejsu `SessionFactory` do właściwości `sessionFactory` obiektu `HibernateSpitterRepository`. Następnie, w metodzie `current`
`➥Session()` przy pomocy tej implementacji uzyskujemy sesję aktualnej transakcji.

Zauważmy też, że klasa została oznaczona adnotacją `@Repository`. Dzięki temu zyskujemy dwie rzeczy. Po pierwsze, adnotacja `@Repository` jest jedną z adnotacji stereotypowych, które są wyszukiwane za pomocą mechanizmu skanowania komponentów. Oznacza to, że zamiast stosować jawną deklarację komponentu `HibernateSpitterRepo`
`➥sitory`, wystarczy, iż klasę repozytorium umieścimy w pakiecie podlegającym skanowaniu.

Adnotacja `@Repository` przyczynia się zatem do zredukowania potrzeby jawnej konfiguracji. Oprócz tego ma jeszcze jedną zaletę. Przypomnijmy, że do zadań klasy szablonowej należy przechwytywanie wyjątków platformy i ponowne ich zgłaszanie w postaci zunifikowanych niekontrolowanych wyjątków Springa. Ale jak dokonać tłumaczenia wyjątków, jeśli zamiast szablonu Hibernate używamy sesji kontekstowych?

Aby umożliwić tłumaczenie wyjątków w niezawierającej szablonu klasie repozytorium Hibernate, musimy dodać komponent `PersistenceExceptionTranslationPostProcessor` w kontekście aplikacji Springa:

```
@Bean
public BeanPostProcessor persistenceTranslation() {
   return new PersistenceExceptionTranslationPostProcessor();
}
```

`PersistenceExceptionTranslationPostProcessor` jest komponentem postprocesora, który dodaje doradcę do każdego oznaczonego adnotacją `@Repository` komponentu, dzięki czemu wyjątki platformy mogą być przechwytywane i zgłaszane ponownie w postaci niekontrolowanych wyjątków dostępu do danych Springa.

Wersja Hibernate naszego repozytorium jest już gotowa. Wyeliminowaliśmy z niej specyficzne dla Springa klasy (używamy jedynie adnotacji `@Repository`). To bezszablonowe podejście możemy zastosować również, tworząc repozytorium na bazie czystego JPA. Podejmijmy więc jeszcze jedną próbę implementacji `SpitterRepository`, tym razem z wykorzystaniem JPA.

11.2. Spring i Java Persistence API

Java Persistence API (JPA) powstało na zgliszczach komponentów encyjnych EJB 2, jako standard utrwalania danych nowej generacji. JPA to mechanizm utrwalania oparty na POJO, czerpiący obficie zarówno z Hibernate, jak i z **Java Data Objects** (JDO) i dorzucający na dokładkę adnotacje Javy 5.

Od wersji 2.0 Spring jest zintegrowany z JPA. Ironią losu jest, iż wielu wini (albo ceni) Springa za upadek EJB. Teraz jednak, kiedy Spring obsługuje już JPA, wielu programistów rekomenduje ten właśnie mechanizm utrwalania w aplikacjach opartych na Springu. Niektórzy twierdzą nawet, że nie ma nic lepszego niż tandem Spring — JPA do programowania POJO.

Pierwszym krokiem w kierunku używania JPA ze Springiem jest konfiguracja fabryki menedżerów encji jako komponentu w kontekście aplikacji Springa.

11.2.1. Konfiguracja fabryki menedżerów encji

Najkrócej rzecz ujmując, aplikacje oparte na JPA używają implementacji interfejsu `EntityManagerFactory` do uzyskania instancji `EntityManager`. Specyfikacja JPA definiuje dwa rodzaje menedżerów encji:

- **Zarządzane przez aplikację** (ang. *application-managed*) — Menedżery encji są tworzone przez fabrykę na bezpośrednie żądanie aplikacji. W przypadku menedżerów zarządzanych przez aplikację to aplikacja jest odpowiedzialna za otwarcie i zamknięcie menedżerów encji oraz za zaangażowanie ich w transakcję. Ten rodzaj menedżera encji jest najlepszym rozwiązaniem dla samodzielnych aplikacji, które nie działają w kontenerze Java EE.
- **Zarządzane przez kontener** (ang. *container-managed*) — Menedżery encji są tworzone i zarządzane przez kontener Java EE. Nie ma interakcji aplikacji z fabryką menedżerów encji. Za to menedżery encji są pozyskiwane bezpośrednio przez wstrzyknięcie lub za pomocą JNDI. Za konfigurację fabryk menedżerów encji odpowiedzialny jest kontener. Użycie tego rodzaju menedżerów będzie odpowiednie, jeśli chcemy, by kontener Java EE miał kontrolę nad konfiguracją JPA, nieograniczającą się do tego, co zdefiniowano w pliku *persistence.xml*.

Obydwa rodzaje menedżerów encji implementują ten sam interfejs `EntityManager`. Kluczową różnicą nie jest interfejs `EntityManager` sam w sobie, ale sposób, w jaki menedżer encji jest tworzony i zarządzany. Menedżery zarządzane przez aplikację są tworzone przez fabrykę `EntityManagerFactory`, która jest z kolei uzyskiwana przez wywołanie metody `createEntityManagerFactory()` implementacji `PersistenceProvider`. `Persistence` ↪`Provider` zawiera też analogiczną metodę tworzenia menedżerów zarządzanych przez kontener — `createContainerEntityManagerFactory()`.

Co to wszystko oznacza dla programistów chcących używać JPA? Niewiele. Niezależnie od tego, który wariant `EntityManagerFactory` wybierzesz, Spring będzie zarządzał menedżerami encji za Ciebie. Jeśli użyjesz menedżera encji zarządzanego przez aplikację, Spring wcieli się w rolę aplikacji i zajmie się za Ciebie menedżerem encji w przezroczysty sposób. W scenariuszu zarządzania przez kontener Spring odegra rolę kontenera.

Każdy wariant fabryki menedżerów encji jest tworzony przez odpowiedni komponent fabryki Springa:

- `LocalEntityManagerFactoryBean` tworzy zarządzany przez aplikację `EntityManagerFactory`.
- `LocalContainerEntityManagerFactoryBean` tworzy zarządzany przez kontener `EntityManagerFactory`.

Należy podkreślić, że wybór pomiędzy `EntityManagerFactory` zarządzaną przez aplikację a `EntityManagerFactory` zarządzaną przez kontener jest zupełnie przezroczysty dla aplikacji opartej na Springu. Gdy korzystamy ze Springa i JPA, te zawiłości i detale są w obu przypadkach ukryte, co pozwala programiście skoncentrować wysiłek na prawdziwym celu: dostępie do danych.

Jedyną istotną różnicą pomiędzy fabrykami zarządzanymi przez aplikację a tymi zarządzanymi przez kontener, jeśli chodzi o Springa, jest ich konfiguracja w kontekście aplikacji Springa. Najpierw zobaczymy, jak skonfigurować zarządzany przez aplikację komponent `LocalEntityManagerFactoryBean` w Springu. Później dowiemy się, jak to samo zrobić z zarządzanym przez kontener komponentem `LocalContainerEntityManagerFactoryBean`.

KONFIGURACJA JPA ZARZĄDZANEGO PRZEZ APLIKACJĘ

Zarządzane przez aplikację fabryki menedżerów encji czerpią większość informacji o swojej konfiguracji z pliku konfiguracyjnego *persistence.xml*. Plik musi znajdować się w katalogu *META-INF* w obrębie ścieżki do klas.

Podstawowym celem pliku *persistence.xml* jest zdefiniowanie jednej lub więcej jednostek utrwalania. Jednostka utrwalania (ang. *persistence unit*) jest grupą jednej lub większej liczby trwałych klas, które odnoszą się do pojedynczego źródła danych. Plik *persistence.xml* wymienia po prostu jedną lub więcej trwałych klas wraz z dodatkową informacją konfiguracyjną (źródła danych, pliki odwzorowań w formacie XML). Typowy plik *persistence.xml* będzie wyglądał podobnie jak ten w aplikacji Spittr:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
   <persistence-unit name="spitterPU">
      <class>com.habuma.spittr.domain.Spitter</class>
      <class>com.habuma.spittr.domain.Spittle</class>
      <properties>
         <property name="toplink.jdbc.driver" value="org.hsqldb.jdbcDriver" />
         <property name="toplink.jdbc.url"
           value="jdbc:hsqldb:hsql://localhost/spitter/spitter" />
         <property name="toplink.jdbc.user" value="sa" />
         <property name="toplink.jdbc.password" value="" />
      </properties>
   </persistence-unit>
</persistence>
```

Z racji tego, że lwia część konfiguracji ma miejsce w pliku *persistence.xml*, niewiele już trzeba (lub nawet można) zrobić w Springu. Poniżej pokazano deklarację komponentu `LocalEntityManagerFactoryBean` w Springu:

```
@Bean
public LocalEntityManagerFactoryBean entityManagerFactoryBean() {
   LocalEntityManagerFactoryBean emfb
      = new LocalEntityManagerFactoryBean();
   emfb.setPersistenceUnitName("spitterPU");
   return emfb;
}
```

Wartość nadana właściwości `persistenceUnitName` odnosi się do nazwy jednostki utrwalania z pliku *persistence.xml*.

Duża rola pliku *persistence.xml* przy tworzeniu zarządzanej przez aplikację `EntityManagerFactory` wynika z samej istoty zarządzania przez aplikację. W scenariuszu zarządzania przez aplikację (bez udziału Springa) cała odpowiedzialność za uzyskanie `EntityManagerFactory` poprzez implementację JPA `PersistenceProvider` spoczywa na aplikacji. Kod aplikacji rozrósłby się do niewyobrażalnych rozmiarów, gdyby musiała ona definiować jednostkę utrwalania przy każdej próbie uzyskania `EntityManagerFactory`. Dzięki plikowi *persistence.xml* JPA może każdorazowo łatwo odszukać definicje jednostek utrwalania.

Spring obsługuje jednak JPA w sposób, który sprawia, że nie operujemy bezpośrednio na `PersistenceProvider`. Dlatego umieszczanie konfiguracji w pliku *persistence.xml* wydaje się nierozsądne. Co więcej, jeśli to zrobimy, konfiguracja `EntityManagerFactory` w Springu (na przykład żeby dostarczyć skonfigurowane za pomocą Springa źródło danych) nie będzie możliwa.

Właśnie dlatego przejdziemy teraz do JPA zarządzanego przez kontener.

KONFIGURACJA JPA ZARZĄDZANEGO PRZEZ KONTENER

JPA zarządzane przez kontener wymaga innego podejścia. Jeżeli aplikacja działa w kontenerze, fabryka `EntityManagerFactory` może zostać utworzona na podstawie informacji dostarczonych przez kontener, w naszym przypadku — Spring.

Zamiast konfigurować szczegóły źródła danych w pliku *persistence.xml*, możesz tę informację umieścić w kontekście aplikacji Springa. Poniższa deklaracja pokazuje, jak skonfigurować JPA zarządzane przez kontener w Springu za pomocą `LocalContainer` ➥`EntityManagerFactoryBean`:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(
      DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {
   LocalContainerEntityManagerFactoryBean emfb = new
      LocalContainerEntityManagerFactoryBean();
   emfb.setDataSource(dataSource);
   emfb.setJpaVendorAdapter(jpaVendorAdapter);
   return emfb;
}
```

We właściwości `dataSource` odnosimy się do skonfigurowanego za pomocą Springa źródła danych. Dopuszczalna jest dowolna implementacja `javax.sql.DataSource`. Chociaż źródło danych może być nadal skonfigurowane w pliku *persistence.xml*, właściwość `dataSource` ma wyższy priorytet.

Właściwość `jpaVendorAdapter` służy do określenia szczegółów implementacji JPA, która ma być używana. Spring daje nam kilka adapterów dostawców do wyboru:

- `EclipseLinkJpaVendorAdapter`,
- `HibernateJpaVendorAdapter`,
- `OpenJpaVendorAdapter`,
- `TopLinkJpaVendorAdapter` (w Springu 3.1 uznany za przestarzały).

W tym przypadku jako implementacji JPA używamy Hibernate, do konfiguracji `jpaVen`
`↪dorAdapter` użyjemy `HibernateJpaVendorAdapter`:

```
@Bean
public JpaVendorAdapter jpaVendorAdapter() {
   HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
   adapter.setDatabase("HSQL");
   adapter.setShowSql(true);
   adapter.setGenerateDdl(false);
   adapter.setDatabasePlatform("org.hibernate.dialect.HSQLDialect");
return adapter;
}
```

Najważniejszą spośród właściwości adaptera dostawcy jest `database`, której nadaliśmy wartość `Hypersonic`, ponieważ to tej bazy danych będziemy używać. Wybrane inne wartości, które właściwość `database` może przyjmować, zostały wymienione w tabeli 11.1.

Tabela 11.1. HibernateJpaVendorAdapter obsługuje szereg baz danych. Określając odpowiednią właściwość, możesz wybrać bazę, która ma zostać użyta

Platforma bazy danych	Wartość właściwości database
IBM DB2	DB2
Apache Derby	DERBY
H2	H2
Hypersonic	HSQL
Informix	INFORMIX
MySQL	MYSQL
Oracle	ORACLE
PostgreSQL	POSTGRESQL
Microsoft SQL Server	SQLSERVER
Sybase	SYBASE

Pewne usługi dynamicznego utrwalania danych wymagają modyfikacji klas obiektów trwałych. Klasy obiektów, których właściwości są leniwie ładowane (nie będą pobrane z bazy aż do momentu próby dostępu), muszą zostać wyposażone w kod umożliwiający pobranie danych przy próbie dostępu. Niektóre mechanizmy do implementacji leniwego ładowania używają dynamicznych klas pośredników (ang. *proxy*). Inne, na przykład JDO, wyposażają klasy w odpowiedni kod w trakcie kompilacji.

Wybór komponentu fabryki menedżera encji zależy w dużym stopniu od tego, do czego będzie używany.

Poniższa wskazówka powinna przekonać Cię do wyboru `LocalEntityManagerFactory➥Bean`.

Głównym zadaniem pliku *persistence.xml* jest wskazanie klas encji w jednostce utrwalania. Począwszy od Springa 3.1, możesz to robić bezpośrednio w pliku `LocalContai➥nerEntityManagerFactoryBean` poprzez ustawienie właściwości `packagesToScan`:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(
      DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {
   LocalContainerEntityManagerFactoryBean emfb = new
      LocalContainerEntityManagerFactoryBean();
   emfb.setDataSource(dataSource);
   emfb.setJpaVendorAdapter(jpaVendorAdapter);
   emfb.setPackagesToScan("com.habuma.spittr.domain");
   return emfb;
}
```

Komponent `LocalContainerEntityManagerFactoryBean` przeskanuje pakiet `com.habuma.➥spittr.domain` w poszukiwaniu klas oznaczonych adnotacją `@Entity`. Nie ma więc potrzeby, żeby je deklarować w sposób jawny w pliku *persistence.xml*. Co więcej, plik ten nie musi wcale istnieć! Po jego usunięciu komponent `LocalContainerEntityManager➥FactoryBean` przygotuje całą konfigurację za nas.

POBIERANIE ENTITYMANAGERFACTORY Z JNDI

Warto zauważyć, że jeśli wdrażasz swoją aplikację Springa na niektórych serwerach aplikacji, być może obiekt `EntityManagerFactory` został już utworzony i jest gotowy do pobrania za pomocą JNDI. W takim przypadku użyj elementu `<jee:jndi-lookup>` z przestrzeni nazw `jee` Springa, aby uchwycić referencję do `EntityManagerFactory`:

```
<jee:jndi-lookup id="emf" jndi-name="persistence/spitterPU" />
```

Komponent `EntityManagerFactory` można też skonfigurować za pomocą konfiguracji Java:

```
@Bean
public JndiObjectFactoryBean entityManagerFactory() {}
   JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();
   jndiObjectFB.setJndiName("jdbc/SpittrDS");
   return jndiObjectFB;
}
```

Chociaż ta metoda nie zwraca obiektu `EntityManagerFactory`, to jej wynikiem jest powstanie komponentu `EntityManagerFactory`. Dzieje się tak dlatego, że zwracany komponent `JndiObjectFactoryBean` jest implementacją interfejsu `FactoryBean`, który z kolei tworzy komponent `EntityManagerFactory`.

Niezależnie od tego, w jaki sposób zdobędziesz instancję `EntityManagerFactory`, jeśli już ją masz — możesz zacząć tworzyć repozytoria. Zatem do dzieła!

11.2.2. Klasa repozytorium na bazie JPA

Tak jak w przypadku pozostałych opcji Springa w zakresie integracji z mechanizmami utrwalania, integracja z JPA oferuje nam szablon `JpaTemplate`. Niemniej jednak, analogicznie do użytych w podrozdziale 11.1.2 sesji kontekstowych Hibernate, zrezygnujemy z podejścia opartego na szablonie JPA na rzecz czystego JPA.

Ponieważ takie podejście ma więcej zalet, w tym podrozdziale skupimy się na budowie opartych na JPA obiektów repozytorium wolnych od kodu Springa. `JpaSpitter` ➥`Repository` z listingu 11.2 jest przykładem opartej na JPA klasy repozytorium zbudowanej bez korzystania z szablonu `JpaTemplate`.

Listing 11.2. Klasa repozytorium bazująca na czystym JPA nie używa szablonów Springa

```
package com.habuma.spittr.persistence;
import java.util.List;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import com.habuma.spittr.domain.Spitter;
import com.habuma.spittr.domain.Spittle;

@Repository
@Transactional
public class JpaSpitterRepository implements SpitterRepository {

   @PersistenceUnit
   private EntityManagerFactory emf; <-- Wstrzykujemy EntityManagerFactory

   public void addSpitter(Spitter spitter) {
      emf.createEntityManager().persist(spitter); <-- Tworzymy i używamy menedżera encji
   }

   public Spitter getSpitterById(long id) {
      return emf.createEntityManager().find(Spitter.class, id);
   }

   public void saveSpitter(Spitter spitter) {
      emf.createEntityManager().merge(spitter);
   }

   ...
}
```

Głównym wartym uwagi elementem na listingu 11.2 jest właściwość `EntityManager-Factory`. Jest ona oznaczona adnotacją `@PersistenceUnit`, co pozwala Springowi na wstrzyknięcie komponentu `EntityManagerFactory` do repozytorium. Mając już tę zależność, metody klasy `JpaSpitterRepository` wykorzystują ją do utworzenia menedżera encji i używają go do przeprowadzenia operacji na bazie danych.

Jedyną pułapką, jaka wiąże się w tej chwili z klasą `JpaSpitterRepository`, jest to, że każda z metod tej klasy kończy się wywołaniem metody `createEntityManager()`. Wiąże

się to z duplikacją kodu, a także powoduje utworzenie nowej instancji menedżera EntityManager przy każdym wywołaniu metody repozytorium. Komplikuje to kwestie związane z transakcjami. Dobrze by było mieć już instancję EntityManager gotową do użycia.

Problem z menedżerem EntityManager polega na tym, że nie jest on bezpieczny ze względu na wątki i nie powinien być wstrzykiwany do współdzielonych komponentów typu singleton, takich jak nasze repozytorium. Nie oznacza to jednak, że nie możemy prosić o jego instancję. Listing 11.3 pokazuje, jak wykorzystać adnotację @PersistenceContext do przekazania repozytorium JpaSpitterRepository instancji menedżera encji.

Listing 11.3. Wstrzykiwanie repozytorium za pomocą obiektu pośredniczącego do menedżera encji EntityManager

```
package com.habuma.spittr.persistence;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import com.habuma.spittr.domain.Spitter;
import com.habuma.spittr.domain.Spittle;

@Repository
@Transactional
public class JpaSpitterRepository implements SpitterRepository {
   @PersistenceContext
   private EntityManager em; <-- Wstrzykujemy menedżera encji
   public void addSpitter(Spitter spitter) {
      em.persist(spitter); <-- Używamy menedżera encji
   }

   public Spitter getSpitterById(long id) {
      return em.find(Spitter.class, id);
   }

   public void saveSpitter(Spitter spitter) {
      em.merge(spitter);
   }

   ...
}
```

Ta nowa wersja repozytorium JpaSpitterRepository otrzymuje teraz bezpośrednio instancję menedżera encji i nie ma potrzeby jego tworzenia za pomocą fabryki EntityManagerFactory w każdej metodzie. Jest to rozwiązanie dużo wygodniejsze, zastanawiasz się jednak prawdopodobnie nad potencjalnymi problemami związanymi z wykorzystaniem wątków, które mogą się pojawić przy pracy z wstrzykniętą instancją menedżera EntityManager.

Prawda jest taka, że adnotacja @PersistenceContext nie powoduje wstrzyknięcia menedżera EntityManager, a przynajmniej nie w sposób dosłowny. Wstrzykiwany jest nie prawdziwy obiekt EntityManager, tylko obiekt do niego pośredniczący. Prawdziwy menedżer EntityManager jest powiązany z bieżącą transakcją lub, jeśli ona nie istnieje, tworzy

nową instancję. Dzięki temu zawsze mamy możliwość pracy z menedżerem encji w sposób bezpieczny ze względu na wątki.

Warto wiedzieć, że adnotacje `@PersistenceUnit` i `@PersistenceContext` nie są adnotacjami Springa. Dostarczane są przez specyfikację JPA. Aby Spring miał możliwość ich zrozumienia i wstrzyknięcia fabryki `EntityManagerFactory` bądź `EntityManager`, musimy skonfigurować procesor `PersistenceAnnotationBeanPostProcessor`. Jeśli korzystamy już z elementów `<context:annotation-config>` bądź `<context:component-scan>`, wszystko jest gotowe, bo elementy te rejestrują automatycznie komponent `PersistenceAnnotationBean` ➥`PostProcessor`. W przeciwnym wypadku musimy zarejestrować ten komponent w sposób jawny:

```
@Bean
public PersistenceAnnotationBeanPostProcessor paPostProcessor() {
   return new PersistenceAnnotationBeanPostProcessor();
}
```

Być może nie uszły także Twojej uwadze adnotacje `@Repository` i `@Transactional`, użyte do oznaczenia `JpaSpitterRepository`. `@Transactional` mówi, że wszystkie metody związane z utrwalaniem danych w naszym repozytorium będą wykonywane w kontekście transakcji.

Rola `@Repository` jest natomiast taka sama jak w przypadku wersji z sesjami kontekstowymi Hibernate naszego repozytorium. Bez dokonującego tłumaczenia wyjątków szablonu, musimy oznaczyć nasze repozytorium adnotacją `@Repository`, dzięki czemu `PersistenceExceptionTranslationPostProcessor` będzie wiedzieć, że jest to komponent, dla którego wyjątki powinny zostać przetłumaczone na jeden zunifikowany wyjątek dostępu do danych Springa.

Skoro jesteśmy już przy `PersistenceExceptionTranslationPostProcessor`, musimy pamiętać o dowiązaniu go jako komponent w Springu, identycznie jak w przykładzie z Hibernate:

```
@Bean
public BeanPostProcessor persistenceTranslation() {
   return new PersistenceExceptionTranslationPostProcessor();
}
```

Warto zauważyć, że zarówno przy JPA, jak i przy Hibernate tłumaczenie wyjątków nie jest obowiązkowe. Jeżeli wolisz, żeby Twoje repozytorium zgłaszało wyjątki JPA lub Hibernate, możesz pominąć użycie `PersistenceExceptionTranslationPostProcessor` i pozwolić na swobodny przepływ wyjątków. Ale jeżeli używasz tłumaczenia wyjątków, możesz zjednoczyć wszystkie związane z dostępem do danych wyjątki w ramach hierarchii wyjątków Springa. To ułatwi wymianę mechanizmu utrwalania danych w przyszłości.

11.3. Automatyczne repozytoria z wykorzystaniem Spring Data

Chociaż metody przedstawione na listingach 11.2 i 11.3 są względnie proste, to zapytania do bazy wykonywane są wciąż bezpośrednio z wykorzystaniem obiektu `EntityManager`. Chwila obserwacji wystarczy, aby zobaczyć, że w dalszym ciągu jest tam sporo nadmiarowego kodu. Na przykład przyjrzyjmy się metodzie `addSpitter()`:

```
public void addSpitter(Spitter spitter) {
   entityManager.persist(spitter);
}
```

W każdej niebanalnej aplikacji pojawi się wiele metod niemal identycznych jak ta powyższa. Założę się, że niejednokrotnie tworzyłeś już podobne metody, ale zapisywałeś pewnie obiekty innego typu. Pozostałe metody w repozytorium `JpaSpitterRepository` też nie są zbyt odkrywcze. Typy domenowe będą inne, lecz takie metody występują w niemal wszystkich tworzonych repozytoriach.

Czy musimy wielokrotnie tworzyć te same metody utrwalania tylko dlatego, że korzystamy z różnych klas domenowych? Spring Data JPA pozwala zerwać z tym szaleństwem. Zamiast wielokrotnego tworzenia tych samych implementacji repozytorium Spring Data wymaga od nas jedynie utworzenia interfejsów. Nie jest potrzebna żadna implementacja.

Przykładowo spójrzmy na listing 11.4, na poniższy interfejs `SpitterRepository`.

Listing 11.4. Tworzenie repozytorium na podstawie definicji interfejsu z wykorzystaniem Spring Data

```
public interface SpitterRepository extends
   JpaRepository<Spitter, Long> { }
```

W tym momencie repozytorium `SpitterRepository` nie wydaje się zbyt użyteczne. Ale to tylko pozory.

Kluczem do tworzenia repozytoriów Spring Data JPA jest rozszerzenie jednego z dostępnych interfejsów. `SpitterRepository` rozszerza interfejs Spring Data JPA `Jpa➥Repository` (o kilku z pozostałych interfejsów powiem za chwilę). `JpaRepository` jest sparametryzowane, dzięki czemu mogliśmy zapisać, że jego przeznaczeniem jest utrwalanie obiektów typu `Spitter`, a ich identyfikator jest typu `Long`. Dziedziczy też 18 metod służących do przeprowadzania najczęstszych operacji utrwalania, takich jak zapisywanie obiektów `Spitter`, ich usuwania i wyszukiwania w oparciu o identyfikator.

Teraz mogłoby się wydawać, że kolejnym krokiem będzie utworzenie klasy, która implementuje wszystkie te 18 metod. Oznaczałoby to, że czeka nas bardzo nudny rozdział. Na szczęście nie będziemy tworzyć żadnej implementacji repozytorium `Spitter➥Repository`. Pozwolimy, żeby Spring Data wykonał tę pracę za nas. Musimy o to tylko poprosić.

W celu utworzenia implementacji repozytorium `SpitterRepository` musimy dodać pojedynczy element do konfiguracji Springa. Listing 11 5 przedstawia konfigurację niezbędną do włączenia mechanizmów Spring Data JPA.

Listing 11.5. Konfigurujemy Spring Data JPA

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xsi:schemaLocation="http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd">
```

```
    <jpa:repositories base-package="com.habuma.spittr.db" />
    ...
</beans>
```

Element `<jpa:repositories>` odpowiada za włączenie całej magii Spring Data JPA. Podobnie jak element `<context:component-scan>`, element `<jpa:repositories>` przyjmuje parametr `base-package`. Podczas gdy element `<context:component-scan>` skanuje pakiet (i jego pakiety potomne) w poszukiwaniu klas oznaczonych adnotacją `@Component`, element `<jpa:repositories>` skanuje pakiet bazowy w poszukiwaniu interfejsów rozszerzających interfejs Spring Data JPA `Repository`. Po wyszukaniu interfejsu rozszerzającego `Repository` automatycznie generuje (przy starcie aplikacji) implementację tego interfejsu.

Nie musimy jednak korzystać z elementu `<jpa:repositories>` i używać w tym celu adnotacji `@EnableJpaRepositories` w klasie konfiguracji Javy. Poniżej znajduje się klasa konfiguracji opatrzona adnotacją `@EnableJpaRepositories`, która ma skanować pakiet `com.habuma.spittr.db`:

```
@Configuration
@EnableJpaRepositories(basePackages="com.habuma.spittr.db")
public class JpaConfiguration {
    ...
}
```

Powróćmy do interfejsu `SpitterRepository`. Rozszerza on interfejs `JpaRepository`, a ten z kolei rozszerza interfejs podstawowy `Repository` (choć nie bezpośrednio). W ten sposób `SpitterRepository` również rozszerza interfejs `Repository`, wyszukiwany podczas skanowania repozytorium. Po odnalezieniu interfejsu przez mechanizmy Spring Data, tworzona jest implementacja `SpitterRepository`, zawierająca deklarację wszystkich 18 metod dziedziczonych z interfejsu `JpaRepository`, `PagingAndSortingRepository` oraz `CrudRepository`.

Istotne jest zrozumienie, że implementacja repozytorium tworzona jest przy uruchamianiu aplikacji, w trakcie tworzenia kontekstu aplikacji Springa. Nie jest wynikiem generowania kodu podczas budowania aplikacji ani nie powstaje przy wywoływaniu metod interfejsu.

Niezłe, co?

Niesamowite jest to, że uzyskujemy dostęp do 18 wygodnych operacji na obiektach Spitter bez potrzeby tworzenia tego kodu utrwalania. Co jednak, jeśli potrzebne jest nam coś więcej, niż te 18 metod jest nam w stanie zaoferować? Na szczęście Spring Data JPA udostępnia kilka sposobów dodania własnych metod do repozytorium. Zobaczmy więc, jak zdefiniować własne zapytanie z wykorzystaniem Spring Data JPA.

11.3.1. Definiujemy metody zapytań

Jedną z funkcjonalności, której wymagamy od naszego repozytorium `SpitterRepository`, jest możliwość wyszukiwania obiektu `Spitter` w oparciu o nazwę użytkownika. Przykładowo zmodyfikujemy interfejs `SpitterRepository` w następujący sposób:

```
public interface SpitterRepository
      extends JpaRepository<Spitter, Long> {
   Spitter findByUsername(String username);
}
```

Nowa wersja metody `findByUsername()` jest prosta i powinna spełnić nasze oczekiwania. Jak teraz wykorzystać Spring Data JPA do utworzenia implementacji tej metody?

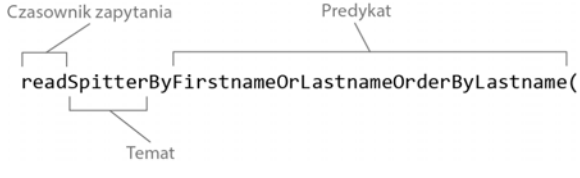
W zasadzie nic więcej nie musimy robić. Sygnatura metody zawiera wystarczająco dużo informacji do utworzenia implementacji metody.

Przy tworzeniu implementacji repozytorium Spring Data bada wszystkie metody interfejsu repozytorium, przetwarza ich nazwę i próbuje zrozumieć cel metody w kontekście utrwalanego obiektu. Mówiąc w skrócie, Spring Data definiuje pewnego rodzaju miniaturowy język DSL (*domain-specific language*), w którym szczegóły utrwalania wyrażone są w sygnaturze metody repozytorium.

Spring Data wie, że zdefiniowana przez nas metoda służy do wyszukiwania obiektów `Spitter`, ponieważ jako jeden z parametrów interfejsu `JpaRepository` ustaliliśmy typ `Spitter`. Nazwa metody, `findByUsername`, jasno wskazuje na fakt, że metoda ma wyszukiwać obiekty `Spitter` poprzez przyrównanie nazwy właściwości `username` z parametrem przekazanym do metody. Co więcej, sygnatura metody wskazuje na to, że oczekujemy pobrania jednego obiektu `Spitter`, a nie ich kolekcji. Pobrany więc zostanie jeden obiekt `Spitter`, którego nazwa odpowiada wartości parametru.

Metoda `findByUsername()` jest prosta, ale Spring Data poradzi sobie nawet z bardziej interesującymi nazwami metod. W repozytorium składają się one z czasownika, opcjonalnego tematu, słowa *By* oraz predykatu. W przypadku metody `findByUsername()` czasownikiem jest *find* (z ang. *szukać*), a predykatem *Username*, temat zaś nie jest określony i w domyśle jest nim `Spitter`.

Spójrzmy na inny przykład tworzenia nazw metod w repozytorium. Rozważmy, jak odwzorowywane są elementy nazwy metody `readSpitterByFirstnameOrLastname()`. Rysunek 11.1 ilustruje rozbicie tej nazwy na poszczególne elementy.



Rysunek 11.1. Metody repozytorium nazywane są w oparciu o wzorzec umożliwiający Spring Data wygenerowanie zapytań do bazy danych

Jak widać, czasownikiem jest *read* (z ang. *czytać*) — w odróżnieniu od *find* z poprzedniego przykładu. Spring Data umożliwia użycie w nazwie metody czterech czasowników: *get*, *read*, *find* oraz *count*. Czasowniki *get*, *read* i *find* są synonimami. Wszystkie trzy służą do definiowania metod odpytujących o dane i zwracających obiekty. Z kolei czasownik *count* (z ang. *liczyć*) zwraca liczbę pasujących obiektów, a nie same obiekty.

Temat metody repozytorium jest opcjonalny. Jego głównym zadaniem jest danie pewnej swobody w nazywaniu metod. Jeśli preferujesz więc nazwę metody w postaci `read ↪SpitterByFirstnameOrLastname`, a nie `readByFirstnameOrLastname`, możesz taką zastosować.

Temat jest niemal zawsze ignorowany i metoda `readSpitterByFirstnameOrLastname` nie różni się niczym od metody `readByFirstnameOrLastname`, a ta od metody `readThoseThings ➥WeWantByFirstnameOrLastname`[1]. Typ pobieranego obiektu określany jest bowiem na podstawie parametru interfejsu `JpaRepository`, a nie tematu w nazwie metody.

Istnieje tylko jeden wyjątek od reguły ignorowania tematu. Jeśli temat zaczyna się od słowa *distinct* (z ang. *różny*), to wygenerowane zapytanie zapewni unikalność otrzymanego zbioru wyników.

Najbardziej interesującym elementem nazwy metody jest predykat. Służy on do określania właściwości zawężających zbiór zwracanych danych. Każdy warunek musi się odnosić do jakiejś właściwości, a dodatkowo może określać operator porównania. Jeżeli zostanie on pominięty, domyślnie zastosowana będzie operacja równości. Możemy jednak wybrać dowolny inny operator porównania, taki jak:

- `IsAfter`, `After`, `IsGreaterThan`, `GreaterThan`;
- `IsGreaterThanEqual`, `GreaterThanEqual`;
- `IsBefore`, `Before`, `IsLessThan`, `LessThan`;
- `IsLessThanEqual`, `LessThanEqual`;
- `IsBetween`, `Between`;
- `IsNull`, `Null`;
- `IsNotNull`, `NotNull`;
- `IsIn`, `In`;
- `IsNotIn`, `NotIn`;
- `IsStartingWith`, `StartingWith`, `StartsWith`;
- `IsEndingWith`, `EndingWith`, `EndsWith`;
- `IsContaining`, `Containing`, `Contains`;
- `IsLike`, `Like`;
- `IsNotLike`, `NotLike`;
- `IsTrue`, `True`;
- `IsFalse`, `False`;
- `Is`, `Equals`;
- `IsNot`, `Not`.

Wartościami, do których nastąpi porównanie, są parametry metody. Pełna sygnatura naszej metody wygląda następująco:

```
List<Spitter> readByFirstnameOrLastname(String first, String last);
```

Jeśli korzystamy z wartości typu String, regułami porównania mogą być też `Ignoring ➥Case` lub `IgnoresCase`, co umożliwia przeprowadzenie porównania z pominięciem wielkości liter w wartościach właściwości `firstname` oraz `lastname`. Przykładowo naszą metodę możemy przerobić tak:

```
List<Spitter> readByFirstnameIgnoringCaseOrLastnameIgnoresCase(
      String first, String last);
```

[1] Co można by przetłumaczyć jako: *Odczytajmy wszystko, co chcemy, z użyciem imienia lub nazwiska — przyp. tłum.*

Warto zwrócić uwagę, że `IgnoringCase` i `IgnoresCase` są synonimami. Możemy wybrać tę nazwę, która bardziej nam odpowiada.

Alternatywą dla warunków `IgnoringCase/IgnoresCase` jest także pominięcie rozróżniania wielkości liter we wszystkich warunkach poprzez wstawienie na ich końcu warunku `AllIgnoringCase` lub `AllIgnoresCase`:

```
List<Spitter> readByFirstnameOrLastnameAllIgnoresCase(
      String first, String last);
```

Nazwy parametrów metody są nieistotne, ich kolejność musi jednak odpowiadać porównaniom ustalonym w nazwie metody.

Na zakończenie mamy możliwość posortowania wyników poprzez dodanie na końcu nazwy `OrderBy`. Przykładowo możemy posortować wyniki rosnąco z wykorzystaniem właściwości `lastname`:

```
List<Spitter> readByFirstnameOrLastnameOrderByLastnameAsc(
      String first, String last);
```

Aby posortować wyniki w oparciu o wiele właściwości, musimy kolejno dodać ich nazwy za tekstem `OrderBy`. Na przykład poniższa metoda umożliwia posortowanie wyników rosnąco — za pomocą właściwości `lastname`, a następnie malejąco — z użyciem właściwości `firstname`:

```
List<Spitter> readByFirstnameOrLastnameOrderByLastnameAscFirstnameDesc(
   String first, String last);
```

Jak już mieliśmy okazję zaobserwować, warunki rozdzielone są słowami `And` (z ang. *i*) oraz `Or` (z ang. *lub*).

Niemożliwe (albo przynajmniej *bardzo* trudne) byłoby podanie pełnej listy typów metod, które można utworzyć z wykorzystaniem ich konwencji nazewniczej. Poniżej przedstawiłem kilka kolejnych przykładów metod stosujących się do przyjętych konwencji:

- `List<Pet> findPetsByBreedIn(List<String> breed;`
- `int countProductsByDiscontinuedTrue();`
- `List<Order> findByShippingDateBetween(Date start, Date end).`

To tylko przedsmak możliwości dotyczących deklaracji metod, dla których Spring Data JPA jest w stanie przygotować gotową implementację. W tej chwili musisz jedynie wiedzieć, że dzięki odpowiednio nadanej nazwie metody Spring Data JPA wygeneruje implementację umożliwiającą tworzenie naprawdę złożonych zapytań do bazy danych.

Mini DSL oferowany przez Spring Data ma jednak swoje ograniczenia i w przypadku niektórych zapytań jego użycie może być niewygodne czy nawet niemożliwe. Spring Data daje nam w takich właśnie sytuacjach wsparcie w postaci adnotacji `@Query`.

11.3.2. Deklarujemy własne zapytania

Przypuśćmy, że chcemy utworzyć metodę repozytorium, która umożliwi wyszukiwanie wszystkich spittersów korzystających ze skrzynki pocztowej Gmail. Jednym ze sposobów byłoby zdefiniowanie metody `findByEmailLike()` i przekazanie do niej jako

parametru tekstu `%gmail.com` za każdym razem, gdy chcemy wyszukać tych użytkowników. Ciekawszym rozwiązaniem byłoby jednak zdefiniowanie wygodniejszej w użyciu metody `findAllGmailSpitters()`, która nie wymaga przekazania fragmentu adresu e-mail:

```
List<Spitter> findAllGmailSpitters();
```

Niestety metoda ta nie stosuje się do konwencji nazewniczej Spring Data. Kiedy Spring Data podejmie próbę wygenerowania implementacji dla tej metody, nie zdoła dopasować elementów nazwy metody do metadanych klasy `Spitter`, co doprowadzi do wyrzucenia wyjątku.

W sytuacjach, gdy nie mamy możliwości sformułowania właściwego zapytania za pomocą nazwy metody, możemy skorzystać z adnotacji `@Query` i dostarczyć Spring Data zapytanie, które ma zostać wywołane. W przypadku metody `findAllGmailSpitters()` zapytanie to może wyglądać następująco:

```
@Query("select s from Spitter s where s.email like '%gmail.com'") List<Spitter>
↪findAllGmailSpitters();
```

W dalszym ciągu nie tworzymy implementacji metody `findAllGmailSpitters()`. Przekazujemy tylko zapytanie, podpowiadając Spring Data JPA, jak ma zaimplementować tę metodę.

Jak widzieliśmy wcześniej, adnotacja `@Query` jest użyteczna wtedy, gdy wyrażenie zapytania za pomocą konwencji nazewniczej jest trudne. Może się też przydać w sytuacji, kiedy trzymanie się konwencji nazewniczej powoduje powstanie metody o bardzo długiej nazwie. Rozważmy dla przykładu następującą metodę:

```
List<Order>
   findByCustomerAddressZipCodeOrCustomerNameAndCustomerAddressState();
```

To dopiero długa nazwa! Musiałem rozdzielić definicję metody po podaniu typu zwracanego, żeby nazwa w ogóle zmieściła się w ramach marginesów książki.

Przykład jest zmyślony, ale i w produkcyjnym kodzie może się pojawić potrzeba przygotowania metody repozytorium służącej do przeprowadzenia zapytania, którego zapisanie wymaga utworzenia metody o wyjątkowo długiej nazwie. Wówczas wolelibyśmy zapewne użyć krótszej nazwy, a zapytanie do bazy zdefiniować za pomocą adnotacji `@Query`.

Adnotacja `@Query` przydaje się przy tworzeniu własnych metod zapytań w interfejsach Spring Data JPA. Pozwala jednak na zastosowanie tylko pojedynczego zapytania JPA. Co zrobić w sytuacji, gdy musimy przeprowadzić bardziej złożoną operację, składającą się z więcej niż jednego zapytania?

11.3.3. Dodajemy własne funkcjonalności

Bardzo prawdopodobne jest, że w pewnym momencie pojawi się potrzeba przygotowania funkcjonalności, której nie można zapisać przy użyciu konwencji nazewniczej metod Spring Data ani nawet za pomocą adnotacji `@Query`. Spring Data JPA ma wspa-

niałe możliwości, ale i on ma pewne ograniczenia i czasem musimy utworzyć metodę repozytorium w starym stylu — korzystając bezpośrednio z menedżera `EntityManager`. Gdy więc zajdzie taka potrzeba, czy mamy zrezygnować ze Spring Data JPA i powrócić do ręcznego tworzenia repozytoriów, jak w podrozdziale 11.2.2?

W skrócie — tak. Kiedy musimy zrobić coś, czego Spring Data JPA nie potrafi, konieczne jest zejście do niższego poziomu niż ten oferowany przez Spring Data JPA. Dobrą wiadomością jest to, że nie musimy całkowicie rezygnować ze Spring Data JPA. Musimy to tylko uczynić w tych metodach, w których jest to niezbędne. W dalszym ciągu możemy natomiast pozostawić w rękach Spring Data te zadania, z którymi jest w stanie sobie poradzić.

Gdy Spring Data JPA generuje implementacje interfejsów dla repozytoriów, wyszukuje też klasy o takich samych nazwach jak nazwy interfejsów, ale zakończone sufiksem `Impl`. Jeśli taka klasa istnieje, Spring Data JPA łączy w jedną całość jej metody z metodami wygenerowanymi przez Spring Data JPA. W przypadku interfejsu `SpitterRepo`
`↪sitory` nazwą wyszukiwanej klasy jest `SpitterRepositoryImpl`.

Aby zilustrować tę sytuację, przypuśćmy, że potrzebna jest nam metoda w repozytorium `SpitterRepository` aktualizująca wszystkich spittersów, którzy wystawili przynajmniej 10 000 spittle'ów, i przydzielająca im status przynależności do grona elity użytkowników. Nie ma dobrego sposobu na zadeklarowanie takiej metody z wykorzystaniem konwencji nazewniczej metod w Spring Data JPA czy adnotacji `@Query`. Najpraktyczniejsze jest w tym przypadku zdefiniowanie metody `eliteSweep()` przedstawionej na listingu 11.6.

Listing 11.6. Repozytorium, które przydziela aktywnym użytkownikom serwisu Spitter status Elite

```
public class SpitterRepositoryImpl implements SpitterSweeper {
   @PersistenceContext
   private EntityManager em;

   public int eliteSweep() {
      String update =
         "UPDATE Spitter spitter " +
         "SET spitter.status = 'Elite' " +
         "WHERE spitter.status = 'Newbie' " +
         "AND spitter.id IN (" +
         "SELECT s FROM Spitter s WHERE (" +
         " SELECT COUNT(spittles) FROM s.spittles spittles) > 10000" +
         ")";
      return em.createQuery(update).executeUpdate();
   }
}
```

Jak widać, metoda `eliteStatus()` nie różni się zbytnio od pozostałych metod repozytorium, które utworzyliśmy w sekcji 11.2.2. Klasa `SpitterRepositoryImpl` nie odznacza się niczym szczególnym.

Zauważmy, że klasa `SpitterRepositoryImpl` nie implementuje interfejsu `SpitterRepository`. Odpowiedzialność za jego implementację spoczywa wciąż na Spring Data JPA. `SpitterRepositoryImpl` implementuje interfejs `SpitterSweeper`, który z naszym repozytorium Spring Data łączy jedynie nazwa i który wygląda następująco:

```
public interface SpitterSweeper{
   int eliteSweep();
}
```

Musimy także sprawić, aby metoda `eliteSweep()` była zadeklarowana w interfejsie `SpitterRepository`. Najprostszym sposobem, by to osiągnąć i uniknąć duplikowania kodu, jest zmiana interfejsu `SpitterRepository` tak, żeby rozszerzał `SpitterSweeper`:

```
public interface SpitterRepository extends
      JpaRepository<Spitter, Long>, SpitterSweeper {
   ...
}
```

Jak wspomniałem wcześniej, Spring Data JPA wiąże implementację klasy z interfejsem w oparciu o nazwę implementacji utworzoną na podstawie nazwy interfejsu. Sufiks `Impl` jest jednak tylko wartością domyślną. Jeśli chcielibyśmy użyć jakiegoś innego sufiksu, musimy go określić w konfiguracji, ustalając w adnotacji `@EnableJpaRepositories` wartość atrybutu `repositoryImplementationPostfix`:

```
@EnableJpaRepositories(
      basePackages="com.habuma.spittr.db",
      repositoryImplementationPostfix="Helper")
```

W przypadku konfiguracji Spring Data JPA opartej na plikach XML z wykorzystaniem elementu `<jpa:repositories>` odpowiedni sufiks możemy określić za pomocą atrybutu `repository-impl-postfix`:

```
<jpa:repositories base-package="com.habuma.spittr.db"
      repository-impl-postfix="Helper" />
```

Po ustawieniu wartości sufiksu na `Helper` jako uzupełnienie interfejsu `SpitterRepository` Spring Data JPA wyszukiwana będzie klasa o nazwie `SpitterRepositoryHelper`.

11.4. Podsumowanie

Przez wiele lat i w wielu aplikacjach bazy relacyjne były jedynym właściwym rozwiązaniem stosowanym do przechowywania danych. Gdy praca z JDBC i odwzorowywaniem obiektów do tabel jest zbyt żmudna, z pomocą przychodzą nam rozwiązania ORM, takie jak Hibernate i JPA, które udostępniają bardziej deklaratywny model utrwalania danych. Chociaż Spring nie oferuje bezpośredniego wsparcia dla systemów ORM, integruje się dobrze z kilkoma popularnymi rozwiązaniami ORM, wliczając w to Hibernate i Java Persistance API.

W tym rozdziale dowiedziałeś się, jak wykorzystać sesje kontekstowe Hibernate w aplikacji springowej, tak aby powstałe repozytoria były w minimalnym stopniu zależne od Springa lub od niego całkowicie niezależne. Zobaczyłeś też, jak utworzyć niezależne

od Springa repozytoria JPA dzięki wstrzyknięciu instancji `EntityManagerFactory` bądź `EntityManager` do implementacji tych repozytoriów.

Następnie spojrzeliśmy po raz pierwszy na Spring Data i dowiedziałeś się, jak zadeklarować interfejsy repozytoriów JPA, żeby umożliwić Spring Data automatyczne generowanie implementacji tych interfejsów w trakcie działania aplikacji. A kiedy te wygenerowane metody repozytorium nie oferują nam wystarczającej funkcjonalności, możemy pomóc Spring Data, używając adnotacji `@Query` i tworząc własną implementację metod repozytorium, co również zobaczyłeś w tym rozdziale.

Ale to zaledwie ułamek możliwości oferowanych przez Spring Data. W następnym rozdziale jeszcze bardziej zagłębimy się w temat języka nazewnictwa metod DSL i przekonamy się, że Spring Data sprawdza się nie tylko w bazach relacyjnych. Zobaczymy bowiem, w jaki sposób Spring Data wspiera wysyp nowych baz NoSQL, które w ostatnich latach tak mocno zyskały na popularności.

12 Pracujemy z bazami NoSQL

W tym rozdziale omówimy:

- Tworzenie repozytoriów opartych na bazach MongoDB i Neo4j
- Przechowywanie danych w wielu bazach
- Pracę ze Springiem i Redisem

W swojej autobiografii Henry Ford zasłynął powiedzeniem: „Może sobie pan zażyczyć samochód w każdym kolorze, byle by był to czarny”[1]. Niektórzy uznają to stwierdzenie za aroganckie i zdradzające przekonanie o nieomylności jego autora. Inni mogą w nim widzieć przejaw poczucia humoru. W rzeczywistości jednak zostało ono wypowiedziane w okresie znaczącej redukcji kosztów, czego wyrazem było użycie farby szybkoschnącej, dostępnej jedynie w kolorze czarnym.

Parafrazując słynne powiedzenie Forda i przenosząc na dziedzinę wyboru bazy danych, można powiedzieć, że przez lata słyszeliśmy, iż możemy użyć dowolnej bazy danych, pod warunkiem że jest to baza relacyjna. Przez długi czas bazy relacyjne miały niemal monopol w tworzonym oprogramowaniu.

Ostatnio ten monopol zaczął jednak słabnąć wraz z pojawieniem się silnych pretendentów do roli nowych liderów. W aplikacjach produkcyjnych wykorzystywane są coraz częściej tak zwane bazy NoSQL. Okazało się, że nie ma jednej, uniwersalnej bazy rozwiązującej każdy problem. Obecnie mamy większe pole manewru i możemy wybrać to rozwiązanie, które najlepiej się sprawdzi w danym zadaniu.

[1] Henry Ford, *Moje życie i dzieło*, Warszawa 1925.

W kilku ostatnich rozdziałach zajmowaliśmy się bazami relacyjnymi, zaczynając od obsługi JDBC w Springu, a następnie systemami ORM (*object-relation mapping*). W szczególności w poprzednim rozdziale poznałeś Spring Data JPA, jeden z kilku projektów rozwijanych w ramach zbiorczego projektu Spring Data. Dowiedziałeś się, w jaki sposób Spring Data JPA ułatwia pracę z JPA poprzez automatyczne generowanie implementacji repozytorium w trakcie działania aplikacji.

Spring Data wspiera również kilka baz NoSQL, wliczając w to MongoDB, Neo4j oraz Redisa. Wsparcie to obejmuje nie tylko automatyczne repozytoria, ale także dostęp do danych w oparciu o szablony i odwzorowania z użyciem adnotacji. W tym rozdziale dowiesz się, jak tworzyć repozytoria współpracujące z bazami nierelacyjnymi, bazami NoSQL. Rozpoczniemy od Spring Data MongoDB i zobaczymy, jak tworzyć repozytoria działające na danych w postaci dokumentów.

12.1. Zapisujemy dane w MongoDB

Najlepszą reprezentacją niektórych danych są dokumenty. Oznacza to, że naszych danych nie rozpraszamy po wielu tabelach, węzłach lub encjach, bo większy sens ma zebranie wszystkich informacji w nieznormalizowanej strukturze (zwanej dokumentem). Chociaż dwa dokumenty (albo więcej) mogą być ze sobą jakoś powiązane, to w większości przypadków dokumenty są niezależnymi bytami. Bazy danych, które są w szczególny sposób przystosowane do pracy z dokumentami, nazywane są bazami dokumentowymi.

Przykładowo przypuśćmy, że tworzymy aplikację przechowującą spis ocen uczniów. Musimy mieć możliwość pobierania ocen na podstawie nazwiska ucznia lub przeszukania spisu pod kątem jakichś wspólnych właściwości. Każdy uczeń jest oceniany indywidualnie, nie ma więc potrzeby, żeby dwa spisy ocen były ze sobą w jakiejś relacji. I chociaż moglibyśmy utworzyć schemat bazy relacyjnej (i prawdopodobnie ktoś już go przygotował) do przechowywania takich danych, być może to właśnie baza dokumentowa jest w tym przypadku lepszym rozwiązaniem.

Kiedy baza dokumentowa nie jest dobrym rozwiązaniem

Wiedza o tym, kiedy użyć bazy dokumentowej, jest bardzo istotna. Równie istotna jest wiedza o tym, kiedy użycie bazy dokumentowej nie ma sensu. Bazy dokumentowe nie są bazami ogólnego przeznaczenia i specjalizują się w rozwiązywaniu ograniczonej liczby problemów.

Bazy dokumentowe nie są przystosowane do przechowywania danych powiązanych ze sobą w dużym stopniu w ramach wzajemnych relacji. Przykładowo sieć społecznościowa reprezentuje wzajemne relacje różnych użytkowników i przechowywanie jej w bazie dokumentowej nie jest najlepszym rozwiązaniem. Przechowywanie takich danych w bazie dokumentowej nie jest niemożliwe, ale wyzwania, jakie się przy tym pojawiają, skutecznie niwelują korzyści odniesione z wykorzystania tej bazy.

Domena aplikacji Spittr nie jest odpowiednia dla bazy dokumentowej. W tym rozdziale spojrzymy na MongoDB w kontekście systemu zamówień.

MongoDB jest oprogramowaniem otwartym i jedną z najpopularniejszych baz dokumentowych. Spring Data MongoDB umożliwia wykorzystanie MongoDB w aplikacjach Springa na trzy sposoby:

- jako adnotacje służące do odwzorowania obiektów na dokumenty;
- jako dostęp do bazy danych w oparciu o szablony z użyciem `MongoTemplate`;
- jako automatyczne generowanie repozytorium w czasie działania.

Widzieliśmy już, jak Spring Data JPA włącza mechanizm automatycznego generowania repozytorium dla bazy opartej na JPA. Spring Data MongoDB udostępnia analogiczny mechanizm dla bazy MongoDB.

Jednak w przeciwieństwie do Spring Data JPA, Spring Data MongoDB udostępnia też adnotacje do odwzorowywania obiektów na dokumenty. (Spring Data JPA nie potrzebuje takich adnotacji dla JPA, ponieważ specyfikacja JPA sama w sobie definiuje adnotacje odwzorowujące obiekty na relacje).

Zanim będziemy mogli skorzystać z tych funkcji, musimy skonfigurować Spring Data MongoDB.

12.1.1. Włączamy MongoDB

Do efektywnej pracy ze Spring Data MongoDB musimy skonfigurować kilka kluczowych komponentów. Na początek trzeba ustawić komponent `MongoClient` umożliwiający dostęp do bazy MongoDB. Potrzebny nam też będzie komponent MongoTemplate, który pozwala na dostęp do bazy w oparciu o szablony. Opcjonalne, ale mocno zalecane, jest włączenie automatycznego generowania repozytorium Spring Data MongoDB.

Listing 12.1 zawiera przykład prostej klasy konfiguracji Spring Data MongoDB, która spełnia opisane wymagania.

Listing 12.1. Podstawowa konfiguracja Spring Data MongoDB

```
package orders.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.core.MongoFactoryBean;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.repository.config.
   EnableMongoRepositories;

import com.mongodb.Mongo;

@Configuration
@EnableMongoRepositories(basePackages="orders.db") <-- Włączamy repozytorium MongoDB
public class MongoConfig {
   @Bean
   public MongoFactoryBean mongo() { <-- Komponent MongoClient
      MongoFactoryBean mongo = new MongoFactoryBean();
      mongo.setHost("localhost");
      return mongo;
   }

   @Bean
   public MongoOperations mongoTemplate(Mongo mongo) { <-- Komponent MongoTemplate
      return new MongoTemplate(mongo, "OrdersDB");
   }
}
```

W poprzednim rozdziale włączyliśmy mechanizm automatycznego generowania repozytorium Spring Data JPA za pomocą adnotacji `@EnableJpaRepositories`. Adnotacja `@Enable`
`↪MongoRepositories` pełni tę samą funkcję w MongoDB.

Poza adnotacją `@EnableMongoRepositories` listing 12.1 dostarcza również dwie metody tworzenia komponentów. Pierwsza metoda wykorzystuje obiekt `MongoFactoryBean` do utworzenia instancji `Mongo`. Komponent ten służy do łączenia Spring Data MongoDB z właściwą bazą danych (pełni podobną funkcję jak źródło danych `DataSource` przy pracy z bazą relacyjną). Chociaż moglibyśmy utworzyć instancję `Mongo` bezpośrednio za pomocą klasy `MongoClient`, musielibyśmy samodzielnie obsłużyć wyjątek `UnknownHost`
`↪Exception` rzucany przez konstruktor klasy `MongoClient`. Fabryka `MognoFactoryBean` konstruuje obiekt `Mongo` za nas i nie zmusza nas do obsłużenia tego wyjątku.

Kolejna metoda deklaruje komponent `MongoTemplate`. Otrzymuje on referencję do obiektu `Mongo` utworzonego z użyciem drugiej metody i nazwy bazy danych. Za chwilę dowiesz się, jak wykorzystać `MongoTemplate` do odpytania bazy danych. Nawet jeśli nigdy nie stosujemy obiektu `MongoTemplate` bezpośrednio, potrzebujemy go do automatycznego generowania repozytoriów, które korzystają z niego w swej implementacji.

Nie musimy deklarować tych komponentów w sposób bezpośredni. Możemy rozszerzyć klasę abstrakcyjną `AbstractMongoConfiguration` i nadpisać jego metody `getData`
`↪baseName()` i `mongo()`. Na listingu 12.2 pokazuję, jak to zrobić.

Listing 12.2. Włączamy Spring Data MongoDB za pomocą adnotacji @EnableMongoRepositories

```
package orders.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.config.
   AbstractMongoConfiguration;
import org.springframework.data.mongodb.repository.config.
   EnableMongoRepositories;
import com.mongodb.Mongo;
import com.mongodb.MongoClient;

@Configuration
@EnableMongoRepositories("orders.db")
public class MongoConfig extends AbstractMongoConfiguration {

   @Override
   protected String getDatabaseName() {          <-- Określamy nazwę bazy danych
      return "OrdersDB";
   }

   @Override
   public Mongo mongo() throws Exception {       <-- Tworzymy klienta Mongo
      return new MongoClient();
   }
}
```

Nowa klasa konfiguracji jest równoważna tej przedstawionej na listingu 12.1, choć trochę prostsza. Najbardziej zauważalną różnicą jest to, że ta klasa konfiguracji nie deklaruje bezpośrednio komponentu `MongoTemplate`, chociaż jest on niejawnie tworzony.

Zamiast tego nadpisujemy metodę `getDatabaseName()`, aby podać nazwę bazy danych. Metoda `mongo()` wciąż tworzy instancję `MongoClient`, ale ponieważ klasa ta rzuca wyjątkiem, możemy z nią pracować bezpośrednio bez użycia `MongoFactoryBean`.

W tej chwili obie wersje konfiguracji z listingów 12.1 i 12.2 dostarczają działającą konfigurację Spring Data MongoDB. Jest ona działająca pod warunkiem, że serwer MongoDB działa na maszynie lokalnej. Jeśli serwer MongoDB działa na innym serwerze, możemy go wskazać przy tworzeniu klienta `MongoClient`:

```
public Mongo mongo() throws Exception {
   return new MongoClient("mongodbserver");
}
```

Możliwe jest też, że serwer MongoDB nie działa na domyślnym porcie (27017). W takim przypadku możemy także przy tworzeniu klienta `MongoClient` wskazać właściwy port:

```
public Mongo mongo() throws Exception {
   return new MongoClient("mongodbserver", 37017);
}
```

Mam również nadzieję, że jeżeli serwer MongoDB działa w środowisku produkcyjnym, to włączone zostało uwierzytelnianie. W takiej sytuacji w celu uzyskania dostępu do bazy danych musimy dostarczyć dane uwierzytelniające. Dostęp do zabezpieczonego serwera MongoDB jest trochę bardziej złożony, co pokazuje listing 12.3.

Listing 12.3. Tworzenie klienta MongoClient w celu uzyskania dostępu do zabezpieczonego serwera MongoDB

```
@Autowired
private Environment env;

@Override
public Mongo mongo() throws Exception {
   MongoCredential credential =
   MongoCredential.createMongoCRCredential(          <-- Tworzymy dane uwierzytelniania MongoDB
      env.getProperty("mongo.username"),
      "OrdersDB",
      env.getProperty("mongo.password").toCharArray());
   return new MongoClient(                           <-- Tworzymy klienta MongoClient
      new ServerAddress("localhost", 37017),
      Arrays.asList(credential));
}
```

Aby klient `MongoClient` mógł uzyskać dostęp do zabezpieczonego serwera, musi zostać utworzony z użyciem listy danych uwierzytelniających `MongoCredentails`. Na listingu 12.3 tworzymy w tym celu pojedynczy obiekt klasy `MongoCredential`. By oddzielić dane uwierzytelniające od klasy konfiguracji, odczytywane są one z użyciem wstrzykniętego obiektu środowiska `Environment`.

Spring Data MongoDB może też być konfigurowany za pomocą XML. Jak już chyba wiesz, preferuję wykorzystywanie konfiguracji zapisanej w klasach Javy. Jeśli masz jednak słabość do konfiguracji w plikach XML, przykład na listingu 12.4 pozwoli Ci odtworzyć naszą konfigurację Spring Data MongoDB z użyciem przestrzeni nazw `mongo`.

Listing 12.4. Spring Data MongoDB udostępnia możliwość konfiguracji w plikach XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
↪xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
↪xmlns:mongo="http://www.springframework.org/schema/data/mongo" ← Deklarujemy przestrzeń nazw mongo
xsi:schemaLocation="
http://www.springframework.org/schema/data/mongo
http://www.springframework.org/schema/data/mongo/spring-mongo.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
   <mongo:repositories base-package="orders.db" /> ← Włączamy generowanie repozytorium

   <mongo:mongo /> ← Deklarujemy klienta MongoClient
   <bean id="mongoTemplate" ← Tworzymy komponent MongoTemplate
      class="org.springframework.data.mongodb.core.MongoTemplate">
      <constructor-arg ref="mongo" />
      <constructor-arg value="OrdersDB" />
   </bean>
</beans>
```

Teraz, po skonfigurowaniu Spring Data MongoDB, jesteśmy już prawie gotowi, aby rozpocząć zapisywanie i pobieranie dokumentów. Na początek musimy odwzorować typy domenowe Javy na dokumenty, żeby umożliwić ich zapis do bazy z użyciem adnotacji Spring Data MongoDB.

12.1.2. Dodajemy adnotacje umożliwiające zapis w MongoDB

Gdy pracowaliśmy z JPA, musieliśmy odwzorować encje Javy na tabele i kolumny w bazie relacyjnej. Specyfikacja JPA dostarcza kilka adnotacji do obsługi odwzorowywania obiektowo-relacyjnego, a niektóre implementacje JPA, takie jak Hibernate, dodają też swoje własne adnotacje.

MongoDB nie dostarcza jednak własnych adnotacji odwzorowywania obiektowo-dokumentowego. Spring Data MongoDB wypełnia tę lukę swoimi adnotacjami, których możemy użyć do odwzorowania klas Javy na dokumenty MongoDB. W tabeli 12.1 znajduje się opis tych adnotacji.

Tabela 12.1. Adnotacje Spring Data MongoDB umożliwiające odwzorowanie obiektowo-dokumentowe

Adnotacja	Opis
`@Document`	Identyfikuje obiekt domenowy, który ma zostać odwzorowany na dokument MongoDB.
`@Id`	Oznacza pole będące identyfikatorem.
`@DbRef`	Oznacza, że pole wskazuje na inny dokument, być może zapisany w innej bazie danych.
`@Field`	Określa własne metadane dla pola dokumentu.
`@Version`	Określa właściwość będącą polem wersji.

Adnotacje `@Document` i `@Id` są odpowiednikami adnotacji `@Entity` i `@Id` w JPA. Będziemy z nich często korzystać. Pojawią się w każdej klasie Javy, która ma służyć do zapisu w bazie MongoDB w postaci dokumentu. Przykładowo listing 12 5 pokazuje, jak dodać adnotacje do klasy `Order`, aby zapisać ją w bazie MongoDB.

Listing 12.5. Adnotacje Spring Data MongoDB odwzorowują klasy Javy na dokumenty

```
package orders;
import java.util.Collection;
import java.util.LinkedHashSet;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;

@Document <-- To jest dokument
public class Order {

   @Id
   private String id; <-- Przydzielamy identyfikator

   @Field("client")
   private String customer; <-- Nadpisujemy domyślną nazwę pola

   private String type;

   private Collection<Item> items = new LinkedHashSet<Item>();

   public String getCustomer() {
      return customer;
   }

   public void setCustomer(String customer) {
      this.customer = customer;
   }

   public String getType() {
      return type;
   }

   public void setType(String type) {
      this.type = type;
   }

   public Collection<Item> getItems() {
      return items;
   }

   public void setItems(Collection<Item> items) {
      this.items = items;
   }

   public String getId() {
      return id;
   }
}
```

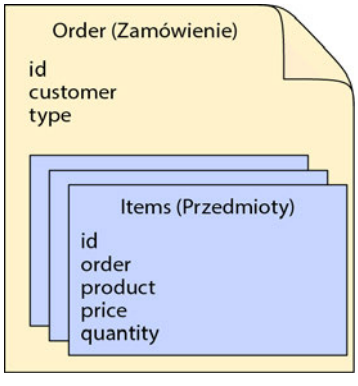
Jak widzisz, klasa `Order` jest opatrzona adnotacją `@Document`, co umożliwia jej przechowywanie za pomocą szablonów `MongoTemplate`, automatycznie generowanego repozytorium lub obu tych sposobów równocześnie. Właściwość `id` opatrzona jest adnotacją `@Id`,

aby wskazać to pole jako identyfikator dokumentu. Dodatkowo właściwość `customer` opatrzona jest adnotacją `@Field`, co powoduje, że przy zapisie dokumentu do bazy właściwość `customer` odwzorowywana jest na pole o nazwie `client`.

Zauważ, że żadne inne pola nie są opatrzone adnotacjami. Wszystkie właściwości obiektu są zapisane jako pola dokumentu, chyba że zostaną oznaczone adnotacją `@Transient`. Podobnie, jeśli nie zostaną oznaczone adnotacją `@Field`, pola dokumentów będą mieć taką samą nazwę jak odpowiadające im właściwości obiektu.

Zwróć też uwagę na właściwość `items`. Jest to kolekcja zamówionych przedmiotów. W tradycyjnej bazie relacyjnej przedmioty te byłyby przechowywane w osobnej tabeli w bazie danych, powiązane kluczem obcym, a pole `items` mogłoby zostać opatrzone adnotacją JPA `@OneToMany`. Tutaj sytuacja wygląda inaczej.

Jak wcześniej wspominałem, dokumenty mogą być ze sobą wzajemnie powiązane, ale nie jest to dziedzina, w której bazy dokumentowe sprawdzają się najlepiej. W przypadku relacji pomiędzy zamówieniem a zamówionymi przedmiotami przedmioty te zostaną umieszczone wewnątrz tego samego dokumentu. Przedstawia to rysunek 12.1.



Rysunek 12.1. Dokumenty reprezentują powiązane, ale nieznormalizowane dane. Powiązane elementy (takie jak zamówione przedmioty) zagnieżdżone są w głównym dokumencie

Nie jest nam potrzebna żadna adnotacja do delegowania tej relacji. W praktyce klasa `Item` nie jest opatrzona żadnymi adnotacjami:

```
package orders;
public class Item {
   private Long id;
   private Order order;
   private String product;
   private double price;
   private int quantity;

   public Order getOrder() {
      return order;
   }
   public String getProduct() {
      return product;
   }
   public void setProduct(String product) {
      this.product = product;
   }
   public double getPrice() {
```

```
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    public Long getId() {
        return id;
    }
}
```

Nie ma potrzeby oznaczania klasy `Item` adnotacją `@Document`, nie jest też potrzebne oznaczanie któregoś z jej pól adnotacją `@Id`. Przyczyną tego jest fakt, że klasa `Item` nie jest nigdy zapisywana jako niezależny dokument. Zawsze jest elementem listy przedmiotów dokumentu `Order` i jest zagnieżdżona w jego wnętrzu.

Moglibyśmy oczywiście oznaczyć jedną z właściwości klasy `Item` za pomocą adnotacji `@Field`, jeśli chcielibyśmy wskazać, pod jaką nazwą ma być to pole zapisane w dokumencie. W naszym przykładzie nie było to po prostu potrzebne.

Mamy już klasę domeny Javy opatrzoną adnotacją persystencji MongoDB. Zobaczmy więc, jak zapisać kilka zamówień za pomocą obiektu klasy `MongoTemplate`.

12.1.3. Dostęp do bazy MongoDB za pomocą szablonów MongoTemplate

Skonfigurowaliśmy już wcześniej komponent `MongoTemplate` w sposób jawny lub poprzez rozszerzenie klasy abstrakcyjnej `AbstractMongoConfiguration` w klasie konfiguracji. Pozostaje nam jedynie wstrzyknąć komponent w miejscu, w którym chcemy z niego skorzystać:

```
@Autowired
MongoOperations mongo;
```

W tym miejscu wstrzykuję obiekt `MongoTemplate` do właściwości typu `MongoOperations`. `MongoOperations` jest interfejsem, który klasa `MongoTemplate` implementuje, a dobrą praktyką jest praca nie z konkretną implementacją, a z jej interfejsem, w szczególności przy wstrzykiwaniu komponentów.

Interfejs `MongoOperations` wystawia kilka użytecznych metod pracy z bazą dokumentową MongoDB. Nie jest możliwe omówienie ich wszystkich tutaj. Możemy jednak spojrzeć na kilka najczęściej wykorzystywanych operacji, takich jak liczenie dokumentów zapisanych w kolekcji. Poprzez wstrzykniętą instancję `MongoOperations` otrzymujemy kolekcję `order` i wywołujemy na niej metodę `count()`:

```
long orderCount = mongo.getCollection("order").count();
```

Przypuśćmy, że chcemy zapisać nowe zamówienie. W tym celu wywołujemy metodę `save()`:

```
Order order = new Order();
... // ustawiamy właściwości i dodajemy pozycje do zamówienia
mongo.save(order, "order");
```

Pierwszym parametrem metody `save()` jest nowo utworzony obiekt zamówienia, a drugim nazwa kolekcji, do której ma trafić zapisywany dokument.

Możemy też wyszukać zamówienie, podając jego identyfikator jako parametr metody `findById()`:

```
String orderId = ...;
Order order = mongo.findById(orderId, Order.class);
```

Bardziej zaawansowane zapytania wymagają skonstruowania obiektu `Query` i przekazania jako parametr metody `find()`. Przykładowo do wyszukania wszystkich zamówień, których pole `client` ma wartość `"Chuck Wagon"`, możemy wykorzystać poniższy fragment kodu:

```
List<Order> chucksOrders = mongo.find(Query.query(
   Criteria.where("client").is("Chuck Wagon")), Order.class);
```

W tym przypadku do konstrukcji obiektu `Query` używamy obiektu `Criteria`, za pomocą którego sprawdzamy jedno pole. Obiekt `Criteria` może jednak służyć do tworzenia dużo ciekawszych zapytań. Chcemy być może pobrać wszystkie zamówienia Chucka złożone przez internet:

```
List<Order> chucksWebOrders = mongo.find(Query.query(
   Criteria.where("customer").is("Chuck Wagon")
      .and("type").is("INTERNET")), Order.class);
```

A gdy zechcemy usunąć dokument, możemy skorzystać z metody `remove()`:

```
mongo.remove(order);
```

Jak już mówiłem, interfejs `MongoOperations` posiada kilka metod do pracy z danymi w postaci dokumentów. Warto się z nimi zapoznać i odkryć pozostałe możliwości interfejsu `MongoOperations`, korzystając z dokumentacji JavaDoc.

Standardowo wstrzyknęlibyśmy implementację `MongoOperations` do własnoręcznie zaprojektowanej klasy repozytorium i wykorzystalibyśmy dostępne w niej działania w implementacji metod repozytorium. Jeśli jednak nie chcemy tworzyć repozytorium samodzielnie, możemy użyć Spring Data MongoDB do automatycznego wygenerowania implementacji repozytorium po uruchomieniu aplikacji. Zobaczmy, jak to zrobić.

12.1.4. Tworzymy repozytorium MongoDB

Aby zrozumieć proces tworzenia repozytorium z użyciem Spring Data MongoDB, spójrzmy jeszcze raz na repozytorium Spring Data JPA, z którym pracowaliśmy w rozdziale 11. Na listingu 11.4 utworzyliśmy interfejs `SpitterRepository`, który rozszerza interfejs `JpaRepository`. W tym samym podrozdziale włączyliśmy również obsługę repozytoriów Spring Data JPA. W rezultacie Spring Data JPA było w stanie automatycznie utworzyć implementację tego interfejsu, wliczając w to kilka metod, które stworzyliśmy w oparciu o konwencje nazewnicze.

Włączyliśmy już repozytoria Spring Data MongoDB za pomocą adnotacji `@Enable`
`↪MongoRepositories`, pozostało nam więc tylko utworzenie interfejsu, na podstawie któ-

rego wygenerowana zostać może jego implementacja. Nie będziemy jednak rozszerzać interfejsu `JpaRepository`, a interfejs `MongoRepository`. Interfejs `OrderRepository`, przedstawiony na listingu 12.6, rozszerza interfejs `MongoRepository`, dzięki czemu uzyskujemy dostęp do podstawowych operacji CRUD na dokumentach typu `Order`.

Listing 12.6. Spring Data MongoDB automatycznie implementuje interfejsy repozytorium

```
package orders.db;
import orders.Order;
import org.springframework.data.mongodb.repository.MongoRepository;
public interface OrderRepository
   extends MongoRepository<Order, String> {
}
```

Interfejs `OrderRepository` rozszerza interfejs `MongoRepository`, a więc w sposób przechodni rozszerza bazowy interfejs `Repository`. Podczas poznawania Spring Data JPA dowiedziałeś się, że każdy interfejs rozszerzający interfejs `Repository` po uruchomieniu zostanie automatycznie zaimplementowany. W tym przypadku jednak, zamiast repozytorium JPA do współpracy z bazami relacyjnymi, otrzymamy implementację interfejsu `Order` ↪`Repository`, która umożliwi nam odczyt i zapis danych z bazy MongoDB.

Interfejs `MongoRepository` przyjmuje dwa parametry. Pierwszym jest typ obiektu opatrzonego adnotacją `@Document`, na którym repozytorium ma wykonywać operacje. Drugim jest typ właściwości opatrzonej adnotacją `@Id`.

Chociaż repozytorium `OrderRepository` nie definiuje żadnych własnych metod, to kilka metod dziedziczy, wliczając w to użyteczne operacje CRUD na dokumentach typu `Order`. W tabeli 12.2 znajduje się opis wszystkich metod dziedziczonych przez repozytorium `OrderRepository`.

Metody w tabeli 12.2 odnoszą się do typów generycznych przekazywanych do metod i zwracanych przez te metody. Interfejs `OrderRepository` rozszerza repozytorium `MongoRepository<Order, String>`, więc `T` odzwierciedla się na `Order`, `ID` na `String`, a `S` na dowolny typ rozszerzający `Order`.

DODAWANIE WŁASNYCH ZAPYTAŃ

Operacje CRUD są z reguły bardzo pomocne, ale w naszym repozytorium możemy potrzebować jeszcze innych metod.

W punkcie 11.3.1 dowiedziałeś się, że Spring Data JPA obsługuje konwencję nazewniczą metod, która pomaga Spring Data przy automatycznym generowaniu implementacji metod stosujących się do tej konwencji. Okazuje się, że ta sama konwencja działa również w Spring Data MongoDB. Oznacza to, że możemy dodać własne metody do repozytorium `OrderRepository`:

```
public interface OrderRepository
      extends MongoRepository<Order, String> {
   List<Order> findByCustomer(String c);
   List<Order> findByCustomerLike(String c);
   List<Order> findByCustomerAndType(String c, String t);
   List<Order> findByCustomerLikeAndType(String c, String t);
}
```

Tabela 12.2. Interfejs repozytorium rozszerza interfejs MongoRepository, dzięki czemu dziedziczy kilka operacji CRUD, które są automatycznie implementowane przez Spring Data MongoDB

Metoda	Opis
`long count();`	Zwraca liczbę dokumentów dla typu repozytorium.
`void delete(Iterable<? extends T);`	Usuwa wszystkie dokumenty powiązane z danymi obiektami.
`void delete(T);`	Usuwa dokument powiązany z danym obiektem.
`void delete(ID);`	Usuwa dokument o danym identyfikatorze.
`void deleteAll();`	Usuwa wszystkie dokumenty repozytorium danego typu.
`boolean exists(Object);`	Zwraca `true`, jeśli istnieje dokument powiązany z danym obiektem.
`boolean exists(ID);`	Zwraca `true`, jeśli istnieje dokument o danym identyfikatorze.
`List<T> findAll();`	Zwraca wszystkie dokumenty dla typu repozytorium.
`List<T> findAll(Iterable<ID>);`	Zwraca wszystkie dokumenty o danych identyfikatorach.
`List<T> findAll(Pageable);`	Zwraca postronicowaną i posortowaną listę dokumentów dla typu repozytorium.
`List<T> findAll(Sort);`	Zwraca posortowaną listę wszystkich dokumentów o danym identyfikatorze.
`T findOne(ID);`	Zwraca pojedynczy dokument dla danego identyfikatora.
`save(Iterable<S>);`	Zapisuje wszystkie dokumenty dla danej kolekcji typu `Iterable`.
`save(S);`	Zapisuje pojedynczy dokument na bazie danego obiektu.

Zdefiniowaliśmy tutaj cztery metody — każda z nich służy do wyszukiwania obiektów typu `Order`, które spełniają określone kryteria. Jedna metoda zwraca listę zamówień, w których właściwość `customer` jest równa wartości przekazanej do tej metody. Inna zwraca listę zamówień, w których właściwość `customer` zawiera ciąg przekazany do tej metody. Kolejna zwraca listę zamówień, w których właściwości `customer` i `type` są równe wartościom przekazanym do tej metody. Ostatnia metoda przypomina poprzednią, ale właściwość `customer` porównuje za pomocą operacji *like*, a nie zwykłej *równości*.

Słowo `find` używane w zapytaniach jest bardzo elastyczne. Równie dobrze moglibyśmy użyć słowa `get`:

```
List<Order> getByCustomer(String c);
```

Możemy też zastosować słowo `read`:

```
List<Order> readByCustomer(String c);
```

Jest jeszcze inne specjalne słowo służące do liczenia pasujących obiektów:

```
int countByCustomer(String c);
```

Podobnie jak w Spring Data JPA, mamy duże możliwości odnośnie do tego, co może się pojawić w zapytaniu pomiędzy czasownikiem a słowem `By`. Przykładowo możemy określić, czego szukamy:

```
List<Order> findOrdersByCustomer(String c);
```

W słowie `Orders` nie ma jednak nic specjalnego. Słowo to nie ma żadnego wpływu na pobierane dane. Równie dobrze moglibyśmy nazwać naszą metodę następująco:

```
List<Order> findSomeStuffWeNeedByCustomer(String c);
```

Nie musimy również zwracać kolekcji `List<Order>`. Jeśli potrzebne jest nam tylko pojedyncze zamówienie, możemy jako zwracany typ ustawić `Order`:

```
Order findASingleOrderByCustomer(String c);
```

W tym przypadku zwrócone zostanie pierwsze zamówienie, które byłoby znalezione przy ustawieniu `List` jako typu zwracanego. Jeśli żaden pasujący dokument nie zostanie znaleziony, metoda zwraca wartość `null`.

TWORZYMY ZAPYTANIA

W punkcie 11.3.2 dowiedziałeś się, że do określenia własnego zapytania dla metody repozytorium możemy wykorzystać adnotację `@Query`. Adnotacja ta działa tak samo dobrze zarówno dla MongoDB, jak i dla JPA. Jedyną zauważalną różnicą jest to, że w MongoDB adnotacja `@Query` przyjmuje jako wartość ciąg JSON w postaci stringa w miejscu zapytania JPA.

Przypuśćmy, że chcemy utworzyć metodę, która wyszuka wszystkie zamówienia danego typu złożone przez klienta „Chuck Wagon". Zadeklarowana poniżej metoda interfejsu `OrderRepository` zwróci pożądany wynik:

```
@Query("{'customer': 'Chuck Wagon', 'type' : ?0}")
List<Order> findChucksOrders(String t);
```

JSON przekazany do adnotacji `@Query` porównywany jest ze wszystkimi dokumentami typu `Order` i zwracane są wszystkie dopasowania. Zauważ, że właściwość `type` odwzorowywana jest na `?0`. Oznacza to, że właściwość ta ma być równa zerowemu parametrowi przekazanemu do metody. Jeśli metoda przyjmowałaby więcej parametrów, moglibyśmy się do nich odwoływać przez symbole `?1`, `?2` itd.

DOŁĄCZANIE WŁASNYCH ZACHOWAŃ DO REPOZYTORIUM

W punkcie 11.3.3 poznałeś sposób na dołączenie własnych implementacji metod do wygenerowanego repozytorium. W JPA oznaczało to utworzenie interfejsu pośredniczącego, który deklarował niestandardowe metody, klasy implementującej te metody i zmianę interfejsu automatycznie generującego repozytorium tak, aby rozszerzał ten pośredniczący interfejs. Te same czynności możemy wykorzystać w przypadku repozytorium Spring Data MongoDB.

Przypuśćmy, że potrzebna jest nam metoda wyszukująca wszystkie obiekty typu `Order`, w których właściwość dokumentu `type` odpowiada danej wartości. Moglibyśmy w prosty sposób utworzyć taką metodę, nadając jej sygnaturę `List<Order> findByType ↪(String t)`. Na potrzeby tego przykładu załóżmy, że jeśli tym typem jest `WEB`, zapytamy o wszystkie zamówienia typu `INTERNET`. To byłoby już trudne do osiągnięcia, nawet z użyciem adnotacji `@Query`. Możemy to jednak osiągnąć, korzystając z dołączonej implementacji.

Na początek zdefiniujemy interfejs pośredniczący:

```
package orders.db;
import java.util.List;
import orders.Order;

public interface OrderOperations {
   List<Order> findOrdersByType(String t);
}
```

Jest to stosunkowo proste. Teraz możemy utworzyć dołączaną implementację. Listing 12.7 pokazuje, jak ta implementacja może wyglądać.

Listing 12.7. Dołączamy własne zachowania do automatycznego repozytorium

```
package orders.db;
import java.util.List;
import orders.Order;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;
public class OrderRepositoryImpl implements OrderOperations {

   @Autowired
   private MongoOperations mongo; <-- Wstrzykujemy MongoOperations
   public List<Order> findOrdersByType(String t) {
      String type = t.equals("WEB") ? "INTERNET" : t;
      Criteria where = Criteria.where("type").is(t); <-- Tworzymy zapytanie
      Query query = Query.query(where);
      return mongo.find(query, Order.class); <-- Wykonujemy zapytanie
   }
}
```

Jak widzisz, dołączana implementacja wstrzykiwana jest do `MongoOperations` (interfejsu, który implementuje klasa `MongoTemplate`). Metoda `findOrdersByType()` wykorzystuje interfejs `MongoOperations` do odpytania bazy danych o dokumenty odpowiadające skonstruowanemu zapytaniu.

Pozostała nam już tylko zmiana interfejsu `OrderRepository` w taki sposób, aby rozszerzał interfejs pośredniczący `OrderOperations`:

```
public interface OrderRepository
      extends MongoRepository<Order, String>, OrderOperations {
   ...
}
```

Wszystkie te elementy wiązane są dzięki temu, że klasa implementująca interfejs `Order`
`↪Repository` ma nazwę `OrderRepositoryImpl`. Jest to taka sama nazwa, jaką ma interfejs, uzupełniona sufiksem `Impl`. Gdy Spring Data MongoDB będzie generował implementację repozytorium, wyszuka tę klasę i dołączy do automatycznie generowanej implementacji.

Jeżeli sufiks `Impl` nam nie odpowiada, możemy skonfigurować Spring Data MongoDB tak, by szukał klas z innym sufiksem w nazwie. Musimy tylko ustawić atrybut `repository`
`↪ImplementationPostfix` adnotacji `@EnableMongoRepositories` (w klasie konfiguracji Springa).

```
@Configuration
@EnableMongoRepositories(basePackages="orders.db",
      repositoryImplementationPostfix="Stuff")
public class MongoConfig extends AbstractMongoConfiguration {
   ...
}
```

W przypadku konfiguracji XML możemy w tym celu ustawić atrybut `repository-impl-postfix` znacznika `<mongo:repositories>`:

```
<mongo:repositories base-package="orders.db"
      repository-impl-postfix="Stuff" />
```

Obie konfiguracje sprawią, że Spring Data MongoDB będzie szukać klasy `OrderRepositoryStuff` zamiast `OrderRepositoryImpl`.

Bazy dokumentowe, takie jak MongoDB, rozwiązują określoną klasę problemów. Ale podobnie jak bazy relacyjne, nie są lekarstwem na wszystkie trudności. Co więcej, w niektórych sytuacjach żadne z tych rozwiązań nie jest odpowiednie. Na szczęście są jeszcze inne możliwości.

Zobaczmy teraz, jak Spring Data wspiera Neo4j, popularną bazę grafową.

12.2. Pracujemy z danymi w postaci grafów w Neo4j

Podczas gdy bazy dokumentowe przechowują dane w „gruboziarnistych" dokumentach, bazy grafowe przechowują dane w kilku „drobnoziarnistych" węzłach, powiązanych ze sobą poprzez relacje. Węzeł w bazie grafowej z reguły reprezentuje w bazie danych koncepcję, w której właściwości określają stan węzła. Relacje wiążą ze sobą dwa węzły i mogą przechowywać własne właściwości.

W najprostszej postaci bazy grafowe są bardziej uniwersalne od baz dokumentowych i mogą stanowić alternatywę dla baz relacyjnych, nie posiadając przy tym ustalonego schematu. Ponieważ jednak dane w bazie grafowej mają strukturę grafu, możliwe jest podążanie wzdłuż relacji i odkrywanie zależności, które byłyby trudne lub nawet nieosiągalne w bazach innego typu.

Spring Data Neo4j oferuje wiele możliwości dostępnych w Spring Data JPA oraz Spring Data MongoDB, wykorzystując przy tym bazę grafową Neo4j. Dostarcza adnotacje umożliwiające odwzorowanie typów Javy do węzłów i relacji, dostęp w oparciu o szablony Neo4j oraz automatyczne generowanie implementacji repozytoriów.

Za chwilę dowiesz się, jak wykorzystać te możliwości do pracy z Neo4j. Najpierw jednak musimy skonfigurować Spring Data Neo4j.

12.2.1. Konfigurujemy Spring Data Neo4j

Kluczem do konfiguracji Spring Data Neo4j jest zadeklarowanie komponentu `GraphDatabaseService` i włączenie automatycznego generowania repozytoriów Neo4j. Listing 12.8 przedstawia podstawową klasę konfiguracji Javy wymaganą przez Spring Data Neo4j.

Listing 12.8. Konfigurujemy Spring Data Neo4j za pomocą adnotacji @EnableNeo4jRepositories

```
package orders.config;
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.neo4j.config.EnableNeo4jRepositories;
import org.springframework.data.neo4j.config.Neo4jConfiguration;

@Configuration
@EnableNeo4jRepositories(basePackages="orders.db") <-- Włączamy automatyczne repozytoria
public class Neo4jConfig extends Neo4jConfiguration {
   public Neo4jConfig() {
      setBasePackage("orders"); <-- Ustawiamy bazowy pakiet modelu
   }


   @Bean(destroyMethod="shutdown")
   public GraphDatabaseService graphDatabaseService() {
      return new GraphDatabaseFactory()
         .newEmbeddedDatabase("/tmp/graphdb"); <-- Konfigurujemy pakiet wbudowanej bazy
      }
}
```

Adnotacja `@EnableNeo4jRepositories` umożliwia Spring Data Neo4j automatyczne generowanie implementacji repozytoriów Neo4j. Jego właściwość `basePackages` przyjmuje wartość `orders.db`, pakietu, wewnątrz którego nastąpi skanowanie interfejsów rozszerzających (bezpośrednio lub pośrednio) bazowy pakiet `Repository`.

Klasa `Neo4jConfig` rozszerza klasę `Neo4jConfiguration`, która udostępnia wygodnie metody konfiguracji Spring Data Neo4j. Jedną z tych metod jest `setBasePackage()`, wywoływana w konstruktorze `Neo4jConfig` i informująca Spring Data Neo4j, że klasy modelu można odnaleźć wewnątrz pakietu `orders`.

Ostatnim elementem układanki jest definicja komponentu `GraphDatabaseService`. W tym przypadku metoda `graphDatabaseService()` wykorzystuje klasę `GraphDatabase`
`↪Factory` do utworzenia instancji wbudowanej (*embedded*) bazy Neo4j. Jeśli chodzi o Neo4j, nie należy mylić wbudowanej bazy z bazą przechowującą dane w pamięci (*in-memory*). „Wbudowana" oznacza tutaj to, że silnik bazy danych działa na tej samej wirtualnej maszynie JVM jako część aplikacji, a nie gdzieś na osobnym serwerze. Dane są w dalszym ciągu zapisywane w systemie plików (w tym przypadku w lokalizacji */tmp/graphdb*).

Możemy też chcieć skonfigurować serwis `GraphDatabaseService`, który odwołuje się do zdalnego serwera Neo4j. Jeżeli w ścieżce klas aplikacji znajduje się biblioteka `spring-`
`↪data-neo4j-rest`, możemy skonfigurować bazę za pomocą klasy `SpringRestGraphDatabase`, co umożliwia dostęp do zdalnej bazy Neo4j za pośrednictwem REST-owego API:

```
@Bean(destroyMethod="shutdown")
public GraphDatabaseService graphDatabaseService() {
   return new SpringRestGraphDatabase(
         "http://graphdbserver:7474/db/data/");
}
```

Jak widać, przedstawiona powyżej konfiguracja SpringRestGraphDatabase zakłada, że zdalna baza nie wymaga uwierzytelnienia. W środowisku produkcyjnym jednakże najprawdopodobniej zechcesz zabezpieczyć serwer bazy danych. W tym przypadku możesz przy tworzeniu instancji SpringRestGraphDatabase podać odpowiednie dane dostępowe:

```
@Bean(destroyMethod="shutdown")
public GraphDatabaseService graphDatabaseService(Environment env) {
   return new SpringRestGraphDatabase(
         "http://graphdbserver:7474/db/data/",
         env.getProperty("db.username"),
         env.getProperty("db.password"));
}
```

W przykładzie dane dostępowe pobierane są za pomocą wstrzykniętej instancji Envi
➥ronment, aby uniknąć ich bezpośredniego umieszczania w kodzie aplikacji.

Spring Data Neo4j umożliwia też konfigurację z wykorzystaniem przestrzeni nazw XML. Jeśli wolimy ten rodzaj konfiguracji, możemy użyć elementów <neo4j:config> i <neo4j: repositories> z tej przestrzeni nazw. Listing 12.9 przedstawia konfigurację równoważną z konfiguracją Javy z listingu 12.8.

Listing 12.9. Spring Data Neo4j można również skonfigurować z wykorzystaniem XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
       xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
       xmlns:neo4j=http://www.springframework.org/schema/data/neo4j
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/data/neo4j
         http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd">
   <neo4j:config                                  Konfigurujemy szczegółowe ustawienia bazy Neo4j
      storeDirectory="/tmp/graphdb"
      base-package="orders" />
   <neo4j:repositories base-package="orders.db" />   Włączamy generowanie repozytoriów
 </beans>
```

Element <neo4j:config> pozwala skonfigurować szczegółowe ustawienia dostępu do bazy danych. W naszym przypadku konfigurujemy Spring Data Neo4j do współpracy z wbudowaną bazą danych. W szczególności atrybut storeDirectory określa ścieżkę w systemie plików, w której mają być zapisywane dane. Atrybut base-package ustawia pakiet, w którym zdefiniowane są klasy modelu.

Element <neo4j:repositories> włącza mechanizm Spring Data Neo4j automatycznego generowania implementacji poprzez skanowanie pakietu orders.db i wyszukiwanie interfejsów rozszerzających interfejs Repository.

Aby skonfigurować połączenie Spring Data Neo4j ze zdalnym serwerem Neo4j, musimy tylko zadeklarować komponent SpringRestGraphDatabase i ustawić atrybut graph
➥DatabaseService elementu <neo4j:config>:

```
<neo4j:config base-package="orders"
              graphDatabaseService="graphDatabaseService" />
<bean id="graphDatabaseService" class=
      "org.springframework.data.neo4j.rest.SpringRestGraphDatabase">
   <constructor-arg value="http://graphdbserver:7474/db/data/" />
   <constructor-arg value="db.username" />
   <constructor-arg value="db.password" />
</bean>
```

Niezależnie od tego, czy zdecydujemy się na konfigurację Spring Data Neo4j za pomocą klas Javy, czy za pomocą plików XML, klasy domenowe muszą znajdować się w pakiecie określonym jako pakiet bazowy (atrybut `basePackages` adnotacji `@EnableNeo4jRepositories` lub atrybut `basePackages` elementu `<neo4j:config>`). Klasy te muszą być też opatrzone adnotacjami jako encje węzłów bądź relacji. Zajmiemy się tym w następnym punkcie.

12.2.2. Dodajemy adnotacje do encji grafów

Neo4j definiuje dwa rodzaje encji: węzły oraz relacje. Encje węzłów reprezentują z reguły różne byty w aplikacji, natomiast relacje definiują powiązania pomiędzy tymi bytami.

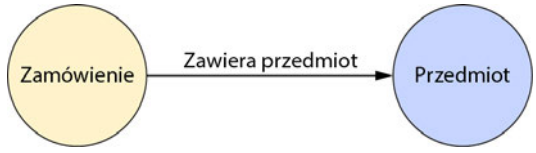
Spring Data Neo4j udostępnia kilka adnotacji, które możemy zastosować na klasach domenowych i ich polach w celu zapisania w Neo4j. Adnotacje te opisane są w tabeli 12.3.

Tabela 12.3. Adnotacje Spring Data Neo4j umożliwiają odwzorowanie klas domenowych na węzły i relacje w grafie

Adnotacja	Opis
`@NodeEntity`	Deklaruje typ Javy jako encję węzła.
`@RelationshipEntity`	Deklaruje typ Javy jako encję relacji.
`@StartNode`	Deklaruje właściwość jako węzeł początkowy w encji relacji.
`@EndNode`	Deklaruje właściwość jako węzeł końcowy w encji relacji.
`@Fetch`	Deklaruje, że właściwość encji ma być ładowana w sposób zachłanny.
`@GraphId`	Deklaruje właściwość jako pole identyfikatora encji (pole musi być typu `Long`).
`@GraphProperty`	Deklaruje właściwość w sposób jawny.
`@GraphTraversal`	Deklaruje, że właściwość ma automatycznie dostarczyć iterator zbudowany poprzez przeszukiwanie grafu.
`@Indexed`	Deklaruje, że właściwość ma zostać zaindeksowana.
`@Labels`	Deklaruje etykiety dla encji węzła.
`@Query`	Deklaruje, że właściwość ma automatycznie dostarczyć iterator zbudowany poprzez wywołanie zapytania Cypher Query Language.
`@QueryResult`	Deklaruje klasę lub interfejs Javy jako mogące przechowywać wynik zapytania.
`@RelatedTo`	Deklaruje prostą relację poprzez właściwość pomiędzy bieżącym węzłem encji a innym węzłem encji.
`@RelatedToVia`	Deklaruje pola encji węzła jako odniesienie do encji relacji, do której ten węzeł należy.
`@RelationshipType`	Deklaruje pole jako typ encji relacji.
`@ResultColumn`	Deklaruje, że właściwość na typie opatrzonym adnotacją `@QueryResult` ma przechwytywać wartość określonego pola z wyniku zapytania.

Zobaczmy działanie kilku z tych adnotacji na przykładzie zamówienia i przedmiotów.

Jednym ze sposobów modelowania danych jest desygnowanie zamówienia jako węzła powiązanego z jednym lub kilkoma przedmiotami. Rysunek 12.2 ilustruje ten model w postaci grafu.



Rysunek 12.2. Prosta relacja łączy dwa węzły, ale nie przechowuje własnych właściwości

Aby desygnować zamówienia jako węzły, musimy opatrzyć klasę Order adnotacją `@NodeEntity`. Listing 12.10 pokazuje klasę Order opatrzoną adnotacją `@NodeEntity`, jak również kilka innych adnotacji z tabeli 12.3.

Listing 12.10. Klasa Order jest oznaczona jako węzeł w grafowej bazie danych

```
package orders;
import java.util.LinkedHashSet;
import java.util.Set;
import org.springframework.data.neo4j.annotation.GraphId;
import org.springframework.data.neo4j.annotation.NodeEntity;
import org.springframework.data.neo4j.annotation.RelatedTo;

@NodeEntity                                          Zamówienia są węzłami
public class Order {
   @GraphId                                          Identyfikator grafu
   private Long id;
   private String customer;
   private String type;
   @RelatedTo(type="HAS_ITEMS")                      Relacja z przedmiotami
   private Set<Item> items = new LinkedHashSet<Item>();
   ...
}
```

Wykorzystaliśmy tutaj adnotację `@NodeEntity` na poziomie klasy. Właściwość `id` opatrzona jest z kolei adnotacją `@GraphId`. Każda encja w Neo4j musi mieć identyfikator grafu. Jest to w zasadzie analogiczne do właściwości opatrzonych adnotacją `@Id` w klasach `@Entity` w JPA lub `@Document` w MongoDB. Wymagane jest też, aby identyfikator węzła był typu `Long`.

Właściwości `customer` oraz `type` są pozbawione jakichkolwiek adnotacji. Dopóki nie zostaną opatrzone adnotacją `@Transient`, i tak będą traktowane jako właściwości węzła w bazie danych.

Właściwość `items` została opatrzona adnotacją `@RelatedTo`, co wskazuje, że zamówienie jest powiązane ze zbiorem przedmiotów. Atrybut `type` służy jako etykieta relacji. Można mu przypisać dowolną wartość, ale najczęściej jest to jakiś przyjazny człowiekowi tekst, który opisuje w skrócie naturę relacji. Etykietę tę wykorzystamy później w zapytaniach między relacjami.

Wracając do klasy Item — na listingu 12.11 pokazano, w jaki sposób został on opatrzony adnotacjami umożliwiającymi persystencję grafu.

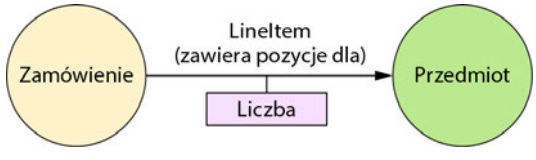
Listing 12.11. Przedmioty też są reprezentowane przez węzły w bazie grafowej

```
package orders;
import org.springframework.data.neo4j.annotation.GraphId;
import org.springframework.data.neo4j.annotation.NodeEntity;

@NodeEntity <-- Przedmioty są węzłami
public class Item {
   @GraphId <-- Identyfikator grafu
   private Long id;
   private String product;
   private double price;
   private int quantity;
   ...
}
```

Podobnie jak klasa `Order`, klasa `Item` jest opatrzona adnotacją `@NodeEntity`, co powoduje jej desygnowanie jako węzła. Właściwość id typu `Long` opatrzona jest też adnotacją `@GraphId`, przez co staje się identyfikatorem grafu. Właściwości `product`, `price` oraz `quantity` również zostaną zapisane jako właściwości węzła w bazie grafowej.

Relacja pomiędzy klasami `Order` i `Item` jest prosta w tym zakresie, że nie przenosi żadnych własnych danych. Dzięki temu adnotacja `@RelatedTo` jest wystarczająca do zdefiniowania relacji. Nie wszystkie relacje są jednak takie proste.

Przemyślmy sposób, w jaki zaprojektowaliśmy nasz model danych, żeby się dowiedzieć, jak pracować z bardziej złożonymi relacjami. W aktualnym modelu połączyliśmy koncepcję pozycji zamówienia i produktu w klasie `Item`. Gdy się nad tym zastanowimy, odkryjemy, że zamówienie powiązane jest z jednym lub większą liczbą produktów. Relacja pomiędzy zamówieniem a produktem stanowi pojedynczą pozycję zamówienia. Rysunek 12.3 przedstawia alternatywny sposób modelowania danych w grafie.



Rysunek 12.3. Encja relacji jest relacją, która posiada własne właściwości

W nowym modelu liczba produktów w zamówieniu jest właściwością pozycji zamówienia, a produkt jest koncepcją zupełnie oddzielną. Tak jak poprzednio, zarówno zamówienia, jak i produkty są węzłami. Pozycje zamówienia są relacją. Ale teraz, gdy pozycja zamówienia musi przenosić wartość liczby zamówionych przedmiotów, relacja nie może być prosta. Musimy zdefiniować klasę, która reprezentuje pozycję zamówienia, taką jak klasa `LineItem` pokazana na listingu 12.12.

Listing 12.12. Klasa LineItem łączy węzeł Order z węzłem Product

```
package orders;
import org.springframework.data.neo4j.annotation.EndNode;
import org.springframework.data.neo4j.annotation.GraphId;
```

```
import org.springframework.data.neo4j.annotation.RelationshipEntity;
import org.springframework.data.neo4j.annotation.StartNode;
@RelationshipEntity(type="HAS_LINE_ITEM_FOR") <-- LineItem jest relacją
public class LineItem {
   @GraphId <-- Identyfikator grafu
   private Long id;
   @StartNode <-- Węzeł początkowy
   private Order order;
   @EndNode <-- Węzeł końcowy
   private Product product;
   private int quantity;
   ...
}
```

Podczas gdy klasa Order została opatrzona adnotacją `@NodeEntity`, aby stać się węzłem, klasa `LineItem` opatrzona jest adnotacją `@RelationshipEntity`. Klasa `LineItem` posiada również właściwość `id` opatrzoną adnotacją `@GraphId`. Powtórzę jeszcze raz: wszystkie encje, zarówno encje węzłów, jak i encje relacji, muszą mieć identyfikator grafu typu `Long`.

Elementem wyróżniającym encje relacji jest to, że łączą ze sobą dwie encje węzłów. Do oznaczenia właściwości definiujących oba końce relacji służą adnotacje `@StartNode` i `@EndNode`. W naszym przypadku `Order` jest węzłem początkowym, a `Product` węzłem końcowym.

Na koniec klasa `LineItem` posiada właściwość `quantity`, która po utworzeniu relacji zostanie zapisana do bazy danych.

Teraz, po oznaczeniu adnotacjami domeny, możemy zacząć zapisywać i odczytywać węzły oraz relacje. Rozpoczniemy od poznania mechanizmu dostępu do danych Spring Data Neo4j przy użyciu szablonów za pośrednictwem `Neo4jTemplate`.

12.2.3. Pracujemy z Neo4jTemplate

Podobnie jak Spring Data MongoDB dostarcza klasę `MongoTemplate` w celu dostępu do danych przy użyciu szablonów, tak Spring Data Neo4j dostarcza klasę Neo4jTemplate, która umożliwia pracę z węzłami i relacjami w bazie grafowej Neo4j. Jeśli skonfigurowaliśmy Spring Data Neo4j w pokazany wcześniej sposób, komponent `Neo4jTemplate` jest już dostępny w kontekście aplikacji Springa. Musimy go tylko wstrzyknąć, gdy będzie nam potrzebny.

Przykładowo możemy skorzystać z metody autowiązania komponentów do właściwości komponentu:

```
@Autowired
private Neo4jOperations neo4j;
```

Neo4jTemplate definiuje kilkadziesiąt metod, wliczając w to metody do zapisywania i usuwania węzłów oraz tworzenia relacji pomiędzy węzłami. Nie mam wystarczająco dużo miejsca, by opisać je wszystkie, spójrzmy jednak na kilka tych najczęściej używanych.

Jedną z pierwszych i najbardziej elementarnych operacji, którą możemy chcieć wykonać z `Neo4jTemplate`, jest zapisanie obiektu jako węzła. Zakładając, że obiekt opatrzony jest adnotacją `@NodeEntity`, możemy to zrobić za pomocą metody `save()` w pokazany poniżej sposób:

```
Order order = ...;
Order savedOrder = neo4j.save(order);
```

Jeśli znamy identyfikator grafu przypisany do obiektu, możemy pobrać ten obiekt za pomocą metody `findOne()`:

```
Order order = neo4j.findOne(42, Order.class);
```

Jeżeli nie istnieje żaden węzeł o podanym identyfikatorze, metoda `findOne()` rzuci wyjątkiem `NotFoundException`.

Żeby pobrać wszystkie obiekty danego typu, możemy zastosować metodę `findAll()`:

```
EndResult<Order> allOrders = neo4j.findAll(Order.class);
```

Zwrócony tutaj obiekt `EndResult` jest typu `Iterable`, co pozwala na jego wykorzystanie w pętli `for-each` i w każdym innym miejscu, w którym możemy użyć typu `Iterable`. Jeśli nie istnieją żadne węzły danego typu, metoda `findAll()` zwróci pusty obiekt `Iterable`.

Jeżeli chcemy tylko sprawdzić, ile obiektów danego typu znajduje się w bazie Neo4j, możemy wywołać metodę `count()`:

```
long orderCount = count(Order.class);
```

Do usuwania obiektów służy z kolei metoda `delete()`:

```
neo4j.delete(order);
```

Jedną z najciekawszych metod dostarczanych przez `Neo4jTemplate` jest metoda `create`
`↪RelationshipBetween()`. Jak łatwo zgadnąć, służy ona do tworzenia relacji pomiędzy dwoma węzłami. Przykładowo moglibyśmy utworzyć relację `LineItem` pomiędzy węzłami `Order` i `Product`:

```
Order order = ...;
Product prod = ...;
LineItem lineItem = neo4j.createRelationshipBetween(
      order, prod, LineItem.class, "HAS_LINE_ITEM_FOR", false);
lineItem.setQuantity(5); neo4j.save(lineItem);
```

Pierwszymi dwoma parametrami metody `createRelationshipBetween()` są obiekty węzłów stanowiących oba końce relacji. Kolejny parametr określa typ oznaczony adnotacją `@RelationshipEntity`, który będzie reprezentował relację. Następnie określamy wartość typu `String`, która opisuje naturę relacji. Ostatni parametr, typu `boolean`, wskazuje, czy dopuszczalne są duplikaty relacji pomiędzy encjami węzłów.

Metoda `createRelationshipBetween()` zwraca instancję klasy relacji. W tym momencie możemy zdefiniować wszystko potrzebne nam właściwości. W powyższym przykładzie ustawiliśmy wartość właściwości `quantity`. Na koniec wywołujemy metodę `save()`, aby zapisać relację w bazie danych.

`Neo4jTemplate` udostępnia prosty sposób pracy z węzłami i relacjami w bazie grafowej Neo4j. Musimy jednak utworzyć samodzielnie własne implementacje repozytoriów, które oddelegowują pracę do `Neo4jTemplate`. Zobaczmy więc, jak wykorzystać mechanizm automatycznego generowania implementacji repozytoriów w Spring Data Neo4j.

12.2.4. Tworzymy automatyczne repozytoria Neo4j

Jedną z najwspanialszych funkcjonalności, dostępną w większości projektów Spring Data, jest automatyczne generowanie implementacji interfejsu repozytorium. Z mechanizmem tym spotkaliśmy się już przy okazji omawiania Spring Data JPA i Spring Data MongoDB. Spring Data Neo4j nie jest w tym zakresie wyjątkiem i także udostępnia taki mechanizm.

Naszą konfigurację opatrzyliśmy wcześniej adnotacją `@EnableNeo4jRepositories`, Spring Data Neo4j jest więc skonfigurowany do generowania repozytoriów. Musimy tylko utworzyć interfejsy. Na dobry początek rozpocznijmy od poniższego interfejsu `OrderRepository`:

```
package orders.db;
import orders.Order;
import org.springframework.data.neo4j.repository.GraphRepository;
public interface OrderRepository extends GraphRepository<Order> {}
```

Tak jak w przypadku innych projektów Spring Data, Spring Data Neo4j generuje repozytoria dla interfejsów rozszerzających interfejs `Repository`. W powyższym przykładzie interfejs `OrderRepository` rozszerza `GraphRepository`, który pośrednio rozszerza `Repository`. W czasie działania aplikacji Spring Data Neo4j wygeneruje więc implementację interfejsu `OrderRepository`.

Zauważ, że repozytorium `GraphRepository` sparametryzowane jest za pomocą typu `Order`, typu encji, z którą repozytorium ma pracować. Ponieważ w Neo4j identyfikator grafu musi mieć typ `Long`, przy rozszerzaniu interfejsu `GraphRepository` nie musimy wskazywać tego typu w sposób jawny.

Dzięki temu otrzymujemy za darmo kilka popularnych operacji CRUD, podobnie jak w przypadku `JpaRepository` i `MongoRepository`. W tabeli 12.4 znajduje się opis metod, które otrzymamy po rozszerzeniu interfejsu `GraphRepository`.

Nie mam wystarczająco dużo miejsca, aby opisać wszystkie te metody, ale kilka z nich bardzo Ci się przyda. Przykładowo poniższa linia kodu zapisuje pojedynczą encję typu `Order`:

```
Order savedOrder = orderRepository.save(order);
```

Po zapisaniu encji za pomocą metody `save()` encja jest zwracana wraz z ustawionym identyfikatorem grafu (pole opatrzone adnotacją `@GraphId`), który mógł mieć wcześniej wartość `null`.

Możemy wyszukać pojedynczą encję za pomocą metody `findOne()`. Przykładowo poniższa linia kodu wyszuka zamówienie o identyfikatorze grafu równym 4:

```
Order order = orderRepository.findOne(4L);
```

Możemy także wyszukać wszystkie zamówienia:

```
EndResult<Order> allOrders = orderRepository.findAll();
```

Mamy też oczywiście możliwość usunięcia encji. Służy do tego metoda `delete()`:

```
delete(order);
```

Tabela 12.4. Dzięki rozszerzeniu interfejsu GraphRepository interfejs repozytorium dziedziczy kilka operacji CRUD implementowanych automatycznie przez Spring Data Neo4j

Metoda	Opis
`long count();`	Zwraca liczbę encji docelowego typu zapisanych w bazie danych.
`void delete(Iterable<? extends T>);`	Usuwa kilka encji.
`void delete(Long id);`	Usuwa pojedynczą encję na podstawie jej identyfikatora.
`void delete(T);`	Usuwa pojedynczą encję.
`void deleteAll();`	Usuwa wszystkie encje docelowego typu.
`boolean exists(Long id);`	Sprawdza, czy istnieje encja o danym identyfikatorze.
`EndResult<T> findAll();`	Pobiera wszystkie encje docelowego typu.
`Iterable<T> findAll(Iterable<Long>);`	Pobiera wszystkie encje docelowego typu dla danych identyfikatorów.
`Page<T> findAll(Pageable);`	Pobiera posortowaną i postronicowaną listę wszystkich encji docelowego typu.
`EndResult<T> findAll(Sort);`	Pobiera posortowaną listę wszystkich encji docelowego typu.
`EndResult<T> findAllBySchemaPropertyValue(String, Object);`	Pobiera wszystkie encje, w których dana właściwość odpowiada podanej wartości.
`Iterable<T> findAllByTraversal(N, TraversalDescription);`	Pobiera wszystkie encje pozyskane w wyniku przeszukiwania grafu, rozpoczynając od danego węzła.
`T findBySchemaPropertyValue(String, Object);`	Pobiera pojedynczą encję, w której dana właściwość odpowiada podanej wartości.
`T findOne(Long);`	Pobiera pojedynczą encję na podstawie jej identyfikatora.
`EndResult<T> query(String, Map<String,Object>);`	Pobiera wszystkie encje odpowiadające danemu zapytaniu Cypher Query Language.
`Iterable<T> save(Iterable<T>);`	Zapisuje kilka encji.
`S save(S);`	Zapisuje pojedynczą encję.

Spowoduje to usunięcie z bazy danych wybranego węzła typu `Order`. Jeśli znamy tylko identyfikator grafu, do metody `delete()` możemy przekazać sam identyfikator zamiast całej encji:

```
delete(orderId);
```

W celu wywołania własnego zapytania moglibyśmy użyć metody `query()` i przekazać do niej arbitralne zapytanie Cypher Query Language. Nie różni się to jednak zbytnio od pracy z metodą `query()` na obiekcie `Neo4jTemplate`. Zamiast tego możemy dodać własną metodę do repozytorium `OrderRepository`.

DODAJEMY METODY ZAPYTAŃ

Dowiedziałeś się już, jak dodać metody zapytań, podążając za ustaloną konwencją nazewniczą, do repozytoriów Spring Data JPA i Spring Data MongoDB. Bylibyśmy straszliwie zawiedzeni, gdyby repozytorium Spring Data Neo4j nie udostępniało podobnych możliwości.

Listing 12.13 utwierdza nas jednak w przekonaniu, że nie mamy powodu czuć się rozczarowani.

Listing 12.13. Definiujemy metody zapytań, stosując się do konwencji nazewniczej

```
package orders.db;
import java.util.List;
import orders.Order;
import org.springframework.data.neo4j.repository.GraphRepository;
public interface OrderRepository
      extends GraphRepository<Order> {
   List<Order> findByCustomer(String customer);
   List<Order> findByCustomerAndType(String customer, String type);
}
```

Metody zapytań

Dodaliśmy dwie nowe metody zapytań. Jedna z nich wyszukuje wszystkie węzły typu `Order`, których właściwość `customer` ma wartość równą wartości typu `String` przekazanej jako parametr metody. Druga metoda jest bardzo podobna, ale oprócz parametru `customer` porównujemy również wartość parametru `type`.

Pisałem już wcześniej na temat konwencji nazewniczych metod zapytań, nie ma więc potrzeby zajmować się dłużej tym tematem. Szczegółowy opis tej konwencji i tworzenia odpowiednio nazwanych metod znajdziesz w poprzednim rozdziale, w części poświęconej Spring Data JPA.

TWORZYMY WŁASNE ZAPYTANIA

Gdy możliwości oferowane przez konwencję nazewniczą nie odpowiadają naszym oczekiwaniom, możemy też opatrzyć metody adnotacją `@Query` i zdefiniować własne zapytania. Widzieliśmy już wcześniej działanie tej adnotacji. W Spring Data JPA wykorzystywaliśmy ją do definiowania zapytania JPA dla metody repozytorium. W Spring Data MongoDB stosowaliśmy ją definiowania zapytań w formacie JSON. W Spring Data Neo4j musimy jednak używać zapytań Cypher:

```
@Query("match (o:Order)-[:HAS_ITEMS]->(i:Item) " +
      "where i.product='Spring w akcji' return o")
List<Order> findSiAOrders();
```

Metoda `findSiOrders()` opatrzona jest adnotacją `@Query`, przyjmującą jako wartość zapytanie Cypher do wyszukania wszystkich węzłów typu `Order` powiązanych z przedmiotem typu `Item`, którego właściwość `product` ma wartość `Spring w akcji`.

DOŁĄCZANIE WŁASNYCH ZACHOWAŃ DO REPOZYTORIUM

Kiedy zarówno konwencja nazewnicza, jak i metoda opatrzona adnotacją `@Query` nie są w stanie sprostać naszym oczekiwaniom, zawsze mamy możliwość dołączenia własnej logiki do repozytorium.

Przypuśćmy, że chcemy utworzyć własną implementację metody `findSiAOrders()`, nie wykorzystując adnotacji `@Query`. Możemy rozpocząć od zdefiniowania interfejsu pośredniczącego do przechowania definicji metody `findSiAOrders()`:

```
package orders.db;
import java.util.List;
import orders.Order;
public interface OrderOperations {
   List<Order> findSiAOrders();
}
```

Następnie modyfikujemy interfejs `OrderRepository` tak, aby oprócz interfejsu `Order➥Operations` rozszerzał również interfejs `GraphRepository`:

```
public interface OrderRepository
      extends GraphRepository<Order>, OrderOperations {
   ...
}
```

Na koniec musimy utworzyć implementację. Podobnie jak w przypadku Spring Data JPA i Spring Data MongoDB, Spring Data Neo4j wyszukuje klasy implementacji o takich samych nazwach, jakie noszą interfejsy repozytoriów, ale są one uzupełnione o sufiks `Impl`. Musimy więc stworzyć klasę `OrderRepositoryImpl`. Listing 12.14 zawiera kod klasy `OrderRepositoryImpl`, która implementuje metodę `findSiAOrders()`.

Listing 12.14. Dołączamy własną funkcjonalność do repozytorium OrderRepository

```
package orders.db;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import orders.Order;
import org.neo4j.helpers.collection.IteratorUtil;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.neo4j.conversion.EndResult;
import org.springframework.data.neo4j.conversion.Result;
import org.springframework.data.neo4j.template.Neo4jOperations;

public class OrderRepositoryImpl implements OrderOperations {   <-- Implementujemy interfejs pośredniczący
   private final Neo4jOperations neo4j;
@Autowired
   public OrderRepositoryImpl(Neo4jOperations neo4j) {   <-- Wstrzykujemy Neo4jOperations

      this.neo4j = neo4j;
   }

   public List<Order> findSiAOrders() {
      Result<Map<String, Object>> result = neo4j.query(   <-- Wykonujemy zapytanie
            "match (o:Order)-[:HAS_ITEMS]->(i:Item) " +
            "where i.product='Spring in Action' return o",
      EndResult<Order> endResult = result.to(Order.class);   <-- Konwertujemy na EndResult<Order>
      Return IteratorUtil.asList(endResult);   <-- Konwertujemy na List<Order>
   }

}
```

Do klasy `OrderRepositoryImpl` wstrzykiwany jest jako zależność komponent typu `Neo4j➥Operations` (w praktyce jest to instancja klasy `Neo4jTemplate`), umożliwiający wykonywanie zapytań na bazie. Metoda `query()` zwraca wynik typu `Result<Map<String, Object>>`,

musimy go więc skonwertować do postaci `List<Order>`. Pierwszym krokiem jest wywołanie metody `to()` na obiekcie `Result` w celu utworzenia obiektu `EndResult<Order>`. Następnie wykorzystujemy metodę Neo4j `IteratorUtil.asList()` do konwersji obiektu typu `EndResult<Order>` do postaci `List<Order>` i zwrócenia powstałego obiektu jako wyniku działania metody.

Bazy grafowe, takie jak Neo4j, sprawdzają się świetnie do przechowywania danych, które można w ładny sposób przedstawić w postaci węzłów i relacji. Jeśli założymy, że otaczający nas świat złożony jest z wielu elementów powiązanych ze sobą wzajemnymi relacjami, bazy grafowe wydają się doskonałym rozwiązaniem w wielu różnych sytuacjach. I przyznam uczciwie, że mam dużą słabość do Neo4j.

Czasem jednak nasze wymagania są dużo mniejsze. Niekiedy chcemy jedynie zapisać gdzieś jakąś wartość i odczytać ją później za pomocą ustalonego klucza. Poznajmy więc możliwości Spring Data w zakresie przechowywania danych w bazach typu klucz-wartość na przykładzie bazy Redis.

12.3. Pracujemy z danymi typu klucz-wartość z użyciem bazy Redis

Redis jest przedstawicielem baz danych nazywanych bazami klucz-wartość. Jak wskazuje sama nazwa, bazy te przechowują dane w postaci pary klucz-wartość. W praktyce bazy klucz-wartość mają wiele cech wspólnych z tablicą haszującą. Nazwanie tych baz trwałymi tablicami haszującymi nie jest zbyt wielkim uproszczeniem.

Gdy o tym pomyślimy, nie ma za dużo możliwości wykonania zapytań na tablicy haszującej… ani na bazie klucz-wartość. Możemy zapisać wartość pod jakimś kluczem i pobrać wartość dla podanego klucza. I to wszystko. W rezultacie obsługa automatycznych repozytoriów Spring Data nie ma zbytniego sensu w przypadku bazy Redis. Z drugiej strony inne istotne możliwości Spring Data, takie jak dostęp do bazy w oparciu o szablony, mogą już się okazać przydatne.

Spring Data Redis dostarcza kilku implementacji szablonów pozwalających na przechowywanie i pobieranie danych zapisanych w bazie Redis. Wkrótce dowiesz się, jak je wykorzystać. Do utworzenia szablonu Spring Data potrzebna jest nam fabryka połączenia z bazą Redis. Na szczęście Spring Data Redis dostarcza do wyboru aż cztery jej implementacje.

12.3.1. Łączymy się z Redisem

Fabryka połączeń Redisa umożliwia utworzenie połączenia z bazą danych serwera Redis. Spring Data Redis dostarcza fabryki połączeń dla czterech implementacji klienta Redisa:

- `JedisConnectionFactory`,
- `JredisConnectionFactory`,
- `LettuceConnectionFactory`,
- `SrpConnectionFactory`.

Wybór należy do Ciebie. Zachęcam do przeprowadzenia własnych testów i benchmarków, aby sprawdzić, który klient Redisa i która fabryka połączeń najlepiej odpowiadają Twoim potrzebom. Z perspektywy Spring Data Redis wszystkie fabryki połączeń są równie dobre.

Po dokonaniu wyboru możemy skonfigurować fabrykę połączeń jako komponent w Springu. Poniżej przedstawiono przykładową konfigurację komponentu `JedisConnec ↪tionFactory`:

```
@Bean
public RedisConnectionFactory redisCF() {
   return new JedisConnectionFactory();
}
```

W przykładzie utworzyliśmy instancję fabryki połączeń z wykorzystaniem jej domyślnego konstruktora. Otrzymana w ten sposób fabryka tworzy połączenia do serwera lokalnego na porcie 6379 bez podawania hasła. Jeśli nasz serwer Redisa działa na innej maszynie lub innym porcie, możemy podać te właściwości przy tworzeniu fabryki:

```
@Bean
public RedisConnectionFactory redisCF() {
   JedisConnectionFactory cf = new JedisConnectionFactory();
   cf.setHostName("redis-server");
   cf.setPort(7379);
   return cf;
}
```

W podobny sposób nasz serwer Redis mógł zostać skonfigurowany tak, aby wymagał uwierzytelniania od klientów. Możemy wówczas ustawić hasło za pomocą metody `setPassword()`:

```
@Bean
public RedisConnectionFactory redisCF() {
   JedisConnectionFactory cf = new JedisConnectionFactory();
   cf.setHostName("redis-server");
   cf.setPort(7379);
   cf.setPassword("foobared");
   return cf;
}
```

We wszystkich przykładach zakładaliśmy, że korzystamy z fabryki `JedisConnection ↪Factory`. Jeśli zdecydowalibyśmy się na inną opcję, to każda implementacja fabryki połączeń mogłaby zostać szybko podmieniona. Na przykład jeżeli postanowimy wykorzystać fabrykę `LettuceConnectionFactory`, możemy użyć następującej konfiguracji:

```
@Bean
public RedisConnectionFactory redisCF() {
   JedisConnectionFactory cf = new LettuceConnectionFactory();
   cf.setHostName("redis-server");
   cf.setPort(7379);
   cf.setPassword("foobared");
   return cf;
}
```

Wszystkie fabryki połączeń Redisa posiadają metody `setHostName()`, `setPort()` oraz `set➥Password()`. To sprawia, że pod względem konfiguracji są niemal identyczne.

Teraz gdy utworzyliśmy już fabrykę połączeń Redisa, możemy rozpocząć pracę z szablonami Spring Data Redis.

12.3.2. Pracujemy z klasą RedisTemplate

Jak wskazuje nazwa, fabryka połączeń Redisa produkuje połączenia (pod postacią obiektów `RedisConnection`) do bazy klucz-wartość Redis. Połączenie `RedisConnection` pozwala nam zapisywać i odczytywać dane. Przykładowo możemy ustanowić połączenie i wykorzystać je do zapisania powitania:

```
RedisConnectionFactory cf = ...;
RedisConnection conn = cf.getConnection();
conn.set("greeting".getBytes(), "Hello World".getBytes());
```

W podobny sposób możemy użyć połączenia RedisConnection do pobrania wartości pozdrowienia:

```
byte[] greetingBytes = conn.get("greeting".getBytes());
String greeting = new String(greetingBytes);
```

Bez wątpienia powyższy kod zadziała. Jednak czy naprawdę chcemy pracować z tablicami bajtów?

Podobnie jak pozostałe projekty Spring Data, Spring Data Redis udostępnia wysokopoziomową opcję dostępu do bazy z wykorzystaniem szablonów. Nawiasem mówiąc, Spring Data Redis udostępnia dwa szablony:

- `RedisTemplate`,
- `StringRedisTemplate`.

Klasa `RedisTemplate` w znacznym stopniu upraszcza dostęp do danych zapisanych w Redisie i pozwala na przechowywanie kluczy i wartości dowolnego typu, nie tylko tablicy bajtów. Jako że zarówno klucze, jak i wartości są najczęściej typu `String`, drugi z szablonów `StringRedisTemplate` rozszerza klasę `RedisTemplate` z naciskiem na obsługę danych tego typu.

Zakładając, że mamy już dostępną fabrykę `RedisConnectionFactory`, szablon `Redis➥Template` możemy skonstruować w następujący sposób:

```
RedisConnectionFactory cf = ...;
RedisTemplate<String, Product> redis =
      new RedisTemplate<String, Product>();
redis.setConnectionFactory(cf);
```

Zwróć uwagę, że klasa `RedisTemplate` sparametryzowana jest za pomocą dwóch typów. Pierwszym typem jest typ klucza, a drugim typ wartości. W powyższym przykładzie zapisywane będą obiekty typu `Product` pod kluczem typu `String`.

Jeśli wiemy, że będziemy korzystać z kluczy i wartości typu `String`, powinniśmy rozważyć zastąpienie szablonu `RedisTemplate` szablonem `StringRedisTemplate`:

```
RedisConnectionFactory cf = ...;
StringRedisTemplate redis = new StringRedisTemplate(cf);
```

Zauważ, że w przeciwieństwie do szablonu `RedisTemplate`, `StringRedisTemplate` posiada konstruktor, który przyjmuje jako parametr fabrykę `RedisConnectionFactory`. Nie ma zatem potrzeby wywoływania metody `setConnectionFactory()` po utworzeniu obiektu.

Chociaż nie jest to wymagane, to w przypadku częstego wykorzystywania szablonów `RedisTemplate` lub `StringRedisTemplate` warto je skonfigurować w postaci komponentu i wstrzykiwać, kiedy są potrzebne. Poniżej znajduje się metoda deklarująca komponent `RedisTemplate`:

```
@Bean
public RedisTemplate<String, Product>
      redisTemplate(RedisConnectionFactory cf) {
   RedisTemplate<String, Product> redis =
         new RedisTemplate<String, Product>();
   redis.setConnectionFactory(cf); return redis;
}
```

A oto metoda deklarująca komponent `StringRedisTemplate`:

```
@Bean
public StringRedisTemplate
      stringRedisTemplate(RedisConnectionFactory cf) {
   return new StringRedisTemplate(cf);
}
```

Po utworzeniu instancji `RedisTemplate` (albo `StringRedisTemplate`) możemy rozpocząć zapisywanie, pobieranie i usuwanie encji klucz-wartość. Większość operacji dostarczanych przez szablon `RedisTemplate` dostępna jest przez API pośredniczące pokazane w tabeli 12 5.

Jak widzisz, API pośredniczące pokazane w tabeli 12.5 dostępne jest poprzez metody szablonu `RedisTemplate` (i `StringRedisTemplate`). Każda z nich dostarcza operacje na encjach w oparciu o typ wartości, który może być typem prostym lub kolekcją.

Pośród tych wszystkich API pośredniczących dostępnych jest kilkadziesiąt metod umożliwiających zapis i pobieranie danych z Redisa. Nie ma tu wystarczającej ilości miejsca, aby opisać je wszystkie, ale spojrzymy na kilka najczęściej spotykanych operacji.

PRACUJEMY Z PROSTYMI WARTOŚCIAMI

Przypuśćmy, że chcemy zapisać produkt za pomocą szablonu `RedisTemplate<String, Product>`, gdzie kluczem jest wartość właściwości `sku`. Poniższy fragment kodu wykona tę czynność za pomocą metody `opsForValue()`:

```
redis.opsForValue().set(product.getSku(), product);
```

W podobny sposób możemy pobrać produkt, którego wartością `sku` jest `123456`:

```
Product product = redis.opsForValue().get("123456");
```

Jeśli nie zostanie znaleziona żadna encja o podanym kluczu, zwrócona zostanie wartość `null`.

PRACUJEMY Z LISTAMI

Praca z listami wartości jest równie prosta, a umożliwia ją metoda `opsForList()`. Możemy na przykład dodać wartość na koniec listy wpisów:

Tabela 12.5. Szablon RedisTemplate udostępnia dużą część swoich możliwości poprzez API pośredniczące, które odróżnia pojedyncze wartości od kolekcji

Metoda	Interfejs API pośredniczącego	Opis
opsForValue()	ValueOperations<K, V>	Operacje umożliwiające pracę z encjami przechowującymi proste wartości.
opsForList()	ListOperations<K, V>	Operacje umożliwiające pracę z encjami przechowującymi listy.
opsForSet()	SetOperations<K, V>	Operacje umożliwiające pracę z encjami przechowującymi zbiory.
opsForZSet()	ZSetOperations<K, V>	Operacje umożliwiające pracę z encjami przechowującymi zbiory uporządkowane (ZSet).
opsForHash()	HashOperations<K, HK, HV>	Operacje umożliwiające pracę z encjami przechowującymi tablice haszujące.
boundValueOps(K)	BoundValueOperations<K,V>	Operacje umożliwiające pracę z prostymi wartościami powiązanymi z danym kluczem.
boundListOps(K)	BoundListOperations<K,V>	Operacje umożliwiające pracę z listami powiązanymi z danym kluczem.
boundSetOps(K)	BoundSetOperations<K,V>	Operacje umożliwiające pracę ze zbiorami powiązanymi z danym kluczem.
boundZSet(K)	BoundZSetOperations<K,V>	Operacje umożliwiające pracę ze zbiorami uporządkowanymi (ZSet) powiązanymi z danym kluczem.
boundHashOps(K)	BoundHashOperations<K,V>	Operacje umożliwiające pracę z tablicami haszującymi powiązanymi z danym kluczem.

```
redis.opsForList().rightPush("koszyk", product);
```

Spowoduje to dodanie encji `Product` na koniec listy przechowywanej pod kluczem `cart`. Jeśli nie istnieje jeszcze lista dla tego klucza, to zostanie utworzona.

Metoda `pushRight()` dodaje element na koniec listy. Aby dodać wartość na początek listy, musimy wykorzystać metodę `leftPush()`:

```
redis.opsForList().leftPush("koszyk", product);
```

Jest kilka sposobów pobierania elementów listy. Możemy wyciągnąć element z początku lub końca listy za pomocą, odpowiednio, metody `leftPop()` oraz `rightPop()`:

```
Product first = redis.opsForList().leftPop("koszyk");
Product last = redis.opsForList().rightPop("koszyk");
```

Poza tym, że pobierają wartości z list, obie metody `pop` mają efekt uboczny w postaci usuwania wyciągniętych elementów z listy. Jeśli chcielibyśmy tylko odczytać wartość (być może nawet ze środka listy), moglibyśmy wykorzystać metodę `range()`:

```
List<Product> products = redis.opsForList().range("koszyk", 2, 12);
```

Metoda `range()` nie usuwa żadnych wartości z listy, ale pobiera jedną bądź więcej wartości na podstawie klucza i zakresu indeksów. Powyższy przykład pobiera do jedenastu wpisów, rozpoczynając od wpisu o indeksie 2 i pobierając je kolejno aż do indeksu 12 (włącznie). Jeżeli zakres przekracza ograniczenia listy, to zwrócone zostaną jedynie wpisy mieszczące się w zakresie indeksów. Jeśli żadne wpisy nie są dostępne w ramach podanych indeksów, zwracana jest pusta lista.

PRZEPROWADZAMY OPERACJE NA ZBIORACH

Poza listami możemy też wykorzystać do pracy zbiory za pośrednictwem metody `opsForSet()`. Najbardziej elementarną operacją, jaką możemy wykonać, jest dodanie elementu zbioru wpisów:

```
redis.opsForSet().add("koszyk", product);
```

Po utworzeniu i wypełnieniu wartościami kilku wpisów w zbiorze możemy przeprowadzić interesujące operacje na tych zbiorach, takie jak znajdowanie różnic pomiędzy nimi, znajdowanie części wspólnej oraz łączenie zbiorów.

```
List<Product> diff = redis.opsForSet().difference("koszyk1", "koszyk2");
List<Product> union = redis.opsForSet().union("koszyk1", "koszyk2");
List<Product> isect = redis.opsForSet().isect("koszyk1", "koszyk2");
```

Możemy też oczywiście usuwać element:

```
redis.opsForSet().remove(product);
```

Możemy nawet pobrać losowy element ze zbioru:

```
Product random = redis.opsForSet().randomMember("koszyk");
```

Zbiory nie mają indeksów ani nie są w jawny sposób posortowane, nie możemy więc wskazać i pobrać pojedynczego elementu.

WIĄZANIE DO KLUCZA

Tabela 12.5 zawiera pięć API pośredniczących w pracy z operacjami powiązanymi z danym kluczem. Niektóre API odzwierciedlają funkcjonalność innych API, ale ich działanie dotyczy wybranego klucza.

Żeby pokazać przykład ich wykorzystania, załóżmy, że przechowujemy obiekty `Product` w postaci listy wpisów, których kluczem jest `koszyk`. W tym scenariuszu zakładamy, że chcemy wyciągnąć element z prawego końca listy, a następnie dodać trzy nowe elementy na jej koniec. Możemy to zrobić za pomocą obiektu typu `BoundListOperations`, zwracanego po wywołaniu metody `boundListOps()`:

```
BoundListOperations<String, Product> cart = redis.boundListOps("koszyk");
Product popped = cart.rightPop();
cart.rightPush(product1);
cart.rightPush(product2);
cart.rightPush(product3);
```

Zauważ, że klucz wpisu wykorzystywany jest tylko raz, w wywołaniu metody `boundListOps()`. Wszystkie operacje przeprowadzone na zwróconym obiekcie typu `BouldListOperations` dotyczą tego właśnie wpisu.

12.3.3. Ustawiamy serializatory kluczy i wartości

Po zapisaniu wpisu w bazie Redis zarówno klucz, jak i wartość poddawane są serializacji z użyciem serializatora Redisa. Spring Data Redis dostarcza kilka takich serializatorów, w tym między innymi:

- `GenericToStringSerializer` — serializacja z użyciem usługi konwersji Springa;
- `JacksonJsonRedisSerializer` — serializacja obiektów na JSON z wykorzystaniem biblioteki Jackson 1;
- `Jackson2JsonRedisSerializer` — serializacja obiektów na JSON z użyciem biblioteki Jackson 2;
- `JdkSerializationRedisSerializer` — wykorzystanie wbudowanej serializacji Javy;
- `OxmSerializer` — serializacja z użyciem mechanizmów przekształcania obiektów na XML (*marshalling*) i XML na obiekty (*unmarshalling*) z wykorzystaniem mechanizmów Springa odwzorowania O/X;
- `StringRedisSerializer` — serializacja kluczy i wartości typu `String`.

Wszystkie te serializatory implementują interfejs `RedisSerializer`, jeśli więc żaden z nich nie spełnia naszych oczekiwań, zawsze możemy utworzyć swój własny serializator.

Klasa `RedisTemplate` wykorzystuje klasę `JdkSerializationRedisSerializer`, co oznacza, że serializacja kluczy i wartości przeprowadzana jest z użyciem mechanizmów serializacji Javy. Jak można przypuszczać, domyślnie `StringRedisTemplate` i `StringRedisSeria`
`↪lizer` konwertują wartości string do i z tablicy bajtów. Te domyślne ustawienia są odpowiednie w większości przypadków, ale czasem przydatne może być zastosowanie innych serializatorów.

Przypuśćmy, że używamy szablonu `RedisTemplate` i chcemy zserializować wartości typu Product na format JSON, korzystając z kluczy typu String. Metody `setKeySeria`
`↪lizer()` oraz `setValueSerializer()` klasy `RedisTemplate` są tym, czego potrzebujemy:

```
@Bean
public RedisTemplate<String, Product>
      redisTemplate(RedisConnectionFactory cf) {
   RedisTemplate<String, Product> redis =
      new RedisTemplate<String, Product>();
   redis.setConnectionFactory(cf);
   redis.setKeySerializer(new StringRedisSerializer());
   redis.setValueSerializer(
      new Jackson2JsonRedisSerializer<Product>(Product.class));
   return redis;
}
```

W przykładzie ustawiliśmy komponent `RedisTemplate` tak, aby do serializacji wartości kluczy zawsze wykorzystywał serializator `StringRedisSerializer`. Do serializacji wartości typu `Product` używamy z kolei wyłącznie serializatora `Jackson2JsonRedisSerializer`.

12.4. Podsumowanie

Minęły już czasy, gdy jedyną opcją przechowywania danych były bazy relacyjne. Istnieje obecnie kilka różnych rodzajów baz danych, a każdy z nich reprezentuje dane w innej postaci i oferuje możliwości odpowiadające wielu modelom domeny. Projekt Spring Data pozwala programistom na używanie tych baz danych w aplikacjach i wykorzystywanie warstwy abstrakcji, która jest względnie spójna pomiędzy różnymi dostępnymi bazami.

W tym rozdziale korzystaliśmy z wiedzy zdobytej w poprzednim rozdziale, dotyczącym Spring Data i JPA, i zastosowaliśmy ją w bazie dokumentowej MongoDB oraz bazie grafowej Neo4j. Spring Data MongDB i Spring Data Neo4j oferują, podobnie jak ich odpowiednik Spring Data JPA, mechanizmy automatycznego generowania repozytorium w oparciu o definicję interfejsu. Dodatkowo dowiedziałeś się, jak wykorzystać adnotacje dostarczane przez projekt Spring Data do odwzorowywania klas domenowych na dokumenty, węzły i relacje.

Spring Data umożliwia również zapis danych w bazie Redis, będącej bazą typu klucz-wartość. Bazy klucz-wartość są wyraźnie prostsze i nie wymagają obsługi automatycznego repozytorium ani adnotacji odwzorowywania danych. Niezależnie od tego Spring Data Redis udostępnia dwie różne klasy szablonów do pracy z bazą Redis.

Bez względu na wybraną bazę pobieranie danych z każdej bazy jest operacją kosztowną. W praktyce zapytania na bazie danych są często największym wąskim gardłem aplikacji. Teraz gdy już wiesz, jak zapisywać i pobierać dane z różnych źródeł, popatrzmy, jak zapobiec powstaniu tego wąskiego gardła. W następnym rozdziale zobaczysz, jak wykorzystać mechanizm deklaratywnego cachowania do uniknięcia niepotrzebnych odczytów z bazy.

13 Cachowanie danych

W tym rozdziale omówimy:

- Włączenie deklaratywnego cachowania
- Cachowanie z użyciem Ehcache, Redisa oraz GemFire
- Cachowanie w oparciu o adnotacje

Czy kiedykolwiek ktoś zadał Ci pytanie, a chwilę po otrzymaniu odpowiedzi zadał je ponownie? Moje dzieci bardzo często zadają mi tego typu pytania:

„Mogę cukierka?",

„Która godzina?",

„Dojechaliśmy już?",

„Mogę cukierka?".

Pod wieloma względami komponenty tworzonej przez nas aplikacji działają w ten sam sposób. Bezstanowe komponenty lepiej się skalują, ale zadają często te same pytania. Ponieważ nie przechowują stanu, to po zakończeniu zadania nie pamiętają żadnej uzyskanej odpowiedzi. Następnym razem, gdy jej potrzebują, muszą więc ponownie zadać to samo pytanie.

Czasem odpowiedź na to pytanie wiąże się z pobraniem jakichś danych lub wykonaniem jakiegoś wyliczenia. Być może musimy pobrać dane z bazy, skorzystać ze zdalnej usługi albo przeprowadzić złożone obliczenia. Wszystko to zabiera cenny czas i zasoby potrzebne do udzielenia odpowiedzi.

Jeśli odpowiedź nie zmienia się często (bądź wcale), marnotrawstwem jest nieustanne powtarzanie tych czynności. Co więcej, może to mieć negatywny wpływ na wydajność naszej aplikacji. Zamiast więc zadawać wciąż to same pytanie, by za każdym razem otrzymać tę samą odpowiedź, lepiej zapytać raz i zapamiętać odpowiedź na przyszłość.

Cachowanie jest sposobem przechowywania często potrzebnych informacji, aby można z nich było łatwo skorzystać. W tym rozdziale spojrzymy na warstwę abstrakcji cachowania dostępną w Springu. Chociaż sam Spring nie implementuje żadnego rozwiązania w zakresie cachowania, to udostępnia deklaratywne wsparcie dla cachowania, które doskonale integruje się z kilkoma popularnymi implementacjami cachowania.

13.1. Włączamy obsługę cachowania

Warstwa abstrakcji cachowania w Springu jest dostępna w dwóch postaciach:

- cachowanie w oparciu o adnotacje;
- cachowanie w oparciu o deklaracje XML.

Najczęstszym sposobem wykorzystania warstwy abstrakcji cachowania w Springu jest oznaczenie metod adnotacjami `@Cacheable` oraz `@CacheEvict`. W tym rozdziale większość czasu spędzimy z tą formą deklaratywnego cachowania. Następnie w podrozdziale 13.3 dowiesz się, jak można zadeklarować granice cachowania w plikach XML.

Zanim rozpoczniemy dodawanie do naszych komponentów adnotacji cachowania, musimy włączyć w konfiguracji Springa obsługę cachowania w oparciu o adnotacje. Jeśli korzystamy z konfiguracji Java, możemy włączyć cachowanie w oparciu o adnotacje, dodając adnotację `@EnableCaching` do jednej z klas konfiguracji. Listing 13.1 przedstawia adnotację `@EnableCaching` w akcji.

Listing 13.1. Włączamy cachowanie w oparciu o adnotacje za pomocą adnotacji @EnableCaching

```
package com.habuma.cachefun;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.concurrent.ConcurrentMapCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableCaching <-- Włączamy cachowanie
public class CachingConfig {
   @Bean
   public CacheManager cacheManager() { <-- Deklarujemy menedżera cachowania
      return new ConcurrentMapCacheManager();
   }
}
```

Jeżeli korzystamy z konfiguracji XML, możemy włączyć cachowanie w oparciu o adnotacje za pomocą elementu `<cache:annotation-driven>` z przestrzeni nazw cache Springa, co prezentuje listing 13.2.

Listing 13.2. Włączamy cachowanie w oparciu o adnotacje za pomocą elementu <cache:annotation-driven>

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cache="http://www.springframework.org/schema/cache"
       xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/cache
          http://www.springframework.org/schema/cache/spring-cache.xsd">
   <cache:annotation-driven />                                    Włączamy cachowanie
   <bean id="cacheManager" class=                                 Deklarujemy
      "org.springframework.cache.concurrent.ConcurrentMapCacheManager" />   menedżera
</beans>                                                                     cachowania
```

Mechanizm działania adnotacji `@EnableCaching` i elementu `<cache:annotation-driven>` jest identyczny. Tworzą one aspekt z punktami przecięcia i wywołują adnotacje cachowania Springa. W zależności od wykorzystanej adnotacji oraz stanu pamięci podręcznej cache aspekt ten pobierze, doda lub usunie wartość z pamięci cache.

Łatwo zauważyć, że kod na listingach 13.1 i 13.2 robi coś więcej niż tylko włączenie cachowania za pomocą adnotacji. Deklarujemy tam również komponent menedżera pamięci podręcznej. Menedżery pamięci podręcznej są sercem abstrakcji mechanizmu cachowania w Springu i umożliwiają integrację z jedną z kilku popularnych implementacji cachowania.

W naszych przykładach zadeklarowaliśmy menedżera `ConcurrentMapCacheManager`. Jest to prosty mechanizm pamięci podręcznej, wykorzystujący do przechowywania danych mapę `java.util.concurrent.ConcurrentHashMap`. Prostota tego rozwiązania czyni go świetnym wyborem na czas tworzenia aplikacji, testowania bądź w prostych aplikacjach. Ponieważ jednak cachowanie danych przebiega w pamięci i w związku z tym jest ściśle powiązane z cyklem życia aplikacji, nie jest to najlepszy wybór dla dużych aplikacji produkcyjnych.

Na szczęście mamy do dyspozycji kilka świetnych menedżerów pamięci podręcznej. Przyjrzyjmy się tym najczęściej używanym.

13.1.1. Konfigurujemy menedżera pamięci podręcznej

Spring 3.1 posiada wbudowaną obsługę pięciu implementacji menedżera pamięci podręcznej:

- `SimpleCacheManager`,
- `NoOpCacheManager`,
- `ConcurrentMapCacheManager`,
- `CompositeCacheManger`,
- `EhCacheCacheManager`.

W Springu 3.2 pojawił się kolejny menedżer pamięci podręcznej do pracy z dostawcami cachowania opartymi na specyfikacji JCache (JSR-107). Spring Data dostarcza dodatkowo dwa menedżery pamięci podręcznej, które nie zostały włączone do jądra Spring Framework:

- `RedisCacheManager` (z projektu Spring Data Redis);
- `GemfireCacheManager` (z projektu Spring Data Gemfire).

Jak widać, mamy duże możliwości wyboru menedżera pamięci podręcznej na potrzeby warstwy abstrakcji cachowania Springa. Wybór zależeć będzie od wykorzystywanego dostawcy cachowania. Wszystkie te rozwiązania dostarczają aplikacji różne możliwości cachowania, a niektóre z nich bardziej niż inne nadają się do użycia w środowisku produkcyjnym. Chociaż dokonany wybór wpływa na sposób, w jaki dane zostaną zapisane, nie rzutuje na sposób deklaracji reguł cachowania w Springu.

Musimy wybrać i skonfigurować menedżera pamięci podręcznej jako komponent w kontekście aplikacji Springa. Widziałeś już, jak skonfigurować menedżera `ConcurrentMapCacheManager`, i dowiedziałeś się, że w prawdziwych aplikacjach może to nie być najlepszy wybór. Zobaczmy teraz, jak skonfigurować część z pozostałych menedżerów pamięci podręcznej, rozpoczynając od menedżera `EhCacheCacheManager`.

CACHOWANIE Z UŻYCIEM EHCACHE

Ehcache jest jednym z najpopularniejszych systemów cachowania. Według informacji na stronie internetowej Ehcache jest to „najczęściej wykorzystywane rozwiązanie cachowania w Javie". Biorąc pod uwagę jego dużą popularność, wydaje się rozsądne, aby Spring dostarczał menedżera pamięci podręcznej umożliwiającego integrację z Ehcache. Menedżerem tym jest `EhCacheCacheManager`.

Jak już przyzwyczaimy się do nazwy, która wygląda trochę tak, jakby ktoś zająknął się przy jej tworzeniu, okazuje się, że konfiguracja EhCache w Springu jest nad wyraz prosta. Listing 13.3 prezentuje konfigurację EhCache z użyciem klasy Javy.

Listing 13.3. Konfiguracja EhCacheCacheManager z użyciem klasy Javy

```
package com.habuma.cachefun;
import net.sf.ehcache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.ehcache.EhCacheCacheManager;
import org.springframework.cache.ehcache.EhCacheManagerFactoryBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
@Configuration
@EnableCaching
public class CachingConfig {
   @Bean
   public EhCacheCacheManager cacheManager(CacheManager cm) {
      return new EhCacheCacheManager(cm);
   }
   @Bean
   public EhCacheManagerFactoryBean ehcache() {
      EhCacheManagerFactoryBean ehCacheFactoryBean =
```

Konfigurujemy EhCacheCacheManager

EhCacheManagerFactoryBean

```
        new EhCacheManagerFactoryBean();
    ehCacheFactoryBean.setConfigLocation(
        new ClassPathResource("com/habuma/spittr/cache/ehcache.xml"));
    return ehCacheFactoryBean;
  }
}
```

Metoda `cacheManager()` na listingu 13.3 tworzy instancję `EhCacheCacheManager` poprzez przekazanie instancji menedżera Ehcache `CacheManager`. Ten element wstrzykiwania może być mylący, bo zarówno Spring, jak i Ehcache definiują typ `CacheManager`. Mówiąc dokładnie, menedżer pamięci podręcznej Ehcache wstrzykiwany jest do menedżera Springa `EhCacheCacheManager` (który jest implementacją interfejsu `CacheManager` Springa).

Kiedy już mamy gotowego do wstrzyknięcia menedżera pamięci podręcznej Ehcache, musimy również zadeklarować komponent `CacheManager`. Aby uprościć to zadanie, Spring dostarcza komponent `EhCacheManagerFactoryBean`, który generuje menedżera Ehcache `CacheManager`. Do utworzenia instancji `EhCacheManagerFactoryBean` służy metoda `ehcache()`. Ponieważ jest to komponent fabryki (co oznacza, że implementuje interfejs Springa `FactoryBean`), w kontekście aplikacji Springa zarejestrowana nie jest instancja `EhCacheManagerFactoryBean`, ale instancja menedżera `CacheManager`, gotowa do wstrzyknięcia do `EhCacheCacheManager`.

Ustawianie komponentów to nie wszystkie możliwości konfiguracji Ehcache. Ehcache definiuje swój własny schemat konfiguracji XML i szczegóły konfiguracji ustawimy z wykorzystaniem tego właśnie schematu w pliku XML. Lokalizację tego pliku XML musimy wskazać przy tworzeniu komponentu fabryki `EhCacheManagerFactoryBean`. Stosujemy tutaj metodę `setConfigLocation()`, przekazując do niej instancję klasy `ClassPathResource`, żeby wskazać względną wobec ścieżki klas lokalizację do pliku XML Ehcache.

Zawartość pliku *ehcache.xml* może się różnić w zależności od aplikacji, musimy jednak zadeklarować przynajmniej podstawowe ustawienia. Przykładowo poniższa konfiguracja deklaruje pamięć podręczną `spittleCache` o maksymalnej pojemności sterty 50 MB i czasie przechowywania danych wynoszącym 100 sekund.

```
<ehcache>
  <cache name="spittleCache"
         maxBytesLocalHeap="50m"
         timeToLiveSeconds="100">
  </cache>
</ehcache>
```

Jest to z pewnością podstawowa konfiguracja Ehcache. W naszych aplikacjach będziemy chcieli skorzystać z szerokiego wachlarza opcji konfiguracji dostarczonych przez Ehcache. Więcej informacji na temat szczegółów konfiguracji Ehcache można znaleźć pod adresem *http://ehcache.org/documentation/configuration*.

CACHOWANIE Z UŻYCIEM REDISA

Gdy się temu dobrze przyjrzymy, to wpis w pamięci podręcznej nie jest niczym innym niż parą klucz-wartość, w której klucz określa operacje i parametry, a na ich podstawie zwracana jest przechowywana wartość. Nikogo nie powinno więc dziwić, że baza klucz-wartość Redis nadaje się świetnie jako pamięć podręczna.

Do przechowywania wpisów pamięci podręcznej w Redisie za pośrednictwem warstwy abstrakcji Springa służy menedżer `RedisCacheManager`, dostarczany w ramach projektu Spring Data Redis i będący implementacją interfejsu `CacheManager`.

Menedżer `RedisCacheManager` komunikuje się z serwerem Redisa za pośrednictwem szablonu `RedisTemplate` i umożliwia przechowywanie w nim wpisów pamięci podręcznej.

Do pracy z menedżerem `RedisCacheManager` potrzebny jest nam komponent `RedisTemplate` oraz komponent implementujący interfejs `RedisConnectionFactory` (taki jak `JedisConnectionFactory`). W rozdziale 12. dowiedziałeś się, jak skonfigurować te komponenty. Gdy przygotujemy już te niezbędne komponenty, możemy przystąpić do konfigurowania menedżera `RedisCacheManager`, co pokazuje listing 13.4.

Listing 13.4. Konfigurujemy menedżera pamięci podręcznej do przechowywania wpisów w bazie Redis

```
package com.myapp;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Bean;
import org.springframework.data.redis.cache.RedisCacheManager;
import org.springframework.data.redis.connection.jedis
      .JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;

@Configuration
@EnableCaching
public class CachingConfig {
   @Bean
   public CacheManager cacheManager(RedisTemplate redisTemplate) {
      return new RedisCacheManager(redisTemplate);
   }
   @Bean
   public JedisConnectionFactory redisConnectionFactory() {
      JedisConnectionFactory jedisConnectionFactory
         new JedisConnectionFactory();
      jedisConnectionFactory.afterPropertiesSet();
      return jedisConnectionFactory;
   }
   @Bean
   public RedisTemplate<String, String> redisTemplate(
         RedisConnectionFactory redisCF) {
      RedisTemplate<String, String> redisTemplate =
         new RedisTemplate<String, String>();
      redisTemplate.setConnectionFactory(redisCF);
      redisTemplate.afterPropertiesSet();
      return redisTemplate;
   }
}
```

Menedżer pamięci podręcznej Redisa

Komponent fabryki połączeń Redisa

Komponent RedisTemplate

Jak widzisz, utworzyliśmy instancję menedżera `RedisCacheManager`, przekazując instancję szablonu `RedisTemplate` w postaci argumentu konstruktora.

PRACUJEMY Z WIELOMA MENEDŻERAMI PAMIĘCI PODRĘCZNEJ

Nie ma żadnych przyczyn, dla których nie moglibyśmy skorzystać z więcej niż jednego menedżera pamięci podręcznej. Jeśli masz problem z wyborem menedżera pamięci podręcznej lub jeśli masz jakiś techniczny powód do użycia więcej niż jednego menedżera, możesz wypróbować komponent Springa `CompositeCacheManager`.

`CompositeCacheManager` skonfigurowany jest za pomocą jednego bądź większej liczby menedżerów pamięci podręcznej, iteruje się po nich wszystkich i próbuje znaleźć zapisaną wcześniej wartość. Listing 13 5 prezentuje sposób tworzenia komponentu `CompositeCacheManager`, który będzie się iterował po menedżerach `JCacheCacheManager`, `EhCacheCacheManager` oraz `RedisCacheManager`.

Listing 13.5. Menedżer CompositeCacheManager iteruje się po liście menedżerów pamięci podręcznej

```
@Bean
public CacheManager cacheManager(
      net.sf.ehcache.CacheManager cm,
      javax.cache.CacheManager jcm) {
   CompositeCacheManager cacheManager = new CompositeCacheManager();
   List<CacheManager> managers = new ArrayList<CacheManager>();
   managers.add(new JCacheCacheManager(jcm));
   managers.add(new EhCacheCacheManager(cm))
   managers.add(new RedisCacheManager(redisTemplate()));
   cacheManager.setCacheManagers(managers);
   return cacheManager;
}
```

Tworzymy menedżera CompositeCacheManager

Dodajemy poszczególne menedżery pamięci podręcznej

Gdy nadejdzie pora odczytania danych z pamięci podręcznej, menedżer `CompositeCache`
`↪Manager` rozpoczyna wyszukiwanie wpisu od menedżera `JCacheCacheManager` w celu sprawdzenia implementacji JCache, następnie sprawdza Ehcache z użyciem menedżera `EhCacheCacheManager`, a na koniec szuka wpisu w bazie Redis z wykorzystaniem menedżera `RedisCacheManager`.

Po skonfigurowaniu menedżera pamięci podręcznej i włączeniu cachowania możemy rozpocząć dodawanie reguł cachowania do metod komponentów. Zobaczmy, jak użyć adnotacji Springa do zdefiniowania cachowania danych.

13.2. Stosowanie adnotacji cachowania na poziomie metod

Jak wspomniałem wcześniej, warstwa abstrakcji Springa w dużym stopniu opiera się na aspektach. Po włączeniu cachowania w Springu tworzony jest aspekt, który wywołuje jedną lub kilka adnotacji cachowania Springa. Tabela 13.1 pokazuje adnotacje cachujące Springa.

Wszystkie wymienione w tabeli 13.1 adnotacje mogą być umieszczone zarówno na poziomie metody, jak i na poziomie klasy. Po opatrzeniu adnotacją metody opisane w niej reguły cachowania dotyczą wyłącznie wskazanej metody. Po umieszczeniu adnotacji na poziomie klasy reguły zastosowane są do wszystkich metod tej klasy.

Tabela 13.1. Spring dostarcza cztery adnotacje dla deklaracji reguł cachowania

Adnotacje	Opis
`@Cacheable`	Wskazuje, że Spring powinien sprawdzić, czy wartość zwracana przez metodę jest zapisana w pamięci podręcznej, zanim nastąpi wywołanie tej metody. Jeśli wartość zostanie odnaleziona w pamięci podręcznej, jest zwracana. W przeciwnym wypadku metoda jest wywoływana, a wartość jest zapisywana w pamięci podręcznej.
`@CachePut`	Wskazuje, że Spring powinien zapisać w pamięci podręcznej wartość zwróconą z metody. Pamięć podręczna nie jest sprawdzana przed inwokacją metody, a metoda jest zawsze wywoływana.
`@CacheEvict`	Wskazuje, że Spring powinien wyrzucić jeden lub więcej wpisów z pamięci podręcznej.
`@Caching`	Adnotacja grupująca, służąca do równoczesnego zastosowania wielu innych adnotacji.

13.2.1. Zapisujemy dane w pamięci podręcznej

Jak widać w tabeli, obie adnotacje, `@Cacheable` oraz `@CachePut`, mogą służyć do umieszczenia danych w pamięci podręcznej. Działają jednak w trochę odmienny sposób.

Adnotacja `@Cacheable` szuka najpierw wpisu w pamięci podręcznej i pomija wywołanie metody w przypadku jego odnalezienia. Jeśli wpis nie zostanie znaleziony, metoda jest wywoływana, a w pamięci podręcznej zapisywana jest wartość zwrócona przez metodę. Adnotacja `@CachePut` z kolei nie sprawdza nigdy pasujących wartości w pamięci cache, zawsze wywołuje metodę, a zwróconą przez nią wartość zapisuje w pamięci podręcznej.

Adnotacje `@Cacheable` i `@CachePut` współdzielą kilka atrybutów, wypisanych poniżej, w tabeli 13.2.

Tabela 13.2. Atrybuty @Cacheable i @CachePut współdzielą zbiór atrybutów

Atrybut	Typ	Opis
`value`	`String[]`	Nazwa wykorzystywanej pamięci podręcznej.
`condition`	`String`	Wyrażenie SpEL, które w przypadku zwrócenia wartości `false` skutkuje nieumieszczeniem wyniku wywoływanej metody w pamięci cache.
`key`	`String`	Wyrażenie SpEL służące do wyliczania niestandardowego klucza cachowania.
`unless`	`String`	Wyrażenie SpEL, które w przypadku zwrócenia wartości `true` zapobiega umieszczeniu w pamięci podręcznej wartości zwracanej przez metodę.

W swojej najprostszej postaci atrybuty `@Cacheable` oraz `@CachePut` określają jedynie nazwę jednej lub większej liczby pamięci podręcznych za pomocą atrybutu `value`. Przykładowo rozważmy metodę `findOne()` z repozytorium `SpittleRepository`. Obiekt wiadomości `Spittle` po zapisaniu z reguły nigdy się nie zmienia. Jeśli jakaś wiadomość jest szczególnie często wyświetlana, marnotrawstwem jest każdorazowe odpytywanie bazy danych o ten konkretny wpis. Poprzez oznaczenie metody `findOne()` adnotacją `@Cacheable` możemy sprawić, że wiadomość zostanie zapisana w pamięci podręcznej, a dzięki temu unikniemy zbędnych zapytań do bazy. Możemy to zaobserwować na listingu 13.6.

Listing 13.6. Wykorzystujemy adnotację @Cacheable do zapisywania i pobierania wartości z pamięci podręcznej

```
@Cacheable("spittleCache") ← Zapisujemy wynik metody w pamięci podręcznej
public Spittle findOne(long id) {
   try {
      return jdbcTemplate.queryForObject(
         SELECT_SPITTLE_BY_ID, new SpittleRowMapper(), id);
   } catch (EmptyResultDataAccessException e) {
      return null;
   }
}
```

Po wywołaniu metody `findOne()` aspekt cachowania przechwytuje wywołanie i w pamięci podręcznej `spittleCache` szuka wartości zwracanej przez tę metodę. Kluczem cachowania jest parametr `id` przekazany do metody `findOne()`. Jeśli dla danego klucza odnaleziona zostanie zapisana wartość, zostanie ona zwrócona, a metoda nie zostanie wywołana. Z drugiej strony, jeżeli wartość nie zostanie odnaleziona, metoda zostanie wywołana, a zwrócona wartość trafi do pamięci podręcznej, przygotowana na kolejne wywołania metody `findOne()`.

W listingu 13.6 adnotacją `@Cacheable` opatrzono implementację metody `findOne()` repozytorium `JdbcSpittleRepository`. Żadna inna implementacja repozytorium `Spittle` ↪`Repository` nie będzie cachowana, dopóki nie zostanie opatrzona adnotacją `@Cacheable`. Warto się więc zastanowić, czy nie umieścić adnotacji na deklaracji metody w interfejsie `SpittleRepository` zamiast na jej implementacji:

```
@Cacheable("spittleCache")
Spittle findOne(long id);
```

Po oznaczeniu metody interfejsu adnotacją `@Cacheable` zostanie ona odziedziczona przez wszystkie implementacje interfejsu `SpittleRepository` i wszystkie będą korzystać z tych samych reguł cachowania.

UMIESZCZAMY WARTOŚCI W PAMIĘCI PODRĘCZNEJ

Podczas gdy adnotacja `@Cacheable` warunkowo wywołuje metodę w zależności od tego, czy pożądana wartość znajduje się w pamięci cache, adnotacja `@CachePut` powoduje bardziej liniowe wywołanie metod. Metoda opatrzona adnotacją `@CachePut` jest wywoływana zawsze, a zwracana przez nią wartość zapisywana jest w pamięci podręcznej. Umożliwia to wygodne wypełnienie pamięci podręcznej danymi, zanim przyjdą zapytania.

Na przykład gdy nowy obiekt `Spittle` zostanie zapisany za pośrednictwem metody `save()` repozytorium `SpittleRepository`, istnieje wysokie prawdopodobieństwo, że nastąpi zapytanie o tę wartość. Po wywołaniu metody `save()` sensowne jest wrzucenie obiektu `Spittle` do pamięci podręcznej, aby był już w niej dostępny, kiedy ktoś będzie chciał go odczytać za pomocą metody `findOne()`. Możemy w tym celu metodę `findOne()` opatrzyć adnotacją `@CachePut`:

```
@CachePut("spittleCache")
Spittle save(Spittle spittle);
```

Po wywołaniu metody `save()` adnotacja ta robi wszystko, co jest potrzebne do zapisania obiektu `Spittle`. Zwrócony przez metodę obiekt `Spittle` umieszczony zostanie w pamięci podręcznej `spittleCache`.

Jest tylko jeden problem: klucz cachowania. Jak wspominałem wcześniej, domyślny klucz opiera się na parametrach przekazanych do metody. Ponieważ jedynym parametrem metody `save()` jest obiekt `Spittle`, wykorzystywany jest on jako klucz cachowania. Czyż nie jest dziwne umieszczanie obiektu `Spittle` pod kluczem, którym jest ten sam obiekt?

Z pewnością domyślna wartość klucza cachowania nie jest w tym przypadku pożądana. Chcemy, aby kluczem cachowania był identyfikator nowo zapisanego obiektu `Spittle`, a nie cały obiekt. Musimy więc określić klucz inny niż domyślny. Zobaczmy, jak ustawić niestandardowy klucz cachowania.

USTAWIAMY WARTOŚĆ KLUCZA CACHOWANIA

Zarówno adnotacja `@Cacheable`, jak i `@CachePut` posiadają atrybut `key`, który umożliwia zamianę domyślnego klucza kluczem wyliczonym za pomocą wyrażenia SpEL. Możemy użyć dowolnego wyrażenia SpEL, najprawdopodobniej chcemy jednak skorzystać z wyrażenia wyliczonego w oparciu o klucz obiektu zapisanego w pamięci podręcznej.

W naszym przypadku kluczem ma być identyfikator zapisanego obiektu `Spittle`. Wiadomość `Spittle` przekazana jako parametr do metody `save()` nie została jeszcze zapisana, nie ma więc identyfikatora. Potrzebna nam jest wartość właściwości id obiektu `Spittle` zwróconego przez metodę `save()`.

Na szczęście Spring pozwala na wykorzystanie w wyrażeniach SpEL metadanych cachowania. Tabela 13.3 zawiera listę dostępnych metadanych.

Tabela 13.3. Spring udostępnia kilka wyrażeń SpEL specjalnie na potrzeby definiowania reguł cachowania

Wyrażenie	Opis
`#root.args`	Argumenty przekazane do cachowanej metody w postaci tablicy
`#root.caches`	Lista pamięci podręcznych, z których ma korzystać metoda podana w postaci tablicy
`#root.target`	Obiekt docelowy
`#root.targetClass`	Klasa obiektu docelowego; skrócona wersja wyrażenia `#root.target.class`
`#root.method`	Cachowana metoda
`#root.methodName`	Nazwa cachowanej metody; skrócona wersja wyrażenia `#root.method.name`
`#result`	Wartość zwracana z wywoływanej metody (niedostępna w adnotacji `@Cacheable`)
`#Argument`	Nazwa dowolnego argumentu metody (na przykład `#nazwaArg`) lub indeks argumentu (na przykład `#a0` lub `#p0`)

W przypadku metody `save()` kluczem musi być właściwość `id` zwracanego obiektu `Spittle`. Wyrażenie `#result` zwraca obiekt `Spittle`. W ten sposób możemy odwoływać się do właściwości `id` obiektu, ustawiając wartość atrybutu `key` na `#result.id`:

```
@CachePut(value="spittleCache", key="#result.id")
Spittle save(Spittle spittle);
```

Takie ustawienie adnotacji `@CachePut` sprawia, że przy wywoływaniu metody `save()` nie korzystamy z pamięci podręcznej. Obiekt `Spittle` zwracany przez tę metodę trafi jednak do pamięci podręcznej pod kluczem równym wartości właściwości `id` obiektu.

WARUNKOWE CACHOWANIE

Zastosowanie jednej z adnotacji cachowania Springa na metodzie stanowi informację dla Springa, że ma utworzyć aspekt cachowania typu around dla tej metody. W niektórych sytuacjach chcielibyśmy jednak wyłączyć cachowanie.

Adnotacje `@Cachable` i `@CachePut` udostępniają dwa atrybuty cachowania warunkowego: `unless` oraz `condition`. Oba przyjmują wyrażenie SpEL. Jeśli wartością atrybutu `unless` wyrażenia SpEL jest `true`, to dane zwrócone z cachowanej metody nie są umieszczane w pamięci podręcznej. Podobnie, jeżeli wartością wyrażenia SpEL atrybutu `condition` jest `false`, cachowanie jest dla tej metody wyłączane.

Wydawać by się mogło, że atrybuty `unless` i `condition` osiągają ten sam cel. Jest jednak pomiędzy nimi subtelna różnica. Atrybut `unless` jedynie zapobiega zapisywaniu obiektu w pamięci podręcznej. Po wywołaniu metody pamięć podręczna jest w dalszym ciągu przeszukiwana, a w przypadku znalezienia pasującego wpisu zostanie on zwrócony. Z drugiej strony, jeśli wartością wyrażenia przypisanego do atrybutu `condition` jest `false`, pamięć podręczna jest całkowicie wyłączana na czas trwania wywołania metody. Pamięć podręczna nie jest więc przeszukiwana ani nie są z niej zwracane żadne wartości.

Na przykład załóżmy (choć będzie to trochę naciągane), że nie chcemy zwracać żadnych obiektów typu `Spittle`, których właściwość `message` zawiera tekst `NieCachować`. Aby zapobiec zapisaniu w pamięci podręcznej obiektów spełniających ten warunek, możemy wykorzystać atrybut `unless`:

```
@Cacheable(value="spittleCache"
      unless="#result.message.contains('NieCachować')")
Spittle findOne(long id);
```

Wyrażenie SpEL przekazane do atrybutu `unless` sprawdza wartość właściwości `message` zwróconego obiektu `Spittle` (dostępnego w wyrażeniu w postaci `#result`). Jeśli właściwość zawiera tekst `NieCachować`, to wartością wyrażenia będzie `true`, a obiekt `Spittle` nie będzie umieszczony w pamięci podręcznej. W przeciwnym wypadku wyrażenie będzie miało wartość `false`, warunek zapisany w atrybucie `unless` nie będzie spełniony, a obiekt `Spittle` nie zostanie zapisany w pamięci podręcznej.

Atrybut `unless` zapobiega zapisaniu wartości w pamięci podręcznej. Możemy jednak chcieć całkowicie ją wyłączyć. To oznacza, że w pewnych warunkach nie chcemy ani dodawać wartości do pamięci cache, ani ich z niej odczytywać.

Przykładowo przypuśćmy, że nie chcemy stosować pamięci cache dla obiektów `Spittle` o identyfikatorach mniejszych niż 10. W tym scenariuszu te obiekty `Spittle` traktujemy jako wpisy testowe wykorzystywane w celach debugowania i ich cachowanie nie przedstawia dla nas żadnej wartości. W celu wyłączenia cachowania obiektów `Spittle` o identyfikatorach mniejszych niż 10 użyjemy atrybutu `conditio` adnotacji `@Cacheable`:

```
@Cacheable(value="spittleCache"
      unless="#result.message.contains('NoCache')"
      condition="#id >= 10")
Spittle findOne(long id);
```

Teraz, jeśli metoda `findOne()` zostanie wywołana z parametrem o wartości mniejszej niż 10, pamięć podręczna nie zostanie przeszukana, a zwrócony obiekt `Spittle` nie zostanie umieszczony w pamięci podręcznej. Aplikacja zachowa się dokładnie tak, jakby metoda nie była wcale opatrzona adnotacją `@Cacheable`.

Jak widzieliśmy we wcześniejszych przykładach, wyrażenie powiązane z atrybutem `unless` może się odwoływać do wartości zwracanej z metody za pośrednictwem wyrażenia `#result`. Działanie takie jest możliwe i zarazem użyteczne, ponieważ atrybut `unless` rozpoczyna swoją pracę dopiero po zwróceniu wartości z metody objętej cachowaniem. Z drugiej strony zadaniem atrybutu `condition` jest wyłączenie mechanizmu cachowania dla metody. Dlatego też z decyzją, czy cachowanie ma zostać wyłączone, nie może czekać do momentu zakończenia wykonywania metody. Oznacza to, że wyrażenie musi zostać wyliczone przed wywołaniem metody i nie ma w związku z tym dostępu do wyrażenia `#result`.

Zapisywaliśmy już elementy w pamięci podręcznej, ale czy możemy je stamtąd usunąć? Za chwilę dowiesz się, jak za pomocą adnotacji `@CacheEvict` wydać zapisanym danym rozkaz zniknięcia nam z oczu.

13.2.2. Usuwamy wpisy z pamięci podręcznej

Adnotacja `@CacheEvict` nie dodaje niczego do pamięci podręcznej. Wprost przeciwnie — wywołanie metody opatrzonej adnotacją `@CacheEvict` powoduje usunięcie jednego lub większej liczby wpisów zapisanych w pamięci podręcznej.

W jakich okolicznościach moglibyśmy chcieć usunąć coś z pamięci podręcznej? Za każdym razem, gdy wartość zapisana w pamięci podręcznej przestanie być aktualna, musimy ją stamtąd usunąć, aby przyszłe odwołania do pamięci podręcznej nie zwracały błędnych lub nieistniejących danych. Jedną z takich sytuacji jest usunięcie danych. Czyni to metodę `remove()` repozytorium `SpittleRepository` idealnym kandydatem na użycie adnotacji `@CacheEvict`:

```
@CacheEvict("spittleCache")
void remove(long spittleId);
```

> **UWAGA.** W odróżnieniu od adnotacji `@Cachable` i `@CachePut`, adnotacja `@CacheEvict` może zostać użyta na metodach niezwracających wartości. Adnotacje `@Cacheable` i `@CachePut` mogą zostać zastosowane jedynie na metodach zwracających wartość innego typu niż `void`, gdyż zwracana wartość umieszczana jest w pamięci podręcznej. Ponieważ adnotacja `@CacheEvict` służy tylko do usuwania elementów z pamięci podręcznej, może być użyta na dowolnej metodzie, nawet zwracającej `void`.

Jak pokazałem wyżej, do usunięcia pojedynczego wpisu z pamięci podręcznej `spittle` ↪`Cache` służy metoda `remove()`. Usunięty zostanie element o kluczu równym wartości przekazanego parametru `spittleId`.

Adnotacja `@CacheEvict` udostępnia kilka atrybutów, wymienionych w tabeli 13.4, które umożliwiają zmianę jego domyślnego zachowania.

Tabela 13.4. Adnotacja @CacheEvict określa, które wpisy cachowania należy usunąć

Atrybut	Typ	Opis
`value`	`String[]`	Nazwa wykorzystywanej pamięci podręcznej.
`key`	`String`	Wyrażenie SpEL, które służy do wyliczania niestandardowego klucza cachowania.
`condition`	`String`	Wyrażenie SpEL, które w przypadku zwrócenia wartości `false` skutkuje nieumieszczeniem wyniku wywoływanej metody w pamięci cache.
`allEntries`	`boolean`	Gdy jego wartością jest `true`, to usuwane są wszystkie wpisy we wskazanej pamięci podręcznej.
`beforeInvocation`	`boolean`	Gdy ma wartość `true`, wpisy usuwane są z pamięci podręcznej przed wywołaniem metody. Kiedy wartością jest `false` (ustawienie domyślne), wpisy usuwane są po udanym wywołaniu metody.

Jak widać, adnotacja `@CacheEvict` współdzieli niektóre ze swoich atrybutów z adnotacjami `@Cacheable` oraz `@CachePut`. Udostępnia też kilka własnych atrybutów. W przeciwieństwie do adnotacji `@Cacheable` i `@CachePut`, adnotacja `@CacheEvict` nie posiada atrybutu `unless`.

Adnotacje cachowania Springa pozwalają w elegancki sposób określić reguły cachowania w kodzie aplikacji. Spring udostępnia też jednak przestrzeń XML przeznaczoną do obsługi cachowania. Zakończmy rozważania na temat cachowania, poznając metodę jej konfiguracji z użyciem reguł zapisanych w pliku XML.

13.3. Deklarujemy cachowanie w pliku XML

Zastanawiasz się może, dlaczego chcielibyśmy deklarować cachowanie w pliku XML. Adnotacje cachowania opisane w tym rozdziale są przecież dużo bardziej eleganckie.

Przychodzą mi jednak na myśl dwa powody:

- Nie czujesz się komfortowo z umieszczaniem adnotacji Springa w kodzie źródłowym swojej aplikacji.
- Chcesz użyć mechanizmów cachowania na komponentach, do których nie masz kodu źródłowego.

W obu tych przypadkach lepszym (lub nawet koniecznym) rozwiązaniem jest trzymanie konfiguracji cachowania w oddzieleniu od kodu, który odpowiada za przygotowanie cachowanych danych. Przestrzeń nazw `cache` Springa pozwala na zadeklarowanie reguł cachowania w pliku XML, co stanowi alternatywę dla adnotacji cachowania. Ze względu na fakt, że cachowanie jest czynnością, której działanie opiera się na użyciu aspektów, przestrzeń nazw `cache` występuje zawsze w parze z przestrzenią nazw `aop`. Umożliwia ona zadeklarowanie punktów przecięcia w cachowanych metodach.

Aby umożliwić deklarację cachowania z użyciem konfiguracji XML, musimy utworzyć plik konfiguracji XML włączający w nagłówku przestrzenie nazw `cache` i `aop`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xmlns:cache="http://www.springframework.org/schema/cache"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="http://www.springframework.org/schema/aop
         http://www.springframework.org/schema/aop/spring-aop.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/cache
         http://www.springframework.org/schema/cache/spring-cache.xsd">
   <!—Tutaj znajdzie się konfiguracja cachowania -->
</beans>
```

Przestrzeń nazw cache definiuje elementy konfiguracji przeznaczone do deklaracji cachowania w pliku konfiguracji Spring XML. W tabeli 13.5 znajduje się opis wszystkich elementów udostępnianych przez przestrzeń nazw `cache`.

Tabela 13.5. Przestrzeń nazw Springa cache udostępnia elementy służące do konfiguracji reguł cachowania w pliku XML

Element	Opis
`<cache:annotation-driven>`	Włącza mechanizm cachowania w oparciu o adnotacje. Odpowiednik adnotacji `@EnableCaching` w konfiguracji Javy.
`<cache:advice>`	Definiuje porady cachowania. W połączeniu z elementem `<aop:advisor>` umożliwia zastosowanie porady na punkcie przecięcia.
`<cache:caching>`	Definiuje określony zbiór reguł cachowania w ramach porady cachowania.
`<cache:cacheable>`	Oznacza metodę jako cachowalną. Odpowiednik adnotacji `@Cacheable`.
`<cache:cache-put>`	Oznacza metodę jako wypełniającą pamięć podręczną danymi (ale z niej niepobierającą). Odpowiednik adnotacji `@CachePut`.
`<cache:cache-evict>`	Oznacza metodę jako usuwającą jeden lub więcej wpisów z pamięci podręcznej. Odpowiednik adnotacji `@CacheEvict`.

Zadaniem elementu `<cache:annotation-driven>`, podobnie jak jego odpowiednika w konfiguracji Javy — adnotacji `@EnableCaching`, jest włączenie cachowania wykorzystującego adnotacje. Pisałem już na temat tego stylu cachowania, nie ma więc powodu, by więcej o nim wspominać.

Pozostałe elementy pokazane w tabeli 13 5 służą do konfiguracji cachowania w oparciu o pliki XML. Listing 13.7 przedstawia sposób użycia tych elementów do konfiguracji cachowania wokół metod komponentu `SpittleRepository`, podobny do sposobu wykorzystanego w przypadku konfiguracji z użyciem klas Javy.

Listing 13.7. Deklarujemy reguły cachowania wokół repozytorium SpittleRepository z wykorzystaniem elementów XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cache="http://www.springframework.org/schema/cache"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/aop
          http://www.springframework.org/schema/aop/spring-aop.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
        http://www.springframework.org/schema/cache
        http://www.springframework.org/schema/cache/spring-cache.xsd">
   <aop:config> <-- Wiążemy poradę cachowania do punktu przecięcia
      <aop:advisor advice-ref="cacheAdvice"
         pointcut=
            "execution(* com.habuma.spittr.db.SpittleRepository.*(..))"/>
   </aop:config>
   <cache:advice id="cacheAdvice">
      <cache:caching>
         <cache:cacheable <-- Ustawiamy metodę jako podlegającą cachowaniu
            cache="spittleCache" method="findRecent" />
         <cache:cacheable <-- Ustawiamy metodę jako podlegającą cachowaniu
            cache="spittleCache"      method="findOne" />
         <cache:cacheable <-- Ustawiamy metodę jako podlegającą cachowaniu
            cache="spittleCache" method="findBySpitterId" />
         <cache:cache-put <-- Dodajemy elementy do pamięci podręcznej przy zapisie
            cache="spittleCache" method="save" key="#result.id" />
         <cache:cache-evict <-- Usuwamy z pamięci podręcznej
            cache="spittleCache" method="remove" />
      </cache:caching>
   </cache:advice>
   <bean id="cacheManager" class=
      "org.springframework.cache.concurrent.ConcurrentMapCacheManager" />
</beans>
```

Pierwszymi widocznymi deklaracjami na listingu 13.7 są elementy `<aop:advisor>`, które odwołują się do porady o identyfikatorze `cacheAdvice`. Element ten pasuje do porady w punkcie przecięcia i tworzy w ten sposób kompletny aspekt. W tym przypadku punkt przecięcia aspektu odpalany jest przy wywołaniu dowolnej metody interfejsu `SpittleRepository`. Jeśli taka metoda wywoływana jest na dowolnym komponencie zarejestrowanym w kontekście aplikacji Springa, następuje odpalenie porady aspektu.

Porada zadeklarowana jest za pomocą elementu `<cache:advice>`. Element `<cache:advice>` zawierać może dowolną liczbę elementów `<cache:caching>`, tyle, ile potrzeba do zdefiniowania wszystkich reguł cachowania w naszej aplikacji. Na listingu 13.7 zdefiniowaliśmy tylko jeden element `<cache:caching>`. Zawiera on trzy elementy `<cache:cacheable>` i jeden element `<cache:cache-put>`.

Każdy z elementów `<cache:cachaeble>` deklaruje metodę z punktu przecięcia jako element podlegający cachowaniu. Jest to odpowiednik adnotacji `@Cacheable`. Metody `findRecent()`, `findOne()` oraz `findBySpitterId()` zadeklarowane są jako podlegające cachowaniu, a zwrócone przez nich wartości zapisywane są w pamięci podręcznej `spittleCache`.

Element `<cache:cache-put>` jest odpowiednikiem adnotacji `@CachePut`. Dzięki temu oznaczeniu wartość zwracana przez tę metodę zapisywana jest w pamięci podręcznej, ale metoda nie pobiera nigdy z pamięci wewnętrznej zwracanej przez siebie wartości. W naszym przykładzie metoda `save()` służy do wypełnienia danymi pamięci podręcznej. Tak jak w przypadku cachowania z użyciem adnotacji, tu również musimy nadpisać wartość domyślnego klucza tak, żeby jako klucz wykorzystywana była właściwość `id` obiektu `Spittle`.

I na sam koniec — element `<cache:cache-evict>` jest odpowiednikiem adnotacji `@CacheEvict`. Usuwa dane z pamięci cache, by przy następnej próbie ich pobrania nie zostały już w niej znalezione. W przykładzie usuwamy obiekt `Spittle` z pamięci podręcznej, korzystając z metody `remove()`. Klucz usuwanego elementu jest taki sam jak identyfikator przekazywany do metody.

Warto dodać, że element `<cache:advice>` posiada atrybut `cache-manager`, który określa komponent pełniący funkcję menedżera cachowania. Domyślnie wykorzystywany jest komponent o identyfikatorze `cacheManager`. Na końcu listingu 13.7 deklarujemy komponent o takim właśnie identyfikatorze, nie mamy więc potrzeby jawnego określania menedżera cachowania. Jeśli jednak menedżerem cachowania miałby zostać komponent posiadający inny identyfikator (co jest konieczne w przypadku deklarowania wielu menedżerów cachowania), możemy go określić za pomocą atrybutu `cache-manager`.

Warto też zauważyć, że elementy `<cache:cacheable>`, `<cache:cache-put>` oraz `<cache:cache-evict>` korzystają z tej samej pamięci podręcznej `spittleCache`. Aby usunąć duplikację kodu, możemy nazwę pamięci podręcznej określić w elemencie `<cache:caching>`:

```
<cache:advice id="cacheAdvice">
   <cache:caching cache="spittleCache">
      <cache:cacheable method="findRecent" />
      <cache:cacheable method="findOne" />
      <cache:cacheable method="findBySpitterId" />
      <cache:cache-put method="save" key="#result.id" />
      <cache:cache-evict method="remove" />
   </cache:caching>
</cache:advice>
```

Element `<cache:caching>` współdzieli z elementami `<cache:cacheable>`, `<cache:cache-put>` oraz `<cache:cache-evict>` kilka atrybutów, wliczając w to:

- `cache` — określa pamięć podręczną służącą do zapisywania i pobierania wartości.
- `condition` — wyrażenie SpEL, które w przypadku wartości `false` wyłącza pamięć podręczną dla tej metody.
- `key` — wyrażenie SpEL wykorzystywane do ustalenia klucza cachowania (przyjmuje domyślnie wartość parametrów metody).
- `method` — nazwa cachowanej metody.

Dodatkowo elementy `<cache:cacheable>` i `<cache:cache-put>` posiadają atrybut `unless`. Jako wartość tego opcjonalnego atrybutu możemy ustawić wyrażenie SpEL, które w przypadku zwrócenia wartości `true` zapobiega zapisaniu zwróconej wartości w pamięci podręcznej.

Element `<cache:cache-evict>` udostępnia kilka atrybutów, dostępnych tylko dla tego elementu:

- `all-entries` — jeśli jego wartością jest `true`, usunięte zostaną wszystkie wpisy w pamięci podręcznej. W przeciwnym wypadku usuwane są jedynie wpisy pasujące do podanego klucza.

- `before-invocation` — jeżeli jego wartością jest `true`, wpis (lub wpisy) w pamięci podręcznej jest usuwany przed wywołaniem metody. W przeciwnym wypadku wpisy usuwane są po wywołaniu metody.

Domyślną wartością atrybutów `all-entries` i `before-invocation` jest `false`. Oznacza to, że użycie `<cache:cache-evict>` bez ich określenia spowoduje usunięcie z pamięci podręcznej pojedynczego wpisu po wywołaniu metody. Usuwany wpis identyfikowany jest przez domyślny klucz (ustalony w oparciu o parametr metody) bądź klucz określony za pomocą wyrażenia SpEL przekazanego do atrybutu `key`.

13.4. Podsumowanie

Cachowanie jest doskonałym sposobem na uniknięcie wielokrotnego przetwarzania, wyliczania lub pobierania tej samej odpowiedzi na zadawane jedno i to samo pytanie. Po wywołaniu metody z danym zbiorem parametrów zwracana wartość może zostać zapisana w pamięci podręcznej, a następnie może być pobierana z pamięci wewnętrznej przy kolejnym zapytaniu z takim samym zbiorem parametrów. W wielu sytuacjach pobranie wartości z pamięci podręcznej jest operacją szybszą albo mniej obciążającą serwery niż ponowne wywołanie metody (na przykład w przypadku zapytań do bazy danych). Dzięki temu cachowanie może mieć pozytywny wpływ na wydajność aplikacji.

W tym rozdziale dowiedziałeś się, jak zadeklarować cachowanie w aplikacji Springa. Na początek zobaczyłeś, jak można zadeklarować jeden lub kilka menedżerów cachowania. Następnie użyliśmy mechanizmów cachowania w naszej aplikacji poprzez dodanie adnotacji `@Cacheable`, `@CachePut` oraz `@CacheEvict` w repozytorium `SpittleRepository`.

Patrzyliśmy także na sposób konfiguracji reguł cachowania w oderwaniu od kodu aplikacji za pomocą pliku XML. Elementy `<cache:cacheable>`, `<cache:cache-put>` oraz `<cache:cache-evict>` odpowiadają adnotacjom wykorzystywanym przez nas wcześniej w tym rozdziale.

W międzyczasie mówiłem o tym, że cachowanie jest czynnością działającą w oparciu o aspekty. W praktyce Spring implementuje cachowanie właśnie w postaci aspektu. Stało się to jasne po zadeklarowaniu reguł cachowania z użyciem konfiguracji XML, w której musieliśmy powiązać poradę cachowania z punktem przecięcia.

Spring wykorzystuje również aspekty przy stosowaniu reguł zabezpieczeń na metodach. W następnym rozdziale dowiesz się, jak za pomocą Spring Security zabezpieczyć metody komponentów.

14 Zabezpieczanie metod

W tym rozdziale omówimy:

- Zabezpieczanie wywoływania metod
- Definiowanie reguł zabezpieczeń za pomocą wyrażeń
- Tworzenie ewaluatorów wyrażeń zabezpieczeń

Ostatnią rzeczą, którą robię przed wyjściem z domu lub pójściem spać, jest sprawdzenie, czy drzwi od domu są zamknięte. Zanim to jednak zrobię, najpierw ustawiam alarm. Dlaczego? Mimo że zamknięcie drzwi jest dobrą formą zabezpieczenia, system alarmowy stanowi drugą linię obrony, jeśli włamywaczowi uda się odbezpieczyć zamek.

W rozdziale 9. dowiedziałeś się, jak wykorzystać Spring Security do zabezpieczenia warstwy internetowej aplikacji. Bezpieczeństwo internetowe jest ważne, bo uniemożliwia użytkownikom uzyskanie dostępu do treści, do których nie powinni mieć dostępu. Co się jednak stanie, jeśli istnieje luka w warstwie internetowej aplikacji? Co, jeżeli w jakiś sposób użytkownik zdoła odpytać o zawartość, której nie powinien zobaczyć?

Chociaż nie mamy powodu, aby przypuszczać, że użytkownik zdoła przełamać zabezpieczenia aplikacji, to luka zabezpieczeń w warstwie internetowej może się nam dość łatwo przydarzyć. Na przykład wtedy, gdy użytkownik wykona żądanie do strony, do której ma dostęp, ale w wyniku niedopatrzenia programisty w kontrolerze wykonywane jest zapytanie do metody pobierającej dane, których użytkownik nie powinien zobaczyć. Jest to zwykła pomyłka, lecz włamania powstają równie często w wyniku pomyłek, co w wyniku przemyślanych akcji hakerów.

Dzięki zabezpieczeniu zarówno warstwy sieciowej aplikacji, jak i metod backendu mamy pewność, że bez odpowiedniej autoryzacji nie zostanie wywołana żadna logika.

W tym rozdziale dowiesz się, jak zabezpieczyć metody komponentów przed wywołaniem przez użytkownika nieposiadającego odpowiednich uprawnień.

Rozpoczniemy od spojrzenia na prostą adnotację, która pozwala oznaczyć metody chronione przed niepowołanym dostępem.

14.1. Zabezpieczamy metody za pomocą adnotacji

Najczęstszym sposobem wprowadzania zabezpieczeń na poziomie metod w Spring Security jest wykorzystanie specjalnej adnotacji na zabezpieczanych metodach. Przynosi to kilka korzyści, a jedną z najważniejszych jest to, że zabezpieczone metody są od razu widoczne w edytorze.

Spring Security udostępnia trzy różne rodzaje adnotacji zabezpieczeń:

- metody oznaczone własną adnotacją `@Secured`,
- metody oznaczone adnotacją JSR-250 `@RolesAllowed`,
- metody oznaczone adnotacjami `@PreAuthorize`, `@PostAuthorize`, `@PreFilter` oraz `@PostFilter`, z wykorzystaniem wyrażeń SpEL.

Adnotacje `@Secured` oraz `@RolesAllowed` są najprostsze w zastosowaniu i ograniczają dostęp w oparciu o uprawnienia przyznane użytkownikowi. Jeśli potrzebna jest nam większa elastyczność w definiowaniu reguł zabezpieczeń na metodach, Spring Security udostępnia adnotacje `@PreAuthorize` i `@PostAuthorize`. Adnotacje `@PreFilter` i `@PostFiler` umożliwiają odrzucenie elementów z kolekcji zwróconej z metody lub do niej przekazanej.

Przed końcem rozdziału zobaczysz wszystkie te adnotacje w działaniu. Na początek zajmijmy się adnotacją `@Secured`, najprostszą z dostępnych w Spring Security adnotacji zabezpieczeń stosowanych na poziomie metod.

14.1.1. Zabezpieczamy metody za pomocą adnotacji @Secured

Kluczem do włączenia zabezpieczeń Springa na poziomie metod jest oznaczenie klasy konfiguracji adnotacją `@EnableGlobalMethodSecurity`:

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled=true)
public class MethodSecurityConfig
      extends GlobalMethodSecurityConfiguration {
}
```

Poza dodaniem adnotacji `@EnableGlobalMethodSecurity` warto zauważyć, że klasa konfiguracji rozszerza klasę `GlobalMethodSecurityConfiguration`. Podobnie jak w przypadku klasy `WebSecurityConfigurerAdapter`, którą rozszerzała klasa konfiguracji zabezpieczeń aplikacji internetowej w rozdziale 9., ona również udostępnia możliwość konfiguracji szczegółowych ustawień zabezpieczeń na poziomie metod.

Przykładowo jeśli nie skonfigurowałeś jeszcze uwierzytelniania w konfiguracji zabezpieczeń warstwy internetowej, możesz to zrobić w tym miejscu za pomocą metody `configure()` klasy `GlobalMethodSecurityConfiguration`:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
      throws Exception {
   auth
      .inMemoryAuthentication()
         .withUser("user").password("password").roles("USER");
}
```

W dalszej części rozdziału, w sekcji 14.2.2, dowiemy się, jak nadpisać metodę `create➥ExpressionHandler()` klasy `GlobalMethodSecurityConfiguration`, aby ustawić własne zachowanie obsługi wyrażeń zabezpieczeń.

Powracając do adnotacji `@EnableGlobalMethodSecurity`, warto zauważyć, że wartość jej atrybutu `secureEnabled` jest równa `true`. Takie ustawienie tego atrybutu powoduje, że tworzone jest przecięcie tak, żeby aspekty Spring Security opakowywały metody komponentów oznaczone adnotacją `@Secured`:

```
@Secured("ROLE_SPITTER")
public void addSpittle(Spittle spittle) {
   // ...
}
```

Adnotacja `@Secured` przyjmuje jako argument tablicę obiektów `String`. Wszystkie te wartości `String` reprezentują uprawnienia, z których przynajmniej jedno jest wymagane do wywołania metody. Przekazując wartość `ROLE_SPITTER`, mówimy Spring Security, że metoda `addSpittle()` nie może być wywołana, o ile `ROLE_SPITTER` nie znajduje się wśród uprawnień przydzielonych uwierzytelnionemu użytkownikowi.

Jeżeli do adnotacji `@Secured` przekazano więcej niż jedną wartość, uwierzytelniony użytkownik musi posiadać przynajmniej jedno z tych uprawnień, aby uzyskać dostęp do metody. Poniższy przykład użycia adnotacji `@Secured` wskazuje, że użytkownik musi posiadać przywilej `ROLE_SPITTER` *lub* `ROLE_ADMIN`, aby wywołać metodę.

```
@Secured({"ROLE_SPITTER", "ROLE_ADMIN"})
public void addSpittle(Spittle spittle) {
   // ...
}
```

W przypadku wywołania metody przez nieuwierzytelnionego użytkownika lub przez użytkownika nieposiadającego wymaganych uprawnień aspekt opakowujący metodę zgłosi jeden z wyjątków Spring Security (prawdopodobnie klasę potomną `Authentica➥tionException` lub `AccesDeniedException`). Są to wyjątki niekontrolowane, ale ktoś musi je w końcu przechwycić i obsłużyć. Jeśli zabezpieczona metoda jest wywoływana w ramach żądania sieciowego, wyjątek zostanie automatycznie obsłużony przez filtry Spring Security. W przeciwnym razie do obsługi wyjątku potrzebny jest specjalny kod.

Jedyną wadą adnotacji `@Secured` jest to, że jest ona specyficzna dla Springa. Jeśli preferujesz standardowe adnotacje, rozważ użycie `@RolesAllowed`.

14.1.2. Adnotacja @RolesAllowed ze specyfikacji JSR-250 w Spring Security

Adnotacja `@RolesAllowed` jest odpowiednikiem adnotacji `@Secured` pod właściwie każdym względem. To, że `@RolesAllowed` jest jedną ze standardowych adnotacji Javy zdefiniowanych w specyfikacji JSR-250, stanowi jedyną istotną różnicę — różnicę bardziej polityczną niż techniczną. Ale użycie standardowej adnotacji `@RolesAllowed` może mieć konsekwencje, jeżeli jest ona używana w kontekście innych frameworków lub API, które przetwarzają tę adnotację.

Mimo wszystko, jeśli chcesz używać `@RolesAllowed`, musisz ją włączyć, ustawiając atrybut `jsr250Enabled` adnotacji `@EnableGlobalMethodSecurity` na `true`:

```
@Configuration
@EnableGlobalMethodSecurity(jsr250Enabled=true)
public class MethodSecurityConfig extends
      GlobalMethodSecurityConfiguration {
}
```

Chociaż włączyliśmy tutaj tylko atrybut `jsr250Enabled`, warto dodać, że nie wyklucza to włączenia również atrybutu `securedEnabled`. Te dwa style adnotacji mogą być włączone jednocześnie.

Ustawienie wartości atrybutu `jsr250Enabled` na `true` powoduje utworzenie takiego punktu przecięcia, że wszystkie metody opatrzone adnotacjami `@RolesAllowed` zostaną opakowane aspektami Spring Security. Umożliwia to użycie adnotacji `@RolesAllowed` na metodach w sposób podobny jak przy wykorzystaniu adnotacji `@Secured`. Na przykład poniższy kod przedstawia tę samą metodę `addSpittle()` opatrzoną adnotacją `@Roles`
`↪Allowed` w miejscu `@Secured`:

```
@RolesAllowed("ROLE_SPITTER")
public void addSpittle(Spittle spittle) {
   //...
}
```

Chociaż zastosowanie adnotacji `@RolesAllowed` daje pewną „polityczną" przewagę nad `@Secured` w sensie wykorzystania w kodzie standardów adnotacji dla metod bezpieczeństwa, to obie współdzielą jedno ograniczenie. Pozwalają ograniczyć wywołanie metod jedynie w oparciu o informację, czy użytkownik posiada określone uprawnienie. W procesie decyzji o przyznaniu dostępu do metody nie są brane pod uwagę żadne inne czynniki. W rozdziale 9. widzieliśmy jednak, że wyrażenia SpEL umożliwiają obejście podobnych ograniczeń przy zabezpieczaniu adresów URL. Za chwilę dowiesz się, jak wykorzystać wyrażenia SpEL za pomocą adnotacji do zabezpieczenia metody przed jej wywołaniem i po nim.

14.2. Korzystamy z wyrażeń do zabezpieczania metod

Chociaż adnotacje `@Secured` i `@RolesAllowed` spełniają swoją rolę, jeśli chodzi o niedopuszczanie nieautoryzowanych użytkowników, jest to właściwie wszystko, co potrafią. Czasem ograniczenia bezpieczeństwa są jednak bardziej złożone niż odrzucanie bądź nie użytkowników na podstawie przydzielonych im uprawnień.

Spring Security 3.0 wprowadził garść nowych adnotacji, które wykorzystują język SpEL do tworzenia ciekawszych ograniczeń bezpieczeństwa na poziomie metody. Te nowe adnotacje opisano w tabeli 14.1.

Tabela 14.1. Spring Security oferuje cztery nowe adnotacje, które mogą zostać użyte do zabezpieczenia metod wyrażeniami SpEL

Adnotacje	Opis
`@PreAuthorize`	Ogranicza dostęp do metody przed wywołaniem na podstawie oszacowania wartości wyrażenia.
`@PostAuthorize`	Pozwala na wywołanie metody, ale zgłasza wyjątek bezpieczeństwa, gdy wynikiem wyrażenia jest `false`.
`@PostFilter`	Pozwala na wywołanie metody, ale filtruje wynik zwracany przez metodę w oparciu o wyrażenie.
`@PreFilter`	Pozwala na wywołanie metody, ale filtruje dane na wejściu metody przed jej rozpoczęciem.

Każda z tych adnotacji przyjmuje wyrażenie SpEL jako wartość parametru. Może to być dowolne wyrażenie SpEL, zawierające dowolne wyrażenia Spring Security spośród dostępnych w tabeli 9.5. Jeśli wynikiem wyrażenia jest `true`, to warunki bezpieczeństwa są spełnione, a w przeciwnym razie nie są. Skutek spełnienia lub niespełnienia warunków różni się w zależności od wykorzystanej adnotacji.

Za chwilę zobaczysz przykład każdej z tych adnotacji. Na początek jednak musimy włączyć możliwość ich użycia, ustawiając wartość atrybutu `prePostEnabled` adnotacji `@EnableGlobalMethodSecurity` na `true`:

```
@Configuration
public class MethodSecurityConfig extends
      GlobalMethodSecurityConfiguration {
}
```

Teraz, po włączeniu adnotacji typu *pre* (przed) i *post* (po), możemy zacząć z nich korzystać. Na początek dowiesz się, jak ograniczyć dostęp do metod za pomocą adnotacji `@PreAuthorize` i `@PostAuthorize`.

14.2.1. Wyrażenia reguł dostępu do metod

Do tej pory widzieliśmy, jak za pomocą adnotacji `@Secured` i `@RolesAllowed` ograniczyć dostęp do metod użytkownikom nieposiadającym odpowiednich uprawnień. Słabością tych adnotacji jest to, że decyzję mogą podjąć jedynie w oparciu o uprawnienia przyznane użytkownikom.

Spring Security udostępnia nam dwie kolejne adnotacje, `@PreAuthorize` oraz `@Post`
`↪Authorize`, które pozwalają ograniczyć dostęp do metod w oparciu o wynik wyrażenia. Wyrażenia znacznie zwiększają możliwości definiowania ograniczeń zabezpieczeń. Umożliwiają przydzielenie lub odrzucenie dostępu do metody z wykorzystaniem niemal dowolnego wyrażenia, które możemy sobie wymyślić.

Kluczowa różnica pomiędzy adnotacjami `@PreAuthorize` i `@PostAuthorize` dotyczy momentu ich ewaluacji. W przypadku adnotacji `@PreAuthrize` wyrażenie wyliczane jest przed wywołaniem metody i uniemożliwia wywołanie metody, jeśli wyliczona wartość

nie jest równa `true`. Adnotacja `@PostAuthorize` oczekuje z wyliczeniem wyrażenia do momentu zwrócenia wyniku przez metodę i na tej podstawie wyrzuca wyjątek bądź go nie wyrzuca.

Na początek przyjrzymy się adnotacji typu *pre*, gdyż jest najczęściej wykorzystywaną adnotacją spośród tych opartych na wyrażeniach. Następnie zobaczysz, jak zabezpieczać dostęp do metod *po* ich wywołaniu.

PREAUTORYZACJA METOD

Na pierwszy rzut oka może się wydawać, że adnotacja `@PreAuthorize` jest wzbogaconym o wyrażenia SpEL odpowiednikiem `@Secured` i `@RolesAllowed`. Właściwie użycie `@Pre`
`↪Authorize` do ograniczenia dostępu na podstawie ról przyznanych uwierzytelnionemu użytkownikowi jest możliwe:

```
@PreAuthorize("hasRole('ROLE_SPITTER')")
public void addSpittle(Spittle spittle) {
   // ...
}
```

Takie użycie adnotacji `@PreAuthorize` nie przynosi nam wyraźnej korzyści w stosunku do adnotacji `@Secured` i `@RolesAllowed`. Jeśli użytkownik posiada rolę `ROLE_SPITTER`, to wywołanie metody będzie możliwe. W przeciwnym wypadku wyrzucony zostanie wyjątek zabezpieczeń i metoda nie zostanie wywołana. Adnotacja `@PreAuthorize` oferuje jednak dużo większe możliwości niż zaprezentowane w tym prostym przykładzie.

Argument `@PreAuthorize` to wyrażenie SpEL. Przy podejmowaniu decyzji o dostępie za pomocą wyrażenia SpEL można utworzyć dużo bardziej złożone ograniczenia bezpieczeństwa. Załóżmy na przykład, że standardowy użytkownik serwisu Spittr może pisać spittle o długości nieprzekraczającej 140 znaków, a długość spittle'ów użytkowników, którzy wykupili usługę premium, nie jest ograniczona.

Adnotacje `@Secured` i `@RolesAllowed` nie zdałyby się tu na wiele, ale `@PreAuthorize` owszem:

```
@PreAuthorize(
      "(hasRole('ROLE_SPITTER') and #spittle.text.length() <= 140)"
      +"or hasRole('ROLE_PREMIUM')")
public void addSpittle(Spittle spittle) {
   // ...
}
```

W powyższym przykładzie `#spittle` odnosi się bezpośrednio do parametru o tej samej nazwie. To pozwala Spring Security na sprawdzenie parametrów przekazanych do metody, a następnie użycie ich przy podejmowaniu decyzji o autoryzacji. W powyższym przykładzie kontrolujemy tekst spittle'a, upewniając się, że nie przekracza on 140 znaków dla standardowego użytkownika. Jeżeli użytkownik wykupił usługę premium, długość tekstu nie ma znaczenia.

POSTAUTORYZACJA METOD

Trochę mniej oczywistym sposobem autoryzacji metody jest jej postautoryzacja. Postautoryzacja polega z reguły na podjęciu decyzji bezpieczeństwa na podstawie obiektu zwróconego przez zabezpieczaną metodę. To oznacza oczywiście, że metoda musi zostać wywołana, aby wyprodukować wynik.

Załóżmy na przykład, że chcemy zabezpieczyć metodę `getSpitterById()` w taki sposób, że dostęp będzie możliwy tylko, jeśli zwracany obiekt `Spittle` należy do uwierzytelnionego użytkownika. Nie mamy jednak możliwości sprawdzenia, czy obiekt `Spittle` należy do aktualnego użytkownika, dopóki tego obiektu nie pobierzemy. Musimy więc najpierw wywołać metodę `getSpittleById()`. Jeśli po pobraniu obiektu `Spittle` okaże się, że nie należy on do aktualnego użytkownika, wyrzucany jest wyjątek zabezpieczeń.

Adnotacja Spring Security `@PostAuthorize` działa bardzo podobnie jak adnotacja `@PreAuthorize`, ale moment zastosowania reguł bezpieczeństwa opóźniony jest do chwili zakończenia wywoływania metody. W tym momencie ma już okazję wykorzystać zwracaną wartość przy podejmowaniu decyzji o przyznaniu użytkownikowi dostępu do metody.

Na przykład żeby zabezpieczyć metodę `getSpittleById()` w opisany powyżej sposób, możemy użyć adnotacji `@PostAuthorize()` następująco:

```
@PostAuthorize("returnObject.spitter.username == principal.username")
public Spittle getSpittleById(long id) {
   //...
}
```

Aby ułatwić dostęp do obiektu zwracanego przez zabezpieczaną metodę, Spring Security dostarcza zmienną `returnObject` w SpEL. Wiemy, że w naszym przypadku zwracany jest obiekt `Spittle`, wyrażenie używa więc właściwości `spitter`, a w dalszej kolejności właściwości `username`.

Po drugiej stronie porównania wyrażenie korzysta z wbudowanego obiektu `principal` i wydobywa z niego właściwość `username`. `principal` (z reguły przechowujący nazwę użytkownika) jest kolejną wbudowaną nazwą Spring Security.

Jeżeli obiekt `Spittle` posiada spittera, którego właściwość `username` jest identyczna z właściwością `username` obiektu `principal`, `Spittle` zostanie zwrócony do wywołującego. W przeciwnym razie zostanie zgłoszony wyjątek `AccessDeniedException` i wywołujący nie zobaczy spittle'a.

Trzeba pamiętać, że w przeciwieństwie do metod oznaczonych adnotacjami `@PreAuthorize`, metody oznaczone adnotacjami `@PostAuthorize` zostaną najpierw wykonane, a dopiero później przerwane. W tej sytuacji należy się upewnić, że metoda nie będzie miała niepożądanych skutków ubocznych, jeśli autoryzacja się nie powiedzie.

14.2.2. Filtrowanie danych wejściowych i wyjściowych metody

Adnotacje `@PreAutorize` i `@PostAuthorize` sprawdzają się świetnie, jeśli wyrażenia mają służyć do zabezpieczania metod. Jednak czasem całkowite ograniczenie dostępu do metody to zbyt dużo. Czasami to nie metoda jest zabezpieczana, ale dane zwracane przez metodę.

Załóżmy na przykład, że mamy metodę `getOffensiveSpittles()`, zwracającą listę obiektów `Spittle` uznanych za obraźliwe. Głównym zadaniem tej metody jest pomoc administratorowi w moderowaniu treści aplikacji Spittr. Metoda ta może być też jednak użyta przez zwykłego użytkownika do sprawdzenia, czy któreś ze stworzonych przez niego spittle'ów zostały uznane za obraźliwe. Sygnatura metody może wyglądać tak, jak pokazano poniżej:

```
public List<Spittle> getOffensiveSpittles() { ... }
```

W tej postaci metoda `getOffensiveSpittles()` nie jest zainteresowana żadnym konkretnym użytkownikiem. Pobiera po prostu listę obraźliwych spittle'ów niezależnie od tego, do kogo należą. Jest to rozwiązanie idealne, gdy chcemy wykorzystać tę metodę w panelu administracyjnym, ale nie sprawdzi się do wyświetlenia listy tylko do tych spittle'ów, które należą do aktualnego użytkownika.

Moglibyśmy oczywiście przeciążyć metodę `getOffensiveSpittles()` i utworzyć drugą wersję, która przyjmuje jako parametr identyfikator użytkownika i pobiera jedynie spittle'e należące do tego użytkownika. Jak jednak wspomniałem na początku rozdziału, zawsze istnieje prawdopodobieństwo, że wykorzystana może być mniej restrykcyjna wersja[1].

Potrzebujemy sposobu na filtrowanie kolekcji obiektów `Spittle` zwróconych przez metodę `getOffensiveSpittles()` i ograniczenie jej do listy, którą aktualny użytkownik ma prawo zobaczyć. Do tego właśnie zadania przygotowana została adnotacja Spring Security `@PostFilter`. Wypróbujmy więc jej działanie.

FILTROWANIE DANYCH ZWRACANYCH PRZEZ METODĘ

Adnotacja `@PostFilter`, podobnie jak adnotacje `@PreAuthorize` i `@PostAuthorize`, przyjmuje jako parametr wyrażenie SpEL. Nie wykorzystuje go jednak do ograniczenia dostępu do metody, ale wylicza wyrażenie na każdym elemencie kolekcji zwracanej z metody, usuwając te elementy, dla których wyrażenie zwraca wartość `false`.

Zademonstrujmy działanie adnotacji `@PostFilter` na metodzie `getOffensiveSpittles()`:

```
@PreAuthorize("hasAnyRole({'ROLE_SPITTER', 'ROLE_ADMIN'})")
@PostFilter( "hasRole('ROLE_ADMIN') || "
   + "filterObject.spitter.username == principal.name")
public List<Spittle> getOffensiveSpittles() { ...
}
```

Adnotacja `@PreAuthorize` pozwala na wykonanie metody tylko użytkownikom z uprawnieniem `ROLE_SPITTER` lub `ROLE_ADMIN`. Jeżeli użytkownik spełni ten warunek, metoda zostanie wykonana i zwróci listę spittle'ów. Wtedy adnotacja `@PostFilter` przefiltruje tę listę, zapewniając, że użytkownik zobaczy tylko obiekty `Spittle`, które ma prawo zobaczyć. W szczególności administrator ma możliwość zobaczenia wszystkich obraźliwych spittle'ów, a użytkownicy niebędący administratorami zobaczą jedynie własne wpisy.

`filterObject` w wyrażeniu odnosi się do pojedynczego elementu (o którym wiemy, że jest obiektem `Spittle`) z listy zwracanej przez metodę. Jeśli nazwa użytkownika

[1] Poza tym, jeśli przeciążylibyśmy metodę `getOffensiveSpittles()`, to musiałbym wymyślić kolejny przykład na przedstawienie sposobu filtrowania wyjścia metody z wykorzystaniem języka SpEL.

obiektu Spitter tego spittle'a jest identyczna z nazwą uwierzytelnionego użytkownika (principal.name w wyrażeniu) lub użytkownik posiada rolę ROLE_ADMIN, element znajdzie się w przefiltrowanej liście. W innym przypadku zostanie on pominięty.

FILTROWANIE PARAMETRÓW METODY

Poza możliwością filtrowania wartości zwracanych przez metodę mamy też możliwość filtrowania listy parametrów trafiających do metody. Technika ta jest rzadziej wykorzystywana, ale czasem może się okazać przydatna.

Przypuśćmy, że mamy listę obiektów Spittle, które chcemy hurtowo usunąć. W tym celu możemy utworzyć metodę z następującą sygnaturą:

```
public void deleteSpittles(List<Spittle> spittles) { ... }
```

Wygląda na nietrudne zadanie, prawda? Co jednak, jeśli chcemy zastosować pewne reguły zabezpieczeń, aby spittle'e mogły być usuwane jedynie przez ich właściciela lub administratora? W takim przypadku moglibyśmy dodać logikę do metody deleteSpittles(), żeby przefiltrować wszystkie spittle'e z listy i usunąć tylko te należące do aktualnego użytkownika (lub wszystkich użytkowników, jeżeli użytkownik jest administratorem).

Takie rozwiązanie mogłoby zadziałać, ale oznaczałoby osadzenie logiki związanej z zabezpieczeniami bezpośrednio w logice metody. Logika zabezpieczeń stanowi oddzielny (choć powiązany) byt od logiki metody związanej z usuwaniem obiektów Spittle. Lepiej by było, aby lista zawierała tylko elementy, które mają zostać rzeczywiście usunięte. Uczyniłoby to logikę metody deleteSpittles() prostszą i skoncentrowaną na zadaniu usuwania spittle'ów.

Adnotacja Spring Security @PreFilter wydaje się idealnym rozwiązaniem tego problemu. Podobnie jak adnotacja @PostFilter, @PreFilter wykorzystuje wyrażenia SpEL do filtrowania kolekcji do elementów spełniających warunek zdefiniowany w wyrażeniu. Nie filtruje jednak danych zwracanych z metody, a elementy kolekcji przekazywane są do metody.

Użycie adnotacji @PreFilter jest dość proste. Poniżej znajduje się definicja metody deleteSpittles(), tym razem z dodaną adnotacją @PreFilter:

```
@PreAuthorize("hasAnyRole({'ROLE_SPITTER', 'ROLE_ADMIN'})")
@PreFilter( "hasRole('ROLE_ADMIN') || "
      + "targetObject.spitter.username == principal.name")
public void deleteSpittles(List<Spittle> spittles) { ... }
```

Tak jak poprzednio, adnotacja @PreAuthorize uniemożliwia wywołanie metody przez użytkowników nieposiadających uprawnienia ROLE_SPITTER ani ROLE_ADMIN. Dodatkowo adnotacja @PreFilter zapewnia, że do metody deleteSpittles() przekazane zostaną tylko te obiekty Spittle, które aktualny użytkownik ma prawo usunąć. Wyrażenie zostanie zastosowane do każdego elementu kolekcji i na liście pozostaną wyłącznie te elementy, dla których wartość wyrażenia zwróciła true. Zmienna targetObject jest kolejną wartością dostarczoną przez Spring Security, która reprezentuje element listy aktualnie przetwarzany w wyrażeniu.

Tak oto zobaczyliśmy już wszystkie cztery adnotacje Spring Security bazujące na wyrażeniach. Wyrażenia oferują znacznie większe możliwości definiowania ograniczeń dostępu niż tylko określenie uprawnień, które dany użytkownik musi posiadać.

Należy jednak przy tym pamiętać, aby nie przesadzić z wykorzystaniem wyrażeń. Należy unikać tworzenia skomplikowanego kodu wyrażeń i umieszczania w nich kodu biznesowego niezwiązanego z zabezpieczeniami. Wyrażenia są przy tym zwykłymi wartościami typu `String` przekazanymi do adnotacji. Sprawia to, że są trudne w testowaniu i znajdowaniu potencjalnych błędów.

Gdy odnosisz wrażenie, że wyrażenia zabezpieczeń stają się nieporęczne i niełatwo nad nimi zapanować, warto rozważyć napisanie własnego ewaluatora uprawnień w celu uproszczenia kodu wyrażeń SpEL. Zobaczmy, jak utworzyć i wykorzystać własny ewaluator uprawnień do uproszczenia naszych wyrażeń filtrowania.

DEFINIUJEMY EWALUATOR WYRAŻEŃ

Wyrażenie, którego użyliśmy w adnotacjach `@PreFilter` i `@PostFilter`, z pewnością nie jest takie skomplikowane. Nie jest też jednak zupełnie trywialne i nietrudno wyobrazić sobie sytuację, w której rozbudowujemy to wyrażenie i stosujemy jeszcze inne reguły zabezpieczeń. Wyrażenie szybko może się stać nieporęczne, zawiłe i trudne w testowaniu.

Co by się wydarzyło, gdybyśmy zamienili całe wyrażenie na znacznie prostsze, wyglądające podobnie do pokazanego poniżej:

```
@PreAuthorize("hasAnyRole({'ROLE_SPITTER', 'ROLE_ADMIN'})")
@PreFilter("hasPermission(targetObject, 'delete')")
public void deleteSpittles(List<Spittle> spittles) { ... }
```

Teraz wyrażenie przekazane do adnotacji `@PreFilter` jest dużo bardziej zwięzłe. Zadaje po prostu pytanie: „Czy użytkownik ma uprawnienia do usunięcia docelowego obiektu?”. Jeśli tak, wynikiem wyrażenia będzie `true`, a obiekt `Spittle` pozostanie na liście przekazanej do metody `deleteSpittles()`. W przeciwnym wypadku zostanie z niej usunięty.

Skąd się jednak wzięła metoda `hasPermission()`? Co oznacza? I co ważniejsze, skąd wie, czy użytkownik ma uprawnienia do usunięcia obiektu `Spittle` z obiektu `targetObject`?

Funkcja `hasPermission()` jest rozszerzeniem języka SpEL dostarczanym przez Spring Security i umożliwia nam, programistom, dodanie logiki wykonywanej przy wyliczaniu wyrażenia. Musimy jedynie napisać i zarejestrować własny ewaluator wyrażeń zawierający logikę wyrażenia (listing 14.1).

Listing 14.1. Ewaluator uprawnień zapewnia logikę hasPermission()

```
package spittr.security;
import java.io.Serializable;
import org.springframework.security.access.PermissionEvaluator;
import org.springframework.security.core.Authentication;
import spittr.Spittle;

public class SpittlePermissionEvaluator implements PermissionEvaluator {
   private static final GrantedAuthority ADMIN_AUTHORITY =
      new GrantedAuthorityImpl("ROLE_ADMIN");
   public boolean hasPermission(Authentication authentication,
         Object target, Object permission) {
```

```
      if (target instanceof Spittle) {
         Spittle spittle = (Spittle) target;
         String username = spittle.getSpitter().getUsername();
         if ("delete".equals(permission)) {
            return isAdmin(authentication) ||
               username.equals(authentication.getName()); }
         }
         throw new UnsupportedOperationException(
            "hasPermission not supported for object <" + target
               + "> and permission <" + permission + ">");
      }

   public boolean hasPermission(Authentication authentication,
         Serializable targetId, String targetType, Object permission) {
      throw new UnsupportedOperationException();
   }

   private boolean isAdmin(Authentication authentication) {
      return authentication.getAuthorities().contains(ADMIN_AUTHORITY);
   }
}
```

`SpittlePermissionEvaluator` implementuje interfejs `PermissionEvaluator` Spring Security, który wymaga implementacji dwóch różnych metod `hasPermission()`. Pierwsza z tych metod przyjmuje jako drugi parametr wartość typu `Object`, wykorzystywaną przy wyznaczaniu uprawnień. Druga z nich przydaje się, kiedy znamy tylko identyfikator obiektu docelowego (identyfikator podawany jest jako obiekt `Serializable` w drugim parametrze).

Do naszych celów przyjmujemy, że będziemy zawsze mieli obiekt `Spittle`, tak więc druga metoda zgłasza jedynie wyjątek `UnsupportedOperationException`.

Jeśli zaś chodzi o pierwszą metodę, sprawdza ona, czy przekazany obiekt jest typu `Spittle`, a następnie, czy może być usunięty. Jeśli tak, nazwa użytkownika spittera porównywana jest z nazwą użytkownika uwierzytelnionego w aplikacji lub ewentualnie następuje sprawdzenie, czy aktualny użytkownik posiada uprawnienie `ROLE_ADMIN`.

Kiedy ewaluator uprawnień jest już gotowy, należy zarejestrować go w Spring Security, żeby można było wykonać operację `hasPermission()` w wyrażeniu przekazanym do `@PostFilter`. Aby to zrobić, musimy zamienić obsługę wyrażenia na korzystającą z naszego ewaluatora wyrażeń.

Domyślnie Spring Security używa klasy `DefaultMethodSecurityExpressionHandler`, do której przekazywana jest instancja `DenyAllPermissionEvaluator`. Jak sama nazwa wskazuje, jej metody `hasPermission()` zawsze zwracają wynik `false`, co powoduje odrzucenie wszelkich prób dostępu do metody. Mamy jednak możliwość dostarczenia Spring Security klasy `DefaultMethodSecurityExpressionHandler`, korzystającej z naszej własnej implementacji ewaluatora `SpittlePermissionEvaluator` poprzez nadpisanie metody `create` ➥`ExpressionHandler` klasy `GlobalMethodSecurityConfiguration`:

```
@Override
protected MethodSecurityExpressionHandler createExpressionHandler() {
   DefaultMethodSecurityExpressionHandler expressionHandler =
      new DefaultMethodSecurityExpressionHandler();
   expressionHandler.setPermissionEvaluator(
         new SpittlePermissionEvaluator()
```

```
    );
    return expressionHandler;
}
```

Od tej pory za każdym razem, gdy zabezpieczymy metodę za pomocą wyrażenia, które stosuje metodę `hasPermission()`, wywołany zostanie ewaluator `SpittlePermissionEvaluator` i podejmie decyzję, czy użytkownik ma uprawnienia do wywołania metody.

14.3. Podsumowanie

Zabezpieczenia na poziomie metod są ważnym uzupełnieniem zabezpieczeń Springa na poziomie warstwy internetowej, omówionej w rozdziale 9. W przypadku aplikacji nieinternetowych zabezpieczenia na poziomie metod są przednią linią obrony aplikacji. W aplikacji internetowej zabezpieczenia na poziomie metod wspomagają reguły zabezpieczeń zadeklarowane dla żądań internetowych.

W tym rozdziale poznawaliśmy sześć adnotacji, które można umieścić na metodach w celu zadeklarowania ograniczeń bezpieczeństwa. Na przykład w przypadku bezpieczeństwa opartego na uprawnieniach pomocna może być adnotacja Spring Security `@Secured`, jak również standardowa adnotacja `@RolesAllowed`. Gdy reguły bezpieczeństwa stają się bardziej złożone, możemy wykorzystać adnotacje `@PreAuthorize` i `@PostAuthorize` w połączeniu z językiem SpEL. Wiesz już też, jak użyć filtrowania wejścia i wyjścia metody za pomocą wyrażeń SpEL i adnotacji `@PreFilter` oraz `@PostFilter`.

Na koniec spojrzeliśmy, jak uczynić reguły zabezpieczeń prostszymi w utrzymaniu, testowaniu i debugowaniu poprzez zdefiniowanie własnego ewaluatora wyrażeń, ukrytego za wywołaniem funkcji `hasPermission()` języka SpEL.

W następnym rozdziale przejdziemy od tematu tworzenia logiki biznesowej aplikacji do integracji Springa z innymi aplikacjami. W kilku następnych rozdziałach przyjrzymy się różnym technikom integracji, włączając w to zdalną komunikację, asynchroniczne wysyłanie komunikatów, REST, a nawet wysyłanie wiadomości e-mail. Na początek poznasz działanie Spring remoting, które omówię już w następnym rozdziale.

Część IV

Integracja w Springu

Żadna aplikacja nie jest odrębną wyspą. Obecnie, aby aplikacje korporacyjne mogły realizować swoje cele, muszą współpracować z innymi systemami. W części IV dowiesz się, jak wyprowadzić aplikację poza jej granice i zintegrować z innymi aplikacjami oraz usługami korporacyjnymi.

Z rozdziału 15., „Praca ze zdalnymi usługami", dowiesz się, jak udostępnić obiekty aplikacji jako zdalne usługi. Poznasz również sposoby przezroczystego dostępu do zdalnych usług, tak jakby były one zwyczajnymi obiektami w aplikacji. W tym celu poznasz różne technologie zdalnego dostępu, takie jak RMI, Hessian/Burlap, oraz usługi sieciowe SOAP wraz z JAX-WS.

W odróżnieniu od rozdziału 15., w którym opisane zostały zdalne usługi opierające się na mechanizmie zdalnego wywołania procedury, PRC, rozdział 16. poświęcony jest usługom typu REST, skoncentrowanym wokół zasobów aplikacji i tworzonym przy wykorzystaniu Spring MVC.

Rozdział 17., „Obsługa komunikatów w Springu", prezentuje inne podejście do integracji aplikacji, pokazując użycie Java Message Service (JMS) oraz Advanced Message Queuing Protocol (AMQP) do prowadzenia asynchronicznej komunikacji pomiędzy aplikacjami.

W coraz większym stopniu od aplikacji internetowych wymaga się błyskawicznej reakcji i operowania na danych dostępnych niemal w czasie rzeczywistym. Rozdział 18.,

„Obsługa komunikatów przy użyciu WebSocket i STOMP", pokazuje nowe możliwości Springa związane z prowadzeniem asynchronicznej komunikacji pomiędzy serwerem a klientem WWW.

Kolejna forma komunikacji asynchronicznej niekoniecznie dotyczy wymiany danych pomiędzy dwiema aplikacjami. Rozdział 19., „Wysyłanie poczty elektronicznej w Springu", pokazuje, jak asynchronicznie wysyłać komunikaty do ludzi — w formie poczty elektronicznej.

Temat zarządzania komponentami Springa i ich monitorowania zostanie poruszony w rozdziale 20., „Zarządzanie komponentami Springa za pomocą JMX". W rozdziale tym pokażemy, jak automatycznie udostępniać komponenty skonfigurowane w Springu jako komponenty zarządzane JMX.

Ostatni rozdział, choć zamieszczony na samym końcu książki, stanowi niezbędne uzupełnienie jej spisu treści. Rozdział 21., „Upraszczanie tworzenia aplikacji przy użyciu Spring Boot", przedstawia ekscytujący i zmieniający reguły gry nowy sposób pisania aplikacji z wykorzystaniem Springa. Przekonasz się w nim, że dzięki Spring Boot można uniknąć żmudnego tworzenia powtarzalnych konfiguracji, tak typowych dla pisania aplikacji z użyciem Springa, i skoncentrować się na implementowaniu funkcjonalności biznesowych.

15 Praca ze zdalnymi usługami

W tym rozdziale omówimy:

- Udostępnianie i dostęp do usług RMI
- Usługi Hessian i Burlap
- Pracę z obiektem wywołującym HTTP Springa
- Używanie Springa z usługami sieciowymi

Wyobraź sobie przez chwilę, że jesteś na bezludnej wyspie. To wcale nie brzmi tak źle. W końcu, kto by nie chciał zakosztować odrobiny samotności na plaży i błogiej separacji od codziennych spraw i świata zewnętrznego?

Ale bezludna wyspa to nie tylko chłodne napoje i opalanie się od rana do wieczora. Nawet jeśli lubisz spokój i odosobnienie, nie minie dużo czasu, zanim poczujesz się głodny, znudzony i samotny. Ile można żywić się kokosami i rybami złowionymi harpunem? Wkrótce zaczniesz odczuwać potrzebę jedzenia, świeżych ubrań i innych artykułów. A jeżeli nie znajdziesz szybko drugiego człowieka, może dojść do tego, że będziesz rozmawiać z piłką do siatkówki!

Wiele aplikacji, które tworzysz, przypomina rozbitków. Na pierwszy rzut oka mogą się wydawać samowystarczalne, w rzeczywistości jednak współpracują z innymi systemami, zarówno tymi w Twojej organizacji, jak i tymi z zewnątrz.

Być może system zaopatrzenia powinien komunikować się z systemem łańcucha dostaw sprzedawcy. A może system kadr w Twojej firmie potrzebuje integracji z systemem płac? Niezależnie od okoliczności, Twoja aplikacja będzie musiała komunikować się z innymi systemami, aby uzyskać zdalny dostęp do usług.

Jako programista Javy masz do wyboru szereg technologii, między innymi:

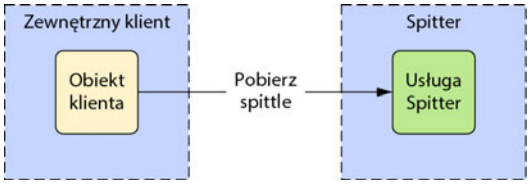
- RMI — zdalne wywoływanie metod (*Remote Method Invocation*),
- Caucho Hessian i Caucho Burlap,
- bazujący na HTTP zdalny dostęp Springa,
- usługi sieciowe z wykorzystaniem JAX-RPC i JAX-WS.

Bez względu na wybraną technologię zdalnego dostępu, Spring zapewnia obszerne wsparcie dla tworzenia i dostępu do zdalnych usług za pomocą szeregu technologii. W tym rozdziale pokażemy, jak Spring zarówno upraszcza, jak i uzupełnia te zdalne usługi. Na początek jednak zobaczmy, jak zdalny dostęp działa w Springu.

15.1. Zdalny dostęp w Springu

Zdalny dostęp (ang. *remoting*) jest konwersacją pomiędzy aplikacją klienta a usługą. Po stronie klienta wymagane są pewne funkcje, które nie są w zasięgu aplikacji. Aplikacja sięga więc do innego systemu, który jest w stanie te funkcje dostarczyć. Zdalna aplikacja udostępnia funkcje poprzez zdalną usługę.

Przypuśćmy, że chcemy udostępnić niektóre funkcje aplikacji Spitter innym aplikacjom jako zdalne usługi. Być może obok istniejącego interfejsu użytkownika w przeglądarce internetowej chcielibyśmy wzbogacić aplikację o interfejsy desktopowy i mobilny (rysunek 15.1). Aby było to możliwe, musimy udostępnić podstawowe funkcje interfejsu `SpitterService` jako zdalną usługę.



Rysunek 15.1. Zewnętrzny klient może komunikować się z aplikacją Spitter poprzez zdalne wywołania udostępnionych usług

Konwersacja pomiędzy aplikacjami a Spitterem zaczyna się od **zdalnego wywołania procedury** (ang. RPC — *remote procedure call*) z aplikacji klientów. Pozornie RPC przypomina wywołanie metody lokalnego obiektu. Obie operacje są synchroniczne, blokują wykonywanie wywołującego kodu do momentu zakończenia procedury.

Różnica polega na odległości, jeżeli porównamy to z ludzką komunikacją. Kiedy dyskutujesz w pracy przy automacie do kawy o wczorajszym meczu, prowadzisz konwersację lokalną — odbywa się ona pomiędzy dwiema osobami w tym samym pomieszczeniu. Podobnie przy lokalnym wywołaniu metody przepływ wykonania przenosi się pomiędzy dwoma blokami kodu w jednej aplikacji.

Z drugiej strony, kiedy podnosisz słuchawkę i dzwonisz do klienta z innego miasta, konwersacja prowadzona jest zdalnie, za pośrednictwem sieci telefonicznej. Podobnie rzecz ma się z RPC. Przepływ wykonania przekazywany jest w tym przypadku pomiędzy dwiema aplikacjami, teoretycznie na innym komputerze w zdalnej lokalizacji, poprzez sieć.

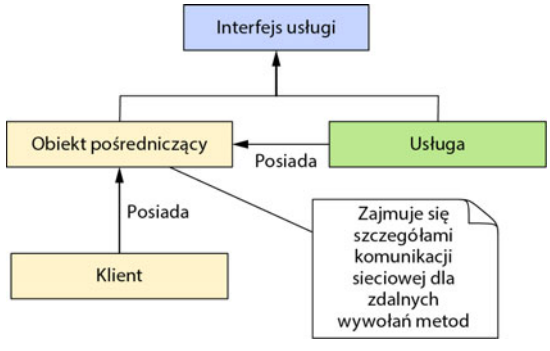
Jak już wspominałem, Spring obsługuje zdalny dostęp dla całej grupy różnych modeli RPC, w tym zdalne wywoływanie metod (RMI), Caucho Hessian, Caucho Burlap i własny obiekt wywołujący HTTP. Tabela 15.1 przedstawia w skrócie każdy z tych modeli i krótko opisuje ich zastosowanie w konkretnych sytuacjach.

Tabela 15.1. Spring obsługuje RPC za pomocą szeregu technologii zdalnego dostępu

Model RPC	Przydatny przy...
Zdalne wywoływanie metod (RMI)	Dostępie do/udostępnianiu usług opartych na Javie, kiedy ograniczenia sieciowe typu firewall nie są brane pod uwagę.
Hessian lub Burlap	Dostępie do/udostępnianiu usług opartych na Javie poprzez HTTP, gdy ograniczenia sieciowe są brane pod uwagę. Hessian jest protokołem binarnym, natomiast Burlap bazuje na XML-u.
Obiekt wywołujący HTTP	Dostępie do/udostępnianiu usług opartych na Springu, gdy ograniczenia sieciowe są brane pod uwagę oraz gdy potrzebna jest serializacja Javy przez XML lub niestandardowa serializacja.
JAX-RPC i JAX-WS	Dostępie do/udostępnianiu niezależnych od platformy, bazujących na SOAP usług sieciowych.

Niezależnie od wyboru modelu zdalnego dostępu, przekonasz się, że wsparcie Springa dla każdego z nich jest realizowane w podobny sposób. Oznacza to, że jeżeli zrozumiesz, jak skonfigurować Springa do pracy z jednym z tych modeli, przejście na inny model nie będzie w przyszłości trudne.

We wszystkich modelach usługi mogą zostać skonfigurowane w aplikacji jako komponenty zarządzane przez Springa. Dzieje się to za sprawą komponentu fabryki obiektów pośredniczących, który umożliwia powiązanie zdalnych usług z właściwościami pozostałych komponentów, tak jakby były lokalnymi obiektami. Na rysunku 15.2 zilustrowano ten mechanizm.



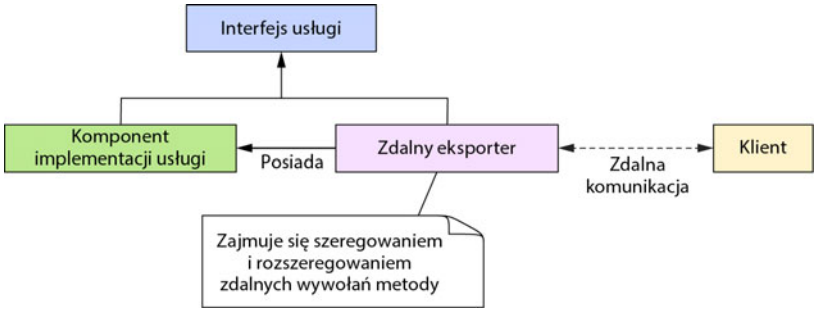
Rysunek 15.2. W Springu dostęp do zdalnych usług odbywa się za pomocą obiektów pośredniczących, tak aby mogły one zostać dowiązane do kodu klienta, jak każdy inny komponent Springa

Klient wywołuje obiekt pośredniczący, tak jakby to on dostarczał funkcjonalność usługi. Obiekt pośredniczący komunikuje się ze zdalną usługą w imieniu klienta. Zajmuje się szczegółami połączenia i wykonania zdalnych wywołań zdalnej usługi.

Co więcej, jeśli wywołanie zdalnej usługi kończy się wyjątkiem `java.rmi.RemoteException`, obiekt pośredniczący przechwytuje ten wyjątek i zgłasza go ponownie, jako niekontrolowany wyjątek `RemoteAccessException`. Zdalne wyjątki z reguły sygnalizują

problemy z siecią i konfiguracją, które nie mogły zostać naprawione. Ponieważ klient z reguły niewiele może zrobić, aby naprawić zdalny wyjątek, zgłoszenie `RemoteAcces`
`↪Exception` sprawia, że obsługa wyjątku przez klienta jest opcjonalna.

Po stronie usługi można udostępnić funkcjonalność każdego komponentu zarządzanego przez Springa jako zdalną usługę, używając jednego z modeli wymienionych w tabeli 15.1. Na rysunku 15.3 pokazano, w jaki sposób zdalne eksportery udostępniają metody komponentów jako zdalne usługi.



Rysunek 15.3. Komponenty zarządzane przez Springa mogą zostać wyeksportowane jako zdalne usługi za pomocą zdalnych eksporterów

Bez względu na to, czy będziesz tworzyć kod, który konsumuje zdalne usługi, implementuje je, czy łączy oba podejścia, praca ze zdalnymi usługami w Springu jest tylko kwestią konfiguracji. Do zdalnego dostępu nie będziesz potrzebować w ogóle kodu Javy. Komponent usługi nie musi w ogóle wiedzieć, że bierze udział w RPC (chociaż komponenty przekazywane do zdalnych wywołań lub z nich zwracane muszą czasem implementować interfejs `java.io.Serializable`).

Odkrywanie obsługi zdalnego dostępu Springa zacznijmy od przyjrzenia się RMI, pierwotnej technologii zdalnego dostępu Javy.

15.2. Praca z RMI

Jeżeli pracujesz w Javie przez jakiś czas, niewątpliwie spotkałeś się ze zdalnym wywoływaniem metod (**RMI** — ang. *Remote Method Invocation*), a nawet go użyłeś. RMI — wprowadzone na platformie Javy w JDK 1.1 — daje programistom Javy ogromne możliwości w zakresie komunikacji pomiędzy programami Javy. Przed RMI jedynymi dostępnymi dla programistów Javy rozwiązaniami zdalnego dostępu były CORBA (którego użycie wiązało się z koniecznością zakupu pośrednika *Object Request Broker*) lub ręczne programowanie gniazd.

Ale rozwijanie i dostęp do usług RMI jest uciążliwe, składa się z wielu kroków, zarówno programowych, jak i manualnych. Spring upraszcza model RMI, dostarczając komponent fabryki obiektów pośredniczących, który umożliwia dowiązanie usług RMI do aplikacji Springa, tak jakby były lokalnymi komponentami JavaBean. Spring posiada również zdalny eksporter, który znacznie upraszcza przekształcenie komponentów zarządzanych Springiem w usługi RMI.

Wracając do aplikacji Spitter, pokażemy, jak dowiązać usługę RMI do kontekstu aplikacji Springa aplikacji klienta. Najpierw zobaczmy jednak, jak użyć eksportera RMI do opublikowania implementacji `SpitterService` jako usługi RMI.

15.2.1. Eksportowanie usługi RMI

Jeśli kiedykolwiek stworzyłeś usługę RMI, to wiesz, że proces ten obejmuje następujące kroki:

1. Przygotowanie klasy implementacji usługi z metodami zgłaszającymi wyjątki `java.rmi.RemoteException`.
2. Utworzenie interfejsu usługi rozszerzającego `java.rmi.Remote`.
3. Wyprodukowanie za pomocą kompilatora RMI (`rmic`) namiastki klienta i klas szkieletowych serwera.
4. Uruchomienie rejestru RMI dla usług.
5. Zarejestrowanie usługi w rejestrze RMI.

Jak na publikację prostej usługi RMI, to bardzo dużo pracy. Ale nawet gorsze od samej złożoności procesu jest to, że, co być może zauważyłeś, w wielu miejscach zmuszani jesteśmy do obsługi wyjątków `RemoteException` i `MalformedURLException`. Wyjątki te najczęściej sygnalizują błąd krytyczny, którego nie można naprawić w bloku `catch`. Mimo to wymaga się od nas nadmiarowego kodu, który przechwyci i obsłuży te wyjątki — nawet jeśli niewiele można z nimi zrobić.

Jak widać, publikacja usługi RMI wiąże się z dużym nakładem pracy i dużą ilością kodu. Czy Spring może temu zaradzić?

KONFIGURACJA USŁUGI RMI W SPRINGU

Na szczęście, Spring pozwala na prostsze publikowanie usług RMI. Zamiast tworzyć klasy specyficzne dla RMI, z metodami zgłaszającymi wyjątki `RemoteException`, wystarczy prosty obiekt POJO, zawierający całą funkcjonalność usługi. Resztą zajmuje się Spring.

Tworzona przez nas usługa RMI udostępnia metody interfejsu `SpitterService`. Na listingu 15.1 pokazano dla przypomnienia kod tego interfejsu.

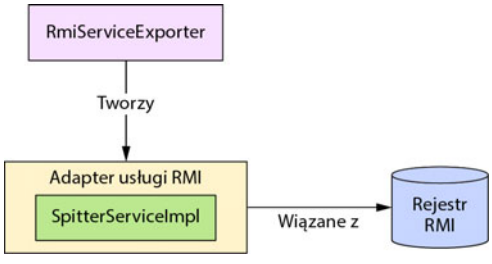
Listing 15.1. Interfejs SpitterService definiuje warstwę usług w aplikacji Spitter

```
package com.habuma.spitter.service;
import java.util.List;
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;

public interface SpitterService {
   List<Spittle> getRecentSpittles(int count);
   void saveSpittle(Spittle spittle);
   void saveSpitter(Spitter spitter);
   Spitter getSpitter(long id);
   void startFollowing(Spitter follower, Spitter followee);
   List<Spittle> getSpittlesForSpitter(Spitter spitter);
   List<Spittle> getSpittlesForSpitter(String username);
   Spitter getSpitter(String username);
   Spittle getSpittleById(long id);
```

```
    void deleteSpittle(long id);
    List<Spitter> getAllSpitters();
}
```

Gdybyśmy udostępniali usługę za pomocą tradycyjnego RMI, wszystkie metody `SpitterService` i `SpitterServiceImpl` musiałyby zgłaszać wyjątek `java.rmi.RemoteException`. Nasza usługa RMI będzie jednak używać eksportera `RmiServiceExporter` Springa, nie ma zatem potrzeby zmiany implementacji.

`RmiServiceExporter` eksportuje komponent zarządzany przez Springa jako usługę RMI. Jak pokazano na rysunku 15.4, `RmiServiceExporter` opakowuje komponent w klasę adaptera. Klasa adaptera jest następnie wiązana z rejestrem RMI i pośredniczy w żądaniach kierowanych do klasy usługi — w tym przypadku `SpitterServiceImpl`.



Rysunek 15.4. RmiServiceExporter przekształca obiekty POJO w usługi RMI, opakowując je w adapter usługi i dowiązując adapter usługi do rejestru RMI

Najprostszym sposobem użycia eksportera `RmiServiceExporter` do udostępnienia `SpitterServiceImpl` jako usługi RMI jest jego konfiguracja w Springu za pomocą następującej metody `@Bean`:

```
@Bean
public RmiServiceExporter rmiExporter(SpitterService spitterService) {
  RmiServiceExporter rmiExporter = new RmiServiceExporter();
  rmiExporter.setService(spitterService);
  rmiExporter.setServiceName("SpitterService");
  rmiExporter.setServiceInterface(SpitterService.class);
  return rmiExporter;
}
```

Komponent `spitterService` jest tu powiązany z właściwością `service`, wskazując, że komponent ma być wyeksportowany jako usługa RMI. Właściwość `serviceName` jest nazwą usługi RMI. Właściwość `serviceInterface` natomiast określa implementowany przez usługę interfejs.

Domyślnie `RmiServiceExporter` próbuje dowiązać usługę do rejestru RMI na porcie 1099 lokalnego komputera. Jeżeli nie uda się znaleźć rejestru RMI na tym porcie, `RmiServiceExporter` uruchamia nowy. W przypadku gdy chcesz dowiązać usługę na innym porcie i na innym adresie hosta, możesz ustawić wartości właściwości `registryPort` i `registryHost`. W przykładzie poniżej `RmiServiceExporter` spróbuje dowiązać usługę do rejestru RMI na porcie 1199 i adresie *rmi.spitter.com*:

```
@Bean
public RmiServiceExporter rmiExporter(SpitterService spitterService) {
  RmiServiceExporter rmiExporter = new RmiServiceExporter();
  rmiExporter.setService(spitterService);
```

```
  rmiExporter.setServiceName("SpitterService");
  rmiExporter.setServiceInterface(SpitterService.class);
  rmiExporter.setRegistryHost("rmi.spitter.com");
  rmiExporter.setRegistryPort(1199);
  return rmiExporter;
}
```

To wszystko, co musimy zrobić, żeby Spring przekształcił komponent w usługę RMI. Teraz, kiedy usługa `Spitter` została udostępniona jako usługa RMI, możemy tworzyć alternatywne interfejsy użytkownika i zachęcać podmioty zewnętrzne do tworzenia nowych klientów Spittera, używających usługi RMI. Twórcy tych klientów łatwo połączą się z usługą RMI `Spitter`, o ile używają Springa. Wrzućmy wyższy bieg i zobaczmy, jak stworzyć klienta usługi RMI `Spitter`.

15.2.2. Dowiązanie usługi RMI

Tradycyjnie klienty RMI używają klasy `Naming` z API RMI do odszukania usługi w rejestrze RMI. Poniższy fragment kodu na przykład mógłby być wykorzystany do uzyskania usługi RMI `Spitter`.

```
try {
   String serviceUrl = "rmi:/spitter/SpitterService";
   SpitterService spitterService = (SpitterService) Naming.lookup(serviceUrl);
   ...
}
catch (RemoteException e) { ... }
catch (NotBoundException e) { ... }
catch (MalformedURLException e) { ... }
```

Chociaż ten fragment kodu z pewnością pozwoli na uzyskanie referencji do usługi RMI `Spitter`, pojawiają się w tym miejscu dwa problemy:

- Tradycyjne wyszukiwanie RMI może zakończyć się jednym z trzech kontrolowanych wyjątków (`RemoteException`, `NotBoundException` i `MalformedURLException`), które muszą zostać przechwycone i zgłoszone ponownie.
- Każdy fragment kodu, który potrzebuje usługi `Spitter`, sam odpowiada za jej uzyskanie. Taki kod integruje aplikację ze środowiskiem, ale prawdopodobnie jest nie do końca spójny z funkcjonalnością klienta.

Wyjątki, które mogą być zgłoszone w trakcie wyszukiwania RMI, najczęściej sygnalizują krytyczny stan aplikacji. `MalformedException` na przykład wskazuje, że przekazany do usługi adres URL jest niepoprawny. Aby spróbować odtworzyć stan sprzed wyjątku, aplikacja musiałaby zostać przynajmniej skonfigurowana i skompilowana ponownie. Rozwiązanie problemu w bloku `try-catch` nie jest możliwe, dlaczego więc zmuszani jesteśmy do przechwytywania tych wyjątków i ich obsługi?

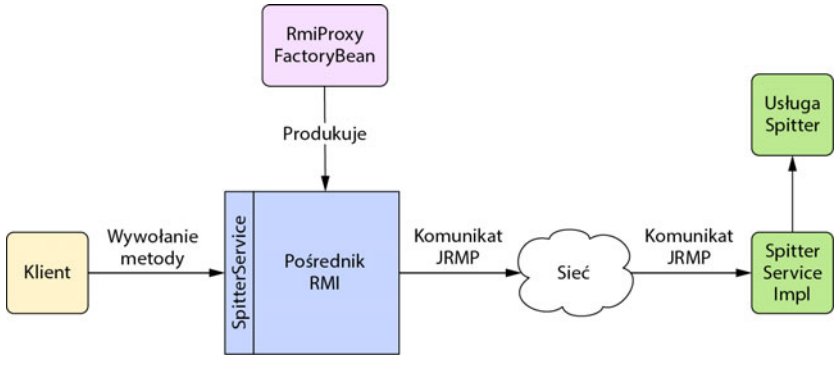
Jeszcze bardziej może niepokoić fakt, że kod ten jest całkowitym zaprzeczeniem zasad wstrzykiwania zależności. Ponieważ kod klienta jest odpowiedzialny za odszukanie usługi `Spitter` i zapewnienie, że usługa jest usługą RMI, nie ma możliwości dostarczenia implementacji `SpitterService` z innego źródła. Idealnym rozwiązaniem byłaby możliwość wstrzyknięcia obiektu `SpitterService` do każdego komponentu, który go potrzebuje,

zamiast jego odszukiwania przez sam komponent. Przy użyciu wstrzykiwania zależności klient `SpitterService` nie musiałby wiedzieć, skąd pochodzi usługa.

Komponent-fabryka Springa `RmiProxyFactoryBean` tworzy obiekt pośrednika dla usługi RMI. Użycie `RmiProxyFactoryBean` w celu odwołania się do `SpitterService` RMI sprowadza się do prostego dodania poniższej metody @Bean w pliku konfiguracyjnym Springa klienta:

```
@Bean
public RmiProxyFactoryBean spitterService() {
  RmiProxyFactoryBean rmiProxy = new RmiProxyFactoryBean();
  rmiProxy.setServiceUrl("rmi://localhost/SpitterService");
  rmiProxy.setServiceInterface(SpitterService.class);
  return rmiProxy;
}
```

Adres URL usługi ustawiany jest za pomocą właściwości `serviceUrl` komponentu `Rmi`
`↪ProxyFactoryBean`. W tym przypadku usługa nazywa się `SpitterService` i jest utrzymywana na lokalnym komputerze. Interfejs dostarczany przez usługę jest z kolei określony we właściwości `serviceInterface`. Interakcję pomiędzy klientem a pośrednikiem RMI pokazano na rysunku 15 5.



Rysunek 15.5. RmiProxyFactoryBean produkuje obiekt pośrednika, który komunikuje się ze zdalnymi usługami RMI w imieniu klienta. Klient komunikuje się z pośrednikiem poprzez interfejs usługi, tak jakby była ona lokalnym obiektem POJO

Teraz, kiedy usługa RMI jest już zadeklarowana jako komponent zarządzany przez Springa, możemy ją dowiązać do innego komponentu jako zależność (tak jak w przypadku komponentów lokalnych). Załóżmy na przykład, że klient potrzebuje usługi `Spitter` do pobrania listy spittle'ów dla danego użytkownika. Możemy również użyć `@Autowired`, aby dowiązać pośrednika usługi do klienta:

```
@Autowired
   SpitterService spitterService;
```

Możemy wtedy wywołać jego metody tak, jakby był lokalnym komponentem.

```
public List<Spittle> getSpittles(String userName) {
   Spitter spitter = spitterService.getSpitter(userName);
   return spitterService.getSpittlesForSpitter(spitter);
}
```

Całe piękno takiego dostępu do usługi RMI polega na tym, że kod klienta nie wie nawet, iż ma do czynienia z usługą RMI. Zostaje mu wstrzyknięty obiekt `SpitterService`, którego pochodzeniem nie musi się interesować. Poza tym — kto miałby poinformować klienta o tym, że używa implementacji bazującej na RMI?

Co więcej, obiekt pośrednika przechwytuje każdy zgłaszany przez usługę wyjątek `RemoteException` i zgłasza go ponownie jako niekontrolowany wyjątek, który można bezpiecznie zignorować. To sprawia, że komponent zdalnej usługi może zostać łatwo wymieniony na inną implementację usługi — być może inną zdalną usługę lub pozorną implementację przy testowaniu kodu klienta.

Chociaż kod klienta nie wie o tym, że przekazana mu `SpitterService` jest zdalną usługą, warto zachować ostrożność przy projektowaniu interfejsu usługi. Zauważ, że klient potrzebuje dwóch wywołań usługi: jednego do odszukania usługi `Spitter` po nazwie użytkownika i drugiego do pobrania listy obiektów `Spittle`. Dwa zdalne wywołania to dodatkowe opóźnienie, co w konsekwencji zmniejsza wydajność klienta. Wiedząc, w jaki sposób usługa będzie użytkowana, warto by wrócić do interfejsu usługi i połączyć te dwa wywołania w jedną metodę. Chwilowo jednak pozostawimy usługę bez zmian.

RMI jest doskonałym sposobem komunikacji pomiędzy zdalnymi usługami, jednak ma pewne ograniczenia. Po pierwsze, napotyka trudności w środowisku, w którym strony komunikacji ukryte są za firewallami. Dzieje się tak, ponieważ RMI wykorzystuje w komunikacji przypadkowe porty — na co przeciętny firewall nie pozwoli. W sieci intranet nie stanowi to problemu. Jeżeli jednak korzystasz z RMI w internecie, napotkasz prawdopodobnie niejeden. I chociaż RMI obsługuje tunelowanie przez HTTP (które firewalle z reguły dopuszczają), konfiguracja tunelowania RMI jest dosyć skomplikowana.

Kolejną ważną rzeczą jest fakt, że RMI oparte jest na Javie. Oznacza to, że zarówno klient, jak i usługa muszą być napisane w tym języku. A ponieważ RMI używa serializacji Javy, typy obiektów przesyłanych w sieci potrzebują dokładnie tej samej wersji środowiska wykonawczego Javy po obu stronach wywołania. To niekoniecznie musi dotyczyć Twojej aplikacji, ale weź to pod uwagę, wybierając RMI do zdalnego dostępu.

Firma Caucho Technology (ta od serwera aplikacji Resin) stworzyła rozwiązanie zdalnego dostępu wolne od ograniczeń RMI. A tak naprawdę dwa rozwiązania: Hessian i Burlap. Zobaczmy, jak ich użyć do pracy ze zdalnymi usługami w Springu.

15.3. Udostępnianie zdalnych usług za pomocą Hessian i Burlap

Hessian i Burlap to dwa rozwiązania firmy Caucho Technology, które umożliwiają lekkie zdalne usługi przez HTTP. Oba mają na celu uproszczenie usług sieciowych, koncentrując się na zachowaniu maksymalnej prostoty zarówno ich API, jak i ich protokołów komunikacyjnych.

Być może zastanawiasz się, dlaczego Caucho oferuje dwa rozwiązania tego samego problemu. Hessian i Burlap są dwiema stronami tego samego medalu, obu jednak używa się w nieco innym celu. Hessian, podobnie jak RMI, do komunikacji między klientem

a usługą używa komunikatów binarnych. Ale w przeciwieństwie do innych binarnych technologii zdalnego dostępu (takich jak RMI) komunikat binarny może być w jego przypadku zrozumiany przez języki inne niż Java (na przykład PHP, Python, C++ i C#).

Burlap jest technologią opartą na XML, co sprawia, że automatycznie może zostać zrozumiany przez każdy język, który potrafi parsować XML. A ponieważ jest to XML, dane mogą zostać łatwiej odczytane przez człowieka niż w przypadku formatu binarnego Hessian. W przeciwieństwie do innych technologii zdalnego dostępu, które działają na bazie XML (jak na przykład SOAP czy XML-RPC), struktura komunikatów jest tak prosta, jak to tylko możliwe, i nie wymaga zewnętrznego języka definicji (jak na przykład WSDL czy IDL).

Zastanawiasz się pewnie, jak wybrać pomiędzy Hessian i Burlap. W przeważającej części są one identyczne. Jedyna różnica polega na tym, że komunikaty Hessian są binarne, a komunikaty Burlap mają format XML. Ponieważ komunikaty Hessian są binarne, rozwiązanie to jest wydajniejsze pod względem przesyłu danych. Jeżeli możliwość odczytania danych przez człowieka jest dla Ciebie ważna (do celów debugowania) lub jeżeli Twoja aplikacja potrzebuje komunikacji z językiem, dla którego Hessian nie ma implementacji, wybierzesz prawdopodobnie komunikaty XML Burlap.

Aby zademonstrować usługi Hessian i Burlap w Springu, powróćmy do naszego przykładu usługi `Spitter` z wykorzystaniem RMI z poprzedniego podrozdziału. Tym razem spróbujemy rozwiązać problem za pomocą Hessian i Burlap jako modeli zdalnego dostępu.

15.3.1. Udostępnianie funkcjonalności komponentu za pomocą Hessian/Burlap

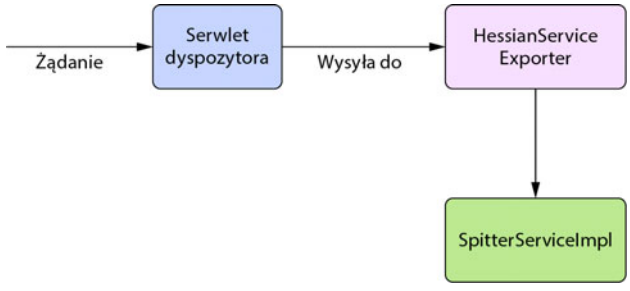
Tak jak wcześniej, załóżmy, że chcemy udostępnić funkcjonalność klasy `SpitterSer`
`↪viceImpl` jako usługę — tym razem usługę Hessian. Nawet bez Springa jest to zadanie trywialne. Należy po prostu stworzyć klasę usługi rozszerzającą klasę `com.caucho.hessian.`
`↪server.HessianServlet` i upewnić się, że wszystkie metody usługi są publiczne (Hessian wszystkie publiczne metody traktuje jako metody usługi).

Ponieważ usługi Hessian są bardzo proste w implementacji, Spring nie wnosi zbyt wiele, jeśli chodzi o dalsze upraszczanie modelu Hessian. W połączeniu ze Springiem Hessian może jednak wykorzystać cały potencjał frameworka Spring, co jest niemożliwe w przypadku czystej usługi Hessian. Może na przykład wykorzystać Spring AOP do umieszczenia w usłudze Hessian rady z usługami systemowymi, jak choćby transakcje deklaratywne.

EKSPORTOWANIE USŁUGI HESSIAN

Eksportowanie usługi Hessian w Springu jest bardzo podobne do implementacji usługi RMI w Springu. Aby udostępnić komponent usługi `Spitter` jako usługę RMI, trzeba skonfigurować komponent `RmiServiceExporter` w pliku konfiguracyjnym Springa. Podobnie, aby udostępnić usługę `Spitter` jako usługę Hessian, musimy skonfigurować inny komponent eksportera. Tym razem `HessianServiceExporter`.

`HessianServiceExporter` spełnia przy usłudze Hessian tę samą rolę, co `RmiService ↪Exporter` przy usłudze RMI: udostępnia publiczne metody obiektu POJO jako metody usługi Hessian. Ale, co pokazano na rysunku 15.6, dokonuje tego w sposób inny niż `RmiServiceExporter`, który eksportował obiekty POJO jako usługi RMI.



Rysunek 15.6. HessianServiceExporter jest kontrolerem Spring MVC, eksportującym obiekt POJO jako usługę Hessian poprzez tłumaczenie otrzymanych żądań Hessian na wywołania obiektu POJO

`HessianServiceExporter` jest kontrolerem Spring MVC (więcej o tym za chwilę), który otrzymuje żądania Hessian i tłumaczy je na wywołania metod eksportowanego obiektu POJO. Eksport komponentu `spitterService` jako usługi Hessian odbywa się za pomocą następującej deklaracji `HessianServiceExporter` w Springu:

```
@Bean
public HessianServiceExporter
      hessianExportedSpitterService(SpitterService service) {
  HessianServiceExporter exporter = new HessianServiceExporter();
  exporter.setService(service);
  exporter.setServiceInterface(SpitterService.class);
  return exporter;
}
```

Tak jak w przypadku `RmiServiceExporter`, właściwość `service` jest wiązana z referencją do komponentu, który implementuje usługę. Tutaj jest to referencja do komponentu `spitterService`. Właściwość `serviceInterface` wskazuje, że usługa implementuje interfejs `SpitterService`.

Inaczej niż w przypadku `RmiServiceExporter`, właściwość `serviceName` nie jest nam potrzebna. Przy RMI właściwość `serviceName` wykorzystywana jest do rejestracji usługi w rejestrze RMI. Hessian nie posiada rejestru, nie ma zatem potrzeby nazywania usługi.

KONFIGURACJA KONTROLERA HESSIAN

Kolejną ważną różnicą pomiędzy `RmiServiceExporter` i `HessianServiceExporter` jest to, że ponieważ Hessian bazuje na HTTP, `HessianServiceExporter` zaimplementowany jest jako kontroler Spring MVC. Oznacza to konieczność wykonania dwóch dodatkowych kroków, aby używać wyeksportowanych usług Hessian:

- Konfiguracji serwletu dyspozytora Springa w pliku *web.xml* i wdrożenia aplikacji jako aplikacji sieciowej.
- Konfiguracji klasy obsługi adresów URL w pliku konfiguracyjnym Springa, aby wysłać adresy URL usługi Hessian do odpowiedniego komponentu usługi Hessian.

Konfigurację serwletu dyspozytora i klasy obsługi adresów URL omówiliśmy w rozdziale 5. Procedura powinna być już mniej więcej znana. Na początek potrzebny jest nam serwlet dyspozytora. Na szczęście, skonfigurowaliśmy go już w pliku *web.xml* aplikacji Spitter. Aby obsłużyć usługi Hessian, serwlet dyspozytora potrzebuje jeszcze elementu `<servlet-mapping>`, który przechwytuje adresy URL w postaci `*.service`:

```
<servlet-mapping>
   <servlet-name>spitter</servlet-name>
   <url-pattern>*.service</url-pattern>
</servlet-mapping>
```

Jeśli serwlet dyspozytora konfigurujemy w Javie, implementując w tym celu interfejs `WebApplicationInitializer`, to wzorzec adresu URL trzeba dodać jako powiązanie do obiektu `ServletRegistration.Dynamic`, uzyskanego podczas dodawania serwletu dyspozytora do kontenera:

```
ServletRegistration.Dynamic dispatcher = container.addServlet(
      "appServlet", new DispatcherServlet(dispatcherServletContext));
  dispatcher.setLoadOnStartup(1);
  dispatcher.addMapping("/");
  dispatcher.addMapping("*.service");
```

Jeżeli natomiast serwlet dyspozytora jest konfigurowany poprzez rozszerzenie klasy `AbstractDispatcherServletInitializer` lub `AbstractAnnotationConfigDispatcherInitializer`, to odwzorowanie należy dodać podczas przesłaniania metody `getServletMappings()`:

```
@Override
  protected String[] getServletMappings() {
    return new String[] { "/", "*.service" };
  }
```

Przy takiej konfiguracji każde żądanie, którego adres URL kończy się ciągiem znaków *.service*, zostanie przekazane do serwletu dyspozytora, który z kolei przekaże je do kontrolera przypisanego danemu adresowi. Tak więc żądania */spitter.service* zostaną ostatecznie obsłużone przez komponent `hessianSpitterService` (który jest tylko pośrednikiem `SpitterServiceImpl`).

Na jakiej podstawie twierdzę, że żądanie trafi do `hessianSpitterService`? Ponieważ skonfigurujemy również odwzorowywanie URL, tak by serwlet dyspozytora wysyłał żądanie do `hessianSpitterService`. Odwzorowanie `SimpleUrlHandlerMapping` przybierze następującą postać:

```
@Bean
public HandlerMapping hessianMapping() {
  SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
  Properties mappings = new Properties();
  mappings.setProperty("/spitter.service",
                       "hessianExportedSpitterService");
  mapping.setMappings(mappings);
  return mapping;
}
```

Alternatywą dla binarnego protokołu Hessian jest protokół Burlap, oparty na XML. Zobaczmy, jak wyeksportować usługę jako usługę Burlap.

EKSPORTOWANIE USŁUGI BURLAP

`BurlapServiceExporter` nie różni się od `HessianServiceExporter` praktycznie niczym, z wyjątkiem tego, że zamiast binarnego używa protokołu opartego na XML. Poniższa deklaracja komponentu pokazuje, jak udostępnić usługę Burlap, korzystając z eksportera `BurlapServiceExporter`:

```
@Bean
public BurlapServiceExporter
      burlapExportedSpitterService(SpitterService service) {
  BurlapServiceExporter exporter = new BurlapServiceExporter();
  exporter.setService(service);
  exporter.setServiceInterface(SpitterService.class);
  return exporter;
}
```

Jak łatwo zauważyć, jedyną różnicą między tym komponentem a jego odpowiednikiem dla Hessian jest identyfikator komponentu i jego klasa. Cała reszta konfiguracji Burlap jest identyczna jak przy Hessian, łącznie z przygotowaniem klasy obsługi adresów URL i serwletu dyspozytora.

Zajmijmy się teraz drugą stroną konwersacji i skonsumujmy usługę, którą opublikowaliśmy za pomocą Hessian (lub Burlap).

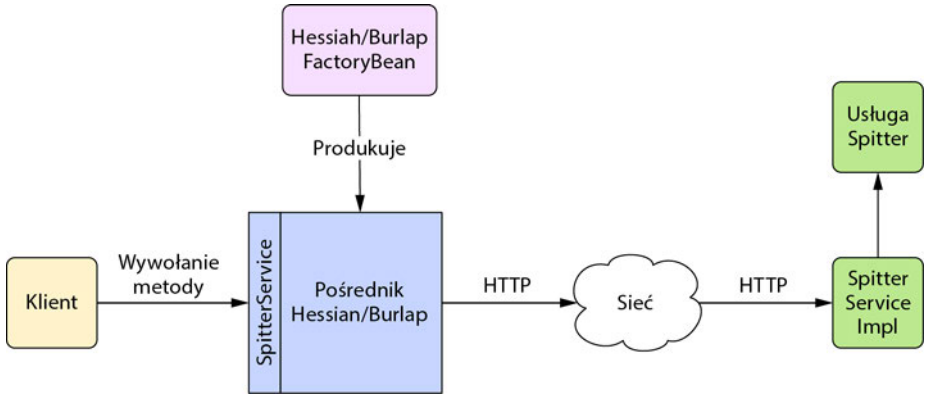
15.3.2. Dostęp do usług Hessian/Burlap

Jak być może pamiętasz z punktu 15.2.2, kod klienta konsumujący usługę `Spitter` za pomocą `RmiProxyFactoryBean` nie miał pojęcia o tym, że ma do czynienia z usługą RMI. Właściwie nie wiedział nawet, że usługa jest sieciowa. Korzystał tylko z interfejsu `SpitterService` — wszystkie szczegóły RMI zostały zawarte w konfiguracji komponentów w pliku konfiguracyjnym Springa. Dobra wiadomość jest taka, że z racji braku wiedzy klienta o implementacji usługi przesiadka z klienta RMI na klienta Hessian jest wyjątkowo łatwa i nie wymaga modyfikacji kodu Javy klienta.

Złą wiadomością jest, że — jeśli uwielbiasz programować w Javie — ten punkt może być dla Ciebie rozczarowaniem. A to dlatego, że jedyną różnicą między dowiązaniem do klienta usługi opartej na RMI a dowiązaniem do niego usługi opartej na Hessian jest użycie `HessianProxyFactoryBean` w miejsce `RmiProxyFactoryBean`. Usługa `Spitter` oparta na Hessian mogłaby być zadeklarowana w kodzie klienta następująco:

```
@Bean
public HessianProxyFactoryBean spitterService() {
  HessianProxyFactoryBean proxy = new HessianProxyFactoryBean();
  proxy.setServiceUrl("http://localhost:8080/Spitter/spitter.service");
  proxy.setServiceInterface(SpitterService.class);
  return proxy;
}
```

Podobnie jak w przypadku usługi bazującej na RMI, właściwość `serviceInterface` określa implementowany przez usługę interfejs. I tak jak przy `RmiProxyFactoryBean`, `serviceUrl` wskazuje adres URL usługi. Ponieważ Hessian bazuje na HTTP, został tu podany adres HTTP (częściowo determinowany przez odwzorowanie URL, które zdefiniowaliśmy wcześniej). Rysunek 15.7 pokazuje interakcję pomiędzy klientem a pośrednikiem produkowanym przez `HessianProxyFactoryBean`.



Rysunek 15.7. HessianProxyFactoryBean i BurlapProxyFactoryBean produkują obiekty pośredników, które komunikują się ze zdalną usługą przez HTTP (Hessian za pomocą kodu binarnego, Burlap za pomocą XML)

Jak się okazuje, dowiązanie usługi Burlap do klienta nie jest bardziej interesujące. Jedyną różnicą jest użycie `BurlapProxyFactoryBean` zamiast `HessianProxyFactoryBean`:

```
@Bean
public BurlapProxyFactoryBean spitterService() {
  BurlapProxyFactoryBean proxy = new BurlapProxyFactoryBean();
  proxy.setServiceUrl("http://localhost:8080/Spitter/spitter.service");
  proxy.setServiceInterface(SpitterService.class);
  return proxy;
}
```

Chociaż określiłem różnice w konfiguracji RMI, Hessian i Burlap jako mało interesujące, nuda jest w tym przypadku zaletą. Podobieństwo tych konfiguracji powoduje, że bez najmniejszego wysiłku można zmieniać różne technologie zdalnego dostępu, bez konieczności przyswajania sobie zupełnie nowego modelu. Kiedy mamy już skonfigurowaną referencję do usługi RMI, zamiana na usługę Hessian lub Burlap wymaga tylko minimalnego wysiłku.

Ponieważ zarówno Hessian, jak i Burlap opierają się na protokole HTTP, nie są narażone na blokadę ze strony firewalla, jak RMI. RMI wygrywa natomiast z Hessian i Burlap, jeśli chodzi o serializację obiektów, które są wysyłane jako komunikaty RPC. Podczas gdy Hessian i Burlap używają własnych mechanizmów serializacji, RMI używa mechanizmu serializacji Javy. Jeżeli model danych jest bardzo złożony, model serializacji Hessian/ Burlap może okazać się niewystarczający.

Istnieje kompromisowe rozwiązanie. Przyjrzyjmy się obiektowi wywołującemu HTTP Springa, który oferuje RPC przez HTTP (podobnie jak Hessian/Burlap), używając jednocześnie serializacji obiektów Javy (podobnie jak RMI).

15.4. Obiekt HttpInvoker

Zespół Springa dostrzegł próżnię pomiędzy usługami RMI a usługami bazującymi na HTTP, jak Hessian i Burlap. Z jednej strony, RMI używa standardowej serializacji obiektów Javy, ale nie najlepiej radzi sobie z firewallami. Z drugiej strony, Hessian

i Burlap nie mają problemu z firewallami, ale używają własnych mechanizmów serializacji obiektów.

Tak narodził się obiekt wywołujący HTTP (ang. *HTTP invoker*) Springa. Obiekt wywołujący HTTP jest nowym modelem zdalnego dostępu, stworzonym jako część frameworka Spring w celu realizacji zdalnego dostępu przez HTTP (aby zadowolić firewalle), przy jednoczesnym użyciu serializacji Javy (aby zadowolić programistów). Praca z usługami bazującymi na obiekcie wywołującym HTTP przypomina pracę z usługami opartymi na Hessian/Burlap.

Spójrzmy zatem na usługę `Spitter` z jeszcze jednej strony — tym razem zostanie ona zaimplementowana jako usługa obiektu wywołującego HTTP.

15.4.1. Udostępnianie komponentów jako usług HTTP

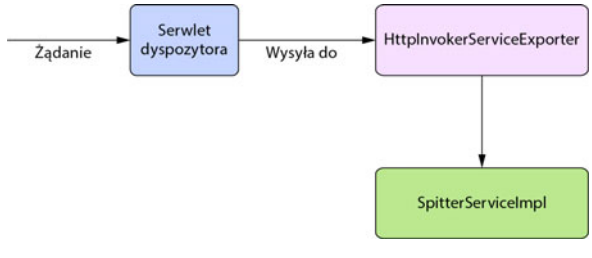
Aby wyeksportować komponent jako usługę RMI, użyliśmy eksportera `RmiServiceExporter`. Żeby wyeksportować go jako usługę Hessian lub Burlap, użyliśmy zaś odpowiednio `HessianServiceExporter` lub `BurlapServiceExporter`. Nie powinno zatem dziwić, że do wyeksportowania usługi obiektu `HttpInvoker` wykorzystamy eksporter o nazwie `HttpInvokerServiceExporter`.

Aby wyeksportować usługę `Spitter` jako usługę bazującą na obiekcie wywołującym HTTP, musimy skonfigurować komponent `HttpInoverServiceExporter` następująco:

```
@Bean
public HttpInvokerServiceExporter
      httpExportedSpitterService(SpitterService service) {
  HttpInvokerServiceExporter exporter =
          new HttpInvokerServiceExporter();
  exporter.setService(service);
  exporter.setServiceInterface(SpitterService.class);
  return exporter;
}
```

Gdzieś to już widziałeś? Rzeczywiście, różnica pomiędzy tą deklaracją komponentu a deklaracjami z punktu 15.3.2 jest ledwo zauważalna. Właściwie jedyną nowością jest tu nazwa klasy: `HttpInvokerServiceExporter`. Poza tym nowy eksporter nie różni się praktycznie niczym od pozostałych.

`HttpInvokerServiceExporter` działa na tej samej zasadzie co `HessianServiceExporter` i `BurlapServiceExporter`, co pokazano na rysunku 15.8. Jest to kontroler Spring MVC, który odbiera żądania od klienta za pomocą serwletu dyspozytora, a następnie tłumaczy je na wywołania metod obiektu POJO, będącego implementacją usługi.



Rysunek 15.8. HttpInvokerServiceExporter działa podobnie jak jego odpowiedniki dla Hessian i Burlap. Odbiera żądania z serwletu dyspozytora Spring MVC i tłumaczy je na wywołania metod komponentu zarządzanego przez Springa

Ponieważ `HttpInvokerServiceExporter` jest kontrolerem Spring MVC, musimy ustawić klasę obsługi adresów URL, która odwzoruje adres URL protokołu HTTP na usługę, podobnie jak w przypadku eksporterów Hessian i Burlap:

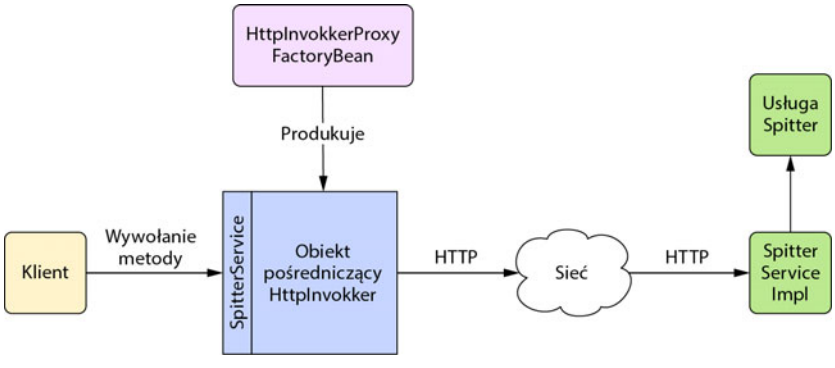
```
@Bean
public HandlerMapping httpInvokerMapping() {
  SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
  Properties mappings = new Properties();
  mappings.setProperty("/spitter.service",
                       "httpExportedSpitterService");
  mapping.setMappings(mappings);
  return mapping;
}
```

Ponadto — tak jak wcześniej — musimy dopilnować, aby określić odpowiednie odwzorowanie, dzięki któremu serwlet dyspozytora będzie prawidłowo obsługiwał żądania z rozszerzeniem **.service*. Szczegółowy opis tego, jak to zrobić, można znaleźć w punkcie 15.3.1.

Wiemy już, jak konsumować zdalne usługi RMI, Hessian i Burlap. Czas teraz stworzyć nową wersję klienta, używającą usługi, którą właśnie udostępniliśmy za pomocą obiektu wywołującego HTTP.

15.4.2. Dostęp do usług przez HTTP

Ryzykując powtarzanie się niczym zacięta płyta, muszę napisać, że konsumpcja usługi bazującej na obiekcie wywołującym HTTP niewiele różni się od tej, którą widzieliśmy przy pozostałych pośrednikach zdalnych usług. Jest praktycznie identyczna. Jak widać na rysunku 15.9, `HttpInvokerProxyFactoryBean` zajmuje to samo miejsce, co pozostałe komponenty fabryk obiektów pośredniczących, o których mówiliśmy w tym rozdziale.



Rysunek 15.9. HttpInvokerProxyFactoryBean jest komponentem fabryki obiektów pośredniczących, który produkuje pośrednika, komunikującego się z usługą za pomocą specyficznego dla Springa protokołu opartego na HTTP

Aby dowiązać usługę bazującą na obiekcie wywołującym HTTP do kontekstu aplikacji Springa klienta, musimy skonfigurować komponent, który pośredniczy w dostępie do niej, za pomocą `HttpInvokerProxyFactoryBean`:

```
@Bean
public HttpInvokerProxyFactoryBean spitterService() {
  HttpInvokerProxyFactoryBean proxy = new HttpInvokerProxyFactoryBean();
  proxy.setServiceUrl("http://localhost:8080/Spitter/spitter.service");
  proxy.setServiceInterface(SpitterService.class);
  return proxy;
}
```

Porównując tę definicję komponentu do definicji z punktów 15.2.2 i 15.3.2, nie zauważysz wielu zmian. Właściwość `serviceInterface` jest w dalszym ciągu używana do wskazania interfejsu implementowanego przez usługę `Spitter`. A właściwość `serviceUrl` nadal określa adres URL zdalnej usługi. Ponieważ obiekt wywołujący HTTP bazuje — podobnie jak Hessian i Burlap — na protokole HTTP, `serviceUrl` może zawierać ten sam adres, jak w przypadku wersji komponentu dla Hessian i Burlap.

Czy ta symetria nie jest piękna?

Obiekt wywołujący HTTP jest kompromisowym rozwiązaniem zdalnego dostępu, łączącym w sobie prostotę komunikacji przez HTTP z wbudowaną serializacją obiektów Javy. To sprawia, że usługi obiektu wywołującego HTTP mogą być atrakcyjną alternatywą zarówno dla RMI, jak i Hessian/Burlap.

Obiekt `HttpInvoker` posiada jedno istotne ograniczenie, o którym warto pamiętać: to rozwiązanie zdalnego dostępu jest oferowane jedynie przez framework Spring. Oznacza to, że zarówno klient, jak i usługa muszą być aplikacjami zbudowanymi w Springu. Wiąże się to też z tym, że — przynajmniej na razie — zarówno klient, jak i usługa muszą być napisane w Javie. A ponieważ rozwiązanie wykorzystuje serializację Javy, obie strony muszą posiadać tę samą wersję klas (podobnie jak w przypadku RMI).

RMI, Hessian, Burlap i obiekt wywołujący HTTP dają bardzo duże możliwości, jeśli chodzi o zdalny dostęp. Ale pod względem wszechobecnego zdalnego dostępu żadne z tych rozwiązań nie może równać się z usługami sieciowymi. W dalszej części rozdziału omówimy wsparcie Springa dla usług sieciowych bazujących na SOAP.

15.5. Publikacja i konsumpcja usług sieciowych

Jednym z bardziej modnych trzyliterowych akronimów w ostatnim czasie jest SOA (*architektura zorientowana na usługi* od ang. *service-oriented architecture*). SOA ma różne znaczenia dla różnych ludzi. Głównym założeniem SOA jest jednak taki sposób projektowania aplikacji, aby polegały one na pewnym trzonie wspólnych usług, dzięki czemu nie ma potrzeby implementowania tych samych funkcji każdej z osobna.

Przykładowo, instytucja finansowa może mieć szereg aplikacji, które wymagają dostępu do informacji o koncie pożyczkobiorcy. Zamiast budować logikę dostępu do konta w każdej aplikacji (duplikując w wielu miejscach kod), aplikacje powinny uzyskiwać ten dostęp za pomocą wspólnych usług.

Wspólna historia Javy i usług sieciowych jest długa, mamy więc do dyspozycji szereg opcji w zakresie pracy z usługami sieciowymi w Javie. Wiele z tych opcji integruje się w jakiś sposób ze Springiem. Na omówienie wszystkich frameworków i zestawów narzędziowych usług sieciowych, które powstały w Springu, nie wystarczyłoby w tej książce miejsca. Sam Spring posiada sprawny mechanizm publikacji i konsumpcji

usług sieciowych SOAP, który wykorzystuje zestaw interfejsów Java API for XML Web Services, bardziej znany jako JAX-WS.

W tym podrozdziale powrócimy jeszcze raz do przykładu usługi `Spitter`. Tym razem udostępnimy i skonsumujemy usługę `Spitter` jako usługę sieciową, wykorzystując wsparcie Springa dla JAX-WS. Na początek sprawdźmy, co jest konieczne do stworzenia usługi sieciowej JAX-WS w Springu.

15.5.1. Tworzenie punktów końcowych JAX-WS w Springu

Wcześniej w tym rozdziale stworzyliśmy zdalne usługi, korzystając z eksporterów usług Springa. Eksportery te w magiczny sposób zmieniają skonfigurowane przez Springa obiekty POJO w zdalne usługi. Dowiedzieliśmy się, jak tworzyć usługi RMI za pomocą `RmiServiceExporter`, usługi Hessian za pomocą `HessianServiceExporter`, usługi Burlap za pomocą `BurlapServiceExporter` i usługi obiektu wywołującego HTTP za pomocą `HttpInvokerServiceExporter`. Prawdopodobnie oczekujesz, że pokażę teraz, jak tworzyć usługi sieciowe za pomocą eksportera usług JAX-WS.

Spring zapewnia eksporter usług JAX-WS, `SimpleJaxWsServiceExporter`, i wkrótce do niego dojdziemy. Zanim to jednak nastąpi, musisz wiedzieć, że nie w każdej sytuacji będzie to najlepszy wybór. `SimpleJaxWsServiceExporter` wymaga bowiem, żeby środowisko uruchomieniowe dawało możliwość publikacji punktów końcowych dla określonego adresu. Chociaż środowisko uruchomieniowe JAX-WS dostarczane z JDK 1.6 firmy Sun spełnia ten warunek, w przypadku innych implementacji JAX-WS (w tym również implementacji referencyjnej) może być już inaczej.

Jeżeli wdrażasz w środowisku uruchomieniowym JAX-WS, które nie obsługuje publikowania dla określonego adresu, musisz stworzyć swoje punkty końcowe JAX-WS w bardziej tradycyjny sposób. To oznacza, że cykl życia punktów końcowych będzie zarządzany przez środowisko uruchomieniowe JAX-WS, a nie przez Springa. Nie znaczy to jednak, że nie mogą one zostać powiązane z komponentami z kontekstu aplikacji Springa.

AUTOMATYCZNE WIĄZANIE PUNKTÓW KOŃCOWYCH JAX-WS W SPRINGU

Model programowania JAX-WS obejmuje między innymi użycie adnotacji do zadeklarowania klasy i jej metod jako operacji usługi sieciowej. Klasa z adnotacją `@WebService` jest punktem końcowym usługi sieciowej, a jej metody, oznaczone adnotacją `@WebMethod`, są operacjami.

Punkt końcowy JAX-WS, podobnie jak każdy obiekt w większej aplikacji, potrzebował będzie do prawidłowego działania innych obiektów. Oznacza to, że punkty końcowe JAX-WS mogą skorzystać z wstrzykiwania zależności. Jeśli jednak cykl życia punku końcowego zarządzany jest przez środowisko uruchomieniowe JAX-WS, a nie przez Springa, dowiązanie komponentów zarządzanych przez Springa do zarządzanej przez JAX-WS instancji punktu końcowego nie będzie możliwe.

Tajemnica wiązania punktów końcowych JAX-WS polega na rozszerzeniu klasy `SpringBeanAutowiringSupport`. Dzięki temu możemy oznaczyć właściwości punktu końcowego adnotacjami `@Autowired`, powodując spełnienie jego zależności. Listing 15.2 ilustruje to na przykładzie klasy `SpitterServiceEndpoint`.

Listing 15.2. SpringBeanAutowiringSupport w punktach końcowych JAX-WS

```
package com.habuma.spitter.remoting.jaxws;
import java.util.List;
import javax.jws.WebMethod;
import javax.jws.WebService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.context.support.SpringBeanAutowiringSupport;
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;

@WebService(serviceName="SpitterService")
public class SpitterServiceEndpoint extends SpringBeanAutowiringSupport {

   @Autowired
   SpitterService spitterService;

   @WebMethod
   public void addSpittle(Spittle spittle) {
      spitterService.saveSpittle(spittle);
   }

   @WebMethod
   public void deleteSpittle(long spittleId) {
      spitterService.deleteSpittle(spittleId);
   }

   @WebMethod
   public List<Spittle> getRecentSpittles(int spittleCount) {
      return spitterService.getRecentSpittles(spittleCount);
   }

   @WebMethod
   public List<Spittle> getSpittlesForSpitter(Spitter spitter) {
      return spitterService.getSpittlesForSpitter(spitter);
   }
}
```

Włączenie automatycznego wiązania

Automatyczne wiązanie SpitterService

Delegowanie do usługi SpitterService

Delegowanie do usługi SpitterService

Właściwość `spitterService` została oznaczona adnotacją `@Autowired`, co wskazuje, że powinien do niej zostać automatycznie wstrzyknięty komponent z kontekstu aplikacji Springa. Potem punkt końcowy może już delegować konkretne zadania do wstrzykniętej usługi `SpitterService`.

EKSPORTOWANIE AUTONOMICZNYCH PUNKTÓW KOŃCOWYCH JAX-WS

Jak już wspomnieliśmy, klasa `SpringBeanAutowringSupport` przydaje się, kiedy cykl życia obiektu, którego właściwości są wstrzykiwane, nie jest zarządzany przez Springa. Ale w sprzyjających okolicznościach można wyeksportować komponent zarządzany przez Springa jako punkt końcowy JAX-WS.

Eksporter Springa `SimpleJaxWsServiceExporter` działa w podobny sposób, co pozostałe poznane przez nas wcześniej eksportery usług. Publikuje on komponenty zarządzane przez Springa jako punkty końcowe usługi w środowisku uruchomieniowym JAX-WS. W przeciwieństwie do pozostałych eksporterów, `SimpleJaxWsServiceExporter`

nie potrzebuje referencji do eksportowanego komponentu. Zamiast tego publikuje wszystkie komponenty oznaczone adnotacjami JAX-WS jako usługi JAX-WS.

`SimpleJaxWsServiceExporter` może zostać skonfigurowany za pomocą następującej metody `@Bean`:

```
@Bean
public SimpleJaxWsServiceExporter jaxWsExporter() {
  return new SimpleJaxWsServiceExporter();
}
```

Jak widać, `SimpleJaxWsServiceExporter` nie potrzebuje do działania nic więcej. Po uruchomieniu rozpocznie przeszukiwanie kontekstu aplikacji Springa pod kątem komponentów z adnotacją `@WebService`. Kiedy taki odnajdzie, opublikuje go jako punkt końcowy JAX-WS o bazowym adresie *http://localhost:8080/*.

Jednym z komponentów, które mogą zostać w ten sposób odnalezione, jest pokazany na listingu 15.3 `SpitterServiceEndpoint`.

Zauważ, że nowa implementacja `SpitterServiceEndpoint` nie jest już rozszerzeniem klasy `SpringBeanAutowiringSupport`. Jako pełnowartościowy komponent Springa, nadaje się do automatycznego wiązania bez konieczności rozszerzania specjalnej klasy pomocniczej.

Listing 15.3. SimpleJaxWsServiceExporter przekształca komponenty w punkty końcowe JAX-WS

```
package com.habuma.spitter.remoting.jaxws;
import java.util.List;
import javax.jws.WebMethod;
import javax.jws.WebService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;

@Component
@WebService(serviceName="SpitterService")
public class SpitterServiceEndpoint {
   @Autowired
   SpitterService spitterService;

   @WebMethod
   public void addSpittle(Spittle spittle) {
      spitterService.saveSpittle(spittle);
   }

   @WebMethod
   public void deleteSpittle(long spittleId) {
      spitterService.deleteSpittle(spittleId);
   }

   @WebMethod
   public List<Spittle> getRecentSpittles(int spittleCount) {
      return spitterService.getRecentSpittles(spittleCount);
   }
```

Automatyczne wiązanie SpitterService

Delegowanie do usługi SpitterService

```
    @WebMethod
    public List<Spittle> getSpittlesForSpitter(Spitter spitter) {
        return spitterService.getSpittlesForSpitter(spitter);
    }
}
```

Delegowanie do usługi SpitterService

Ponieważ bazowym adresem `SimpleJaxWsServiceEndopoint` domyślnie jest *http://localhost:8080/* i jako że klasa `SpitterServiceEndpoint` posiada adnotację `@WebService (serviceName= ↪"SpitterService")`, kombinacja tych dwóch komponentów da w rezultacie usługę sieciową pod adresem: *http://localhost:8080/SpitterService*. Masz jednak pełną kontrolę nad adresem URL usługi, w każdej chwili możesz więc zmienić adres bazowy. Poniższa konfiguracja `SimpleJaxWsServiceEndpoint` na przykład publikuje ten sam punkt końcowy usługi pod adresem *http://localhost:8888/services/SpitterService*.

```
@Bean
public SimpleJaxWsServiceExporter jaxWsExporter() {
  SimpleJaxWsServiceExporter exporter =
          new SimpleJaxWsServiceExporter();
  exporter.setBaseAddress("http://localhost:8888/services/");
}
```

Przy całej prostocie klasy `SimpleJaxWsServiceEndpoint` trzeba pamiętać, że działa ona jedynie w środowisku uruchomieniowym JAX-WS, które obsługuje publikacje punktów końcowych z adresem. Jednym z takich środowisk jest środowisko uruchomieniowe JAX-WS, dostarczane wraz z JDK 1.6 firmy Sun. Inne środowiska uruchomieniowe JAX-WS, takie jak implementacja referencyjna JAX-WS 2.1, nie obsługują tego rodzaju publikacji i dlatego nie mogą zostać użyte razem z `SimpleJaxWsServiceEndpoint`.

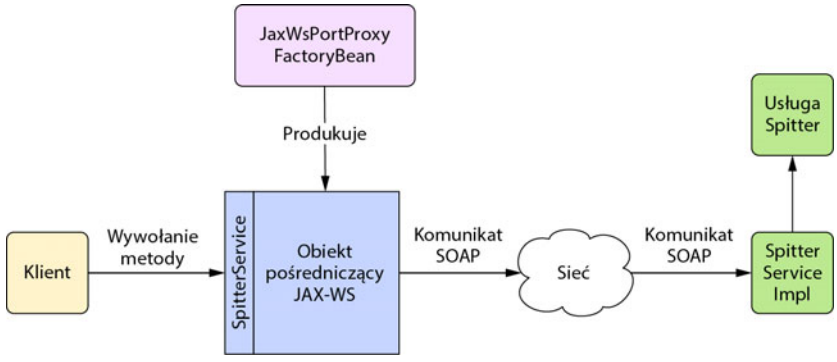
15.5.2. Pośrednik usług JAX-WS po stronie klienta

Publikacja usług sieciowych za pomocą Springa różni się, jak widać, od publikacji usług w RMI, Hessian, Burlap i za pomocą obiektu wywołującego HTTP. Ale jak zaraz się przekonamy, konsumpcja usług sieciowych przez Springa, w której udział biorą obiekty pośredniczące po stronie klienta, odbywa się na podobnej zasadzie, co konsumpcja innych technologii zdalnego dostępu przez klienty bazujące na Springu.

Przy użyciu `JaxWsPortProxyFactoryBean` możemy powiązać usługę sieciową `Spitter`, jak każdy inny komponent. `JaxWsPortProxyFactoryBean` jest komponentem-fabryką Springa, który produkuje pośrednika komunikującego się z usługą sieciową SOAP. Obiekt pośredniczący implementuje interfejs usługi (rysunek 15.10). W rezultacie, `JaxWsPortPro ↪xyFactoryBean` umożliwia powiązanie i użycie zdalnej usługi sieciowej, tak jakby była ona lokalnym obiektem POJO.

W naszej konfiguracji `JaxWsPortProxyyFactoryBean` odniesiemy się do usługi sieciowej `Spitter` w następujący sposób:

```
@Bean
public JaxWsPortProxyFactoryBean spitterService() {
  JaxWsPortProxyFactoryBean proxy = new JaxWsPortProxyFactoryBean();
  proxy.setWsdlDocument(
          "http://localhost:8080/services/SpitterService?wsdl");
  proxy.setServiceName("spitterService");
  proxy.setPortName("spitterServiceHttpPort");
```



Rysunek 15.10. JaxWsPortProxyFactoryBean produkuje obiekty pośredniczące, które komunikują się ze zdalnymi usługami sieciowymi. Obiekty te mogą następnie zostać dowiązane do innych komponentów, tak jak lokalne obiekty POJO

```
  proxy.setServiceInterface(SpitterService.class);
  proxy.setNamespaceUri("http://spitter.com");
  return proxy;
}
```

Jak widać, komponent `JaxWsPortProxyFactoryBean` potrzebuje do działania ustawienia kilku właściwości. Właściwość `wsdlDocumentUrl` identyfikuje adres pliku definicji zdalnej usługi sieciowej. `JaxWsPortProxyFactoryBean` użyje dostępnego pod tym adresem kodu WSDL do skonstruowania obiektu pośredniczącego dla usługi. Wyprodukowany obiekt będzie implementował interfejs `SpitterService`, zgodnie z wartością właściwości `service ➥Interface`.

Wartości pozostałych trzech właściwości najczęściej wynikają z kodu WSDL usługi. Żeby to zilustrować, załóżmy, że kod WSDL usługi `Spitter` wyglądał następująco:

```
<wsdl:definitions targetNamespace="http://spitter.com">
...
   <wsdl:service name="spitterService">
      <wsdl:port name="spitterServiceHttpPort" binding="tns:spitterServiceHttpBinding">
...
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Chociaż jest to mało prawdopodobne, to jednak istnieje możliwość definicji wielu usług lub (i) portów w kodzie WSDL usługi. Dlatego `JaxWsPortProxyFactoryBean` wymaga określenia nazw portów i usług we właściwościach `portName` i `serviceName`. Szybki rzut oka na atrybuty `name` elementów `<wsdl:port>` i `<wsdl:service>` w kodzie WSDL pomoże zorientować się, jakie wartości powinny zostać nadane tym właściwościom.

Wreszcie, właściwość `namespaceUri` określa przestrzeń nazw usługi. Przestrzeń nazw ma przede wszystkim pomóc komponentowi `JaxWsPortProxyFactoryBean` zlokalizować definicję usługi w kodzie WSDL. Podobnie jak w przypadku nazw portu i usługi, poprawną wartość tej właściwości znaleźć można, zaglądając do kodu WSDL. Z reguły jest ona dostępna w atrybucie `targetNamespace` elementu `<wsdl:definition>`.

15.6. Podsumowanie

Praca ze zdalnymi usługami to najczęściej nużący obowiązek. Spring oferuje jednak obsługę zdalnego dostępu, która czyni ją tak prostą, jak praca ze standardowymi komponentami JavaBean.

Po stronie klienta Spring dostarcza komponenty fabryk pośredników, dzięki którym można skonfigurować zdalne usługi w aplikacji Springa. Bez względu na to, czy do zdalnego dostępu używasz RMI, Hessian, Burlap, obiektu wywołującego HTTP Springa, czy usług sieciowych, możesz dowiązać zdalne usługi do swojej aplikacji, tak jakby były obiektami POJO. Spring przechwytuje nawet zgłaszane wyjątki `RemoteException` i zgłasza je ponownie jako wyjątki czasu wykonania `RemoteAccessException`, uwalniając nas od konieczności obsługi wyjątku, po którym aplikacja i tak nie może powrócić do stanu wyjściowego.

Chociaż Spring ukrywa wiele szczegółów zdalnej usługi, sprawiając, że przypomina ona lokalny komponent JavaBean, warto pamiętać o konsekwencjach, które wiążą się ze zdalnymi usługami. Zdalne usługi są z natury mniej efektywne od usług lokalnych. Powinno się brać to pod uwagę przy tworzeniu kodu, który korzysta ze zdalnych usług, ograniczając zdalne wywołania, aby uniknąć wąskich gardeł wydajności.

W tym rozdziale pokazaliśmy, jak można użyć Springa do udostępnienia i skonsumowania usług na bazie pewnych podstawowych technologii zdalnego dostępu. Choć te ostatnie przydają się przy aplikacjach rozproszonych, był to tylko przedsmak pracy z architekturą zorientowaną na usługi.

Dowiedzieliśmy się też, jak eksportować komponenty jako usługi bazujące na SOAP. Chociaż ten sposób programowania usług sieciowych jest dość łatwy z punku widzenia architektury, może nie być najlepszym rozwiązaniem. W następnym rozdziale omówimy inne podejście do budowy rozproszonych aplikacji, udostępniając części aplikacji jako zasoby REST.

16 Tworzenie API modelu REST przy użyciu Spring MVC

W tym rozdziale omówimy:

- Tworzenie kontrolerów, które obsługują zasoby REST
- Reprezentację zasobów w XML, JSON oraz w innych formatach
- Konsumowanie zasobów REST

Dane są najważniejsze.

Jako programiści, często skupiamy się na budowie oprogramowania, które w profesjonalny sposób rozwiąże problemy firmy. Dane są dla nas tylko surowcem, który musi zostać przetworzony. Ale jeśliby zapytać właścicieli firm, co dla nich ważniejsze — dane czy oprogramowanie — najprawdopodobniej wybraliby te pierwsze. Dane to fundament w biznesie. Oprogramowanie prawie zawsze można zastąpić, dane gromadzone latami — nigdy.

Czy to nie zastanawiające, że przy tak dużym znaczeniu danych w procesie tworzenia oprogramowania są one tak często spychane na drugi plan? Weźmy na przykład zdalne usługi z poprzedniego rozdziału. Koncentrowały się one głównie na działaniach i procesach, nie na informacji i danych.

REST (*Representational State Transfer*) wyłonił się w ostatnich latach jako skoncentrowana na informacji alternatywa dla tradycyjnych usług sieciowych opartych na SOAP. O ile SOAP koncentruje się na danych i przetwarzaniu, to REST skupia się na przetwarzanych danych.

Zaczynając od wersji 3.0, w Springu wprowadzono doskonałe mechanizmy służące do tworzenia API modelu REST. Implementacja tych mechanizmów zmieniała się i rozwijała w wersjach Springa 3.1 i 3.2, a obecnie także 4.0.

Dobrą wiadomością jest to, że obsługa REST w Springu bazuje na Spring MVC, tak więc omówiliśmy już znaczną część z tego, co powinniśmy wiedzieć przy pracy z REST w Springu. W tym rozdziale skorzystamy ze zdobytej wcześniej wiedzy o Spring MVC, aby zbudować kontrolery, które obsługują żądania zasobów REST. Zanim jednak przejdziemy do tych zagadnień, sprawdźmy, na czym w ogóle polega praca z REST.

16.1. Zrozumienie REST

Z całą pewnością przynajmniej słyszałeś wcześniej o REST. W ostatnich latach dużo się o tym modelu mówi. W kręgach programistów zapanowała pewna moda na krytykowanie usług sieciowych opartych na SOAP i jednoczesne promowanie REST jako alternatywy.

Na pewno wiele aplikacji potrzebuje tylko ułamka zbioru funkcji SOAP, a REST jest dużo prostszą alternatywą. Co więcej, wiele nowoczesnych aplikacji dysponuje mobilnymi, rozbudowanymi klientami napisanymi w JavaScripcie, które konsumują API w modelu REST działające na serwerze.

Problem w tym, że nie każdy zdaje sobie do końca sprawę z tego, czym tak naprawdę jest REST. W rezultacie pojawia się dużo błędnych informacji na ten temat, a etykietę „REST" dodaje się do wielu rozwiązań, które w rzeczywistości nie odpowiadają przeznaczeniu tego modelu. Zanim omówimy szczegółowo obsługę REST w Springu, musimy ustalić, czym właściwie jest sam model REST.

16.1.1. Fundamenty REST

Często popełnianym błędem w podejściu do REST jest traktowanie go jak „usługi sieciowej z adresami URL" — jak kolejnego mechanizmu zdalnego wywołania procedury (RPC), podobnego do SOAP, ale wywoływanego przez zwykłe adresy HTTP, bez używania ciężkich przestrzeni nazw XML obecnych w SOAP.

Tymczasem jest zupełnie na odwrót: REST ma niewiele wspólnego z RPC. Podczas gdy RPC bazuje na usługach i koncentruje się na działaniach i czasownikach, architektura REST korzysta z zasobów, kładąc nacisk na rzeczy i rzeczowniki, które opisują aplikację.

W zrozumieniu, na czym polega model REST, pomaga rozbicie akronimu na części składowe:

- **Reprezentacyjny** (ang. *Representational*) — Zasoby REST mogą być reprezentowane w praktycznie każdej formie, między innymi w XML, JavaScript Object Notation (JSON), a nawet w HTML — w zależności od potrzeb konsumenta tych zasobów.
- **Stanu** (ang. *State*) — Podczas pracy z REST bardziej koncentrujemy się na stanie zasobu niż na działaniach, które możemy względem zasobu podjąć.

- **Transfer** (ang. *Transfer*) — REST umożliwia transfer danych zasobów, w konkretnej formie reprezentacji, z jednej aplikacji do drugiej.

Mówiąc bardziej zwięźle, model REST polega na transferze stanu zasobów — w odpowiedniej formie, która jest najwłaściwsza dla klienta bądź serwera — z serwera do klienta (lub na odwrót).

W REST zasoby są identyfikowane i lokalizowane na podstawie adresów URL. Nie istnieją żadne ścisłe reguły określające, jak mają wyglądać adresy URL, by były one zgodne z konwencją REST, jednak powinny określać zasoby, a nie reprezentować polecenia przekazywane serwerowi. Nasza uwaga koncentruje się na rzeczach, a nie czynnościach.

Mimo to w modelu REST są dostępne akcje — odpowiadają one metodom protokołu HTTP. Konkretnie rzecz ujmując, czasownikami REST są: `GET`, `POST`, `PUT`, `DELETE`, `PATCH` oraz inne metody HTTP. Metody te są zazwyczaj odwzorowywane na zestaw operacji CRUD następująco:

- tworzenie (ang. *Create*) — `POST`,
- odczyt (ang. *Read*) — `GET`,
- aktualizacja (ang. *Update*) — `PUT` lub `PATCH`,
- usuwanie (ang. *Delete*) — `DELETE`.

Choć taki sposób odwzorowywania metod protokołu HTTP na czasowniki CRUD jest popularny, to jednak nie jest on ściśle wymagany. Istnieją sytuacje, w których do utworzenia zasobu można używać metody `PUT`, a do jego aktualizacji — metody `POST`. W rzeczywistości nieidempotentna natura metody `POST` sprawia, że jej przeznaczenie nie jest precyzyjnie określone i może być ona stosowana do wykonywania operacji, które nie odpowiadają semantyce pozostałych metod HTTP.

Korzystając z takiego opisu modelu REST, będę się starał unikać terminów takich jak *usługi REST*, *sieciowe usługi REST* oraz innych, które błędnie nadają większe znaczenie akcjom. Zamiast tego wolę podkreślać fakt, że REST koncentruje się na zasobach, i pisać o *zasobach REST* (ang. *RESTful resources*).

16.1.2. Obsługa REST w Springu

Spring od dawna oferował wszystkie elementy potrzebne do udostępniania zasobów REST. Wraz z pojawieniem się Springa 3.0 moduł Spring MVC zaczęto rozbudowywać o doskonałą obsługę REST. Obecnie, w wersji 4.0, Spring umożliwia tworzenie zasobów REST na kilka, opisanych poniżej, sposobów:

- Kontrolery mogą obsłużyć żądania wszystkich metod HTTP, w tym czterech głównych metod REST: `GET`, `PUT`, `DELETE` i `POST`. Spring 3.2 oraz jego nowsze wersje obsługują także metodę `PATCH`.
- Nowa adnotacja `@PathVariable` pozwala kontrolerom obsłużyć żądania sparametryzowanych adresów URL (adresów, które zawierają zmienne).
- Zasoby mogą być reprezentowane na wiele różnych sposobów przy użyciu widoków i producentów widoków Springa, włączając w to nowe implementacje widoków, służące do wyświetlania danych modelu w postaci XML, JSON, Atom i RSS.

- Najbardziej odpowiednia reprezentacja dla klienta może zostać wybrana przy użyciu producenta widoków `ContentNegotiatingViewResolver`.
- Wyświetlanie oparte na widokach można całkiem pominąć, używając nowej adnotacji `@ResponseBody` i różnych implementacji `HttpMethodConverter`.
- Podobnie, nowa adnotacja `@RequestBody` w połączeniu z implementacjami `HttpMethodConverter` może konwertować przychodzące dane HTTP na obiekty Javy, przekazywane do metod obsługi kontrolera.
- Aplikacje Spring mogą konsumować zasoby REST używając w tym celu szablonu `RestTemplate`.

W tym rozdziale pokażę, jak można pracować w Springu w konwencji REST, a rozpocznę od sposobu generowania zasobów REST przy użyciu Spring MVC. Następnie, w podrozdziale 16.4, skoncentrujemy się na klientach REST i zobaczymy, jak można konsumować utworzone wcześniej zasoby. Zacznijmy jednak od stworzenia kontrolera Spring MVC zgodnego z modelem REST.

16.2. Tworzenie pierwszego punktu końcowego REST

Jedną z bardzo miłych cech wsparcia Springa dla REST jest to, że już teraz wiemy całkiem sporo na temat tworzenia kontrolerów zgodnych z konwencją REST (określanych także jako kontrolery typu *RESTful*). Informacje dotyczące tworzenia aplikacji sieciowych, zdobyte w rozdziałach od 5. do 7., możemy w tej chwili wykorzystać do udostępnienia zasobów w formie API modelu REST. Zacznijmy od utworzenia pierwszego punktu końcowego REST w nowym kontrolerze o nazwie `SpittleApiController`.

Zamieszczony poniżej listing 16.1 przedstawia początek nowego kontrolera REST, który będzie obsługiwać zasoby `Spittle`. Początki są skromne, lecz na tym kontrolerze będziemy bazować w dalszej części rozdziału podczas poznawania tajników modelu programowania REST w Springu.

Listing 16.1. Kontroler Spring MVC typu RESTful

```
package spittr.api;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import spittr.Spittle;
import spittr.data.SpittleRepository;

@Controller
@RequestMapping("/spittles")
public class SpittleController {

  private static final String MAX_LONG_AS_STRING="9223372036854775807";

  private SpittleRepository spittleRepository;
```

```
  @Autowired
  public SpittleController(SpittleRepository spittleRepository) {
    this.spittleRepository = spittleRepository;
  }

  @RequestMapping(method=RequestMethod.GET)
  public List<Spittle> spittles(
      @RequestParam(value="max",
                    defaultValue=MAX_LONG_AS_STRING) long max,
      @RequestParam(value="count", defaultValue="20") int count) {

    return spittleRepository.findSpittles(max, count);
  }

}
```

Przyjrzyj się dokładnie temu listingowi. Czy widzisz, w jaki sposób przedstawiony na nim kontroler udostępnia zasób REST, a nie zwyczajną stronę WWW?

Zapewne nie. W obecnej postaci tego kontrolera nie ma niczego, co sugerowałoby, że jest on kontrolerem typu RESTful, udostępniającym zasoby REST. Jednak najprawdopodobniej rozpoznałeś metodę `spittles()` — została ona zaprezentowana już wcześniej, w rozdziale 5. (a konkretnie w punkcie 5.3.1).

Jak sobie zapewne przypominasz, kiedy zostanie odebrane żądanie `GET` dotyczące adresu `/spittles`, zostaje wywołana metoda `spittles()`. Odszukuje ona i zwraca listę obiektów `Spittle`, pobraną ze wstrzykniętego `SpittleRepository`. Lista ta zostaje umieszczona w modelu, który będzie użyty do wygenerowania widoku. W przypadku aplikacji sieciowej wykorzystywanej w przeglądarce będzie to zapewne oznaczać, że dane widoku zostaną wygenerowane na stronie WWW.

Mamy się jednak zajmować API modelu REST. W tym przypadku kod HTML nie stanowi właściwej reprezentacji danych.

Reprezentacja jest ważnym aspektem REST. Dotyczy ona sposobu komunikowania klienta i serwera odnośnie do zasobu. Każdy zasób może być reprezentowany praktycznie w dowolnej postaci. Jeśli konsument zasobów preferuje format JSON, to zasób może być przedstawiony w tym formacie. Jeżeli natomiast klient jest miłośnikiem nawiasów kątowych, to ten sam zasób może być przedstawiony w formie kodu XML. Jednocześnie człowiek przeglądający ten sam zasób w przeglądarce WWW wolałby zapewne, by został on wyświetlony w formie dokumentu HTML (albo PDF, arkusza Excel bądź też w jakiejś innej, zrozumiałej dla nas formie). Sam zasób się nie zmienia — zmieniana jest wyłącznie jego reprezentacja.

> **UWAGA** Choć Spring obsługuje wiele formatów do reprezentacji zasobów, to podczas definiowania API modelu REST nie trzeba stosować ich wszystkich. Dla większości klientów w zupełności wystarczy reprezentacja JSON i XML.

Oczywiście jeśli będziemy prezentowali treści przeznaczone do oglądania przez ludzi, to zapewne trzeba będzie obsługiwać zasoby formatowane w formie kodu HTML. Można się nawet zdecydować na prezentowanie zasobów w postaci dokumentów PDF lub arkuszy kalkulacyjnych Excel, lecz to już zależy od samego zasobu oraz natury aplikacji Spring.

Jeżeli konsumentami zasobów nie są ludzie, a na przykład inne aplikacje bądź kod wywołujący nasze punkty końcowe REST, to najczęściej stosowanymi sposobami reprezentacji są XML i JSON. Udostępnienie obu tych reprezentacji w Springu jest całkiem łatwe, więc nie trzeba się ograniczać i wybierać tylko jednej z nich.

Niemniej jednak sugeruję, by *absolutnym minimum* było udostępnianie zasobów w formacie JSON. Używanie tego formatu jest co najmniej równie łatwe jak korzystanie z XML-a (a wiele osób uważa, że jest znacznie prostsze). A jeśli klient został napisany w JavaScripcie (co obecnie zdarza się coraz częściej), to JSON jest bezdyskusyjnym zwycięzcą, gdyż stosowanie danych w formacie JSON w kodzie JavaScript nie wymaga żadnego szeregowania ani rozszeregowywania.

Warto przy tym zauważyć, że kontrolery rzadko kiedy zwracają uwagę na sposób reprezentacji zasobów. Kontrolery operują na danych wyrażonych w formie definiujących je obiektów Javy. Samo przekształcenie tych danych do postaci, która najbardziej odpowiada klientowi, następuje dopiero wtedy, gdy kontroler zakończy wykonywane przez siebie operacje.

Spring udostępnia dwie metody przekształcania zasobu reprezentowanego przez obiekty Javy do postaci, która zostanie przekazana klientowi:

- Negocjowanie zawartości (ang. *content negotiation*) — zostaje wybrany widok, który może wyświetlić model do reprezentacji, a ta z kolei może zostać przekazana klientowi.
- Konwersja komunikatów (ang. *message conversion*) — konwerter komunikatu przekształca obiekt przekazany przez kontroler do reprezentacji, która może zostać przekazana klientowi.

Ponieważ producenci widoków zostali opisani w rozdziałach 5. i 6., a zagadnienia związane z generowaniem widoków poznałeś w rozdziale 6., zacznę od pokazania, w jaki sposób użyć negocjowania zawartości do wyboru widoku lub producenta widoku, który będzie w stanie wyświetlić zasób w postaci akceptowanej przez klienta.

16.2.1. Negocjowanie reprezentacji zasobu

Być może pamiętasz z rozdziału 5. (oraz rysunku 5.1), że kiedy metoda obsługi kontrolera kończy pracę, zwraca zazwyczaj logiczną nazwę widoku. Nawet jeśli logiczna nazwa widoku nie jest bezpośrednio zwracana (metoda może na przykład zwracać wartość `void`), uzyskiwana jest z adresu URL żądania. Serwlet dyspozytora przekazuje wtedy nazwę widoku producentowi widoku, pytając o nazwę widoku, który powinien wyświetlić wynik żądania.

W aplikacjach sieciowych używanych przez ludzi wybrany widok jest prawie zawsze wyświetlany w formie HTML. Produkcja widoku jest czynnością jednowymiarową. Jeżeli nazwa widoku pasuje do widoku, zostaje on wybrany.

Kiedy produkcja widoków na podstawie nazw widoków dotyczy widoków, które mogą generować różne reprezentacje zasobu, dochodzi dodatkowy wymiar. Oprócz dopasowania widoku do nazwy widoku, wybrany widok musi być jeszcze odpowiedni dla klienta. Jeżeli klient oczekuje XML, widok wyświetlający HTML nie wystarczy — nawet jeśli dopasowano nazwę widoku.

ContentNegotiatingViewResolver jest specjalnym producentem widoków Springa, który bierze pod uwagę oczekiwany przez klienta typ zawartości. W jego najprostszej postaci ContentNegotiatingViewResolver można skonfigurować następująco:

```
@Bean
public ViewResolver cnViewResolver() {
  return new ContentNegotiatingViewResolver();
}
```

W tej prostej deklaracji komponentu dzieje się bardzo dużo. Zrozumienie działania producenta ContentNegotiatingViewResolver wymaga poznania dwuetapowego procesu negocjowania zawartości:

1. Określenie żądanego typu MIME (żądanych typów MIME).
2. Znalezienie najlepszego widoku dla określonego typu MIME (określonych typów MIME).

Przyjrzyjmy się tym etapom i zobaczmy, jaka jest rola ContentNegotiatingViewResolver na każdym z nich. Zaczniemy od ustalenia, jakiego rodzaju zawartości oczekuje klient.

OKREŚLENIE ŻĄDANYCH TYPÓW MIME

Pierwszy etap negocjacji zawartości polega na ustaleniu, jakiego rodzaju reprezentacji zasobu oczekuje klient. Na pozór wydaje się to dosyć proste. Czyż nie od tego jest nagłówek Accept? Powinien on być przecież jasną wskazówką odnośnie rodzaju reprezentacji, która ma zostać wysłana do klienta.

Niestety, na nagłówku Accept nie zawsze możemy polegać. Jeżeli klientem jest przeglądarka internetowa, nie ma żadnej gwarancji, że to, czego oczekuje klient, pokrywa się z informacją wysyłaną przez przeglądarkę w nagłówku Accept. Przeglądarki z reguły akceptują tylko typy zawartości przyjazne dla człowieka (jak na przykład text/html) i nie ma sposobu ustawienia innego rodzaju zawartości (bez wtyczek dla programistów).

ContentNegotiatingViewResolver rozważy użycie nagłówka Accept i wskazanych przez niego typów MIME, ale wcześniej sprawdzi rozszerzenie pliku adresu URL. Jeśli adres URL kończy się rozszerzeniem pliku, producent ContentNegotiatingViewResolver spróbuje określić typ na podstawie tego rozszerzenia. Jeżeli tym rozszerzeniem będzie *.json*, to odpowiednim typem MIME będzie musiał być application/json. Jeśli z kolei rozszerzenie ma postać *.xml*, to klient prosi o typ application/xml. Oczywiście jeżeli rozszerzeniem będzie *.html*, to wiadomo, że klient prosi o wyświetlenie zasobu w HTML-u (text/html).

Dopiero kiedy nie uda się ustalić typów MIME na podstawie rozszerzenia pliku, nagłówek Accept w żądaniu zostanie wzięty pod uwagę. W takim przypadku typ (lub typy) MIME oczekiwany przez klienta zostanie odczytany z wartości nagłówka Accept i nie ma dalszej potrzeby odszukiwania go.

W końcu, jeśli nagłówek żądania nie posiada pozycji Accept, wykorzystany zostanie domyślny typ MIME, co będzie oznaczać, że klient musi zadowolić się dowolną reprezentacją zasobu, która zostanie zwrócona przez serwer.

Po określeniu typu MIME producent ContentNegotiatingViewResolver musi przekształcić logiczną nazwę widoku na obiekt View, który zostanie użyty do wyświetlenia

modelu. W odróżnieniu od innych producentów widoków Springa, `ContentNegotiating ↪ViewResolver` nie robi tego samodzielnie. Przekazuje to zadanie do innych producentów, prosząc o jego wykonanie.

`ContentNegotiatingViewResolver` prosi innych producentów widoków o przekształcenie logicznej nazwy widoku na obiekt widoku. Każdy określony w ten sposób obiekt widoku zostaje dodany do listy potencjalnych widoków. Po przygotowaniu tej listy `ContentNegotiatingViewResolver` przegląda wszystkie żądane typy MIME, starając się odszukać na liście widoków te, które będą w stanie wygenerować dane odpowiedniego typu. Pierwsze odnalezione dopasowanie zostanie użyte do wyświetlenia modelu.

WPŁYWANIE NA PROCES WYBORU TYPÓW MIME

Opisany powyżej proces wyboru typów MIME jest domyślną strategią ustalania żądanych typów MIME. Można go jednak zmienić, przekazując komponent `ContentNegotia ↪tingManager`. Poniższa lista przedstawia kilka możliwości, które obiekt ten nam zapewnia:

- podawanie domyślnego typu MIME, który zostanie użyty, jeśli nie można będzie go określić na podstawie żądania;
- określanie typu MIME przy użyciu parametru żądania;
- ignorowanie nagłówka `Accept` żądania;
- odwzorowywanie rozszerzeń żądania na konkretne typy MIME;
- zastosowanie Java Activation Framework (JAF) jako opcji do określania typów MIME na podstawie rozszerzeń.

`ContentNegotiationManager` można konfigurować na trzy sposoby:

- bezpośrednio deklarując komponenty typu `ContentNegotiationManager`;
- tworząc komponent pośrednio, przy użyciu komponentu fabryki `ContentNegotia ↪tionManagerFactoryBean`;
- przesłaniając metodę `configureContentNegotiation()` adaptera `WebMvcConfigurer ↪Adapter`.

Bezpośrednie tworzenie komponentu `ContentNegotiationManager` jest dosyć złożone i raczej nie jest czymś, co chcielibyśmy robić, nie mając ważkiego powodu. Pozostałe dwa sposoby istnieją właśnie po to, by ułatwić tworzenie tych obiektów.

Ogólnie rzecz biorąc, komponent fabryki `ContentNegotiationManagerFactoryBean` jest najbardziej użyteczny w przypadku konfigurowania komponentów `ContentNegotiation ↪Manager` przy użyciu XML-a. Na przykład poniżej pokazano, jak skonfigurować w tym komponencie domyślny typ MIME `application/json`:

```
<bean id="contentNegotiationManager"
  class="org.springframework.http.ContentNegotiationManagerFactoryBean"
  p:defaultContentType="application/json">
```

Ponieważ `ContentNegotiationManagerFactoryBean` jest implementacją `FactoryBean`, to zastosowanie powyższego kodu powoduje utworzenie komponentu `ContentNegotiation ↪Manager`. Komponent ten można następnie wstrzyknąć do właściwości `contentNegotia ↪tionManager` producenta widoków `ContentNegotiatingViewResolver`.

ContentNegotiationManager dodano w Springu 3.2

`ContentNegotiationManager` jest stosunkowo nowym dodatkiem, gdyż wprowadzono go w wersji 3.2 Springa. We wcześniejszych wersjach przeważająca większość możliwości funkcjonalnych producenta `ContentNegotiatingViewResolver` była konfigurowana poprzez określanie wartości jego właściwości. Jednak w Springu 3.2 większość metod ustawiających klasy `ContentNegotiationResolver` została wycofana, a zalecanym sposobem konfigurowania producenta widoków jest użycie komponentu `ContentNegotiationManager`.

Choć w tym rozdziale nie przedstawię wcześniejszej metody konfigurowania producenta widoków `ContentNegotiatingViewResolver`, to warto wiedzieć, że wiele właściwości określanych podczas tworzenia komponentów `ContentNegotiationManager` ma odpowiadające im właściwości w klasie `ContentNegotiatingViewResolver`. Dlatego odwzorowanie nowego sposobu konfigurowania na stary, w razie stosowania starszej wersji Springa, nie powinno przysporzyć większych problemów.

W przypadku konfigurowania komponentu `ContentNegotiationManager` w kodzie Javy najprościej to zrobić, rozszerzając klasę `WebMvcConfigurationAdapter` i przesłaniając metodę `configureContentNegotiation()`. Istnieje spore prawdopodobieństwo, że klasa ta została już rozszerzona na początku prac nad aplikacją Spring MVC. W naszej przykładowej aplikacji Spittr dysponujemy już klasą, która rozszerza `WebMvcConfigurationAdapter` — nosi ona nazwę `WebConfig`. A zatem jedyne, co musimy zrobić, to przesłonić metodę `confi`
`↪gureContentNegotiation()` tej klasy. Przedstawiona poniżej implementacja tej metody określa domyślny typ zawartości:

```
@Override
public void configureContentNegotiation(
    ContentNegotiationConfigurer configurer) {
  configurer.defaultContentType(MediaType.APPLICATION_JSON);
}
```

Jak widać, do metody `configureContentNegotiation()` przekazywany jest obiekt `Content`
`↪NegotiationConfigurer`, na którym może ona operować. Obiekt ten udostępnia kilka metod, odpowiadających metodom ustawiającym komponentu `ContentNegotiationManager` i pozwalających określać wszelkie sposoby negocjowania zawartości przez komponent `ContentNegotiationManager`, który zostanie utworzony. W tym przypadku wywołana została metoda `defaultContentType()`, która określi domyślny typ zawartości na `appli`
`↪cation/json`.

Kiedy już będziemy dysponować komponentem `ContentNegotiationManager`, pozostanie nam jedynie wstrzyknąć go do właściwości `contentNegotiationManager` producenta widoku `ContentNegotiatingViewResolver`. Wymaga to wprowadzenia niewielkiej zmiany w metodzie `@Bean`, w której producent został zadeklarowany:

```
@Bean
public ViewResolver cnViewResolver(ContentNegotiationManager cnm) {
  ContentNegotiatingViewResolver cnvr =
      new ContentNegotiatingViewResolver();
  cnvr.setContentNegotiationManager(cnm);
  return cnvr;
}
```

W wywołaniu tej metody przekazywany jest komponent `ContentNegotiationManager`, który następnie zostaje użyty jako argument wywołania metody `setContentNegotiation➥Manager()`. W efekcie producent `ContentNegotiatingViewResolver` przejmuje zachowania zdefiniowane w przekazanym komponencie `ContentNegotiationManager`.

Istnieje tyle różnych wariantów konfiguracji komponentu `ContentNegotiationManager`, że przedstawienie ich wszystkich w tej książce nie byłoby możliwe. Na zamieszczonym poniżej listingu 16.2 zaprezentowałem stosunkowo prostą konfigurację, której zazwyczaj używam podczas korzystania z producenta `ContentNegotiatingViewResolver`:

Listing 16.2. Konfigurowanie komponentu ContentNegotiationManager

```
@Bean
public ViewResolver cnViewResolver(ContentNegotiationManager cnm) {
  ContentNegotiatingViewResolver cnvr =
      new ContentNegotiatingViewResolver();
  cnvr.setContentNegotiationManager(cnm);
  return cnvr;
}

@Override
public void configureContentNegotiation(
    ContentNegotiationConfigurer configurer) {
  configurer.defaultContentType(MediaType.TEXT_HTML); <-- Domyślnym typem zawartości będzie HTML
}

@Bean
public ViewResolver beanNameViewResolver() { <-- Widoki będą odnajdywane jako komponenty
  return new BeanNameViewResolver();
}

@Bean
public View spittles() {
  return new MappingJackson2JsonView(); <-- Widok spittle'ów używający formatu JSON
}
```

Oprócz producenta przedstawionego na listingu 16.2 aplikacja miałaby do dyspozycji zapewne także producenta służącego do generowania widoku HTML (takiego jak `InternalResourceViewResolver` lub `TilesViewResolver`). W większości przypadków producent `ContentNegotiatingViewResolver` przyjmuje, że klient chce danych przedstawionych w formie kodu HTML, zgodnie z konfiguracją określoną w komponencie `Content➥NegotiationManager`. Jeśli jednak klient zaznaczy, że chce otrzymać kod JSON (przesyłając w tym celu żądanie z rozszerzeniem *.json* albo umieszczając odpowiednią wartość w nagłówku `Accept`), to producent `ContentNegotiatingViewResolver` spróbuje odnaleźć odpowiedniego producenta, który jest w stanie generować widoki JSON.

Jeżeli logiczną nazwą widoku jest „spittles", to skonfigurowany producent widoku `BeanNameViewResolver` utworzy obiekt `View` zadeklarowany w metodzie `spittles()`. Stanie się tak dlatego, że nazwa komponentu będzie odpowiadać logicznej nazwie widoku. W przeciwnym razie, jeśli nie będzie żadnego pasującego widoku, `ContentNegotiating➥ViewResolver` użyje ustawienia domyślnego, czyli widoku generującego kodu HTML.

Kiedy producent widoków `ContentNegotiatingViewResolver` już pozna typy MIME oczekiwane przez klienta, nadejdzie czas na odnalezienie widoku, który będzie w stanie wygenerować klientowi zawartość tego typu.

ZALETY I OGRANICZENIA PRODUCENTA CONTENTNEGOTIATINGVIEWRESOLVER

Podstawowa zaleta stosowania producenta widoków `ContentNegotiatingViewResolver` polega na tym, że umieszcza on określanie reprezentacji zasobów REST ponad Spring MVC i nie wymusza modyfikowania kodu kontrolerów. Dokładnie ten sam kontroler, który generuje kod HTML przeznaczony dla ludzi przeglądających zasoby, może także generować kod JSON lub XML przeznaczony dla innych klientów.

Negocjowanie zawartości jest wygodnym rozwiązaniem, kiedy interfejsy dla ludzi oraz innych klientów w znacznym stopniu się pokrywają. Jednak w praktyce widoki generujące treści dla ludzi rzadko kiedy dysponują równie wysokim poziomem szczegółowości co API modelu REST. Gdy interfejsy dla ludzi oraz dla innych klientów nie pokrywają się w dużym stopniu, znacznie trudniej jest zauważyć zalety stosowania producenta `ContentNegotiatingViewResolver`.

Producent widoków `ContentNegotiatingViewResolver` ma także poważne ograniczenia. Ponieważ stanowi on implementację `ViewResolver`, to zapewnia jedynie możliwość określania sposobu wyświetlania zasobu. Nie ma jednak nic do powiedzenia, jeśli chodzi o to, jakie reprezentacje zasobu przesyłane przez klienta kontroler będzie mógł konsumować. Jeżeli klient może przesyłać dane JSON bądź XML, producent `ContentNego`
`↪tiatingViewResolver` nie będzie w stanie pomóc w wyborze jednej z nich.

Jest jeszcze jedna sprawa związana ze stosowaniem tego producenta widoków. Wybrany obiekt `View` generuje dla klienta model, a nie zasób. To subtelna, lecz bardzo ważna różnica. Kiedy klient zażąda listy obiektów `Spittle` zapisanych w formie kodu JSON, to prawdopodobnie będzie oczekiwał odpowiedzi wyglądającej podobnie jak poniższy fragment kodu:

```
[
  {
    "id": 42,
    "latitude": 28.419489,
    "longitude": -81.581184,
    "message": "Witaj, świecie!",
    "time": 1400389200000
  },
  {
    "id": 43,
    "latitude": 28.419136,
    "longitude": -81.577225,
    "message": "Odpalamy!",
    "time": 1400475600000
  }
]
```

Jednak model jest mapą par klucz-wartość, dlatego odpowiedź będzie wyglądać tak:

```
{
    "spittleList" : [
        {
```

```
        "id": 42,
        "latitude": 28.419489,
        "longitude": -81.581184,
        "message": "Witaj, świecie!",
        "time": 1400389200000
      },
      {
        "id": 43,
        "latitude": 28.419136,
        "longitude": -81.577225,
        "message": "Odpalamy!",
        "time": 1400475600000
      }
  ]
}
```

Choć nie jest to nic strasznego, to może nie być tym, czego klient oczekuje.

Ze względu na te ograniczenia osobiście wolę nie używać producenta `ContentNego↪tiatingViewResolver`. Zamiast tego, jeśli chodzi o tworzenie reprezentacji zasobów, zdecydowanie skłaniam się do stosowania konwerterów komunikatów Springa. Przekonajmy się zatem, w jaki sposób można wykorzystać konwertery komunikatów w metodach kontrolerów Springa.

16.2.2. Stosowanie konwerterów komunikatów HTTP

Konwersja komunikatów jest bardziej bezpośrednim sposobem przekształcania danych przygotowanych przez kontroler do reprezentacji, która zostanie przekazana klientowi. W przypadku ich stosowania serwlet dyspozytora nie zaprząta sobie głowy przekazywaniem do widoku danych modelu. W rzeczywistości nie ma bowiem żadnego modelu ani żadnego widoku. Istnieją jedynie dane przygotowane przez kontroler oraz reprezentacja zasobu wygenerowana przez konwerter komunikatu podczas konwertowania tych danych.

Spring udostępnia szereg konwerterów komunikatów przydatnych przy najczęściej spotykanych konwersjach obiekt-reprezentacja. Pełną ich listę można znaleźć w tabeli 16.1.

Załóżmy na przykład, że klient określił poprzez nagłówek `Accept` żądania, że akceptuje dane typu `application/json`. Zakładając, że biblioteka Jackson JSON znajduje się w ścieżce do klas aplikacji, obiekt zwrócony przez metodę obsługi zostanie przekazany konwerterowi `MappingJacksonHttpMessageConverter`, który skonwertuje go na reprezentację JSON, a następnie zwróci klientowi. Jeśli jednak z nagłówka żądania wynika, że klient preferuje typ `text/xml`, do wygenerowania odpowiedzi XML dla klienta zatrudniony zostanie konwerter `Jaxb2RootElementHttpMessageConverter`.

Zwróć uwagę, że za wyjątkiem pięciu, wszystkie konwertery z tabeli 16.1 są rejestrowane domyślnie, nie wymagają zatem konfiguracji Springa. Może się jednak zdarzyć, że do ich użycia konieczne będzie dodanie dodatkowych bibliotek do ścieżki klas aplikacji. Jeśli chcesz na przykład, aby `MappingJacksonHttpMessageConverter` konwertował komunikaty JSON na obiekty Javy, musisz dodać bibliotekę Jackson JSON Processor do

Tabela 16.1. Spring dostarcza szereg konwerterów komunikatów HTTP, które dokonują konwersji reprezentacji zasobów z obiektów i na obiekty Javy

Konwerter komunikatu	Opis
`AtomFeedHttpMessageConverter`	Dokonuje konwersji obiektów `Feed` projektu Rome na kanały (z kanałów) Atom (typ MIME `application/ atom+xml`). *Rejestrowany, gdy biblioteka Rome jest obecna w ścieżce do klas*
`BufferedImageHttpMessageConverter`	Dokonuje konwersji obiektu typu `BufferedImage` na dane binarne (z danych binarnych) obrazka.
`ByteArrayHttpMessageConverter`	Odczytuje (zapisuje) tablice bajtowe. Odczytuje wszystkie typy MIME (*/*), a zapisuje jako `application/octet-stream`. *Rejestrowany domyślnie*
`FormHttpMessageConverter`	Odczytuje zawartość `application/x-www-form-urlencoded` i zapisuje ją jako `MultiValueMap<String, String>`. Zapisuje także `MultiValueMap<String, String>` jako `application/x-www-form-urlencoded` oraz `MultiValueMap<String, Object>` jako `multipart/form-data`.
`Jaxb2RootElementHttpMessageConverter`	Odczytuje i zapisuje XML (`text/xml` lub `application/xml`), zapisuje jako obiekt z adnotacjami JAXB2 i vice versa. *Rejestrowany, gdy biblioteki JAXB 2 są obecne w ścieżce do klas*
`MappingJacksonHttpMessageConverter`	Odczytuje JSON, zapisując go w postaci obiektów z typami lub obiektu `HashMap` bez typów i vice versa. *Rejestrowany, gdy biblioteka Jackson JSON jest obecna w ścieżce do klas*
`MappingJackson2HttpMessageConverter`	Odczytuje JSON, zapisując go w postaci obiektów z typami lub obiektu `HashMap` bez typów i vice versa. *Rejestrowany, gdy biblioteka Jackson 2 JSON jest obecna w ścieżce do klas*
`MarshallingHttpMessageConverter`	Odczytuje i zapisuje XML za pomocą wstrzykniętych obiektów szeregujących i rozszeregowujących, jak na przykład Castor, JAXB2, JIBX, XMLBeans czy XStream.
`ResourceHttpMessageConverter`	Odczytuje i zapisuje dane typu `org.springframework.core.io.Resource`.
`RssChannelHttpMessageConverter`	Odczytuje kanały RSS zapisując je jako obiekty `Channel` Rome i na odwrót. *Rejestrowany, gdy biblioteka Rome jest obecna w ścieżce do klas*
`SourceHttpMessageConverter`	Odczytuje XML zapisując go jako obiekty `javax.xml.transform.Source`.
`StringHttpMessageConverter`	Odczytuje wszystkie typy MIME (*/*) jako `String`. Zapisuje obiekty `String` jako `text/plain`.
`XmlAwareFormHttpMessageConverter`	Rozszerza `FormHttpMesssageConverter`. Dodaje obsługę fragmentów opartych na XML za pomocą `SourceHttpMessageConverter`.

ścieżki do klas. Podobnie w przypadku konwersji komunikatów z kodu XML na obiekty Javy i na odwrót, wykonywanej przy użyciu konwertera `Jaxb2RootElementHttpMessageConverter`, konieczne będzie zastosowanie biblioteki JAXB. Natomiast konwersja

komunikatów w formatach Atom i RSS przy użyciu konwerterów `AtomFeedHttpRequestConverter` oraz `RssChannelHttpMessageConverter` będzie wymagać biblioteki Rome.

Jak można się spodziewać, w celu wykorzystania konwersji komunikatów trzeba będzie nieco odejść od tradycyjnego modelu programowania Spring MVC. Spróbujmy zatem trochę zmodyfikować kontroler z listingu 16.1, tak by korzystał z konwersji komunikatów.

ZWRACANIE STANU OBIEKTU W TREŚCI ŻĄDANIA

Zazwyczaj, kiedy metoda obsługi zwraca obiekt Javy (cokolwiek innego od `String` lub implementacji `View`), trafia on do modelu, który zostanie użyty podczas wyświetlania widoku. Jednak w przypadku stosowania konwersji komunikatów musimy poinformować Spring, by pominął standardowy sposób działania bazujący na obiektach modelu i widoku i by zamiast niego zastosował konwersję komunikatów. Można to zrobić na kilka sposobów, ale najprostszym z nich jest dodanie do metody kontrolera adnotacji `@ResponseBody`.

Wracając do metody `spittles()` z listingu 16.1 — możemy dodać do niej adnotację `@ResponseBody`, aby Spring skonwertował zwróconą listę `List<Spittles>` na zawartość odpowiedzi:

```
@RequestMapping(method=RequestMethod.GET,
                produces="application/json")
public @ResponseBody List<Spittle> spittles(
    @RequestParam(value="max",
                  defaultValue=MAX_LONG_AS_STRING) long max,
    @RequestParam(value="count", defaultValue="20") int count) {

  return spittleRepository.findSpittles(max, count);
}
```

Adnotacja `@ResponseBody` informuje Spring, że zwrócony obiekt chcemy przesłać do klienta jako zasób skonwertowany do jakiejś postaci, którą klient jest w stanie przyjąć. Konkretnie rzecz biorąc, serwlet dyspozytora przeanalizuje nagłówek `Accept` żądania i poszuka konwertera komunikatów, który będzie w stanie wygenerować zasób w postaci oczekiwanej przez klienta.

Na przykład jeśli nagłówek `Accept` określa, że klient akceptuje odpowiedź typu `application/json`, i jeśli biblioteka Jackson JSON jest dostępna w ścieżce klas aplikacji, to zostanie wybrany konwerter `MappingJacksonHttpMessageConverter` lub `MappingJackson2HttpMessageConverter` (zależnie od tego, która wersja biblioteki Jackson będzie dostępna). Konwerter komunikatów skonwertuje listę obiektów `Spittle`, zwróconą przez kontroler, do postaci dokumentu JSON, który zostanie przesłany w zawartości odpowiedzi. Postać takiej odpowiedzi mogłaby przypominać tablicę JSON przedstawioną poniżej:

```
[
  {
    "id": 42,
    "latitude": 28.419489,
    "longitude": -81.581184,
    "message": "Witaj, świecie!",
```

```
    "time": 1400389200000
  },
  {
    "id": 43,
    "latitude": 28.419136,
    "longitude": -81.577225,
    "message": "Odpalamy!",
    "time": 1400475600000
  }
]
```

Jackson domyślnie używa odzwierciedlania

Trzeba pamiętać, że do tworzenia zasobów JSON reprezentujących obiekty zwracane przez kontroler biblioteka Jackson JSON domyślnie używa techniki odzwierciedlania (ang. *reflection*). W przypadku prostych reprezentacji takie rozwiązanie może dawać zadowalające wyniki. Jeśli jednak typ Javy będzie refaktoryzowany poprzez dodawanie, usuwanie czy zmianę nazw właściwości, to zmiany te będą także wprowadzane w wynikowym kodzie JSON (co w skrajnym wypadku może nawet doprowadzić do awarii klienta).

Jednak poprzez zastosowanie odpowiednich adnotacji w typie Javy można modyfikować sposób, w jaki biblioteka Jackson będzie go konwertować do kodu JSON. Uzyskujemy dzięki temu większą kontrolę nad wynikową postacią kodu JSON i możemy uchronić się przed zmianami, które mogłyby wywołać problemy w naszym API oraz w korzystających z niego klientach.

Prezentacja wspomnianych adnotacji znacznie wykracza poza ramy tematyczne tej książki; informacje o nich można znaleźć w dokumentacji na stronie *http://wiki.fasterxml.com/JacksonAnnotations*.

Skoro wspominam o nagłówku `Accept`, warto zwrócić uwagę na adnotację `@Request ↪Mapping` dodaną do metody `spittle()`. Dodałem do niej atrybut `produces`, aby zadeklarować, że metoda ta ma obsługiwać wyłącznie te żądania, które oczekują odpowiedzi w formacie JSON. Oznacza to, że ta metoda będzie obsługiwać tylko te żądania, których nagłówek `Accept` zawiera `application/json`. Wszelkie inne żądania, nawet żądania `GET`, których adres URL pasuje do podanej ścieżki, nie będą obsługiwane przez tę metodę. Albo zostaną one obsłużone przez inną metodę obsługi (jeśli odpowiednia metoda będzie dostępna), albo do klienta zostanie przesłana odpowiedź HTTP 406 (`Not Acceptable` — niedozwolone).

ODBIERANIE STANU ZASOBU W TREŚCI ŻĄDANIA

Jak na razie koncentrowaliśmy się wyłącznie na punktach końcowych REST, które udostępniają zasoby klientom. Jednak REST nie jest modelem tylko do odczytu. API modelu REST mogą także przyjmować reprezentacje zasobów nadsyłane przez klienty. Nie byłoby to szczególnie wygodne, gdyby nasz kontroler musiał konwertować reprezentacje JSON lub XML zasobu, nadsyłane przez klienty, na obiekty Javy, na których operuje. Konwertery komunikatów Springa były w stanie konwertować obiekty do odpowiednich reprezentacji przesyłanych do klientów — czy zatem teraz będą w stanie zrobić to samo z danymi odbieranymi przez naszą aplikację?

Podobnie jak adnotacja `@ResponseBody` pozwala używać konwerterów komunikatów do konwersji danych wysyłanych do klienta, tak adnotacja `@RequestBody` nakazuje, by

Spring odnalazł konwerter potrafiący skonwertować reprezentację zasobu przesyłaną przez klienta na obiekt Javy. Na przykład załóżmy, że musimy zapewnić klientowi możliwość przysyłania nowych obiektów `Spittle`, które aplikacja ma zapamiętać. Poniżej przedstawiona została metoda kontrolera, która będzie obsługiwać takie żądania:

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
public @ResponseBody
    Spittle saveSpittle(@RequestBody Spittle spittle) {
  return spittleRepository.save(spittle);
}
```

Gdyby pominąć adnotacje, to `saveSpittle()` byłaby naprawdę prostą metodą. Pobiera ona obiekt `Spittle` jako parametr, zapisuje go, używając `spittleRepository`, i zwraca obiekt `Spittle` zwrócony przez wywołanie `spittleRepository.save()`.

Jednak dzięki zastosowaniu adnotacji metoda `saveSpittle()` staje się znacznie bardziej interesująca i potężna. Adnotacja `@RequestMapping` oznacza, że będzie ona obsługiwać wyłącznie żądania `POST` nadsyłane na adres `/spittles` (zadeklarowany w adnotacji `@RequestMapping` na poziomie klasy). Zawartość żądania `POST` ma obejmować reprezentację zasobu `Spittle`. Ponieważ do parametru `Spittle` została dodana adnotacja `@RequestBody`, to Spring przeanalizuje nagłówek `Content-Type` żądania i spróbuje znaleźć konwerter komunikatów, który będzie w stanie skonwertować zawartość żądania do obiektu `Spittle`.

Na przykład jeśli klient prześle dane `Spittle` zapisane w postaci kodu JSON, to nagłówek `Content-Type` takiego żądania może mieć postać `application/json`. W takim przypadku serwlet dyspozytora poszuka konwertera komunikatów, który będzie w stanie skonwertować dane JSON na obiekty Javy. Jeżeli w ścieżce klas aplikacji będzie dostępna biblioteka Jackson 2, to zastosowany zostanie konwerter `MappingJackson2Http` ↪`MessageConverter`, który skonwertuje dane JSON na obiekt `Spittle`, a ten zostanie w efekcie przekazany w wywołaniu metody `saveSpittle()`. W przedstawionej metodzie została także użyta adnotacja `@ResponseBody`, sprawiająca, że zwrócony obiekt `Spittle` zostanie skonwertowany do odpowiedniej postaci, która zostanie przekazana do klienta.

Warto zauważyć, że adnotacja `@RequestMapping` zawiera atrybut `consumes` o wartości `application/json`. Atrybut ten działa bardzo podobnie jak atrybut `produces`, lecz odnosi się do nagłówka żądania `Content-Type`. W naszym przypadku atrybut ten informuje, że metoda będzie obsługiwać żądania `POST` przesyłane na adres `/spittles`, lecz wyłącznie wtedy, gdy nagłówek `Content-Type` będzie zawierał `application/json`. W przeciwnym razie żądanie zostanie obsłużone przez jakąś inną metodę (o ile taka będzie dostępna).

DOMYŚLNE KONTROLERY DO KONWERSJI KOMUNIKATÓW

Adnotacje `@ResponseBody` oraz `@RequestBody` są zwięzłymi, ale potężnymi sposobami pozwalającymi na stosowanie konwerterów komunikatów Springa podczas obsługi żądań. Jednak w przypadkach, kiedy tworzony kontroler ma kilka metod, z których wszystkie muszą korzystać z konwerterów komunikatów, to określanie ich przy użyciu tych adnotacji może być nieco nadmiarowe.

W Springu 4.0 wprowadzona została adnotacja `@RestController`, która pozwala rozwiązać ten problem. Jeśli do kontrolera zamiast `@Controller` zostanie dodana adno-

tacja `@RestController`, Spring zastosuje konwersję komunikatów do wszystkich metod obsługi zdefiniowanych w danym kontrolerze. W takiej sytuacji dodawanie adnotacji `@ResponseBody` do wszystkich metod nie będzie już konieczne. Przedstawiony wcześniej kontroler `SpittleController` można zatem zmodyfikować w sposób przedstawiony na listingu 16.3.

Listing 16.3. Zastosowanie adnotacji @RestController

```
package spittr.api;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import spittr.Spittle;
import spittr.data.SpittleRepository;

@RestController ← Domyślnie będzie stosowana konwersja komunikatów
@RequestMapping("/spittles")
public class SpittleController {

  private static final String MAX_LONG_AS_STRING="9223372036854775807";

  private SpittleRepository spittleRepository;

  @Autowired
  public SpittleController(SpittleRepository spittleRepository) {
    this.spittleRepository = spittleRepository;
  }

  @RequestMapping(method=RequestMethod.GET)
  public List<Spittle> spittles(
        @RequestParam(value="max",
                      defaultValue=MAX_LONG_AS_STRING) long max,
        @RequestParam(value="count", defaultValue="20") int count) {
    return spittleRepository.findSpittles(max, count);
  }

  @RequestMapping(
      method=RequestMethod.POST
      consumes="application/json")
  public Spittle saveSpittle(@RequestBody Spittle spittle) {
    return spittleRepository.save(spittle);
  }
}
```

Kluczową rzeczą, na jaką należy zwrócić uwagę na listingu 16.3, jest to, czego nie ma w kodzie. Żadna z metod obsługi przedstawionego kontrolera nie zawiera adnotacji `@ResponseBody`. Jednak ponieważ do kontrolera została dodana adnotacja `@RestController`, to obiekty zwracane przez te metody i tak będą konwertowane do postaci wymaganej przez klienta przy użyciu odpowiednich konwerterów komunikatów.

Jak na razie dowiedziałeś się, jak stosować model programowania Spring MVC do publikowania zasobów w modelu REST w zawartości odpowiedzi. Jednak odpowiedzi

nie składają się wyłącznie z samej zawartości. W ich skład wchodzą również nagłówki oraz kody statusu, które także mogą przekazywać przydatne informacje o odpowiedzi. Zobaczmy zatem, jak określać nagłówki odpowiedzi i ustawiać kody statusu podczas generowania zasobów.

16.3. Zwracanie zasobów to nie wszystko

Adnotacja `@ResponseBody` jest przydatna podczas przekształcania zwracanych przez kontroler obiektów Javy do postaci reprezentacji zasobu przesyłanej do klienta. Jak się jednak okazuje, przesyłanie do klientów reprezentacji zasobów to tylko część historii. Dobre API modelu REST robi znacznie więcej niż przesyłanie zasobów pomiędzy klientem a serwerem. Przekazuje też klientom metadane, które pomagają im zrozumieć zasób lub dowiedzieć się, co zaszło podczas obsługi żądania.

16.3.1. Przekazywanie błędów

Zacznijmy na przykład od dodania do kontrolera `SpittleControler` nowej metody obsługi, która będzie zwracać pojedynczy obiekt `Spittle`:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public @ResponseBody Spittle spittleById(@PathVariable long id) {
  return spittleRepository.findOne(id);
}
```

Identyfikator jest przekazywany jako wartość parametru `id` i służy do odszukania obiektu przy użyciu metody `findOne()`. Obiekt `Spittle` zwrócony przez metodę `findOne()` zostanie zwrócony jako wynik metody obsługi, a przygotowaniem reprezentacji zasobu wymaganej przez klienta zajmie się konwerter komunikatów.

Dosyć proste, prawda? Nie można wymyślić nic lepszego. A może jednak można?

Jak sądzisz, co się stanie, kiedy nie będzie obiektu `Spittle` o podanym identyfikatorze i metoda `findOne()` zwróci `null`? Do klienta nie zostaną przekazane żadne użyteczne informacje. Niemniej jednak domyślny kod statusu HTTP przekazywany w odpowiedzi ma wartość `200 (OK)`, czyli informuje, że wszystko poszło zgodnie z planem.

Ale żądanie nie zostało obsłużone prawidłowo. Klient poprosił o obiekt `Spittle`, lecz go nie otrzymał. Nie otrzymał ani obiektu, ani jakiejkolwiek informacji sugerującej, że coś poszło nie tak. W zasadzie serwer stwierdził: „Oto masz bezużyteczną odpowiedź, ale powinieneś wiedzieć, że wszystko poszło świetnie!".

A teraz zastanówmy się, co *powinno* się zdarzyć w takiej sytuacji. W najprostszym przypadku kod statusu powinien mieć wartość inną niż `200`. Powinien mieć wartość `404 (Not Found` — nie znaleziono), która poinformuje klienta, że zasób, o który prosił, nie został znaleziony. Poza tym miło by było, gdyby w zawartości odpowiedzi znalazł się komunikat o błędzie.

Spring udostępnia kilka sposobów obsługi takich scenariuszy:

- Kody statusu można określać przy użyciu adnotacji `@ResponseStatus`.
- Metody obsługi mogą zwracać obiekty `ResponseEntity` zawierające więcej metadanych dotyczących odpowiedzi.

- Błędy można obsługiwać przy użyciu odpowiednich procedur obsługi wyjątków, dzięki czemu same metody obsługi mogą się skoncentrować na ścieżce reprezentującej prawidłowy przebieg przetwarzania żądania.

To kolejny obszar, w którym Spring zapewnia dużą elastyczność i w którym nie istnieje jedno, jedynie słuszne rozwiązanie. Zamiast próbować określić jedną strategię postępowania z błędami, takimi jak ten opisany powyżej, lub obsługiwać wszystkie możliwe sytuacje, przedstawię kilka sposobów poprawienia metody `spittleById()` tak, by prawidłowo radziła sobie z sytuacjami, w których nie uda się odnaleźć poszukiwanego obiektu `Spittle`.

STOSOWANIE OBIEKTÓW RESPONSEENTITY

W ramach alternatywy dla `@ResponseBody` metody kontrolerów mogą zwracać obiekty `ResponseEntity`. Obiekty `ResponseEntity`, oprócz samego obiektu, który zostanie skonwertowany i zwrócony jako reprezentacja zasobu, zawierają metadane (takie jak nagłówki i kody statusu) dotyczące odpowiedzi.

Ponieważ obiekty `ResponseEntity` pozwalają na określanie kodu statusu, wydają się one doskonałym rozwiązaniem do przekazania kodu błędu HTTP o wartości `404`, informującego o problemach z odnalezieniem obiektu `Spittle`. Poniżej przedstawiona została nowa wersja metody `spittleById()`, zwracająca obiekt `ResponseEntity`:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<Spittle> spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  HttpStatus status = spittle != null ?
                      HttpStatus.OK : HttpStatus.NOT_FOUND;
  return new ResponseEntity<Spittle>(spittle, status);
}
```

Podobnie jak wcześniej, także w tej wersji metody identyfikator ze ścieżki zostaje użyty do pobrania z repozytorium odpowiedniego obiektu `Spittle`. Jeśli obiekt zostanie znaleziony, status odpowiedzi zostaje ustawiony na `HttpStatus.OK` (wcześniej używany domyślnie). Jeżeli jednak repozytorium zwróciło wartość `null`, zostaje wykorzystany status `HttpStatus.NOT_FOUND`, który odpowiada kodowi statusu HTTP o wartości `404`. Na końcu metody tworzony jest obiekt `ResponseEntity`, w którym zostanie umieszczony zarówno zwracany obiekt `Spittle`, jak i kod status.

Warto zauważyć, że do metody `spittleById()` nie została dodana adnotacja `@Res`
`↪ponseBody`. Poza dostarczaniem nagłówków odpowiedzi, kodu statusu oraz samej zawartości odpowiedzi zwrócenie obiektu `ResponseEntity` ma dokładnie to samo znaczenie co użycie adnotacji `@ResponseBody` — oznacza ono, że zawartość obiektu zostanie wyświetlona w treści odpowiedzi identycznie jak w przypadku, gdy w metodzie zostałaby użyta adnotacja `@ResponseBody`. A zatem jeśli metoda zwraca obiekt `ResponseEntity`, nie ma potrzeby dodawać do niej adnotacji `@ResponseBody`.

Bez wątpienia jest to krok we właściwym kierunku. Teraz, jeżeli obiekt `Spittle`, o który prosił klient, nie zostanie znaleziony, aplikacja zwróci odpowiedni kod statusu. Jednak w takim przypadku zawartość odpowiedzi wciąż pozostaje pusta, a chcielibyśmy umieścić w niej dodatkowe informacje o błędzie.

Spróbujmy więc jeszcze raz. Zacznijmy od zdefiniowania obiektu Error, który będzie zawierał informacje o błędzie:

```
public class Error {
  private int code;
  private String message;

  public Error(int code, String message) {
    this.code = code;
    this.message = message;
  }

  public int getCode() {
    return code;
  }

  public String getMessage() {
    return message;
  }
}
```

Następnie możemy zmienić metodę `spittleById()`, by zwracała obiekt `Error`:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<?> spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  if (spittle == null) {
    Error error = new Error(4, "Spittle [" + id + "] nie został znaleziony");
    return new ResponseEntity<Error>(error, HttpStatus.NOT_FOUND);
  }
  return new ResponseEntity<Spittle>(spittle, HttpStatus.OK);
}
```

Teraz przedstawiona metoda obsługi powinna działać zgodnie z naszymi oczekiwaniami. Jeśli uda się znaleźć obiekt `Spittle`, to zostanie on zwrócony wewnątrz obiektu `ResponseEntity`, który oprócz niego będzie także zawierać kod statusu `200 (OK)`. Jeżeli natomiast metoda `findOne()` zwróci wartość `null`, metoda utworzy obiekt `Error`, który zostanie umieszczony w obiekcie `ResponseEntity` wraz z kodem statusu `404 (Not Found)` i zwrócony.

Przypuszczam, że na tym moglibyśmy poprzestać. W końcu metoda działa już dokładnie tak, jak chcieliśmy. Martwi mnie jednak kilka rzeczy.

Przede wszystkim jest ona trochę bardziej skomplikowana niż wersja początkowa. Zawiera nieco więcej logiki, w tym nawet instrukcję warunkową. A poza tym zwracanie przez metodę wyniku `ResponseEntity<?>` wydaje się niewłaściwym rozwiązaniem. Takie uogólnione zastosowanie typu `ResponseEntity` pozostawia zbyt wiele pola do interpretacji i potencjalnych błędów.

Na szczęście problemy te można wyeliminować dzięki zastosowaniu obsługi błędów.

OBSŁUGA BŁĘDÓW

Instrukcja `if` użyta w metodzie `spittleById()` służy do obsługi błędu. Jednak do tego samego zadania doskonale nadają się metody obsługi błędów kontrolerów. Pozwalają one obsługiwać niemiłe realia, czyli to wszystko, co może pójść źle, oraz zapewniają

zwyczajnym metodom obsługi możliwość błogiego skoncentrowania się na ścieżce prawidłowej obsługi żądania.

Spróbujmy zatem zrefaktoryzować fragment przedstawionego wcześniej kodu i zastosować w nim metodę obsługi błędów. Zacznijmy od zdefiniowania takiej metody, która będzie wywoływana w odpowiedzi na zgłoszenie wyjątku `SpittleNotFoundException`:

```
@ExceptionHandler(SpittleNotFoundException.class)
public ResponseEntity<Error> spittleNotFound(
                                  SpittleNotFoundException e) {
  long spittleId = e.getSpittleId();
  Error error = new Error(4, "Spittle [" + spittleId + "] nie został znaleziony");
  return new ResponseEntity<Error>(error, HttpStatus.NOT_FOUND);
}
```

Dodanie do metody adnotacji `@ExceptionHandler` sprawia, że metoda ta będzie używana do obsługi konkretnych wyjątków. W powyższym przykładzie informuje, że jeśli któraś metoda obsługi zdefiniowana w tym samym kontrolerze zgłosi wyjątek `SpittleNotFound` ↪`Exception`, to w celu jego obsługi należy wywołać metodę `spittleNotFound()`.

Jeżeli natomiast chodzi o klasę `SpittleNotFoundException`, to stanowi ona całkiem prostą klasę wyjątku:

```
public class SpittleNotFoundException extends RuntimeException {
  private long spittleId;
  public SpittleNotFoundException(long spittleId) {
    this.spittleId = spittleId;
  }

  public long getSpittleId() {
    return spittleId;
  }
}
```

Teraz możemy już usunąć kod obsługi błędu z metody `spittleById()`:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<Spittle> spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  if (spittle == null) { throw new SpittleNotFoundException(id); }
  return new ResponseEntity<Spittle>(spittle, HttpStatus.OK);
}
```

Jak widać, udało nam się w całkiem dużym stopniu uprościć kod metody `spittleById()`. Z wyjątkiem sprawdzenia wartości `null` koncentruje się ona wyłącznie na ścieżce prawidłowej obsługi żądania, zakładającej, że obiekt `Spittle` został pomyślnie odnaleziony. A przede wszystkim udało nam się wyeliminować dziwne wykorzystanie typów parametryzowanych w wyniku zwracanym przez metodę.

Jednak ten kod można jeszcze bardziej uprościć. Skoro mamy pewność, że metoda `spittleById()` będzie zwracać obiekt `Spittle`, a kod statusu zawsze będzie mieć wartość `200` (`OK`), nie musimy już używać obiektu `ResponseEntity` i możemy go zastąpić adnotacją `@ResponseBody`:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public @ResponseBody Spittle spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
```

```
  if (spittle == null) { throw new SpittleNotFoundException(id); }
  return spittle;
}
```

Oczywiście jeśli do klasy kontrolera została dodana adnotacja `@RestController`, to nawet adnotacja `@ResponseBody` nie będzie potrzebna:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public Spittle spittleById(@PathVariable long id) {
  Spittle spittle = spittleRepository.findOne(id);
  if (spittle == null) { throw new SpittleNotFoundException(id); }
  return spittle;
}
```

Skoro wiemy, że metoda obsługi błędów zawsze będzie zwracać obiekt `Error` i używać kodu statusu `404` (`Not Found`), to także ją możemy zmodyfikować w podobny sposób:

```
@ExceptionHandler(SpittleNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public @ResponseBody Error spittleNotFound(SpittleNotFoundException e) {
  long spittleId = e.getSpittleId();
  return new Error(4, "Spittle [" + spittleId + "] nie został znaleziony");
}
```

Ponieważ metoda `spittleNotFound()` zawsze zwraca obiekt `Error`, jednym argumentem przemawiającym za dalszym korzystaniem z obiektu `ResponseEntity` była chęć zapewnienia możliwości określania kodu statusu. Jednak dodając do metody `spittleNotFound()` adnotację `@ResponseStatus(HttpStatus.NOT_FOUND)`, uzyskujemy dokładnie ten sam efekt, a jednocześnie możemy pozbyć się obiektu `ResponseEntity`.

Również w przypadku tej metody, jeśli do klasy kontrolera zostanie dodana adnotacja `@RestController`, będziemy mogli wyeliminować adnotację `@ResponseBody` i dodatkowo uprościć kod metody:

```
@ExceptionHandler(SpittleNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public Error spittleNotFound(SpittleNotFoundException e) {
  long spittleId = e.getSpittleId();
  return new Error(4, "Spittle [" + spittleId + "] nie został znaleziony");
}
```

W pewnym sensie zatoczyliśmy pełne koło. Aby móc określać kod statusu, zaczęliśmy stosować obiekt `ResponseEntity`. Jednak później okazało się, że możemy skorzystać z metod obsługi błędów oraz adnotacji `@ResponseStatus`, by wyeliminować obiekty `ResponseEntity` i uprościć kod.

Można by sądzić, że stosowanie obiektów `ResponseEntity` w ogóle nie jest potrzebne. Jest jednak coś, co obiekty te robią doskonale, a czego nie można zrobić ani przy użyciu adnotacji, ani za pomocą metod obsługi wyjątków. Przekonajmy się, jak można określać nagłówki odpowiedzi.

16.3.2. Ustawianie nagłówków odpowiedzi

W przypadku metody `saveSpittle()` w trakcie obsługi żądania `POST` tworzony jest nowy obiekt `Spittle`. Ale w obecnej postaci (przedstawionej na listingu 16.3) metoda nie komunikuje tego faktu klientowi.

Po obsłużeniu żądania przez metodę `saveSpittle()` serwer wysyła do klienta odpowiedź z reprezentacją nowego obiektu `Spittle` oraz kodem statusu `200 (OK)`. Choć takie rozwiązanie nie jest złe, to nie jest także całkowicie prawidłowe.

Oczywiście zakładając, że żądanie doprowadzi do prawidłowego utworzenia zasobu, użycie wspomnianego wcześniej kodu statusu jest poprawne. Ale warto by przekazać klientowi coś więcej niż lakoniczne „OK". W końcu coś zostało utworzone, a istnieje kod statusu HTTP, który może o tym poinformować. Kod HTTP 201 powiadamia o tym, że żądanie zostało obsłużone prawidłowo oraz że w jego wyniku coś utworzono. Jeśli zależy nam na pełnej i precyzyjnej komunikacji z klientem, to czy zamiast kodu statusu `200 (OK)` nie powinniśmy użyć kodu `201 (Created)`?

Jeżeli skorzystamy ze zdobytych wiadomości, taka modyfikacja nie przysporzy żadnych problemów. Wystarczy dodać do metody `saveSpittle()` następującą adnotację `@ResponseStatus`:

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Spittle saveSpittle(@RequestBody Spittle spittle) {
  return spittleRepository.save(spittle);
}
```

I to powinno wystarczyć. Teraz kod statusu dokładnie odpowiada temu, co zaszło podczas obsługi żądania — informuje klienta, że utworzono nowy zasób. I problem został rozwiązany.

Jest jednak jeszcze jedna sprawa. Klient wie, że utworzono nowy zasób, ale czy nie sądzisz, że mogłoby go interesować, *gdzie* on został utworzony? W końcu to nowy zasób, a zatem jest z nim skojarzony jakiś adres URL. Czy klient musi zgadywać, jaki jest adres URL tego nowego zasobu? A może moglibyśmy mu jakoś przekazać tę informację?

Przyjęło się, że w przypadku tworzenia nowych zasobów w dobrym tonie jest przekazywanie klientowi adresu URL tego zasobu przy użyciu nagłówka odpowiedzi `Location`. Oznacza to, że niezbędny jest jakiś sposób określania tych nagłówków odpowiedzi. A w tym może nam pomóc nasz stary znajmy — obiekt `ResponseEntity`.

Listing 16.4 przedstawia nową wersję metody `saveSpittle()`, która zwraca obiekt `ResponseEntity`, by przekazać klientowi informację o adresie utworzonego zasobu.

Listing 16.4. Określanie nagłówków odpowiedzi przy użyciu ResponseEntity

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
public ResponseEntity<Spittle> saveSpittle(
                      @RequestBody Spittle spittle) {
```

```
  Spittle spittle = spittleRepository.save(spittle);  <-- Pobranie obiektu Spittle

  HttpHeaders headers = new HttpHeaders();  <-- Ustawienie nagłówka odpowiedzi Location
  URI locationUri = URI.create(
      "http://localhost:8080/spittr/spittles/" + spittle.getId());
  headers.setLocation(locationUri);

  ResponseEntity<Spittle> responseEntity =  <-- Utworzenie obiektu ResponseEntity
        new ResponseEntity<Spittle>(
          spittle, headers, HttpStatus.CREATED);
  return responseEntity;
}
```

W tej nowej wersji metody tworzony jest obiekt `HttpHeaders`, a w nim zapisywane są wartości nagłówków HTTP, które chcemy umieścić w odpowiedzi. `HttpHeaders` jest specjalną implementacją klasy `MultiValueMap<String, String>`, wyposażoną w kilka wygodnych metod ustawiających (takich jak `setLocation()`), ułatwiających określanie wartości nagłówków HTTP. Po określeniu adresu URL nowego zasobu `Spittle` nagłówki HTTP zostają użyte do utworzenia obiektu `ResponseEntity`.

O rany! Prosta metoda `savaSpittle()` nagle „przybrała nieco na wadze". Znacznie bardziej problematyczne jest jednak to, że wyznacza ona wartość nagłówka `Location`, korzystając z wartości podanych na stałe w kodzie. Największe obawy wzbudzają określenie serwera (`localhost`) oraz numer portu (`8080`), gdyż wartości te na pewno nie będą się nadawały do użycia, jeśli aplikacja zostanie wdrożona gdziekolwiek indziej niż na naszym lokalnym komputerze.

Zamiast tworzyć adres URI ręcznie, możemy wykorzystać udostępnione przez Springa narzędzie, które może nam w tym pomóc. Jest nim klasa `UriComponentsBuilder`. Pozwala ona na tworzenie obiektu `UriComponents` poprzez niezależne określanie poszczególnych fragmentów URI (takich jak adres komputera, numer portu, ścieżka i łańcuch zapytania). Na podstawie obiektu `UriComponents` utworzonego przez obiekt `UriComponents` ↪`Builder` możemy pobrać adres URI zasobu i zapisać go w nagłówku `Location`.

Aby skorzystać z możliwości obiektu `UriComponentsBuilder`, należy poprosić o niego, dodając odpowiedni parametr do metody obsługi, jak pokazano na listingu 16.5.

Listing 16.5. Użycie UriComponentsBuilder do tworzenia adresu URI zasobu

```
@RequestMapping(
    method=RequestMethod.POST
    consumes="application/json")
public ResponseEntity<Spittle> saveSpittle(
      @RequestBody Spittle spittle,
      UriComponentsBuilder ucb) {  <-- Używając tego obiektu UriComponentsBuilder...

  Spittle spittle = spittleRepository.save(spittle);

  HttpHeaders headers = new HttpHeaders();  <-- ...możemy utworzyć adres URI zasobu
  URI locationUri =
    ucb.path("/spittles/")
      .path(String.valueOf(spittle.getId()))
      .build()
      .toUri();
```

```
  headers.setLocation(locationUri);

  ResponseEntity<Spittle> responseEntity =
      new ResponseEntity<Spittle>(
          spittle, headers, HttpStatus.CREATED);

  return responseEntity;
}
```

Obiekt `UriComponentsBuilder` przekazywany do metody obsługi jest wstępnie konfigurowany przy użyciu znanych informacji, takich jak nazwa komputera, numer portu i zawartość serwletu. Informacje te są pobierane z obsługiwanego żądania. Do określenia całego adresu obiekt `UriComponents` wymaga jedynie określenia ścieżki.

Warto zwrócić uwagę, że ścieżka tworzona jest w dwóch etapach. W pierwszym etapie wywoływana jest metoda `path()`, która ustawia ścieżkę na `/spittles/`, czyli ścieżkę bazową obsługiwaną przez kontroler. Następnie, w drugim wywołaniu metody `path()`, zostaje przekazany identyfikator nowego obiektu `Spittle`. Jak się można domyślić, każde wywołanie metody `path()` bazuje na efektach poprzednich wywołań.

Po określeniu kompletnej ścieżki wywoływana jest metoda `build()`, która tworzy obiekt `UriComponents()`. Obiekt ten jest następnie używany do wywołania metody `toUri()`, która zwraca adres URI nowego zasobu `Spittle`.

Udostępnianie zasobów w API modelu REST reprezentuje tylko jedną ze stron konwersacji. W końcu publikowanie API nie ma większego sensu, jeśli nie będzie nikogo, kto chciałby go używać. Najczęściej klientami API modelu REST są aplikacje mobilne oraz aplikacje pisane w języku JavaScript; nic jednak nie stoi na przeszkodzie, by z takich zasobów korzystały także aplikacje Spring. A zatem zmieńmy bieg i przekonajmy się, jak można pisać kod Spring działający jako klient konwersacji w modelu REST.

16.4. Konsumowanie zasobów REST

Tworzenie kodu, który komunikuje się z zasobem REST jako klient, może być nieco żmudne, a sam kod — wtórny. Załóżmy na przykład, że musimy napisać metodę, która korzysta z Graph API Facebooka, by pobierać czyjś profil. Jednak kod niezbędny do pobrania danych profilu jest trochę bardziej złożony, co pokazano na listingu 16.6.

Listing 16.6. Pobieranie profilu Facebooka przy użyciu Apache HTTP Client

```
public Profile fetchFacebookProfile(String id) {
  try {
    HttpClient client = HttpClients.createDefault(); <-- Tworzymy klienta

    HttpGet request = new HttpGet("http://graph.facebook.com/" + id); <-- Tworzymy żądanie
    request.setHeader("Accept", "application/json");

    HttpResponse response = client.execute(request); <-- Wykonujemy żądanie

    HttpEntity entity = response.getEntity();
    ObjectMapper mapper = new ObjectMapper(); <-- Odwzorowujemy odpowiedź na obiekt
    return mapper.readValue(entity.getContent(), Profile.class);
  } catch (IOException e) {
```

```
      throw new RuntimeException(e);
    }
  }
```

Jak widać, na proces konsumpcji zasobu REST składa się wiele elementów. Tutaj ułatwiłem sobie jeszcze dodatkowo zadanie, używając klienta HTTP fundacji Apache do wykonania żądania oraz procesora Jackson JSON do przetworzenia odpowiedzi.

Przyglądając się bliżej metodzie `fetchFacebookProfile()`, zauważysz, że tylko niewielka jej część odpowiada za pobranie danych profilu Facebooka. Znaczna część jej kodu jest wtórna. Wygląd metody konsumującej inny zasób REST byłby prawdopodobnie bardzo zbliżony do wyglądu metody `fetchFacebookProfile()`.

Co więcej, w kilku miejscach może zostać zgłoszony wyjątek `IOException`. Ponieważ `IOException` jest wyjątkiem kontrolowanym, trzeba go albo przechwycić, albo zgłosić. W tym przypadku zdecydowałem się na przechwycenie wyjątku i zgłoszenie niekontrolowanego `RuntimeException` w jego miejsce.

Przy takich ilościach powtarzalnego kodu potrzebnych do konsumpcji zasobu najlepszym wyjściem byłaby jakaś forma enkapsulacji kodu i parametryzacja jego wersji. Tu z pomocą przychodzi nam szablon Springa `RestTemplate`. Podobnie jak szablon `Jdbc➥Template`, który zajmuje się za nas najmniej przyjemną częścią pracy z dostępem do danych JDBC, `RestTemplate` uwalnia nas od monotonii przy konsumpcji zasobów REST.

Już za chwilę zobaczymy, jak można zmodyfikować metodę `fetchFacebookProfile()` przy użyciu `RestTemplate`, upraszczając ją i pozbywając się niepotrzebnego kodu. Najpierw jednak zbadajmy dostępne dzięki `RestTemplate` operacje REST.

16.4.1. Operacje szablonu RestTemplate

Klasa `RestTemplate` definiuje 36 metod służących do interakcji z zasobami REST, a większość z nich odpowiada metodom HTTP. W tej książce nie mam na tyle miejsca, by opisać wszystkie 36 metod tej klasy, lecz na szczęście okazuje się, że istnieje tylko 11 unikalnych operacji. Dziesięć z nich jest dostępnych w trzech przeciążonych wersjach, natomiast jedenasta — aż w sześciu. Wszystko to daje w sumie 36 metod. W tabeli 16.2 opisano 11 unikalnych operacji oferowanych przez szablon `RestTemplate`.

`RestTemplate` zawiera wszystkie czasowniki HTTP z wyjątkiem `TRACE`. Dodatkowo, `execute()` i `exchange()` są metodami ogólnego przeznaczenia na niższym poziomie, które mogą być użyte z dowolnymi metodami HTTP.

Większość operacji z tabeli 16.2 może zostać przeciążona i przybrać jedną z trzech form:

- Taką, która przyjmuje klasę `java.net.URI` jako specyfikację URL bez obsługi sparametryzowanych adresów URL.
- Taką, która przyjmuje `String` jako specyfikację URL z parametrami określonymi jako obiekt `Map`.
- Taką, która przyjmuje `String` jako specyfikację URL z parametrami określonymi jako zmienna lista argumentów.

Poznanie 11 operacji dostarczanych przez `RestTemplate` i działania ich poszczególnych wariantów jest podstawą do tworzenia aplikacji klienckich konsumujących zasoby REST.

Tabela 16.2. RestTemplate definiuje 11 unikalnych operacji, z których każda może zostać przeciążona, co daje łącznie 36 metod

Metoda	Opis
`delete()`	Wykonuje żądanie HTTP DELETE pod określonym adresem URL.
`exchange()`	Wykonuje określoną metodę HTTP pod adresem URL, zwracając obiekt `ResponseEntity`, zawierający obiekt odwzorowany z treści odpowiedzi.
`execute()`	Wykonuje określoną metodę HTTP pod adresem URL, zwracając obiekt odwzorowany z treści odpowiedzi.
`getForEntity()`	Wysyła żądanie HTTP `GET`, zwracając obiekt `ResponseEntity`, zawierający treść odpowiedzi odwzorowaną na obiekt.
`getForObject()`	Wysyła żądanie HTTP `GET`, zwracając treść odpowiedzi odwzorowaną na obiekt.
`headForHeaders()`	Wysyła żądanie HTTP `HEAD`, zwracając nagłówki HTTP dla określonego zasobu URL.
`optionsForAllow()`	Wysyła żądanie HTTP `OPTIONS`, zwracając nagłówek `Allow` dla określonego adresu URL.
`postForEntity()`	Wysyła żądanie HTTP `POST` na adres `URL`, zwracając obiekt `ResponseEntity`, zawierający obiekt odwzorowany z treści odpowiedzi.
`postForLocation()`	Wysyła żądanie HTTP `POST` na adres `URL`, zwracając adres URL nowego zasobu.
`postForObject()`	Wysyła żądanie HTTP `POST` na adres `URL`, zwracając treść odpowiedzi odwzorowaną na obiekt.
`put()`	Wysyła na podany adres URL żądanie HTTP `PUT`, zawierające dane zasobu.

Przeanalizujemy teraz operacje `RestTemplate`, które obsługują cztery podstawowe metody HTTP: `GET`, `PUT`, `DELETE` i `POST`. Zaczniemy od metod `GET`: `getForObject()` i `getFor`
`↪Entity()`.

16.4.2. Pobieranie zasobów za pomocą GET

Być może zauważyłeś, że w tabeli 16.2 znalazły się dwa rodzaje metod wykonujących żądanie `GET`: `getForObject()` i `getForEntity()`. Jak już wspomnieliśmy, każda z tych metod jest przeciążana i może przyjąć trzy formy. Sygnatury trzech wariantów metody `getFor`
`↪Object()` wyglądają tak:

```
<T> T getForObject(URI url, Class<T> responseType) throws RestClientException;

<T> T getForObject(String url, Class<T> responseType,
  Object... uriVariables) throws RestClientException;

<T> T getForObject(String url, Class<T> responseType,
  Map<String, ?> uriVariables) throws RestClientException;
```

Analogiczne sygnatury metod `getForEntity()` przedstawiają się następująco:

```
<T> ResponseEntity<T> getForEntity(URI url, Class<T> responseType)
  throws RestClientException;

<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
  Object... uriVariables) throws RestClientException;

<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
  Map<String, ?> uriVariables) throws RestClientException;
```

Z wyjątkiem zwracanych typów, metody `getForObject()` są lustrzanym odbiciem metod `getForEntity()`. I w rzeczy samej, ich działanie jest bardzo podobne. Obie wykonują żądanie `GET`, pobierając zasób na podstawie adresu URL. Obie odwzorowują także zasób na typ określony w parametrze `responseType`. Jedyna różnica polega na tym, że `getFor`
`↪Object()` zwraca po prostu obiekt żądanego typu, podczas gdy `getForEntity()`zwraca ten obiekt wraz z dodatkową informacją o odpowiedzi.

W pierwszej kolejności przyjrzymy się prostszej metodzie `getForObject()`. Później zobaczymy, jak można uzyskać więcej informacji z odpowiedzi `GET`, używając metody `getForEntity()`.

16.4.3. Pobieranie zasobów

Działanie pobierającej zasób metody `getForObject()` jest bardzo konkretne. Żądamy zasobu i otrzymujemy ten zasób odwzorowany na wybrany typ Javy. Jako próbkę możliwości `getForObject()` rozważmy kolejną próbę implementacji `fetchFacebookProfile()`:

```
public Profile fetchFacebookProfile(String id) {
  RestTemplate rest = new RestTemplate();
  return rest.getForObject("http://graph.facebook.com/{spitter}",
      Profile.class, id);
}
```

Na listingu 16.6 metoda `fetchFacebookProfile()` składała się z więcej niż 10 wierszy kodu. Za pomocą `RestTemplate` udało nam się zredukować jej rozmiar do zalewie kilku wierszy (mogło ich nawet być jeszcze mniej, gdyby nie konieczność dzielenia długich wierszy, które nie mieszczą się na stronie).

`fetchFacebookProfile()` zaczyna od skonstruowania instancji `RestTemplate` (alternatywna implementacja mogłaby użyć zamiast niej wstrzykniętej instancji). Następnie wywołuje metodę `getForObject()`, aby pobrać profil portalu Facebook. Oczekuje przy tym wyników w postaci obiektu typu `Profile`. Gdy tylko uzyska ten obiekt, przekazuje go wywołującemu.

Zauważ, że w nowej wersji `fetchFacebookProfile()` do wygenerowania adresu URL nie stosujemy konkatenacji łańcuchów. Zamiast tego wykorzystujemy fakt, że `Rest`
`↪Template` akceptuje sparametryzowane adresy URL. Pod symbol zastępczy `{id}` w adresie URL zostanie podstawiony parametr `id` metody. Ostatnim argumentem `getFor`
`↪Object()` jest zmienna lista argumentów, z których każdy wstawiany jest w kolejności występowania w miejsce symbolu zastępczego do określonego adresu URL.

Ewentualnie moglibyśmy umieścić parametr `id` w obiekcie `Map` pod kluczem `id` i przekazać obiekt `Map` jako ostatni parametr metody `getForObject()`:

```
public Spittle[] fetchFacebookProfile(String id) {
  Map<String, String> urlVariables = new HashMap<String, String();
  urlVariables.put("id", id);
  RestTemplate rest = new RestTemplate();
  return rest.getForObject("http://graph.facebook.com/{spitter}",
      Profile.class, urlVariables);
}
```

Brakuje tu tylko jakiegokolwiek parsowania JSON oraz odwzorowania obiektu. Metoda `getForObject()` dokonuje za nas po cichu konwersji treści odpowiedzi na obiekt.

Wykorzystuje do tego ten sam zbiór konwerterów komunikatów HTTP z tabeli 16.1, który używany jest przez Spring MVC przy metodach obsługi oznaczonych adnotacją `@Res ➥ponseBody`.

W kodzie tej metody nie ma również przechwytywania wyjątków. Nie dlatego, że `getForObject()` nie może zgłosić wyjątku, ale ponieważ każdy zgłaszany przez nią wyjątek jest niekontrolowany. Jeżeli coś pójdzie nie tak w metodzie `getForObject()`, zgłoszony zostanie wyjątek `RestClientException` (lub wyjątek którejś z podklas). Możesz go przechwycić, jeśli chcesz — ale kompilator tego nie wymaga.

16.4.4. Odczyt metadanych z odpowiedzi

Jako alternatywę dla `getForObject()`, `RestTemplate` oferuje metodę `getForEntity()`. Jej działanie nie różni się bardzo od działania metod `getForObject()`. Ale podczas gdy `getForObject()` zwraca tylko zasób (przekształcony w obiekt Javy przez konwerter komunikatów HTTP), `getForEntity()` zwraca ten sam obiekt, umieszczony w obiekcie `Respon ➥seEntity`. `ResponseEntity` zawiera także dodatkowe informacje o odpowiedzi, takie jak kod HTTP i nagłówki odpowiedzi.

Jedną z rzeczy, do których możesz wykorzystać obiekt `ResponseEntity`, jest odczytanie wartości jednego z nagłówków odpowiedzi. Załóżmy na przykład, że obok samego zasobu potrzebujesz jeszcze informacji o tym, kiedy został on zmodyfikowany. Zakładając, że serwer dostarcza taką informację w nagłówku `Last-Modified`, możesz użyć metody `getHeaders()` w następujący sposób:

```
Date lastModified = new Date(response.getHeaders().getLastModified());
```

Metoda `getHeaders()` zwraca obiekt `HttpHeaders`, który udostępnia szereg metod złożonych, do odczytu nagłówków odpowiedzi, w tym metodę `getLastModified()`, zwracającą liczbę milisekund, która upłynęła od 1 stycznia 1970.

Oprócz `getLastModified()` obiekt `HttpHeaders` zawiera następujące metody, umożliwiające odczyt informacji nagłówka:

```
public List<MediaType> getAccept() { ... }
public List<Charset> getAcceptCharset() { ... }
public Set<HttpMethod> getAllow() { ... }
public String getCacheControl() { ... }
public List<String> getConnection() { ... }
public long getContentLength() { ... }
public MediaType getContentType() { ... }
public long getDate() { ... }
public String getETag() { ... }
public long getExpires() { ... }
public long getIfNotModifiedSince() { ... }
public List<String> getIfNoneMatch() { ... }
public long getLastModified() { ... }
public URI getLocation() { ... }
public String getOrigin() { ... }
public String getPragma() { ... }
public String getUpgrade() { ... }
```

Obiekt `HttpHeaders` udostępnia także bardziej uniwersalne metody dostępu do nagłówka HTTP, w tym metodę `get()` i metodę `getFirst()`. Obie przyjmują argument typu `String`,

identyfikujący klucz wybranego nagłówka. Metoda `get()` zwraca listę wartości typu `String`, po jednej na każdą wartość nagłówka. Metoda `getFirst()` natomiast zwraca tylko pierwszą wartość nagłówka.

Jeżeli interesuje Cię kod odpowiedzi HTTP, możesz użyć metody `getStatusCode()`. Jako przykład wykorzystamy zamieszczoną poniżej implementację metody, która pobiera obiekt `Spittle`:

```
public Spittle fetchSpittle(long id) {
  RestTemplate rest = new RestTemplate();
  ResponseEntity<Spittle> response = rest.getForEntity(
      "http://localhost:8080/spittr-api/spittles/{id}",
      Spittle.class, id);
  if(response.getStatusCode() == HttpStatus.NOT_MODIFIED) {
    throw new NotModifiedException();
  }
  return response.getBody();
}
```

Jeśli serwer odpowie z kodem 304, zawartość na serwerze nie została zmodyfikowana od czasu jej ostatniego żądania przez klienta. W takim przypadku zgłaszany jest niestandardowy wyjątek `NotModifiedException`, informujący klienta, że dane zasobu powinien odczytać ze swojej pamięci podręcznej.

16.4.5. Umieszczanie zasobów na serwerze za pomocą PUT

Szablon `RestTemplate` definiuje zbiór metod `put()` do wykonania operacji `PUT` na zasobie. Podobnie jak w przypadku pozostałych metod `RestTemplate`, istnieją trzy warianty metody `put()`:

```
void put(URI url, Object request) throws RestClientException;
void put(String url, Object request, Object... uriVariables)
        throws RestClientException;
void put(String url, Object request, Map<String, ?> uriVariables)
        throws RestClientException;
```

W najprostszej wersji metoda `put()` przyjmuje jako argument wartość `java.net.URI`, która identyfikuje (i lokalizuje) wysyłany na serwer zasób, oraz obiekt będący reprezentacją Javy tego zasobu.

Wersja metody `put()` bazująca na obiekcie `URI` mogłaby zostać użyta do uaktualnienia zasobu `Spitter` na serwerze, na przykład tak:

```
public void updateSpittle(Spittle spittle) throws SpitterException {
  RestTemplate rest = new RestTemplate();
  String url = "http://localhost:8080/spittr-api/spittles/"
               + spittle.getId();
  rest.put(URI.create(url), spittle);
}
```

Chociaż sygnatura metody była dosyć prosta, użycie `java.net.URI` pociąga za sobą pewne konsekwencje. Najpierw, aby utworzyć adres URL dla uaktualnianego obiektu `Spittle`, musimy połączyć wartości `String`.

A jak mogliśmy się już przekonać przy okazji metod `getForObject()` oraz `getFor`
`↪Entity()`, stosowanie jednej z dwóch pozostałych wersji metody `put()`, które bazują na

wartości String, oszczędza nam trudu związanego z tworzeniem obiektu URI. Co więcej, metody te pozwalają nam na określenie szablonu URI i podstawienie doń odpowiednich wartości. Oto zmodyfikowana metoda updateSpittle(), która korzysta z jednej z wersji metody put() bazujących na wartości String:

```
public void updateSpittle(Spittle spittle) throws SpitterException {
  RestTemplate rest = new RestTemplate();
  rest.put("http://localhost:8080/spittr-api/spittles/{id}",
          spittle, spittle.getId());
}
```

Adres URI ma tutaj formę prostego szablonu String. Kiedy RestTemplate wyśle żądanie PUT, w miejsce {*id*} w szablonie adresu wstawiona zostanie wartość zwrócona przez metodę spittle.getId(). Podobnie jak w przypadku metod getForObject() i getForEnti
➥ty(), ostatni argument tej wersji metody put() jest listą argumentów o zmiennej długości. Każdy z nich przypisywany jest do symboli zastępczych, w kolejności występowania.

Zmienne szablonu można również przekazać w formie obiektu Map:

```
public void updateSpittle(Spittle spittle) throws SpitterException {
  RestTemplate rest = new RestTemplate();
  Map<String, String> params = new HashMap<String, String>();
  params.put("id", spittle.getId());
  rest.put("http://localhost:8080/spittr-api/spittles/{id}",
            spittle, params);
}
```

Gdy korzystamy z obiektu Map do przekazania zmiennych szablonu, każdy klucz obiektu Map odpowiada zmiennej wyrażonej symbolem zastępczym w szablonie URI o tej samej nazwie.

We wszystkich wersjach metody put() drugim argumentem jest obiekt Javy reprezentujący zasób umieszczany na serwerze za pomocą metody PUT. W naszym przykładzie jest to obiekt Spittle. RestTemplate użyje jednego z konwerterów komunikatów z tabeli 16.1, aby przekształcić obiekt Spittle w reprezentację, która zostanie wysłana na serwer w treści żądania.

Typ zawartości konwertowanego obiektu zależy w dużej mierze od typu przekazywanego do metody put(). Jeśli jest to wartość String, użyty zostanie konwerter String
➥HttpMessageConverter: wartość zostanie zapisana bezpośrednio w treści żądania, a typ zawartości ustawiony na text/plain. Jeśli natomiast będzie to mapa MultiValueMap
➥<String,String>, poszczególne wartości mapy zostaną zapisane w treści żądania w formie application/x-www-form-urlencoded przez konwerter FormHttpMessageConverter.

Ponieważ my przekazujemy obiekt typu Spittle, potrzebujemy konwertera, który będzie pracował z dowolnymi obiektami. Jeżeli w ścieżce do klas znajduje się biblioteka Jackson 2, konwerter MappingJackson2HttpMessageConverter zapisze obiekt Spittle do żądania jako application/json.

16.4.6. Usuwanie zasobów za pomocą DELETE

Kiedy chcesz, żeby zasób nie był już przechowywany na serwerze, użyjesz jednej z metod `delete()` szablonu `RestTemplate`. Podobnie jak metodę `put()`, metodę `delete()` również cechuje prostota. Metoda ta występuje tylko w trzech wersjach, których sygnatury wyglądają następująco:

```
void delete(String url, Object... uriVariables) throws RestClientException;
void delete(String url, Map<String, ?> uriVariables) throws RestClientException;
void delete(URI url) throws RestClientException;
```

Pod względem prostoty metody `delete()` wygrywają bez trudu z pozostałymi metodami `RestTemplate`. Wystarczy im tylko przekazać adres URI usuwanego zasobu. Aby pozbyć się obiektu `Spittle` o danym `id`, można na przykład wywołać metodę `delete()` w następujący sposób:

```
public void deleteSpittle(long id) {
  RestTemplate rest = new RestTemplate();
  rest.delete(
      URI.create("http://localhost:8080/spittr-api/spittles/" + id));
}
```

Jest to w miarę proste, ale po raz kolejny polegamy na łączeniu wartości `String`, aby utworzyć obiekt `URI`. Możemy sobie ułatwić zadanie, używając jednej z prostszych wersji metody `delete()`:

```
public void deleteSpittle(long id) {
  RestTemplate rest = new RestTemplate();
  rest.delete("http://localhost:8080/spittr-api/spittles/{id}", id));
}
```

Tak zdecydowanie lepiej. Czyż nie?

Przegląd najprostszych metod szablonu `RestTemplate` mamy już za sobą. Przejdziemy teraz do najbardziej zróżnicowanego zbioru metod `RestTemplate` — tych, które obsługują żądania HTTP `POST`.

16.4.7. Wysyłanie danych zasobu za pomocą POST

Spoglądając na tabelę 16.2, zauważysz, że szablon `RestTemplate` definiuje trzy rodzaje metod umożliwiających wysyłanie żądań `POST`. Jeśli pomnożymy tę liczbę przez trzy warianty przeciążenia metody, otrzymamy łącznie dziewięć metod do wysyłania danych na serwer za pomocą `POST`.

Nazwy dwóch z tych metod wyglądają znajomo. Metody `postForObject()` i `postFor`
`↪Entity()` spełniają taką samą rolę przy żądaniach `POST`, co metody `getForObject()` i `getForEntity()` przy żądaniach `GET`. Ostatnia metoda, `postForLocation()`, jest unikalna dla żądań `POST`.

16.4.8. Odbieranie obiektów odpowiedzi z żądań POST

Powiedzmy, że chcesz za pomocą `RestTemplate` wysłać nowy obiekt `Spitter` do REST API aplikacji Spitter. Ponieważ jest to zupełnie nowy obiekt `Spitter`, serwer nic jeszcze o nim nie wie. Oficjalnie nie jest jeszcze zatem zasobem REST i nie posiada adresu

URL. Ponadto klient nie zna identyfikatora obiektu `Spitter`, dopóki nie zostanie on utworzony na serwerze.

Jedną z metod umieszczania zasobu na serwerze za pomocą `POST` jest użycie metody `postForObject()` szablonu `RestTemplate`. Trzy warianty `postForObject()` mają takie oto sygnatury:

```
<T> T postForObject(URI url, Object request, Class<T> responseType)
        throws RestClientException;

<T> T postForObject(String url, Object request, Class<T> responseType,
        Object... uriVariables) throws RestClientException;

<T> T postForObject(String url, Object request, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

We wszystkich przypadkach pierwszy parametr jest adresem URL, na który zasób powinien zostać wysłany, drugi parametr jest wysyłanym obiektem, a trzeci — typem Javy, który powinien zostać zwrócony. W przypadku dwóch wersji, które przyjmują adres URL w postaci `String`, czwarty parametr identyfikuje zmienne URL (w formie zmiennej listy argumentów lub obiektu `Map`).

Nowe zasoby `Spitter` powinny być wysyłane przy użyciu metody `POST` na adres *http://localhost:8080/Spitter/spitters*, pod którym metoda obsługi kontrolera czeka, gotowa zapisać obiekt. Ponieważ ten adres URL nie wymaga żadnych zmiennych URL, możemy użyć dowolnej wersji `postForObject()`. Ale starając się zachować prostotę, zrobimy to w następujący sposób:

```
public Spitter postSpitterForObject(Spitter spitter) {
  RestTemplate rest = new RestTemplate();
  return rest.postForObject("http://localhost:8080/spittr-api/spitters",
          spitter, Spitter.class);
}
```

Metodzie `postSpitterForObject()` przekazywany jest nowo utworzony obiekt `Spitter`, który wysyłany jest następnie na serwer za pomocą `postForObject()`. W odpowiedzi otrzymuje ona obiekt `Spitter` i zwraca go wywołującemu.

Podobnie jak to miało miejsce w przypadku metod `getForObject()`, możemy potrzebować jakiejś części metadanych, które wracają z żądaniem. W takim przypadku najlepszym wyborem będzie metoda `postForEntity()`. `postForEntity()` posiada zbiór sygnatur, bliźniaczo podobnych do sygnatur `postForObject()`:

```
<T> ResponseEntity<T> postForEntity(URI url, Object request,
        Class<T> responseType) throws RestClientException;

<T> ResponseEntity<T> postForEntity(String url, Object request,
        Class<T> responseType, Object... uriVariables)
        throws RestClientException;

<T> ResponseEntity<T> postForEntity(String url, Object request,
        Class<T> responseType, Map<String, ?> uriVariables)
        throws RestClientException;
```

Powiedzmy zatem, że oprócz zasobu `Spitter` chcesz także zobaczyć wartość nagłówka `Location` w odpowiedzi. Możesz wtedy wywołać metodę `postForEntity()` w taki oto sposób:

```
RestTemplate rest = new RestTemplate();
ResponseEntity<Spitter> response = rest.postForEntity(
    "http://localhost:8080/spittr-api/spitters",
    spitter, Spitter.class);
Spitter spitter = response.getBody();
URI url = response.getHeaders().getLocation();
```

Podobnie jak `getForEntity()`, metoda `postForEntity()` również zwraca obiekt `ResponseEntity<T>`. Wywołując metodę `getBody()` tego obiektu, otrzymasz obiekt zasobu (w tym przypadku `Spitter`). Metoda `getHeaders()` pozwala natomiast na uzyskanie obiektu `HttpHeaders`, który umożliwia dostęp do różnych nagłówków HTTP, zwróconych w odpowiedzi. W naszym przykładzie wywołujemy `getLocation()`, aby pobrać nagłówek `Location` w postaci `java.net.URI`.

16.4.9. Pobranie informacji o lokalizacji po żądaniu POST

Metoda `postForEntity()` przydaje się zarówno do pobrania wysłanego zasobu, jak i nagłówków odpowiedzi. Często jednak nie potrzebujesz odsyłania zasobu (w końcu to Ty go wysyłasz na serwer). Jeżeli wartość nagłówka `Location` jest jedyną informacją, której potrzebujesz, łatwiej będzie użyć metody `postForLocation()` szablonu `RestTemplate`.

Podobnie jak pozostałe metody `POST`, `postForLocation()` wysyła zasób na serwer w treści żądania `POST`. Ale zamiast przesłania tego samego obiektu zasobu w odpowiedzi, `postForLocation()` odpowiada tylko lokalizacją nowo utworzonego zasobu. Metoda ta posiada trzy sygnatury:

```
URI postForLocation(String url, Object request, Object... uriVariables)
        throws RestClientException;

URI postForLocation(
        String url, Object request, Map<String, ?> uriVariables)
        throws RestClientException;

URI postForLocation(URI url, Object request) throws RestClientException;
```

Aby zademonstrować działanie metody `postForLocation()`, spróbujmy wysłać obiekt `Spitter` ponownie. Tym razem w odpowiedzi chcemy uzyskać adres URL zasobu:

```
public String postSpitter(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForLocation(
              "http://localhost:8080/spittr-api/spitters",
              spitter).toString();
}
```

Przekazujemy tu docelowy adres URL jako `String` wraz z obiektem `Spitter`, który ma zostać wysłany (tym razem nie ma zmiennych URL). Jeżeli po utworzeniu zasobu serwer odpowie adresem URL nowego zasobu w nagłówku `Location` odpowiedzi, metoda `postForLocation()` zwróci ten URL jako `String`.

16.4.10. Wymiana zasobów

Jak dotychczas omówiliśmy szereg metod szablonu RestTemplate, związanych z wykonywaniem żądań GET, PUT, DELETE i POST na zasobach. Wśród nich dwie szczególne metody, getForEntity() i postForEntity(), które zwracały wynikowy zasób opakowany w obiekt RequestEntity, umożliwiający pobranie nagłówków i kodów odpowiedzi.

Możliwość odczytania nagłówków żądania bardzo się przydaje. Ale co zrobić, gdy chcemy ustawić nagłówki żądania wysyłanego na serwer? W takim przypadku pomocne okażą się metody exchange() szablonu RestTemplate.

Tak jak pozostałe metody RestTemplate, metoda exchange() posiada trzy przeciążone sygnatury. Pierwsza z nich pobiera obiekt java.net.URI określający docelowy adres URL, natomiast dwie pozostałe pobierają łańcuchy zawierające zmienne odpowiadające fragmentom adresu URL:

```
<T> ResponseEntity<T> exchange(URI url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType)
        throws RestClientException;

<T> ResponseEntity<T> exchange(String url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType,
        Object... uriVariables) throws RestClientException;

<T> ResponseEntity<T> exchange(String url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

Metoda exchange() przyjmuje również parametr typu HttpMethod, wskazując odpowiedni czasownik HTTP. W zależności od wartości nadanej temu parametrowi metoda exchange() potrafi wykonać te same zadania, co pozostałe metody RestTemplate.

Przykładem jednego ze sposobów pobrania zasobu Spitter z serwera jest następujące użycie metody getForEntity() szablonu RestTemplate:

```
ResponseEntity<Spitter> response = rest.getForEntity(
          "http://localhost:8080/spittr-api/spitters/{spitter}",
          Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

To samo można osiągnąć również za pomocą metody exchange(), jak pokazano w kolejnym przykładzie:

```
ResponseEntity<Spitter> response = rest.exchange(
          "http://localhost:8080/spittr-api/spitters/{spitter}",
          HttpMethod.GET, null, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

Przekazując HttpMethod.GET jako czasownik HTTP, każemy metodzie exchange() wysłać żądanie GET. Trzeci argument służy do wysyłania zasobu w żądaniu, ale ponieważ jest to żądanie GET, można mu nadać wartość null. Kolejny argument wskazuje, że chcemy, aby odpowiedź przybrała formę obiektu typu Spitter. Ostatnim argumentem jest wartość, która powinna zostać wstawiona w miejsce symbolu zastępczego {*spitter*} w określonym szablonie URL.

Użyta w ten sposób metoda exchange() niewiele różni się od getForEntity() z wcześniejszego przykładu. Ale w przeciwieństwie do getForEntity() i getForObject() metoda

`exchange()` pozwala nam na ustawienie nagłówków wysyłanego żądania. Zamiast więc przekazywać do niej wartość `null`, przekażemy obiekt `HttpEntity`, zawierający odpowiednie nagłówki.

Bez określenia nagłówków metoda `exchange()` wyśle żądanie `GET` obiektu `Spitter` z następującymi nagłówkami:

```
GET /Spitter/spitters/habuma HTTP/1.1
Accept: application/xml, text/xml, application/*+xml, application/json
Content-Length: 0
User-Agent: Java/1.6.0_20
Host: localhost:8080
Connection: keep-alive
```

Zwróć uwagę na nagłówek `Accept`. Wymienia on akceptowane typy zawartości (kilka typów XML i `application/json`). Pozostawia to serwerowi dość dużą swobodę co do formatu odsyłanego zasobu. Załóżmy, że chcemy, aby serwer wysłał odpowiedź w formacie JSON. W takim przypadku nagłówek `Accept` powinien zawierać wyłącznie wartość `application/json`.

Do ustawienia nagłówków żądania wystarczy skonstruować obiekt `HttpEntity`, który zostanie przekazany do metody `exchange()`. W konstruktorze należy podać wartość `MultiValueMap` z załadowanymi odpowiednimi nagłówkami:

```
MultiValueMap<String, String> headers =
    new LinkedMultiValueMap<String, String>();
headers.add("Accept", "application/json");
HttpEntity<Object> requestEntity = new HttpEntity<Object>(headers);
```

Utworzyliśmy tu wartość `LinkedMultiValueMap` i dodaliśmy nagłówek `Accept` o wartości `application/json`. Następnie skonstruowaliśmy obiekt `HttpEntity` (uogólnionego typu `Object`), przekazując wartość `MultiValueMap` jako argument konstruktora. Gdybyśmy mieli do czynienia z żądaniem `PUT` lub `POST`, przekazalibyśmy `HttpEntity` także obiekt do przesłania w treści żądania — przy żądaniu `GET` nie ma takiej potrzeby.

Możemy teraz wywołać metodę `exchange()`, przekazując nasz obiekt `HttpEntity`:

```
ResponseEntity<Spitter> response = rest.exchange(
          "http://localhost:8080/spittr-api/spitters/{spitter}",
          HttpMethod.GET, requestEntity, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

Na pierwszy rzut oka wyniki są identyczne. Powinniśmy otrzymać obiekt `Spitter`, o który prosiliśmy. Zaglądając pod maskę żądania, zobaczymy jednak, że zostało ono wysłane z następującymi nagłówkami:

```
GET /Spitter/spitters/habuma HTTP/1.1
Accept: application/json
Content-Length: 0
User-Agent: Java/1.6.0_20
Host: localhost:8080
Connection: keep-alive
```

Zakładając, że serwer potrafi dokonać serializacji odpowiedzi typu `Spitter` do formatu JSON, treść odpowiedzi powinna mieć natomiast format JSON.

16.5. Podsumowanie

Architektura REST wykorzystuje standardy sieciowe do integracji aplikacji, zapewniając prostą i naturalną komunikację. Zasoby w systemie są identyfikowane za pomocą adresów URL, modyfikowane za pomocą metod HTTP i reprezentowane w jednej lub kilku właściwych dla klienta formach.

W tym rozdziale dowiedzieliśmy się, jak tworzyć kontrolery Spring MVC, które odpowiadają na żądania modyfikacji zasobów REST. Używając sparametryzowanych wzorców URL i kojarzenia metod obsługi kontrolera z konkretnymi metodami HTTP, umożliwiamy kontrolerom odpowiedź na żądania `GET`, `POST`, `PUT` i `DELETE` dotyczące zasobów aplikacji.

W odpowiedzi na te żądania Spring potrafi reprezentować dane tych zasobów w formacie najlepszym dla klienta. Dla odpowiedzi bazujących na widokach `ContentNe` ↪`gotiatingViewResolver`, współpracując z innymi producentami widoków, potrafi wybrać najodpowiedniejszy pod względem typu zawartości dla klienta widok. Alternatywnie metoda obsługi kontrolera może zostać oznaczona adnotacją `@ResponseBody`, co spowoduje pominięcie generacji widoku i wykorzystanie jednego z wielu konwerterów komunikatów do przekształcenia zwracanej wartości na odpowiedź dla klienta.

API modelu REST udostępniają możliwości funkcjonalne aplikacji, dzięki czemu mogą z nich korzystać klienty. Co więcej, sposoby, w jakie to robią, mogą wykraczać poza początkowe oczekiwania projektantów. Bardzo często klientami API modelu REST są aplikacje mobilne bądź aplikacje JavaScript działające w przeglądarkach WWW. Nic jednak nie stoi na przeszkodzie, by przy użyciu szablonów `RestTemaplate` korzystały z nich także inne aplikacje Spring.

Zasoby REST zdefiniowane w tym rozdziale są elementami publicznego API. Oznacza to, że gdybyśmy opublikowali je na serwerze działającym gdzieś w internecie, to nic nie mogłoby powstrzymać innych programistów przed napisaniem klientów, które by z nich korzystały. W następnym rozdziale zaczniemy blokować dostęp do tych zasobów — będzie on bowiem poświęcony sposobom zabezpieczania zasobów REST, tak aby mogły ich używać tylko autoryzowane klienty.

17 Obsługa komunikatów w Springu

W tym rozdziale omówimy:

- Wprowadzenie do asynchronicznej wymiany komunikatów
- Wymianę komunikatów przy użyciu JMS
- Wysyłanie komunikatów przy użyciu Springa i AMQP
- Obiekty POJO sterowane komunikatami

Jest piątek, godzina 16:55. Już tylko minuty dzielą Cię od długo oczekiwanego urlopu. Masz akurat tyle czasu, ile potrzeba, aby dojechać na lotnisko i wsiąść do samolotu. Zanim się jednak spakujesz i wyruszysz, musisz mieć pewność, że Twój szef i koledzy wiedzą, na jakim etapie jest projekt, aby bez problemu mogli kontynuować pracę nad nim w poniedziałek. Niestety, część kolegów urwała się przed weekendem wcześnie, a szef jest na spotkaniu. Co robisz?

Możesz do szefa zadzwonić, ale nie ma sensu przerywać spotkania z powodu zwykłego raportu o stanie projektu. Możesz też spróbować poczekać, aż spotkanie się skończy, nikt jednak nie wie, ile potrwa, a samolot z pewnością nie będzie czekał. A może przykleić mu karteczkę do monitora? Tuż obok 100 innych, które już tam są…

Okazuje się, że najpraktyczniejszym sposobem na poinformowanie szefa o stanie pracy i niespóźnienie się przy tym na samolot będzie krótka wiadomości e-mail do szefa i kolegów, zawierająca opis postępów i obietnicę przysłania kartki z wakacji. Nie wiesz, gdzie się teraz znajdują ani kiedy przeczytają wiadomość, ale masz pewność, że prędzej czy później usiądą przy biurku i to zrobią. Tymczasem Ty jesteś już w drodze na lotnisko.

Niektóre sytuacje wymagają kontaktu bezpośredniego. Jeżeli zrobisz sobie krzywdę, do wezwania karetki użyjesz najprawdopodobniej telefonu — raczej nie będziesz kontaktować się ze szpitalem za pomocą poczty elektronicznej. Często jednak wystarczy wysłanie wiadomości. Ta forma komunikacji ma nawet kilka dodatkowych zalet. Możesz na przykład cieszyć się wakacjami już od samego początku weekendu.

Kilka rozdziałów temu pokazaliśmy, jak dzięki RMI, Hessian, Burlap, obiektowi wywołującemu HTTP i usługom sieciowym możemy umożliwić komunikację między aplikacjami. Każdy z tych mechanizmów opiera się na synchronicznej komunikacji, w której aplikacja kliencka kontaktuje się ze zdalną usługą bezpośrednio i oczekuje na zakończenie zdalnej procedury przed kontynuacją.

Komunikacja synchroniczna ma wiele zastosowań, ale nie jest bynajmniej jedynym stylem komunikacji między aplikacjami dostępnym dla programistów. **Asynchroniczna obsługa komunikatów** jest podejściem pozwalającym na pośrednie wysyłanie komunikatów z jednej aplikacji do drugiej, bez potrzeby czekania na odpowiedź. Rozwiązanie to ma w niektórych sytuacjach przewagę nad komunikatami przesyłanymi synchronicznie, o czym już wkrótce się przekonamy.

Spring udostępnia kilka sposobów asynchronicznej wymiany komunikatów. W tym rozdziale przyjrzymy się, jak można wysyłać i odbierać komunikaty w Springu, wykorzystując Java Message Service (JMS) oraz protokół AMQP (*Advanced Message Queuing Protocol*). Oprócz zwykłego wysyłania i odbierania komunikatów omówimy również obsługę przez Springa obiektów POJO sterowanych komunikatami, prostego sposobu odbierania komunikatów, który przypomina komponenty MDB (ang. *message-driven beans*) technologii EJB.

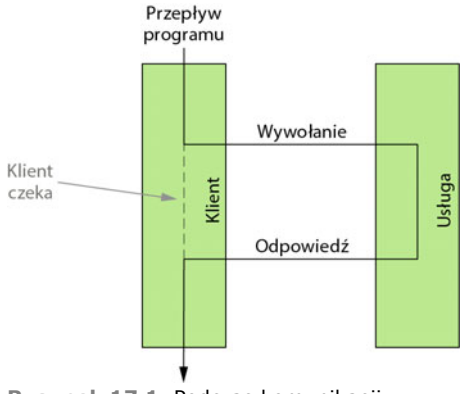
17.1. Krótkie wprowadzenie do asynchronicznej wymiany komunikatów

Podobnie jak w przypadku mechanizmów zdalnego dostępu i interfejsów REST, którymi zajmowaliśmy się wcześniej w tej części książki, asynchroniczna wymiana komunikatów służy do nawiązywania komunikacji pomiędzy aplikacjami. Jednak różni się ona od przedstawionych wcześniej mechanizmów sposobem przekazywania informacji pomiędzy systemami.

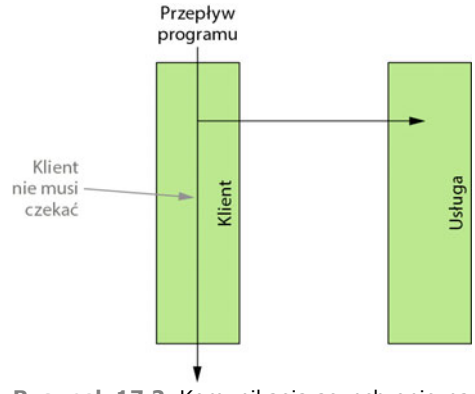
Rozwiązania zdalnego dostępu typu RMI czy Hessian/Burlap są synchroniczne. Jak pokazano na rysunku 17.1, klient wywołujący zdalną metodę nie może kontynuować działania, dopóki metoda się nie zakończy. Nawet jeśli zdalna metoda nie zwraca żadnego wyniku do klienta, i tak musi on wstrzymać swoje działanie na czas jej wykonania.

Z drugiej strony, kiedy komunikaty są przesyłane asynchronicznie, jak pokazano na rysunku 17.2, klient nie musi czekać, aż usługa przetworzy komunikat, ani nawet aż zostanie on dostarczony. Klient wysyła komunikat i kontynuuje działanie, zakładając, że prędzej czy później dotrze on do usługi i zostanie przez nią przetworzony.

Komunikacja asynchroniczna jest lepsza od komunikacji synchronicznej pod kilkoma względami. Opowiemy o nich już za chwilę. Najpierw jednak zobaczmy, w jaki sposób można asynchronicznie wysyłać komunikaty.



Rysunek 17.1. Podczas komunikacji synchronicznej klient musi czekać na zakończenie operacji



Rysunek 17.2. Komunikacja asynchroniczna nie wymaga oczekiwania

17.1.1. Wysyłanie komunikatów

Większość z nas uważa usługi świadczone przez pocztę za oczywistość. Każdego dnia ludzie powierzają pracownikom tej instytucji miliony listów, kartek i paczek, ufając, że dotrą one do adresata. Świat jest za duży, abyśmy dostarczali każdą przesyłkę własnoręcznie, zdajemy się więc w tym zakresie na system pocztowy. Adresujemy ją, naklejamy znaczek i wrzucamy do skrzynki, nie zastanawiając się nawet, jak dotrze do celu.

Kluczowym aspektem usługi pocztowej jest pośrednictwo. Doręczenie kartki bezpośrednio do babci w dniu jej urodzin byłoby raczej kłopotliwe. W zależności od tego, gdzie mieszka, mogłoby zająć od kilku godzin do kilku dni. Na szczęście, poczta jest w stanie dostarczyć kartkę, podczas gdy my zajmujemy się swoimi sprawami.

Pośrednictwo jest również kluczowe przy asynchronicznej wymianie komunikatów. Kiedy jedna aplikacja wysyła komunikat do drugiej, nie istnieje bezpośrednie połączenie między aplikacjami. Zamiast tego wysyłająca aplikacja powierza komunikat usłudze, której zadaniem jest jego dostarczenie aplikacji odbierającej.

Dwa najważniejsze pojęcia związane z asynchroniczną wymianą komunikatów to: **brokery komunikatów** (ang. *message brokers*) i **miejsca docelowe** (ang. *destinations*). Kiedy aplikacja wysyła komunikat, przekazuje go brokerowi komunikatów. Broker komunikatów jest odpowiednikiem poczty. Zapewni on doręczenie komunikatu do określonego adresata, nie angażując w cały proces nadawcy.

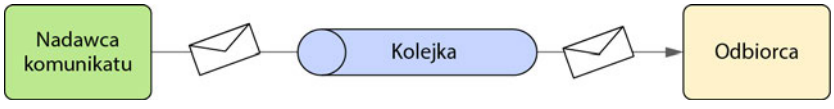
Gdy wysyłasz list pocztą, ważne jest, by był on odpowiednio zaadresowany, dzięki czemu pracownicy poczty będą wiedzieć, gdzie mają go dostarczyć. Także asynchronicznie przesyłane komunikaty posiadają rodzaj adresu — miejsce docelowe. Miejsca docelowe można porównać do skrzynek pocztowych, w których umieszczane są komunikaty czekające, aż ktoś je odbierze.

Ale w przeciwieństwie do adresów pocztowych, które mogą wskazywać określoną osobę lub ulicę i numer domu, miejsca docelowe są mniej konkretne. Miejsca docelowe skupiają się tylko na tym, *gdzie* komunikat będzie odebrany — nie na tym, *kto* go odbierze. Pod tym względem komunikaty przypominają wysyłanie listów „do aktualnego lokatora".

Choć różne systemy obsługi komunikatów mogą udostępniać wiele różnych systemów ich rozsyłania i kierowania, to można wskazać dwa najpopularniejsze rodzaje miejsc docelowych: **kolejki** (ang. *queues*) i **tematy** (ang. *topics*). Każde z nich jest związane z określonym modelem obsługi komunikatów — punkt-punkt (ang. *point-to-point*) w przypadku kolejek i publikacja-subskrypcja (ang. *publish-subscribe*) w przypadku tematów.

OBSŁUGA KOMUNIKATÓW TYPU PUNKT-PUNKT

W modelu punkt-punkt każdy komunikat ma dokładnie jednego nadawcę i jednego odbiorcę, co pokazano na rysunku 17.3. Broker komunikatów po otrzymaniu komunikatu umieszcza go w kolejce. Kiedy odbiorca zgłasza się po następny komunikat z kolejki, komunikat jest z niej pobierany i dostarczany odbiorcy. Ponieważ podczas dostarczania komunikat jest usuwany z kolejki, możemy być pewni, że nie trafi do więcej niż jednego odbiorcy.



Rysunek 17.3. Kolejka komunikatów oddziela nadawcę komunikatu od odbiorcy. Kolejka może mieć kilku odbiorców, natomiast każdy komunikat ma dokładnie jednego

To, że każdy komunikat w kolejce jest doręczany tylko jednemu odbiorcy, nie oznacza, że tylko jeden odbiorca pobiera komunikaty z kolejki. Komunikaty z kolejki mogą być przetwarzane przez kilku odbiorców. Każdy z nich przetwarza jednak swoje własne komunikaty.

Proces można porównać do czekania w kolejce w banku. Przy transakcji może Ci pomóc jeden z kilku kasjerów. Po obsłużeniu klienta kasjer jest wolny i prosi następną osobę z kolejki. Gdy nadchodzi Twoja kolej, zostajesz poproszony do okienka i obsłużony przez jednego kasjera. Pozostali kasjerzy obsłużą innych klientów.

Kolejną analogią z bankiem jest to, że podczas gdy stoisz w kolejce, z reguły nie wiesz, który kasjer Cię obsłuży. Możesz policzyć liczbę oczekujących w kolejce, skonfrontować ją z liczbą kasjerów i spróbować zgadnąć, który kasjer zawoła Cię do okienka. Szanse, że się pomylisz, są jednak bardzo duże.

Podobnie jest w przypadku modelu obsługi komunikatów punkt-punkt, jeśli wielu odbiorców nasłuchuje komunikatów z kolejki, nie wiadomo, który ostatecznie przetworzy konkretny komunikat. Ta niepewność jest dobra, umożliwia bowiem aplikacji zwiększenie zaangażowania w przetwarzanie komunikatów poprzez proste dodanie kolejnego odbiorcy.

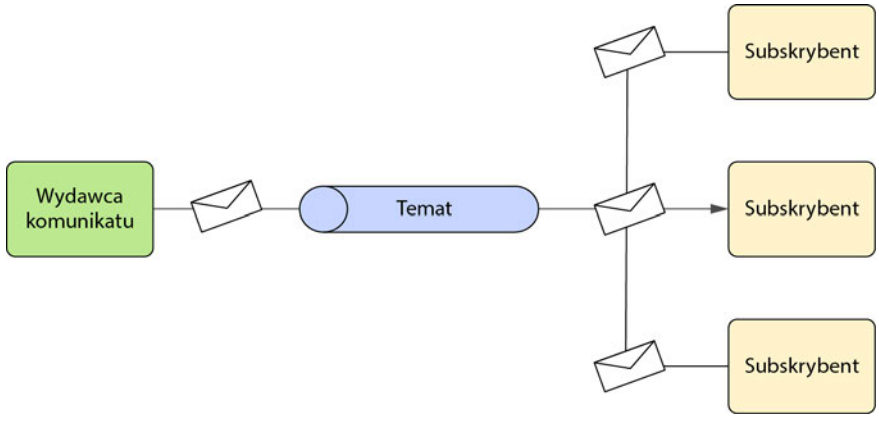
OBSŁUGA KOMUNIKATÓW TYPU PUBLIKACJA-SUBSKRYPCJA

W modelu obsługi komunikatów publikacja-subskrypcja komunikaty są wysyłane do tematu. Tak jak w przypadku kolejek, wielu odbiorców nasłuchuje komunikatów z tematu. Ale w przeciwieństwie do kolejek, gdzie dany komunikat jest doręczany tylko i wyłącznie jednemu odbiorcy, wszyscy subskrybenci tematu otrzymają kopię komunikatu (rysunek 17.4).

Jak łatwo wywnioskować z nazwy, model publikacja-subskrypcja jest analogią do wydawcy czasopisma i jego prenumeratorów. Czasopismo (komunikat) jest publikowane i wysyłane pocztą, każdy prenumerator otrzymuje jedną kopię.

Analogia z czasopismem upada, kiedy zdamy sobie sprawę, że w przypadku asynchronicznej wymiany komunikatów wydawca nie ma pojęcia o tym, kto jest subskrybentem. Wydawca wie tylko, że komunikat zostanie opublikowany w danym temacie — nie ma żadnych informacji o odbiorcach tematu. A co za tym idzie, nie wie, w jaki sposób komunikat zostanie przetworzony.

Teraz, kiedy omówiliśmy już podstawy asynchronicznej wymiany komunikatów, spróbujmy porównać ją do synchronicznego RPC.



Rysunek 17.4. Podobnie jak kolejki, tematy oddzielają nadawców komunikatów od ich odbiorców, z tą różnicą, że komunikat tematu może zostać dostarczony do wielu subskrybentów tematu

17.1.2. Szacowanie korzyści związanych ze stosowaniem asynchronicznej wymiany komunikatów

Chociaż intuicyjna i prosta w instalacji, komunikacja synchroniczna narzuca pewne ograniczenia po stronie klienta zdalnej usługi. Oto kilka najważniejszych:

- *Komunikacja synchroniczna wiąże się z oczekiwaniem.* Kiedy klient wywołuje metodę zdalnej usługi, musi poczekać na jej zakończenie przed wykonaniem kolejnych zadań. Jeśli klient komunikuje się ze zdalną usługą często lub (i) oczekiwanie na odpowiedź zdalnej usługi trwa długo, może to negatywnie wpłynąć na wydajność aplikacji klienta.
- *Klient jest uzależniony do usługi przez jej interfejs, którego używa.* Jeżeli interfejs usługi się zmieni, konieczna będzie również modyfikacja klientów usługi.
- *Klient jest uzależniony od adresu usługi.* Musi mu zostać podany adres usługi, aby mógł się z nią połączyć. Jeśli topologia sieci się zmieni, klient będzie musiał zostać skonfigurowany ponownie, z uwzględnieniem nowego adresu.
- *Klient jest uzależniony od dostępności usługi.* Gdy usługa jest niedostępna, klient nie może z niej skorzystać.

Chociaż komunikacja synchroniczna ma swoje zastosowania, przy ocenianiu potrzeb aplikacji w zakresie mechanizmu komunikacji powinniśmy wziąć pod uwagę jej wszystkie wyżej wymienione wady. Jeżeli ograniczenia te są dla Ciebie istotne, z pewnością zainteresuje Cię, jak radzi sobie z nimi asynchroniczna wymiana komunikatów.

BEZ CZEKANIA

Kiedy komunikat jest wysyłany asynchronicznie, klient nie musi czekać na jego przetworzenie ani nawet dostarczenie. Zostawia komunikat w brokerze komunikatów i kontynuuje działanie, ufając, że komunikat dotrze do odpowiedniego miejsca docelowego.

Ponieważ nie musi czekać, klient dostaje wolną rękę w wykonywaniu dalszych działań. Powoduje to znaczący wzrost wydajności klienta.

CENTRALNA ROLA KOMUNIKATÓW I ODDZIELENIE NADAWCY OD ODBIORCY

W przeciwieństwie do komunikacji RPC, która najczęściej koncentruje się wokół wywołania metody, asynchronicznie wysyłane komunikaty skupiają się na danych. Oznacza to, że klient nie jest przypisany na stałe do konkretnej sygnatury metody. Każdy odbiorca kolejki lub subskrybent tematu, który potrafi przetworzyć przesłane przez klienta dane, potrafi przetworzyć komunikat. Klient nie musi znać szczegółów usługi.

NIEZALEŻNOŚĆ OD ADRESU

Synchroniczne usługi RPC są z reguły lokalizowane za pomocą adresu sieciowego. Na skutek tego aplikacje klienckie nie są odporne na zmiany w topologii sieci. Jeśli adres IP usługi ulegnie zmianie lub jeśli zacznie ona nasłuchiwać na innym porcie, klient musi zostać odpowiednio zmodyfikowany, inaczej nie będzie mógł skorzystać z usługi.

Aplikacje klienckie korzystające z asynchronicznej wymiany komunikatów nie mają natomiast pojęcia, kto przetworzy ich komunikaty ani gdzie znajduje się usługa. Klient zna tylko kolejkę lub temat, przez które komunikat zostanie wysłany. Nie ma dla niego znaczenia lokalizacja usługi, liczy się tylko możliwość pobierania komunikatów z kolejki lub tematu.

W modelu punkt-punkt dzięki niezależności od adresu można utworzyć klaster usług. Skoro klient nie musi znać adresu usługi, a jedynym jej wymaganiem jest, aby miał dostęp do brokera komunikatów, nie ma powodu, dla którego wiele usług nie może pobierać komunikatów z tej samej kolejki. Jeśli usługa jest nadmiernie obciążona i nie nadąża z przetwarzaniem, wystarczy dodać kilka nowych instancji usługi odbierających komunikaty z tej samej kolejki.

Niezależność od adresu ma jeszcze jeden interesujący efekt uboczny w modelu publikacja-subskrypcja. Wiele usług może subskrybować ten sam temat, otrzymując podwójne kopie tych samych komunikatów. Ale każda mogłaby przetworzyć ten komunikat inaczej. Powiedzmy na przykład, że mamy zestaw usług, które przetwarzają komunikat zawierający szczegóły zatrudnienia nowego pracownika. Jedna z usług może dodać pracownika do systemu płac, druga do portalu HR, jeszcze inna dopilnować, żeby pracownik miał dostęp do systemów, które będą mu potrzebne w pracy. Każda usługa operuje niezależnie na tych samych danych, pobranych z tematu.

GWARANCJA DOSTARCZENIA

Aby klient mógł połączyć się z synchroniczną usługą, usługa musi nasłuchiwać na określonym porcie pod określonym adresem IP. W razie awarii usługi klient nie będzie mógł kontynuować działania.

Przy asynchronicznym wysyłaniu komunikatów klient ma pewność, że jego komunikaty będą dostarczone. Nawet gdy usługa jest niedostępna podczas wysyłania komunikatu, komunikat zostanie przechowany do czasu jej wznowienia.

Teraz gdy znamy już podstawy asynchronicznej wymiany komunikatów, możemy przyjrzeć się jej w działaniu. Zaczniemy od wysyłania i odbierania komunikatów przy użyciu JMS.

17.2. Wysyłanie komunikatów przy użyciu JMS

Java Message Service (w skrócie: JMS) to standard Javy definiujący wspólny interfejs API służący do korzystania z brokerów komunikatów. Przed wprowadzeniem JMS każdy broker komunikatów udostępniał swój własny API, znacząco ograniczając możliwości przenoszenia kodu aplikacji i wykorzystania innego brokera. Jednak obecnie dzięki JMS wszystkie implementacje zgodne z tym standardem mogą być obsługiwane przy użyciu jednego, wspólnego interfejsu — podobnie jak JDBC udostępnia wspólny interfejs do obsługi baz danych.

Spring obsługuje JMS przy użyciu abstrakcji bazującej na szablonach, a konkretnie — szablonu `JmsTemplate`. Korzystając z niego, można w prosty sposób wysyłać komunikaty do kolejek i tematów (po stronie producenta) oraz odbierać komunikaty (po stronie klienta). Spring obsługuje także notację obiektów POJO sterowanych komunikatami: zwyczajnych obiektów Javy reagujących na komunikaty asynchronicznie nadsyłane do kolejki lub tematu.

W tym rozdziale przyjrzymy się mechanizmom korzystania z JSM dostępnym w Springu, w tym szablonowi `JmsTemplate` oraz obiektom POJO sterowanym komunikatami. Jednak zanim będziemy mogli wysyłać i odbierać komunikaty, musimy przygotować brokera komunikatów, który będzie pośredniczył w ich wymianie pomiędzy producentami a konsumentami. Zacznijmy zatem naszą przygodę z JMS w Springu od skonfigurowania brokera komunikatów.

17.2.1. Konfiguracja brokera komunikatów w Springu

ActiveMQ, broker komunikatów o otwartym kodzie, jest doskonałym wyborem, jeśli chodzi o asynchroniczną obsługę komunikatów za pomocą JMS. W momencie pisania tych słów najnowsza wersja ActiveMQ ma numer 5.11.1. Aby rozpocząć pracę z ActiveMQ, musimy pobrać plik dystrybucji binarnej z *http://activemq.apache.org*. Po pobraniu rozpakujemy zawartość archiwum na lokalny dysk. W katalogu *lib* rozpakowanej dystrybucji znajdziemy plik *activemq-core-5.11.1.jar*. Plik ten musi zostać dodany do ścieżki do klas aplikacji, aby korzystanie z API ActiveMQ było możliwe.

W katalogu *bin* znajdziemy szereg podkatalogów dla różnych systemów operacyjnych. To w nich znajdują się skrypty służące do uruchomienia ActiveMQ. Na przykład, aby uruchomić ActiveMQ w systemie OS X[1], wydaj komendę `activemq start` z katalogu *macosx*. Już po chwili ActiveMQ będzie gotowy do przetwarzania komunikatów.

TWORZENIE FABRYKI POŁĄCZEŃ

W tym rozdziale pokażemy różne przykłady użycia Springa do wysyłania i odbierania komunikatów za pomocą JMS. W każdym z nich potrzebować będziemy fabryki połączeń, aby móc wysyłać komunikaty przez brokera komunikatów. Jako że naszym brokerem komunikatów jest ActiveMQ, będziemy musieli skonfigurować fabrykę połączeń JMS do połączenia z ActiveMQ. ActiveMQ dostarcza fabrykę połączeń JMS `ActiveMQConnectionFactory`, którą konfiguruje się w Springu następująco:

```
<bean id="connectionFactory"
  class="org.apache.activemq.spring.ActiveMQConnectionFactory">
</bean>
```

Domyślnie `ActiveMQConnectionFactory` zakłada, że broker ActiveMQ nasłuchuje na porcie 61616 lokalnego komputera (`localhost`). Takie rozwiązanie w zupełności wystarcza na potrzeby tworzenia aplikacji, choć produkcyjny broker ActiveMQ najprawdopodobniej będzie musiał działać na innym komputerze bądź porcie. W takim przypadku adres URL brokera można określić przy użyciu właściwości `brokerURL`:

```
<bean id="connectionFactory"
      class="org.apache.activemq.spring.ActiveMQConnectionFactory"
      p:brokerURL="tcp://localhost:61616"/>
```

Ewentualnie, ponieważ wiemy, że mamy do czynienia z ActiveMQ, do deklaracji fabryki połączeń możemy też użyć konfiguracyjnej przestrzeni nazw Springa dla ActiveMQ (dostępnej dla wszystkich wersji ActiveMQ, począwszy od wersji 4.1). Zaczniemy od deklaracji przestrzeni nazw `amq` w pliku konfiguracyjnym XML Springa:

```
<?xml version="1.0" encoding="UTF-8"?>
  <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jms="http://www.springframework.org/schema/jms"
    xmlns:amq="http://activemq.apache.org/schema/core"
    xsi:schemaLocation="http://activemq.apache.org/schema/core
      http://activemq.apache.org/schema/core/activemq-core.xsd
      http://www.springframework.org/schema/jms
      http://www.springframework.org/schema/jms/spring-jms.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
  ...
</beans>
```

Następnie użyjemy elementu `<amq:connectionFactory>` do deklaracji fabryki połączeń:

```
<amq:connectionFactory id="connectionFactory"
                       brokerURL="tcp://localhost:61616"/>
```

[1] Instrukcje instalacji ActiveMQ w pozostałych systemach operacyjnych można znaleźć pod *adresem http://activemq.apache.org/getting-started.html — przyp. tłum.*

Zwróć uwagę, że element <amq:connectionFactory> jest charakterystyczny dla ActiveMQ. Dla innej implementacji brokera komunikatów konfiguracyjna przestrzeń nazw Springa może, ale nie musi istnieć. W przypadku jej braku fabryka połączeń musi zostać dowiązana jako <bean>.

W dalszej części rozdziału będziemy używać komponentu connectionFactory bardzo często. W tej chwili jednak wystarczy nam wiedza, że atrybut brokerURL informuje fabrykę połączeń o adresie brokera komunikatów. W naszym przykładzie podany w atrybucie brokerURL adres URL sugeruje fabryce połączeń połączenie z ActiveMQ na porcie 61616 lokalnego komputera (na tym porcie ActiveMQ nasłuchuje domyślnie).

DEKLARACJA MIEJSCA DOCELOWEGO KOMUNIKATÓW ACTIVEMQ

Oprócz fabryki połączeń potrzebować będziemy miejsca docelowego, do którego komunikaty będą dostarczane. Miejsce docelowe może być albo kolejką, albo tematem, w zależności od potrzeb aplikacji.

Bez względu na to, czy używamy kolejki czy tematu, musimy skonfigurować komponent miejsca docelowego w Springu za pomocą implementacji klasy specyficznej dla brokera komunikatów. Na przykład poniższy komponent deklaruje kolejkę ActiveMQ:

```
<bean id="queue"
   class="org.apache.activemq.command.ActiveMQQueue"
   c:_="spitter.queue"/>
</bean>
```

Analogiczny komponent deklarujący temat ActiveMQ przedstawia się następująco:

```
<bean id="topic"
      class="org.apache.activemq.command.ActiveMQTopic"
      c:_="spitter.queue" />
```

W obu przypadkach do konstruktora jest przekazywana nazwa kolejki, po której jest ona identyfikowana przez brokera komunikatów. W naszym przykładzie jest to spitter.
➥topic.

Podobnie jak przy fabryce połączeń, przestrzeń nazw ActiveMQ oferuje nam alternatywną metodę deklaracji kolejek i tematów. Dla kolejek możemy użyć elementu <amq:queue>:

```
<amq:queue id="spittleQueue" physicalName="spitter.alert.queue" />
```

A dla tematów JMS elementu <amq:topic>:

```
<amq:topic id="spittleTopic" physicalName="spitter.alert.topic" />
```

W obu przypadkach atrybut physicalName jest nazwą kanału komunikatów.

Na tym etapie wiemy już, jak zadeklarować wszystkie komponenty niezbędne do pracy z JMS, niezależnie od tego, czy chcemy wysyłać komunikaty, czy je odbierać. Jesteśmy już więc gotowi do rozpoczęcia komunikacji. Użyjemy do tego szablonu JmsTemplate, który stanowi trzon obsługi JMS przez Springa. Najpierw jednak, aby docenić korzyści płynące z tego szablonu, zobaczmy, jak wygląda JMS bez JmsTemplate.

17.2.2. Szablon JMS Springa

Jak już wiemy, JMS daje programistom Javy standardowe API do interakcji z brokerami komunikatów oraz wysyłania i obierania komunikatów. Mało tego: praktycznie każda implementacja brokera komunikatów obsługuje JMS. Nie ma więc potrzeby nauki niestandardowego API obsługi komunikatów przy każdym nowym brokerze.

Ale choć JMS oferuje interfejs uniwersalny dla wszystkich brokerów komunikatów, nie dostajemy tego za darmo. Wysyłanie i odbieranie komunikatów za pomocą JMS nie jest tak proste, jak przyklejenie znaczka na kopertę. Używając przenośni, można by powiedzieć, że wymaga jeszcze dodatkowo zatankowania furgonetki przewoźnika poczty.

KOD JMS A OBSŁUGA WYJĄTKÓW

W punkcie 10.3.1 zaprezentowałem przykład tradycyjnego kodu JDBC, przypominającego bardziej bezładną masę kodu do obsługi połączeń, wyrażeń, zbiorów wynikowych i wyjątków. Niestety, tradycyjny kod JMS wydaje się podążać tą samą drogą, co da się zaobserwować na listingu 17.1.

Listing 17.1. Wysyłanie komunikatu przy użyciu tradycyjnego JMS (bez Springa)

```
ConnectionFactory cf =
      new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
  conn = cf.createConnection();
  session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
  Destination destination = new ActiveMQQueue("spitter.queue");
  MessageProducer producer = session.createProducer(destination);
  TextMessage message = session.createTextMessage();

  message.setText("Witaj, świecie!");
  producer.send(message); <-- Wyślij komunikat
} catch (JMSException e) {
  // obsługa wyjątku?
} finally {
  try {
    if (session != null) {
      session.close();
    }
    if (conn != null) {
      conn.close();
    }
  } catch (JMSException ex) {
  }
}
```

Gdzieś to już chyba mówiłem, ale to całkiem pokaźny kawałek kodu! Zupełnie jak w przykładzie JDBC, prawie 20 wierszy tylko po to, żeby wysłać prosty komunikat „Witaj, świecie!”. Za samo wysłanie komunikatu odpowiada tak naprawdę tylko kilka wierszy kodu. Reszta służy tylko do stworzenia warunków dla tej operacji.

Po stronie odbiorcy sytuacja wygląda niewiele lepiej; spójrzmy na listing 17.2.

Listing 17.2. Odbieranie komunikatu przy użyciu tradycyjnego JMS (bez Springa)

```
ConnectionFactory cf =
      new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
  conn = cf.createConnection();
  conn.start();
  session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
  Destination destination =
      new ActiveMQQueue("spitter.queue");
  MessageConsumer consumer = session.createConsumer(destination);
  Message message = consumer.receive();
  TextMessage textMessage = (TextMessage) message;
  System.out.println("OTRZYMANO KOMUNIKAT: " + textMessage.getText());
  conn.start();
} catch (JMSException e) {
// obsługa wyjątku?
} finally {
try {
  if (session != null) {
    session.close();
  }
  if (conn != null) {
    conn.close();
  }
  } catch (JMSException ex) {
  }
}
```

Podobnie jak na listingu 17.1, to zdecydowanie za dużo kodu na coś tak prostego. Porównując oba listingi wiersz po wierszu, zauważysz, że są niemal identyczne. I każdy z tysiąca innych przykładów JMS byłby uderzająco podobny. Niektóre uzyskiwałyby fabryki połączeń z JNDI, inne używały tematu zamiast kolejki. Wszystkie byłyby jednak skonstruowane według tego samego wzorca.

W ten sposób, pracując z JMS, powielasz każdorazowo duże fragmenty swojego kodu JMS. Albo, co nawet gorsze — czyjegoś.

W rozdziale 10. zaprezentowaliśmy szablon `JdbcTemplate`, dzięki któremu udało się ograniczyć kod JDBC do niezbędnego minimum. Teraz spróbujemy się uporać z nadmiarowym kodem JMS w analogiczny sposób, za pomocą szablonu `JmsTemplate`.

PRACA Z SZABLONAMI JMS

Szablon `JmsTemplate` jest odpowiedzią Springa na rozwlekły i pełen powtórzeń kod JMS. `JmsTemplate` zajmuje się tworzeniem połączenia, uzyskiwaniem sesji i wreszcie wysyłaniem oraz odbieraniem komunikatów. Dzięki temu programista może się skupić na generowaniu nowych komunikatów i przetwarzaniu otrzymanych.

Ponadto, `JmsTemplate` potrafi obsłużyć kłopotliwy wyjątek `JMSException`, który może zostać w każdej chwili zgłoszony. Jeśli podczas pracy z `JmsTemplate` zgłoszony zostanie wyjątek `JMSException`, `JmsTemplate` przechwyci go i zgłosi ponownie w postaci jednego z niekontrolowanych wyjątków, będących rozszerzeniem klasy `JmsException` Springa.

W tabeli 17.1 zestawiono standardowe wyjątki `JMSException` i odpowiadające im niekontrolowane wyjątki Springa `JmsException`.

Trzeba oddać API JMS, że klasa `JMSException` posiada dosyć obszerny i opisowy zbiór podklas, które dają nam pewne pojęcie o charakterze błędu. Niemniej jednak wszystkie one są klasami wyjątków kontrolowanych, które muszą być przechwycone. `JmsTemplate` zajmuje się tym za nas, przechwytując te wyjątki i zgłaszając je ponownie jako niekontrolowane podklasy `JmsException`.

Tabela 17.1. Szablon JmsTemplate Springa przechwytuje standardowe wyjątki JMSException i zgłasza je ponownie jako niekontrolowane podklasy JmsException Springa

Spring (`org.springframework.jms.*`)	**Standardowe JMS (`javax.jms.*`)**
`DestinationResolutionException`	Specyficzny dla Springa — zgłaszany, gdy Spring nie jest w stanie uzyskać nazwy miejsca docelowego
`IllegalStateException`	`IllegalStateException`
`InvalidClientIDException`	`InvalidClientIDException`
`InvalidDestinationException`	`InvalidDestinationException`
`InvalidSelectorException`	`InvalidSelectorException`
`JmsSecurityException`	`JmsSecurityException`
`ListenerExecutionFailedException`	Specyficzny dla Springa — zgłaszany, gdy nie uda się wykonać metody odbiorcy
`MessageConversionException`	Specyficzny dla Springa — zgłaszany, gdy konwersja komunikatu się nie powiedzie
`MessageEOFException`	`MessageEOFException`
`MessageFormatException`	`MessageFormatException`
`MessageNotReadableException`	`MessageNotReadableException`
`MessageNotWriteableException`	`MessageNotWriteableException`
`ResourceAllocationException`	`ResourceAllocationException`
`SynchedLocalTransactionFailedException`	Specyficzny dla Springa — zgłaszany przy błędzie zsynchronizowanej lokalnej transakcji
`TransactionInProgressException`	`TransactionInProgressException`
`TransactionRolledBackException`	`TransactionRolledBackException`
`UncategorizedJmsException`	Specyficzny dla Springa — zgłaszany w sytuacji, gdy nie można zastosować żadnego innego wyjątku

Aby użyć szablonu `JmsTemplate`, musimy zadeklarować go jako komponent w pliku konfiguracyjnym Springa. Poniższy fragment kodu XML powinien wystarczyć:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate"
      c:_-ref="connectionFactory" />
```

Ponieważ `JmsTemplate` powinien wiedzieć, jak pozyskiwać połączenia do brokera komunikatów, we właściwości `connectionFactory` musimy podać referencję do komponentu implementującego interfejs `ConnectionFactory` JMS. W przykładzie powyżej dowiązaliśmy referencję do zadeklarowanego przez nas wcześniej (w punkcie 17.2.1) komponentu `connectionFactory`.

Tak skonfigurowany szablon JmsTemplate jest gotowy do użycia. Czas wysłać komunikaty!

WYSYŁANIE KOMUNIKATÓW

Jedną z wbudowanych funkcji aplikacji Spittr jest opcja powiadamiania (być może za pomocą poczty elektronicznej) innych użytkowników o pojawieniu się nowego spittle'a. Moglibyśmy wbudować ją bezpośrednio w aplikację, w punkcie, w którym dodawany jest spittle. Ale decyzja, do kogo te powiadomienia wysłać, a zwłaszcza samo ich wysłanie, może zająć chwilę. Ten dodatkowy czas może zaważyć na wydajności aplikacji. Podczas tworzenia nowego spittle'a oczekujemy, że aplikacja odpowie błyskawicznie.

Zamiast poświęcać cenny czas na wysyłanie tych komunikatów w momencie dodawania spittle'a, bardziej sensownym rozwiązaniem wydaje się umieszczenie tego zadania w kolejce, do wykonania już po otrzymaniu odpowiedzi przez użytkownika. Czas potrzebny na wysłanie komunikatu do kolejki komunikatów jest nieporównywalnie krótszy od czasu, który musielibyśmy przeznaczyć na rozsyłanie powiadomień użytkownikom.

W asynchronicznym wysyłaniu powiadomień o nowych spittle'ach pomoże nam nowa usługa aplikacji Spittr — AlertService:

```
package com.habuma.spittr.alerts;
import com.habuma.spittr.domain.Spittle;

public interface AlertService {
   void sendSpittleAlert(Spittle spittle);
}
```

Jak widać, AlertService jest interfejsem definiującym tylko jedną operację: sendSpittle
➥Alert().

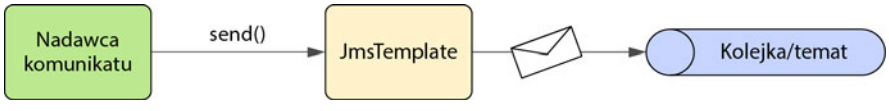
AlertServiceImpl jest implementacją interfejsu AlertService, która za pomocą wstrzykniętego JmsTemplate (interfejsu używanego przez JmsTemplate) wysyła obiekty Spittle do kolejki komunikatów celem późniejszego przetworzenia. Kod klasy pokazano na listingu 17.3.

Listing 17.3. Wysyłanie spittle'a z wykorzystaniem JmsTemplate

```
package com.habuma.spittr.alerts;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsOperations;
import org.springframework.jms.core.MessageCreator;
import com.habuma.spittr.domain.Spittle;

public class AlertServiceImpl implements AlertService {
  private JmsOperations jmsOperations;

  @Autowired
  public AlertServiceImpl(JmsOperations jmsOperatons) {   <-- Wstrzykuje szablon JMS
    this.jmsOperations = jmsOperations;
```

```
  }

  public void sendSpittleAlert(final Spittle spittle) {
    jmsOperations.send(                                    <-- Wysyła komunikat
      "spittle.alert.queue",                               <-- Określa miejsce docelowe
      new MessageCreator() {
        public Message createMessage(Session session)
                  throws JMSException {
          return session.createObjectMessage(spittle);     <-- Tworzy komunikat
        }
      }
    );
  }
}
```

Pierwszym parametrem metody `send()` szablonu `JmsTemplate` jest nazwa miejsca docelowego JMS, do którego komunikat zostanie wysłany. Po wywołaniu metody `send()` `Jms`
`↪Template` postara się uzyskać połączenie JMS i sesję, a następnie wyśle komunikat w imieniu nadawcy (rysunek 17 5).



Rysunek 17.5. JmsTemplate wykonuje złożoną operację wysłania komunikatu w imieniu nadawcy

Sam komunikat natomiast jest konstruowany za pomocą kreatora komunikatów `Message`
`↪Creator`, tu zaimplementowanego jako anonimowa klasa wewnętrzna. Metoda kreatora `createMessage()` zwraca obiekt komunikatu z sesji na podstawie przekazanego obiektu `Spittle`, z którego buduje komunikat.

I po wszystkim! Zauważ, że metoda `sendSpittleAlert()` koncentruje się wyłącznie na wygenerowaniu i wysłaniu komunikatu. Nie trzeba specjalnego kodu do połączenia ani zarządzania sesją; `JmsTemplate` wyręcza nas w tej kwestii. Nie ma też potrzeby przechwytywania wyjątku `JMSException`. `JmsTemplate` przechwyci wszystkie wyjątki `JMSEx`
`↪ception` i zgłosi je ponownie jako jeden z niekontrolowanych wyjątków Springa z tabeli 17.1.

USTAWIANIE DOMYŚLNEGO MIEJSCA DOCELOWEGO

Na listingu 17.3 określiliśmy miejsce docelowe, do którego komunikat zostanie wysłany w metodzie `send()`. Ta forma metody `send()` nadaje się do sytuacji, w których chcemy programowo wybrać miejsce docelowe. Ale w przypadku `AlertServiceImpl` komunikat spittle będzie wysyłany zawsze w to samo miejsce docelowe, nie korzystamy więc z tej możliwości.

Zamiast określać jawnie miejsce docelowe za każdym razem, gdy wysyłamy komunikat, możemy dowiązać domyślne miejsce docelowe do `JmsTemplate`:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate"
      c:_-ref="connectionFactory"
      p:defaultDestinationName="spittle.alert.queue" />
```

W tym przypadku miejsce docelowe komunikatów jest określane jako `spittle.alert.queue`. Jednak to tylko nazwa: nie określa ona typu miejsca docelowego, z którym mamy do czynienia. Jeśli istnieje już kolejka lub temat o tej nazwie, to zostaną one użyte. W przeciwnym razie zostanie utworzone nowe miejsce docelowe (przy czym zazwyczaj będzie to kolejka). Jeżeli jednak chcemy być bardziej precyzyjni i określić typ miejsca docelowego, które zostanie utworzone, to możemy dowiązać referencję do zadeklarowanej wcześniej kolejki bądź miejsca docelowego:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate"
      c:_-ref="connectionFactory"
      p:defaultDestination-ref="spittleTopic" />
```

Wywołanie metody `send()` szablonu JMS można teraz nieco uprościć, usuwając pierwszy parametr:

```
jmsTemplate.send(
   new MessageCreator() {
   ...
   }
);
```

Ta forma metody `send()` jako parametr przyjmuje tylko obiekt `MessageCreator`. Nie ma potrzeby określania miejsca docelowego, ponieważ wszystkie komunikaty trafiają do domyślnego miejsca docelowego.

Pozbycie się jawnego określenia miejsca docelowego w wywołaniu metody `send()` nieco uprościło sprawę. Jednak wysyłanie komunikatów może być jeszcze łatwiejsze — wystarczy skorzystać z ich konwerterów.

KONWERTOWANIE KOMUNIKATÓW PODCZAS ICH WYSYŁANIA

Oprócz metody `send()` szablon JMS udostępnia także metodę `convertAndSend()`. W odróżnieniu od `send()` metoda `convertAndSend()` nie pobiera argumentu typu `MessageCreator`. Wynika to z faktu, że do utworzenia komunikatu używa ona wbudowanego konwertera komunikatów.

W przypadku stosowania tej metody naszą metodę `sendSpittleAlert()` można uprościć do następującej postaci:

```
public void sendSpittleAlert(Spittle spittle) {
  jmsOperations.convertAndSend(spittle);
}
```

W niemal magiczny sposób obiekt `Spittle` zostaje skonwertowany przed wysłaniem na obiekt `Message`. Jednak, jak to zazwyczaj bywa z magicznymi sztuczkami, szablon JMS miał w zanadrzu coś, co mu pomogło. Okazuje się, że w celu wykonania konwersji wysyłanych obiektów na obiekty `Message` korzysta on z implementacji interfejsu `MessageConverter`.

`MessageConverter` to interfejs zdefiniowany przez Springa, który deklaruje jedynie dwie metody:

```
public interface MessageConverter {
  Message toMessage(Object object, Session session)
```

```
                    throws JMSException, MessageConversionException;
  Object fromMessage(Message message)
                    throws JMSException, MessageConversionException;
}
```

Choć zaimplementowanie tego interfejsu jest dosyć proste, to jednak w większości sytuacji nie będziemy musieli tworzyć jego własnych implementacji. Spring udostępnia bowiem kilka takich implementacji — zostały one przedstawione w tabeli 17.2.

Tabela 17.2. Spring udostępnia kilka konwerterów komunikatów, służących do wykonywania najpopularniejszych rodzajów konwersji (wszystkie są dostępne w pakiecie org.springframework.jms.support.converter)

Konwerter komunikatów	Przeznaczenie
`MappingJacksonMessageConverter`	Używa biblioteki Jackson JSON, by konwertować komunikaty na kod JSON i na odwrót.
`MappingJackson2MessageConverter`	Używa biblioteki Jackson JSON, by konwertować komunikaty na kod JSON i na odwrót.
`MarshallingMessageConverter`	Używa JAXB do konwertowania komunikatów na XML i na odwrót.
`SimpleMessageConverter`	Konwertuje łańcuch znaków (`String`) na `TextMessage` (i na odwrót), tablice bajtów na `BytesMessage` (i na odwrót), obiekty `Map` na `MapMessage` (i na odwrót) oraz obiekty `Serializable` na `ObjectMessage` (i na odwrót).

Podczas przesyłania komunikatów za pomocą metody `convertAndSend()` szablon JMS używa domyślnie konwertera `SimpleMessageConverter`. Można to jednak zmienić, deklarując konwerter komunikatów jako komponent i wstrzykując go do właściwości `messageConverter` szablonu JMS. Na przykład jeśli mamy zamiar stosować komunikaty JSON, to możemy zadeklarować komponent `MappingJacksonMessageConverter`:

```
<bean id="messageConverter"
    class="org.springframework.jms.support.converter.MappingJacksonMessageConverter" />
```

Następnie wystarczy dowiązać go do szablonu JMS w następujący sposób:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate"
      c:_-ref="connectionFactory"
      p:defaultDestinationName="spittle.alert.queue"
      p:messageConverter-ref="messageConverter" />
```

Różne konwertery komunikatów mogą udostępniać dodatkowe opcje konfiguracyjne, pozwalające na dokładniejszą kontrolę procesu konwersji. Na przykład konwerter `MappingJackskonMessageConverter` umożliwia konfigurowanie takich aspektów konwersji jak używane kodowanie oraz stosowanie niestandardowych obiektów `ObjectMapper`. Dokładniejsze informacje na temat sposobu konfiguracji tych szczegółowych aspektów działania poszczególnych konwerterów komunikatów można znaleźć w ich dokumentacji.

KONSUMOWANIE KOMUNIKATÓW

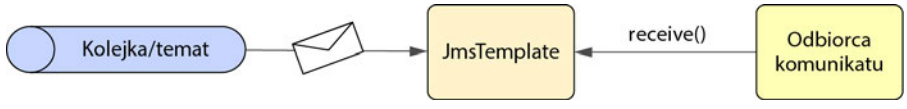
Wiemy już, jak wysłać komunikat za pomocą `JmsTemplate`. A co z drugą stroną? Czy `JmsTemplate` da się też wykorzystać do odbierania komunikatów?

Owszem. I jest to nawet prostsze niż wysyłanie komunikatów. Wystarczy tylko wywołać metodę `receive()` szablonu JMS w sposób pokazany na listingu 17.4.

Listing 17.4. Odbieranie komunikatu z wykorzystaniem JmsTemplate

```
public Spittle receiveSpittleAlert() {
  try {
    ObjectMessage receivedMessage =
        (ObjectMessage) jmsOperations.receive();   <-- Odbierz komunikat

    return (Spittle) receivedMessage.getObject();   <-- Pobierz obiekt
  } catch (JMSException jmsException) {
    throw JmsUtils.convertJmsAccessException(jmsException);   <-- Zgłoś wyjątek konwersji
  }
}
```

Metoda `receive()` szablonu JMS w momencie wywołania spróbuje pobrać komunikat z brokera. W przypadku braku dostępnych komunikatów poczeka, aż jakiś się pojawi. Interakcję tę pokazano na rysunku 17.6.



Rysunek 17.6. Odbieranie komunikatów z tematu lub kolejki za pomocą JmsTemplate jest równie proste, co wywołanie metody receive(). JmsTemplate zajmuje się resztą

Ponieważ wiemy, że komunikat spittle został wysłany w postaci obiektu komunikatu, może on zostać zrzutowany na typ `ObjectMessage` po nadejściu. W dalszej kolejności wywołujemy metodę `getObject()`, aby uzyskać obiekt typu `Spittle` z obiektu komunikatu, który jest następnie zwracany.

Jest jedno „ale". Musimy coś zrobić z potencjalnym wyjątkiem `JMSException`. Jak wcześniej wspominałem, `JmsTemplate` znakomicie sobie radzi z obsługą kontrolowanych wyjątków `JMSException` i ponownym ich zgłaszaniem w postaci niekontrolowanych wyjątków `JmsException` Springa. Dotyczy to jednak tylko wywołań metod `JmsTemplate`. Szablon JMS jest bezradny przy wyjątku `JMSException`, który może się pojawić przy wywołaniu metody `getObject()` obiektu typu `ObjectMessage`.

Dlatego musimy albo przechwycić ten wyjątek, albo zadeklarować go w sygnaturze metody. Zgodnie z filozofią Springa, by unikać kontrolowanych wyjątków, nie chcemy, żeby wyjątek `JMSException` wymknął się metodzie, tak więc spróbujemy go przechwycić. W bloku `catch` możemy skorzystać z metody `convertJmsAccessException()` z klasy Springa `JmsUtils` do przekształcenia kontrolowanego wyjątku `JMSException` w niekontrolowany `JmsException`. Osiągniemy w ten sposób to samo, co daje nam `JmsTemplate` w pozostałych przypadkach.

Jedną z rzeczy, którą można zrobić, by przetworzyć otrzymany komunikat w metodzie `receiveSpittleAlert()`, jest skorzystanie z konwertera komunikatów. Przekonaliśmy się już, jak te konwertery mogą przekształcać nasze obiekty do postaci obiektów `Message` w metodzie `convertAndSend()`. Okazuje się jednak, że można ich także używać podczas odbierania komunikatów za pomocą metody `receiveAndConvert()`:

```
public Spittle retrieveSpittleAlert() {
  return (Spittle) jmsOperations.receiveAndConvert();
}
```

Teraz nie musimy rzutować obiektów `Message` na `ObjectMessage`, pobierać obiektów `Spittle` za pomocą metody `getObject()` ani zawracać sobie głowy kontrolowanymi wyjątkami `JMSException`. Nowa wersja metody `retrieveSpitterAlert()` jest znacznie bardziej przejrzysta. Wciąż jednak występuje pewien, niezbyt oczywisty, problem.

Dużym minusem konsumpcji komunikatów za pomocą szablonu `JmsTemplate` jest synchroniczność metod `receive()` oraz `receiveAndConvert()`. Oznacza ona, że odbiorca musi cierpliwie czekać na nadejście komunikatu, jako że metoda `receive()` wstrzyma wykonanie do momentu odebrania komunikatu (lub przekroczenia limitu czasowego). Czyż nie jest dziwne, że konsumujemy synchronicznie komunikat, który został wysłany asynchronicznie?

Tu właśnie przydadzą nam się obiekty POJO sterowane komunikatami. Zobaczmy, jak odbierać komunikaty asynchronicznie, wykorzystując komponenty, które reagują na komunikaty, zamiast na nie czekać.

17.2.3. Tworzenie obiektów POJO sterowanych komunikatami

Pewnego lata, będąc jeszcze w college'u, miałem przyjemność pracować w Parku Narodowym Yellowstone. Posada nie była tak prestiżowa jak strażnik parku czy człowiek sterujący z ukrycia wybuchami gejzeru Old Faithful[2]. Zamiast tego wykonywałem czynności pomocy domowej, zmieniając prześcieradła, czyszcząc łazienki i odkurzając podłogę. Mało ekskluzywne zajęcie, ale przynajmniej dane mi było pracować w jednym z najpiękniejszych zakątków świata.

Każdego dnia po pracy udawałem się do lokalnej placówki pocztowej sprawdzić, czy nie nadeszła nowa korespondencja. Przebywałem poza domem przez kilka tygodni i zawsze było miło dostać list albo kartkę od szkolnych przyjaciół. Nie miałem własnej skrzynki pocztowej, podchodziłem więc do człowieka za ladą i pytałem, czy przyszło coś do mnie. Wtedy zaczynało się oczekiwanie.

Widzisz, mężczyzna ten miał, na oko, jakieś 195 lat. Jak przystało na człowieka w tym wieku, poruszał się bardzo wolno, o ile w ogóle. Podnosił się wtedy dostojnie z krzesła i ciężko powłócząc nogami, kierował się w stronę zaplecza. Po kilku chwilach wracał równie ślamazarnym krokiem, opadając z powrotem na krzesło. Wtedy patrzył na mnie i mówił: „Nie ma dziś listów do pana".

Metoda `receive()` szablonu JMS jest niczym ten leciwy pracownik poczty. Po wywołaniu odchodzi i szuka komunikatów w kolejce lub temacie i nie kończy się, dopóki nie nadejdzie komunikat albo nie zostanie przekroczony limit czasowy. Tymczasem aplikacja pozostaje bezczynna, czekając na komunikat. Czy nie byłoby lepiej, gdyby aplikacja mogła kontynuować działanie i zostać powiadomiona w momencie nadejścia komunikatu?

Jedną z najważniejszych nowości specyfikacji EJB 2 było włączenie do niej **komponentu sterowanego komunikatami** (ang. *message-driven bean*, w skrócie MDB). MDB

[2] Jeden z najpopularniejszych gejzerów w Parku Narodowym Yellowstone — *przyp. tłum.*

są komponentami EJB przetwarzającymi komunikaty asynchronicznie. Innymi słowy, komponenty MDB reagują na komunikaty w miejscu docelowym JMS jak na zdarzenia i odpowiadają na te zdarzenia. W przeciwieństwie do synchronicznych odbiorców komunikatów, wstrzymujących wykonanie do czasu nadejścia komunikatu.

Komponenty MDB były jasnymi punktami EJB. Nawet najzagorzalsi krytycy EJB doceniali ich elegancję przy obsłudze komunikatów. Jedyną ich niedoskonałością była konieczność implementowania przez nie interfejsu `javax.ejb.MessageDrivenBean`. To z kolei pociągało za sobą konieczność implementacji kilku metod zwrotnych cyklu życia EJB. Mówiąc najogólniej, komponenty MDB EJB 2 nie przypominały w niczym obiektów POJO.

W specyfikacji EJB 3 komponentom MDB nadano charakter bardziej zbliżony do obiektów POJO. Nie muszą już implementować interfejsu `MessageDrivenBean`. Zamiast niego implementują bardziej uniwersalny `javax.jms.MessageListener` i korzystają z adnotacji `@MessageDriven`.

Spring 2.0 rozwiązuje problem asynchronicznej konsumpcji komunikatów, dostarczając swój własny rodzaj komponentów sterowanych komunikatami, podobnych do komponentów MDB specyfikacji EJB 3. W tym podrozdziale pokażemy, jak Spring obsługuje asynchroniczną konsumpcję komunikatów za pomocą **obiektów POJO sterowanych komunikatami** (ang. *message-driven POJO*, w skrócie MDP).

TWORZENIE ODBIORCY KOMUNIKATÓW

Gdybyśmy chcieli zbudować klasę obsługi powiadomień o spittle'ach za pomocą modelu sterowanego komunikatami EJB, musiałaby ona posiadać adnotację `@MessageDriven`. I, chociaż nie jest to bezwzględnie wymagane, zaleca się, by komponent MDB implementował interfejs `MessageListener`. Całość mogłaby wyglądać następująco:

```
@MessageDriven(mappedName="jms/spittle.alert.queue")
  public class SpittleAlertHandler implements MessageListener {

  @Resource
  private MessageDrivenContext mdc;

  public void onMessage(Message message) {
    ...
  }
}
```

Spróbuj sobie wyobrazić przez chwilę lepszy świat, w którym komponenty sterowane komunikatami nie muszą implementować interfejsu `MessageListener`. W takim świecie słońce świeciłoby jaśniej, ptaki zawsze nuciły Twoją ulubioną melodię, a Ty nie musiałbyś implementować metody `onMessage()` ani wstrzykiwać `MessageDrivenContext`.

Być może wymagania narzucone komponentom MDB przez specyfikację EJB 3 nie są aż takie kłopotliwe. Nie da się jednak zaprzeczyć, że implementacja `SpitterAlert`
`↪Handler` na bazie EJB 3 jest zbyt przywiązana do sterowanego komunikatami API EJB, co odróżnia ją od obiektów POJO. Chcielibyśmy, żeby klasa obsługi powiadomień była w stanie obsłużyć komunikaty, ale jednocześnie nie była skonstruowana tak, jak gdyby wiedziała, że będzie to robić.

Spring umożliwia obsługę komunikatów z kolejki lub tematu JMS przez metodę POJO. Na listingu 17 5 pokazano przykład implementacji obiektu POJO `SpittleAlert➥Handler`.

Listing 17.5. Obiekt MDP Springa asynchronicznie odbiera i przetwarza komunikaty

```
package com.habuma.spittr.alerts;
import com.habuma.spittr.domain.Spittle;

public class SpittleAlertHandler {

   public void processSpittle(Spittle spittle) {   <-- Metoda obsługi
      // ...tutaj implementacja...
   }
}
```

Chociaż jasność słońca i tresura ptaków leżą poza zasięgiem Springa, listing 17.5 pokazuje, że lepszy świat, który opisałem, jest do pewnego stopnia realny. Wnętrze metody `processSpittle()` uzupełnimy później. Teraz zauważ, że w tej wersji `Spitter➥AlertHandler` nie ma najmniejszego śladu JMS. Jest to obiekt POJO w pełnym tego słowa znaczeniu. Mimo to potrafi obsłużyć komunikaty w takim samym stopniu, jak jego odpowiednik oparty na EJB. Potrzebuje tylko trochę dodatkowej konfiguracji Springa.

KONFIGURACJA ODBIORCÓW KOMUNIKATÓW

Kluczem do uwolnienia zdolności POJO do odbierania komunikatów jest jego konfiguracja jako odbiorcy komunikatów w Springu. Przestrzeń `jms` Springa ma wszystko, co jest nam do tego potrzebne. Na początek musimy zadeklarować klasę obsługi jako `<bean>`:

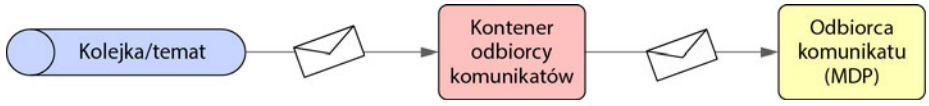
```
<bean id="spittleHandler"
      class="com.habuma.spittr.alerts.SpittleAlertHandler" />
```

Następnie, aby zmienić `SpittleAlertHandler` w sterowany komunikatami obiekt POJO, możemy zadeklarować, że komponent ten jest odbiorcą komunikatów:

```
<jms:listener-container connection-factory="connectionFactory">
  <jms:listener destination="spitter.alert.queue"
        ref="spittleHandler" method="handleSpittleAlert" />
</jms:listener-container>
```

Mamy tutaj odbiorcę komunikatów, który zawarty jest w kontenerze odbiorcy komunikatów. **Kontener odbiorcy komunikatów** (ang. *message listener container*) jest specjalnym komponentem, którego zadanie polega na obserwacji miejsca docelowego JMS w oczekiwaniu na komunikat. Kiedy komunikat się pojawia, jest pobierany i przekazywany do wszystkich zainteresowanych odbiorców. Działanie kontenera odbiorcy komunikatów zilustrowano na rysunku 17.7.

Do konfiguracji kontenera odbiorcy komunikatów oraz odbiorcy komunikatów w Springu używamy dwóch elementów przestrzeni nazw `jms` Springa. Element `<jms:lis➥tener-container>` zawiera elementy `<jms:listener>`. W jego atrybucie `connectionFactory` podano referencję do obiektu `connectionFactory`, która zostanie użyta przez elementy



Rysunek 17.7. Kontener odbiorcy komunikatów oczekuje komunikatu z kolejki/tematu. Kiedy komunikat się pojawia, przekazywany jest do odbiorcy komunikatów (na przykład do sterowanego komunikatami obiektu POJO)

<jms:listener> podczas odbierania komunikatów. W tym przypadku atrybut connection ➥factory mógł zostać pominięty, jako że domyślnie przyjmuje on wartość connec ➥tionFactory.

Element <jms:listener> jest z kolei używany do identyfikacji komponentu i metody, które powinny obsłużyć przychodzące komunikaty. Aby obsługa komunikatów powiadomień o spittle'ach była możliwa, atrybut ref odnosi się do komponentu spittleHandler. Kiedy komunikat pojawia się w kolejce spitter.alert.queue (atrybut destination określa miejsce docelowe), wywołana zostaje metoda processSpittle() komponentu spittle ➥Handler (co określono w atrybucie method).

Warto także zauważyć, że jeśli komponent określony w atrybucie ref implementuje interfejs MessageListener, to nie trzeba określać atrybutu method. Domyślnie zostanie wywołana metoda onMessage().

17.2.4. Używanie RPC opartego na komunikatach

W rozdziale 15. omówiliśmy szereg opcji Springa w zakresie udostępniania metod komponentów jako zdalnych usług i wywołań tych usług z poziomu aplikacji klienckich. W tym rozdziale dowiedzieliśmy się, jak przesyłać komunikaty pomiędzy aplikacjami przez kolejki i tematy. Teraz spróbujemy połączyć te rozwiązania w jedno i użyć zdalnych wywołań, które wykorzystują do transportu JMS.

W celu obsługi RPC opartego na komunikatach Spring udostępnia JmsInvokerService ➥Exporter do eksportowania komponentów jako usług sterowanych komunikatami oraz JmsInvokerProxyFactoryBean dla klientów konsumujących te usługi. Jak się wkrótce przekonamy, rozwiązania te są bardzo podobne, ale i jedno, i drugie ma swoje wady i zalety. Zaprezentuję oba podejścia, a Ty zdecydujesz, które jest najlepsze dla Ciebie. Na początek przyjrzymy się usługom wykorzystującym JMS i ich obsłudze przez Springa.

Jak zapewne pamiętasz z rozdziału 15., Spring udostępnia kilka możliwości eksportowania komponentów jako usług zdalnych. Używaliśmy RmiServiceExporter, by wyeksportować komponent jako usługi RMI, HessianExporter oraz BurlapExporter, by tworzyć odpowiednio usługi oparte na protokołach Hessian i Burlap, oraz HttpInvoker ➥ServiceExporter, by tworzyć usługi obiektu wywołującego HTTP. Jednak Spring oferuje jeszcze jeden eksporter usługi, o którym nie wspominałem w rozdziale 15.

EKSPORTOWANIE USŁUG BAZUJĄCYCH NA JMS

JmsInvokerServiceExporter w bardzo dużym stopniu przypomina inne eksportery, które przedstawiłem w rozdziale 15. Warto zwrócić uwagę na symetrię nazw: JmsInvokerService ➥Exporter oraz HttpInvokerServiceExporter. Skoro HttpInvokerServiceExporter eksportuje usługi komunikujące się przy użyciu protokołu HTTP, oznacza to, że JmsInvoker ➥ServiceExporter musi eksportować usługi porozumiewające się za pomocą JMS.

Aby zobaczyć, jak działa eksporter `JmsInvokerServiceExporter`, przeanalizujmy klasę `AlertServiceImpl`, zaprezentowaną na listingu 17.6.

Listing 17.6. AlertServiceImpl: wolny od JMS obiekt POJO obsługujący komunikaty JMS JSM-free

```
package com.habuma.spittr.alerts;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.stereotype.Component;
import com.habuma.spittr.domain.Spittle;

@Component("alertService")
public class AlertServiceImpl implements AlertService {

  private JavaMailSender mailSender;
  private String alertEmailAddress;

  public AlertServiceImpl(JavaMailSender mailSender,
                          String alertEmailAddress) {
    this.mailSender = mailSender;
    this.alertEmailAddress = alertEmailAddress;
  }

  public void sendSpittleAlert(final Spittle spittle) {   <-- Wysyła powiadomienie
    SimpleMailMessage message = new SimpleMailMessage();
    String spitterName = spittle.getSpitter().getFullName();
    message.setFrom("noreply@spitter.com");
    message.setTo(alertEmailAddress);
    message.setSubject("Nowy spittle od " + spitterName);
    message.setText(spitterName + " mówi: " + spittle.getText());
    mailSender.send(message);
  }

}
```

Na razie nie warto zaprzątać sobie zbyt mocno głowy szczegółami działania metody `sendSpittleAlert()`. Więcej informacji na temat wysyłania wiadomości pocztą elektroniczną przy użyciu Springa podam później, w rozdziale 20. W tym przypadku najważniejszą rzeczą, na którą należy zwrócić uwagę, jest to, że `AlertServiceImpl` jest zwyczajną klasą obiektów POJO, a w jej kodzie nie ma niczego, co mogłoby sugerować, iż będzie używana do obsługi komunikatów JMS. Klasa ta, co pokazałem poniżej, implementuje interfejs `AlertService`:

```
package com.habuma.spittr.alerts;
import com.habuma.spittr.domain.Spittle;
public interface AlertService {
  void sendSpittleAlert(Spittle spittle);
}
```

Jak widać, klasa `AlertServiceImpl` została opatrzona adnotacją `@Component`, dzięki czemu zostanie automatycznie wykryta i zarejestrowana pod identyfikatorem `alertService` jako komponent w kontekście aplikacji Springa. Do tego komponentu możemy się odwołać podczas konfigurowania eksportera `JmsInvokerServiceExporter` w następujący sposób:

```
<bean id="alertServiceExporter"
    class="org.springframework.jms.remoting.JmsInvokerServiceExporter"
    p:service-ref="alertService"
    p:serviceInterface="com.habuma.spittr.alerts.AlertService" />
```

Właściwości tego komponentu mówią nam o tym, jak eksportowana usługa powinna wyglądać. Właściwość `service` odnosi się do komponentu `alertService`, który jest implementacją zdalnej usługi. Właściwość `serviceInterface` przyjmuje tymczasem jako wartość pełną nazwę oferowanego przez usługę interfejsu.

Właściwości eksportera milczą na temat sposobu transportu usługi przez JMS. Dobrą wiadomością jest jednak fakt, że `JmsInvokerServiceExporter` można zakwalifikować jako odbiorcę JMS. Możemy go zatem skonfigurować w elemencie `<jms:lis ↪tener-container>`:

```
<jms:listener-container connection-factory="connectionFactory">
   <jms:listener destination="spitter.alert.queue"
                 ref="alertServiceExporter" />
</jms:listener-container>
```

Kontenerowi odbiorcy JMS przekazujemy fabrykę połączeń, aby wiedział, jak połączyć się z brokerem komunikatów. Deklaracja `<jms:listener>` tymczasem zawiera miejsce docelowe zdalnego komunikatu.

KONSUMOWANIE USŁUG BAZUJĄCYCH NA JMS

Na tym etapie usługa powiadomień bazująca na JMS powinna być już gotowa i czekać, aż w kolejce o nazwie `spitter.alert.queue` pojawią się komunikaty RPC. Po stronie klienta dostęp do usługi zapewni `InvokerProxyFactoryBean`.

`JmsInvokerProxyFactoryBean` niewiele różni się od pozostałych komponentów fabryk obiektów pośredniczących w zdalnym dostępie, omówionych przez nas w rozdziale 15. Ukrywa szczegóły dostępu do zdalnej usługi za wygodnym interfejsem, poprzez który klient komunikuje się z usługą. Największa różnica polega na tym, że zamiast pośredniczyć w dostępie do usług opartych na RMI czy HTTP, `JmsInvokerProxyFactoryBean` pośredniczy w dostępie do usług JMS, eksportowanych przez `JmsInvokerServiceExporter`.

Aby skonsumować usługę powiadomień, możemy dowiązać komponent `JmsInvoker ↪ProxyFactoryBean` w następujący sposób:

```
<bean id="alertService"
  class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean"
  p:connectionFactory-ref="connectionFactory"
  p:queueName="spittle.alert.queue"
  propp:serviceInterface="com.habuma.spittr.alerts.AlertService" />
```

Właściwości `connectionFactory` i `queueName` określają sposób dostarczania komunikatów RPC — w tym przypadku z udziałem kolejki o nazwie `spitter.alert.queue` i brokera komunikatów, skonfigurowanego w określonej fabryce połączeń. Właściwość `ser ↪viceInterface` wskazuje natomiast, że obiekt pośredniczący powinien zostać udostępniony poprzez interfejs `AlertService`.

Przez wiele lat JMS był optymalnym rozwiązaniem pozwalającym na stosowanie komunikatów w aplikacjach pisanych w Javie. Jednak nie jest to jedyne narzędzie służące

do wymiany komunikatów dostępne dla programistów używających Javy i Springa. W ciągu ostatnich kilku lat bardzo duże uznanie zdobył także **Advanced Message Queuing Protocol** (AMQP, zaawansowany protokół kolejkowania komunikatów). Jak się okazuje, Spring zapewnia wsparcie dla wysyłania komunikatów przy jego użyciu, o czym przekonasz się w następnym podrozdziale.

17.3. Obsługa komunikatów przy użyciu AMQP

Być może zastanawiasz się, do czego jest nam potrzebna kolejna specyfikacja wymiany komunikatów. Czy JMS nie jest dostatecznie dobry? Co takiego wnosi AMQP, czego wcześniej nie miał JMS?

Jak się okazuje, AMQP w porównaniu z JMS wypada pod kilkoma względami lepiej. Przede wszystkim AMQP jest protokołem warstwy połączenia (ang. *wire-level protocol*), natomiast JMS definiuje specyfikację API. Specyfikacja ta zapewnia, że wszystkie implementacje JMS będą mogły być stosowane przy użyciu wspólnego API, nie gwarantuje jednak, że komunikaty wysyłane z jednej implementacji JMS będą mogły być konsumowane przez inne implementacje. Z drugiej strony protokół AMQP określa format, w jakim zostanie zapisany komunikat przesyłany pomiędzy producentem a konsumentem. W konsekwencji AMQP zapewnia większe możliwości współdziałania niż JMS — obejmują one bowiem nie tylko różne implementacje AMQP, lecz także inne języki i platformy[3].

Kolejną zaletą AMQP, której nie posiada JMS, jest to, że ma on znacznie bardziej elastyczny i transparentny model obsługi komunikatów. W przypadku JMS istnieją dwa modele obsługi komunikatów: punkt-punkt oraz publikacja-subskrypcja. Oba ten modele są także dostępne w AMQP, jednak AMQP pozwala dodatkowo na stosowanie różnych sposobów kierowania komunikatów, a zapewnia je poprzez oddzielenie producenta komunikatów od kolejki (lub kolejek), w której dany komunikat ma zostać umieszczony.

Spring AMQP jest rozszerzeniem Spring Framework, pozwalającym na obsługiwanie komunikatów w aplikacjach Spring w stylu charakterystycznym dla AMQP. Jak się niebawem przekonasz, Spring AMQP udostępnia API, dzięki któremu korzystanie z AMQP staje się bardzo podobne do stosowania dostępnej w Springu abstrakcji JMS. To z kolei oznacza, że będziemy mogli skorzystać ze znacznej części zamieszczonych wcześniej informacji o JMS, aby ułatwić sobie zrozumienie sposobów wysyłania i odbierania komunikatów przy użyciu Spring AMQP.

Już wkrótce zobaczysz, jak można stosować Spring AMQP. Zanim jednak zajmiemy się szczegółami wysyłania i odbierania komunikatów AMQP w Springu, warto się dowiedzieć, jak działa AMQP.

---

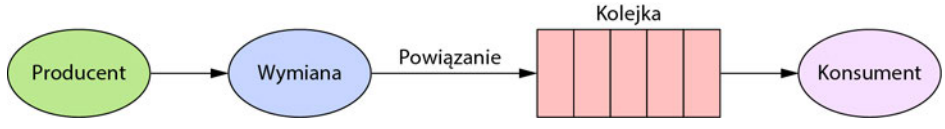
[3] Jeśli czytając to, pomyślałeś, że AMQP wykracza poza język Java i jego platformę, to doskonale zrozumiałeś, o co chodzi.

17.3.1. Krótkie wprowadzenie do AMQP

W zrozumieniu modelu obsługi komunikatów wykorzystywanego w AMQP może pomóc przypomnienie modelu używanego w JMS. W przypadku JMS w wymianie komunikatów bierze udział trzech głównych uczestników: producent, konsument oraz kanał (kolejka lub temat), którym komunikaty są przekazywane od producenta do konsumenta. Te trzy podstawowe elementy modelu wymiany komunikatów JMS zostały przedstawione na rysunkach 17.3 i 17.4.

W JMS kanał pomaga w oddzieleniu producenta od konsumenta, jednak oba te elementy są ściśle powiązane z samym kanałem. Producent publikuje swoje komunikaty do określonej kolejki bądź tematu i analogicznie konsument pobiera komunikaty ze ściśle określonej kolejki lub tematu. Kanał wykonuje zatem podwójne zadanie: dostarcza komunikaty oraz określa, gdzie będą one przekazywane; kolejki dostarczają komunikaty według algorytmu punkt-punkt, a tematy wykorzystują model publikacja-subskrypcja.

Natomiast w przypadku AMQP producenci nie publikują komunikatów bezpośrednio w kolejce. AMQP wprowadza bowiem dodatkowy poziom, oddzielający producenta od kolejki, która będzie przekazywać komunikaty. Chodzi o tak zwaną **wymianę** (ang. *exchange*). Zależności pomiędzy tymi wszystkimi elementami zostały zilustrowane na rysunku 17.8.



Rysunek 17.8. W AMQP producenci są odseparowani od kolejek przez wymianę, która zajmuje się kierowaniem komunikatów

Jak widać, producent publikuje komunikaty do wymiany. Z kolei wymiana, powiązana z jedną lub kilkoma kolejkami, kieruje komunikaty do odpowiedniej kolejki (bądź kolejek). Konsumenci pobierają komunikaty z kolejki i przetwarzają je.

Na rysunku 17.8 nie widać natomiast, że działanie wymiany nie jest zwyczajnym przekazywaniem komunikatu do kolejki. AMQP definiuje cztery różne typy wymian, z których każdy posiada własny algorytm kierowania komunikatów, określający, czy należy je umieszczać w kolejce. W zależności od algorytmu wymiany pod uwagę mogą być brane **klucz trasowania** (ang. *routing key*) oraz argumenty, które są następnie porównywane z kluczem trasowania i argumentami powiązania pomiędzy wymianą i kolejką. (Klucz trasowania można sobie w przybliżeniu wyobrazić jako adres podawany w wiadomości z poczty elektronicznej i określający jej odbiorcę). Jeśli algorytm zaakceptuje porównanie, komunikat zostanie skierowany do danej kolejki. W przeciwnym razie nie trafi do niej.

Poniżej przedstawione zostały cztery standardowe typy wymian AMQP:

- Wymiana typu *direct* (bezpośrednia) — komunikat zostanie skierowany do kolejki, jeżeli jego klucz trasowania bezpośrednio odpowiada kluczowi w powiązaniu.
- Wymiana typu *topic* (temat) — komunikat zostanie skierowany do kolejki, jeśli jego klucz trasowania będzie pasował do klucza w powiązaniu przy wykorzystaniu dopasowywania z użyciem znaków wieloznacznych.

- Wymiana typu *headers* (nagłówki) — komunikat zostanie skierowany do kolejki, jeżeli nagłówki i wartości umieszczone w tablicy argumentów będą odpowiadać nagłówkom i wartościom dostępnym w tablicy argumentów powiązania. Przy użyciu specjalnego nagłówka o nazwie `x-match` można określić, czy wszystkie (`all`) wartości muszą sobie odpowiadać, czy też wystarczy, by pasowała dowolna (`any`) z nich.
- Wymiana typu *fanout* (do wszystkich) — komunikat zostanie wysłany do wszystkich kolejek powiązanych z wymianą, niezależnie od klucza trasowania czy nagłówków i wartości zapisanych w tablicy argumentów.

Dysponując tymi czterema typami wymian, nie trudno wyobrazić sobie, jak można zdefiniować wiele różnych schematów kierowania komunikatów, znacznie wykraczających poza proste modele punkt-punkt oraz publikacja-subskrypcja[4]. Na szczęście okazuje się, że te różne algorytmy kierowania komunikatów mają bardzo mały wpływ na sposób implementacji producentów i konsumentów komunikatów. Najprościej rzecz ujmując, producent publikuje komunikat do wymiany o określonym kluczu, a konsument odbiera komunikat z kolejki.

Na tym się kończy krótkie wprowadzenie do wymiany komunikatów przy użyciu AMQP — powinieneś już dysponować wystarczająca wiedzą, by móc wysyłać i odbierać komunikatu za pomocą Spring AMQP. Zachęcam jednak do dokładniejszego poznania zagadnień związanych z AMQP, a konkretnie do lektury specyfikacji oraz pozostałych materiałów dostępnych na stronie *www.amqp.org* bądź do sięgnięcia po książkę *Rabbit in Action* napisaną przez Alvara Videlę i Jasona J.W. Williamsa (wydaną w 2012 roku przez wydawnictwo Manning, *www.manning.com/videla/*).

A teraz zostawmy te abstrakcyjne rozważania o możliwościach AMQP i zajmijmy się pisaniem kodu, który będzie wysyłał i odbierał komunikaty przy użyciu Spring AMQP. Zaczniemy od przedstawienia wspólnej konfiguracji Spring AMQP, wymaganej zarówno przez producentów, jak i konsumentów wiadomości.

17.3.2. Konfigurowanie Springa do wymiany komunikatów przy użyciu AMQP

Kiedy zaczynaliśmy używać JMS w Springu, pierwszą wykonaną czynnością było skonfigurowanie fabryki połączeń. Podobnie jest w przypadku AMQP. Choć oczywiście tym razem zamiast konfigurować fabrykę połączeń JMS, skonfigurujemy fabrykę połączeń AMQP. A konkretnie rzecz biorąc, skonfigurujemy fabrykę połączeń RabbitMQ.

Najprostszym sposobem skonfigurowania fabryki połączeń RabbitMQ jest skorzystanie z konfiguracyjnej przestrzeni nazw `rabbit`, udostępnianej przez Spring AMQP. W tym celu koniecznie trzeba zadbać o to, by w konfiguracyjnym pliku XML Springa znalazła się odpowiednia deklaracja schematu:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/rabbit"
  xmlns:beans="http://www.springframework.org/schema/beans"
```

[4] A nawet nie wspomniałem o tym, że można powiązać jedną wymianę z innymi, tworząc w ten sposób zagnieżdżoną hierarchię wymian.

Czym jest RabbitMQ?

RabbitMQ jest popularnym, otwartym brokerem komunikatów implementującym AMQP. Spring AMQP zapewnia możliwości korzystania z RabbitMQ i zawiera fabrykę połączeń RabbitMQ, szablon oraz konfiguracyjną przestrzeń nazw.

Zanim będzie można wysyłać i odbierać komunikaty przy użyciu RabbitMQ, należy je zainstalować. Instrukcje dotyczące instalacji można znaleźć na stronie *http://www.rabbitmq.com/download.html*. Różnią się one w zależności od stosowanego systemu operacyjnego, dlatego sam będziesz musiał je znaleźć i wykonać.

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/rabbit
    http://www.springframework.org/schema/rabbit/spring-rabbit-1.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
...
</beans:beans>
```

Choć nie jest to konieczne, to w powyższym przykładzie zdecydowałem się zadeklarować przestrzeń nazw `rabbit` jako główną, natomiast przestrzeń nazw `bean` jako drugorzędną. Przyjęte rozwiązanie wynika z faktu, że przewiduję, iż w pliku konfiguracyjnym znajdzie się więcej elementów związanych z komunikatami RabbitMQ niż z komponentami. Dlatego wolę poprzedzić prefiksem `beans:` te kilka komponentów, a uniknąć dodawania prefiksów do elementów związanych z komunikatami.

Przestrzeń nazwy `rabbit` zawiera kilka elementów służących do konfigurowania obsługi RabbitMQ w Springu. Na obecnym etapie prac najbardziej interesuje nas element `<connection-factory>`. Poniżej przedstawiona została najprostsza postać konfiguracji fabryki połączeń RabbitMQ:

```
<connection-factory/>
```

Choć taki sposób konfiguracji zadziała, to jednak utworzony komponent fabryki połączeń nie będzie dysponował żadnym identyfikatorem, przez co trudno go będzie powiązać z jakimś innym komponentem, który będzie go potrzebował. Dlatego najprawdopodobniej będziemy chcieli określić identyfikator komponentu, używając w tym celu atrybutu `id`:

```
<connection-factory id="connectionFactory" />
```

Domyślnie fabryka połączeń zakłada, że serwer RabbitMQ nasłuchuje połączeń przy wykorzystaniu adresu `localhost` i portu `5672`, a nazwa użytkownika i hasło dostępu mają postać *guest*. Te ustawienia domyślne są wystarczające dla środowiska służącego do pisania aplikacji, ale w przypadku środowisk produkcyjnych zapewne będziemy chcieli je zmienić. Poniższy element `<connection-factory>` zmienia te ustawienia domyślne:

```
<connection-factory id="connectionFactory"
  host="${rabbitmq.host}"
  port="${rabbitmq.port}"
  username="${rabbitmq.username}"
  password="${rabbitmq.password}" />
```

Do określenia wartości zastosowaliśmy tutaj symbole zastępcze, dzięki czemu działanie aplikacji będzie można określać poza plikami konfiguracyjnymi Springa (na przykład w plikach właściwości).

Oprócz fabryki połączeń istnieje także kilka innych elementów konfiguracyjnych, z których pewnie zechcemy skorzystać. Przekonajmy się, jak skonfigurować Spring AMQP, aby leniwie tworzyć kolejki, wymiany i powiązania.

DEKLAROWANIE KOLEJEK, WYMIAN I POWIĄZAŃ

W odróżnieniu od JMS, w którym sposób kierowania komunikatów przez kolejki i tematy jest ściśle określony przez specyfikację, model kierowania komunikatów stosowany w AMQP jest bogatszy i bardziej elastyczny. Dlatego to do nas należy zdefiniowanie kolejek i wymian oraz określenie, jak będą one ze sobą powiązane. Jednym ze sposobów deklarowania kolejek, wymian i powiązań jest korzystanie z wielu metod interfejsu `Channel`. Jednak używanie go bezpośrednio jest dosyć złożone. A zatem czy Spring AMQP może nam pomóc w deklarowaniu komponentów związanych z wymianą komunikatów?

Na szczęście przestrzeń nazw `rabbit` deklaruje kilka elementów, których można używać do deklarowania kolejek i wymian oraz ich wzajemnych powiązań. Elementy te zostały przedstawione w tabeli 17.3.

Tabela 17.3. Przestrzeń rabbit Spring AMQP deklaruje kilka elementów służących do leniwego tworzenia kolejek, wymian i powiązań pomiędzy nimi

Element	Przeznaczenie
`<queue>`	Tworzy kolejkę.
`<fanout-exchange>`	Tworzy wymianę typu *fanout*.
`<header-exchange>`	Tworzy wymianę typu *headers*.
`<topic-exchange>`	Tworzy wymianę typu *topic*.
`<direct-exchange>`	Tworzy wymianę typu *direct*.
`<bindings> <binding/> </bindings>`	Element `<bindings>` definiuje zestaw elementów `<binding>`. Z kolei element `<binding>` tworzy powiązanie pomiędzy wymianą i kolejką.

Te elementy konfiguracyjne są stosowane wraz z elementem `<admin>`. Element ten tworzy administracyjny komponent RabbitMQ, który automatycznie tworzy (w brokerze RabbitMQ, o ile jeszcze nie istnieją) wszelkie kolejki, wymiany i powiązania zadeklarowane przy użyciu elementów zaprezentowanych w tabeli 17.3.

Na przykład aby zadeklarować kolejkę o nazwie `spittle.alert.queue`, wystarczy dodać do pliku konfiguracyjnego Springa dwa poniższe elementy:

```
<admin connection-factory="connectionFactory" />
<queue id="spittleAlertQueue" name="spittle.alerts" />
```

Na potrzeby prostej wymiany komunikatów taka konfiguracja w zupełności wystarczy. Dzieje się tak, gdyż istnieje domyślna wymiana typu *direct*, która nie ma żadnego klucza trasowania. Z wymianą tą są powiązane wszystkie kolejki, przy czym używany przez

nie klucz trasowania odpowiada nazwie kolejki. Dzięki tej prostej konfiguracji możemy wysyłać komunikaty do domyślnej, pozbawionej nazwy wymiany i zastosować jedynie klucz `spittle.alert.queue`, by komunikaty trafiły do naszej kolejki. W efekcie rozwiązanie to stanowi odpowiednik modelu punkt-punkt rozsyłania komunikatów w JMS.

Bardziej interesujące sposoby wymiany komunikatów będą jednak wymagały zadeklarowania jednej lub kilku wymian i powiązania ich z kolejkami. Na przykład żeby komunikaty były kierowane do wielu kolejek bez względu na klucze trasowania, można zastosować wymianę typu *fanout* oraz kilka kolejek:

```
<admin connection-factory="connectionFactory" />
<queue name="spittle.alert.queue.1" />
<queue name="spittle.alert.queue.2" />
<queue name="spittle.alert.queue.3" />
<fanoutexchange name="spittle.fanout">
  <bindings>
    <binding queue="spittle.alert.queue.1" />
    <binding queue="spittle.alert.queue.2" />
    <binding queue="spittle.alert.queue.3" />
  </bindings>
</fanoutexchange>
```

Dzięki elementom z tabeli 17.3 istnieje niemal nieskończenie wiele sposobów na skonfigurowanie wymiany komunikatów w RabbitMQ. Jednak w tej książce nie dysponuję nieskończenie wieloma stronami, by opisać wszystkie te sposoby konfiguracji. Dlatego aby kontynuować prezentowanie Spring AMQP, pozostawię Ci możliwość wykazania się kreatywnością w zakresie konfigurowania wymiany komunikatów, a sam przejdę do opisu sposobu ich wysyłania.

17.3.3. Wysyłanie komunikatów przy użyciu RabbitTemplate

Zgodnie z tym, co sugeruje nazwa, fabryka połączeń RabbitMQ służy do tworzenia połączeń z serwerem RabbitMQ. Gdybyśmy chcieli go użyć do wysyłania komunikatów, to *moglibyśmy* wstrzyknąć do naszej klasy `AlertServiceImpl` właściwość `connection ↪Factory`, następnie zastosować to połączenie do utworzenia obiektu `Channel`, a ten z kolei do wysłania komunikatu do wymiany.

No tak… *moglibyśmy* tak zrobić.

Jednak wymagałoby to dużego nakładu pracy i konieczności napisania rozbudowanego, wtórnego kodu. A wtórny kod jest jedną z rzeczy, których Spring nie znosi. Poznaliśmy już kilka przykładów rozwiązań, w których Spring udostępniał szablony pozwalające na zminimalizowanie niepotrzebnego powielania kodu. Jednym z nich był, przedstawiony we wcześniejszej części rozdziału, szablon `JmsTemplate`, umożliwiający wyeliminowanie wtórnego kodu związanego z obsługą komunikatów JMS. Nie powinno zatem być wielkim zaskoczeniem, że Spring AMQP udostępnia szablon `RabbitTemplate`, który eliminuje wtórny kod związany z wysyłaniem i odbieraniem komunikatów przy użyciu RabbitMQ.

Najprostszym sposobem konfiguracji szablonu `RabbitTemplate` jest skorzystanie z elementu `<template>` dostępnego w konfiguracyjnej przestrzeni nazw `rabbit`:

```
<template id="rabbitTemplate"
    connection-factory="connectionFactory" />
```

Teraz, aby wysłać wiadomość, wystarczy tylko wstrzyknąć komponent szablonu do obiektu `AlertServiceImpl` i użyć go do wysłania obiektu `Spittle`. Listing 17.7 przedstawia nową wersję klasy `AlertServiceImpl`, która wysyła komunikat z obiektem `Spittle`, wykorzystując w tym celu szablon `RabbitTemplate`, a nie, jak było wcześniej, szablon `JmsTemplate`.

Listing 17.7. Wysyłanie komunikatu z obiektem Spittle przy użyciu szablonu RabbitTemplate

```
package com.habuma.spitter.alerts;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;

import com.habuma.spitter.domain.Spittle;

public class AlertServiceImpl implements AlertService {

  private RabbitTemplate rabbit;

  @Autowired
  public AlertServiceImpl(RabbitTemplate rabbit) {
    this.rabbit = rabbit;
  }

  public void sendSpittleAlert(Spittle spittle) {
    rabbit.convertAndSend("spittle.alert.exchange",
                          "spittle.alerts",
                          spittle);
  }
}
```

Jak widać, tym razem metoda `sendSpittleAlert()` wywołuje metodę `convertAndSend()` wstrzykniętego szablonu `RabbitTemplate`. Do tej metody są przekazywane trzy parametry: nazwa wymiany, klucz trasowania oraz obiekt, który należy wysłać. Warto zwrócić uwagę na to, że w żaden sposób nie jest natomiast określane, gdzie komunikat zostanie skierowany, do jakich kolejek ma trafić ani jacy konsumenci mają go otrzymać.

Szablon `RabbitTemplate` definiuje kilka przeciążonych wersji metody `convertAndSend()`, ułatwiających jego stosowanie. Na przykład jedna z przeciążonych wersji tej metody pozwala na pominięcie nazwy wymiany w wywołaniu:

```
rabbit.convertAndSend("spittle.alerts", spittle);
```

Z kolei inna wersja metody pozwala na pominięcie zarówno nazwy wymiany, jak i klucza trasującego:

```
rabbit.convertAndSend(spittle);
```

W sytuacji, gdy w wywołaniu zostanie pominięta nazwa wymiany bądź zarówno nazwa wymiany, jak i klucz trasujący, szablon `RabbitTemplate` zastosuje nazwę domyślnej

wymiany oraz domyślny klucz. W przypadku użycia przedstawionej wcześniej konfiguracji szablonu domyślna nazwa wymiany jest pusta (podobnie jak podczas korzystania z wymiany, której nazwa nie została podana), tak samo zresztą jak domyślny klucz trasowania. Stosując atrybuty `exchange` oraz `routing-key` elementu `<template>`, można jednak zmienić te ustawienia domyślne:

```
<template id="rabbitTemplate"
  connection-factory="connectionFactory"
  exchange="spittle.alert.exchange"
  routing-key="spittle.alerts" />
```

Niezależnie do zastosowanych wartości domyślnych zawsze można je przesłonić, podając odpowiednie argumenty w wywołaniu metody `convertAndSend()`.

Można się także zastanowić nad wysyłaniem komunikatów przy użyciu innej metody szablonu `RabbitTemplate`. Można chociażby skorzystać z metody `send()`, operującej na nieco niższym poziomie, która pozwala na wysyłanie obiektów `org.springframework.amqp.Message`. Oto przykład jej zastosowania:

```
Message helloMessage =
    new Message("Witaj, świecie!".getBytes(), new MessageProperties());
rabbit.send("hello.exchange", "hello.routing", helloMessage);
```

Metoda `send()`, podobnie jak `convertAndSend()`, udostępnia kilka przeciążonych wersji, które umożliwiają wysyłanie komunikatów bez podawania nazwy wymiany bądź klucza trasującego.

Cała sztuka związana z używaniem metody `send()` polega na utworzeniu obiektu `Message`. W powyższym przykładzie stworzyliśmy go, przekazując w wywołaniu konstruktora tablicę bajtów reprezentującą łańcuch znaków. Takie rozwiązanie bez trudu można wykorzystać w przypadku przesyłania łańcuchów, jednak szybko staje się ono kłopotliwe przy przesyłaniu złożonych obiektów.

Właśnie z tego względu dostępna jest metoda `convertAndSend()`, która automatycznie konwertuje wysyłany obiekt do postaci obiektu `Message`. Domyślnym używanym konwerterem komunikatów jest w tym przypadku `SimpleMessageConverter`, który doskonale nadaje się do wysyłania łańcuchów znaków, instancji klas implementujących interfejs `Serializable` oraz tablic bajtów. Spring AMQP udostępnia także kilka innych konwerterów wiadomości, które mogą okazać się bardzo pomocne. Znajdziemy wśród nich również konwertery do obsługi danych JSON i XML.

Skoro udało nam się już wysłać komunikat, zajmijmy się drugą stroną konwersacji i zobaczmy, jak można taki komunikat odebrać.

17.3.4. Odbieranie komunikatów AMQP

Jak zapewne pamiętasz, mechanizmy obsługi JMS w Springu udostępniają dwa sposoby pobierania komunikatów z kolejek: synchroniczny, bazujący na użyciu szablonu `JmsTemplate`, oraz asynchroniczny, korzystający z obiektów POJO sterowanych komunikatami. Podobne możliwości oferuje Spring AMQP. Ponieważ dysponujemy już skonfigurowanym szablonem `RabbitTemplate`, w pierwszej kolejności zobaczymy, jak można go używać do synchronicznego pobierania komunikatów z kolejki.

POBIERANIE KOMUNIKATÓW PRZY UŻYCIU SZABLONU RABBITTEMPLATE

Szablon `RabbitTemplate` udostępnia kilka metod do pobierania komunikatów. Najprostszymi z nich są metody należące do grupy o nazwie `receive()`, które stanowią stosowane po stronie konsumenta wiadomości odpowiedniki metod `send()`. Metody te pozwalają na pobranie z kolejki obiektu typu `Message`:

```
Message message = rabbit.receive("spittle.alert.queue");
```

Gdyby ktoś chciał, to można także skonfigurować domyślną kolejkę służącą do odbierania komunikatów. W tym celu podczas konfigurowania szablonu wystarczy określić wartość atrybutu `queue`:

```
<template id="rabbitTemplate"
    connection-factory="connectionFactory"
    exchange="spittle.alert.exchange"
    routing-key="spittle.alerts"
    queue="spittle.alert.queue" />
```

W takim przypadku można pobierać komunikaty z domyślnej kolejki, wywołując metodę `receive()` bez podawania żadnych argumentów:

```
Message message = rabbit.receive();
```

Po pobraniu obiektu `Message` będziemy zapewne musieli skonwertować tablicę bajtów zapisaną w jego właściwości `body` na dowolny obiekt, który chcemy otrzymać. Wcześniej nie było łatwo skonwertować obiekt domeny do postaci obiektu `Message`, który można przesłać w komunikacie, i podobnie jest w tym przypadku, gdy musimy skonwertować obiekt `Message` do postaci obiektu domeny. Dlatego zamiast korzystać z metody `receive()`, warto zastanowić się nad użyciem metody `reveiveAndConvert()`:

```
Spittle spittle =
    (Spittle) rabbit.receiveAndConvert("spittle.alert.queue");
```

Można również pominąć w jej wywołaniu nazwę kolejki, aby zastosować nazwę kolejki domyślnej:

```
Spittle spittle = (Spittle) rabbit.receiveAndConvert();
```

Metoda `receiveAndConvert()` używa tego samego konwertera komunikatów co metoda `sendAndConvert()`.

Jeśli w kolejce nie ma żadnych komunikatów oczekujących na pobranie, to wywołania metod `receive()` oraz `receiveAndConvert()` kończą się natychmiast i zwracają przy tym najprawdopodobniej wartość `null`. Z tego względu obowiązek zarządzania odpytywaniem kolejki i obsługą wątków spoczywa na naszych barkach.

Zamiast synchronicznie odpytywać kolejkę i oczekiwać na odebranie komunikatu, Spring AMQP udostępnia także możliwość skorzystania z obiektów POJO sterowanych komunikatami, stanowiącą odpowiednik tej samej możliwości Spring JMS. Zobaczmy zatem, jak konsumować komunikaty, używając Spring AMQP i obiektów POJO sterowanych komunikatami.

DEFINIOWANIE OBIEKTÓW POJO STEROWANYCH KOMUNIKATAMI WSPÓŁDZIAŁAJĄCYCH ZE SPRING AMQP

Aby asynchronicznie skonsumować obiekt `Spittle` przy użyciu obiektu POJO sterowanego komunikatami, w pierwszej kolejności będziemy potrzebowali takiego obiektu POJO. Poniżej przedstawiłem klasę `SpitterAlertHandler`, której obiekty wykorzystamy w tym celu:

```
package com.habuma.spittr.alerts;
import com.habuma.spittr.domain.Spittle;

public class SpittleAlertHandler {

  public void handleSpittleAlert(Spittle spittle) {
    // ... miejsce na implementację ...
  }

}
```

Warto zwrócić uwagę, że jest to dokładnie ta sama klasa `SpitterAlertHandler`, którą stosowaliśmy podczas konsumowania komunikatów `Spittle` przesyłanych przy użyciu JMS. Ten sam obiekt POJO możemy wykorzystać dlatego, że w kodzie klasy nie ma niczego, co w jakikolwiek sposób wiązałoby go z JMS lub AMQP. To zwyczajna klasa obiektów POJO, które są gotowe do przetwarzania obiektów `Spittle` niezależnie od tego, jaki mechanizm przesyłania komunikatów został zastosowany w celu ich dostarczenia.

Dodatkowo musimy także zadeklarować `SpittleAlertHandler` jako komponent w kontekście aplikacji Spring:

```
<bean id="spittleListener"
    class="com.habuma.spittr.alert.SpittleAlertHandler" />
```

Również ten komponent zadeklarowaliśmy już wcześniej, tworząc MDP korzystające z JMS. Ten niczym się nie różni.

W końcu musimy też zadeklarować kontener odbiorcy oraz samego odbiorcę, który wywoła obiekt `SpittleAlertHandler` w momencie odebrania komunikatu. Robiliśmy to już w przypadku MDP obsługujących komunikaty JMS, jednak jeśli chodzi o odbieranie komunikatów AMQP, jest pewna różnica:

```
<listener-container connection-factory="connectionFactory">
    <listener ref="spittleListener"
          method="handleSpittleAlert"
          queue-names="spittle.alert.queue" />
</listener-container>
```

Czy zauważyłeś tę różnicę? Zgadzam się, że nie jest ona aż tak oczywista. Elementy `<listener-container>` oraz `<listener>` wydają się podobne do swych odpowiedników stosowanych podczas korzystania z JMS. Ale w tym przypadku oba te elementy pochodzą z przestrzeni nazw `rabbit`, a nie z przestrzeni nazw JMS.

Jak już zaznaczyłem — to wcale nie jest takie oczywiste.

No dobrze, w powyższym kodzie jest jeszcze jedna drobna różnica. Zamiast określać sprawdzaną kolejkę lub temat przy użyciu atrybutu `destination` (jak to było podczas korzystania z JMS), tutaj określamy nazwę kolejki, z której będą pobierane komunikatu, stosując w tym celu atrybut `queue-names`. Jednak z wyjątkiem tych drobnych różnic obiekty POJO używane do obsługi komunikatów AMQP i JMS działają bardzo podobnie.

Gdyby ktoś się zastanawiał, to tak — atrybut `queue-names` sugeruje liczbę mnogą. W naszym przypadku podaliśmy nazwę tylko jednej kolejki, która ma być sprawdzana, niemniej jednak można podać dowolną ich liczbę, oddzielając je od siebie przecinkami.

Innym sposobem określenia kolejek, z których mają być pobierane komunikaty, jest podanie odwołań do komponentów tych kolejek, zadeklarowanych przy użyciu elementów `<queue>`. Należy to zrobić za pomocą atrybutu `queues`:

```
<listener-container connection-factory="connectionFactory">
    <listener ref="spittleListener"
          method="handleSpittleAlert"
          queues="spittleAlertQueue" />
</listener-container>
```

Także w tym atrybucie można podać listę identyfikatorów kolejek, oddzielonych od siebie przecinkami. Oczywiście aby skorzystać z tej możliwości, należy wcześniej zadeklarować identyfikatory kolejek. Poniżej zamieściłem deklarację naszej przykładowej kolejki, w której tym razem został określony jej identyfikator:

```
<queue id="spittleAlertQueue" name="spittle.alert.queue" />
```

Należy zwrócić uwagę, że w celu określenia identyfikatora komponentu kolejki w kontekście aplikacji Springa został zastosowany atrybut `id`. Z kolei atrybut `name` określa nazwę kolejki w brokerze RabbitMQ.

17.4. Podsumowanie

Asynchroniczna obsługa komunikatów ma kilka zalet w porównaniu do synchronicznego RPC. Mniej bezpośrednia komunikacja powoduje, że aplikacje są ze sobą luźniej powiązane, co zmniejsza wpływ awarii jednego z systemów na całość. W dodatku, ponieważ komunikaty przekazywane są do odbiorców, nadawca nie musi czekać na odpowiedź. W wielu okolicznościach zwiększa to wydajność aplikacji i zapewnia możliwość jej skalowania.

Chociaż JMS zapewnia standardowe API wszystkim aplikacjom Javy, które chcą brać udział w asynchronicznej komunikacji, jego użycie może być kłopotliwe. Spring eliminuje wtórny kod i kod obsługi wyjątków, dzięki czemu asynchroniczna obsługa komunikatów jest dużo łatwiejsza w użyciu.

W tym rozdziale pokazaliśmy kilka sposobów Springa na ustanowienie asynchronicznej komunikacji pomiędzy dwoma aplikacjami za pomocą brokerów komunikatów i JMS. Szablon JMS Springa eliminuje zbędny kod, często wymagany w tradycyjnym modelu programowania JMS. A komponenty zarządzane komunikatami pozwalają na deklarację metod komponentów, które reagują na komunikaty pojawiające się w kolejce lub temacie. Dowiedziałeś się też, jak za pomocą obiektu wywołującego JMS Springa używać komponentów Springa do obsługi wywołań RPC sterowanych komunikatami.

W tym rozdziale poznałeś sposoby asynchronicznej komunikacji pomiędzy aplikacjami. W następnym rozdziale także zajmiemy się podobną problematyką, a konkretnie zobaczymy, jak przy wykorzystaniu WebSocket zapewnić asynchroniczną komunikację pomiędzy klientami działającymi w przeglądarkach WWW a serwerem.

18 Obsługa komunikatów przy użyciu WebSocket i STOMP

W tym rozdziale omówimy:

- Przesyłanie komunikatów pomiędzy przeglądarką i serwerem
- Obsługę komunikatów przy użyciu kontrolerów MVC Springa
- Przesyłanie komunikatów przeznaczonych dla konkretnego użytkownika

W poprzednim rozdziale zostały przedstawione sposoby przesyłania komunikatów pomiędzy aplikacjami Springa przy wykorzystaniu JMS i AMQP. Asynchroniczna wymiana komunikatów jest bardzo często stosowaną formą komunikacji pomiędzy aplikacjami. Jednak w przypadku gdy jedna z tych aplikacji działa w przeglądarce WWW, potrzebne jest nieco inne rozwiązanie.

WebSocket jest protokołem dwukierunkowej komunikacji przy użyciu jednego gniazda. Pozwala on między innymi na asynchroniczne przesyłanie komunikatów pomiędzy przeglądarką WWW i serwerem. Komunikacja dwukierunkowa oznacza, że zarówno serwer może przesyłać komunikaty do przeglądarki, jak i przeglądarka może wysyłać komunikaty na serwer.

W Springu 4.0 wprowadzono wsparcie dla WebSocket, a w jego skład wchodzą:

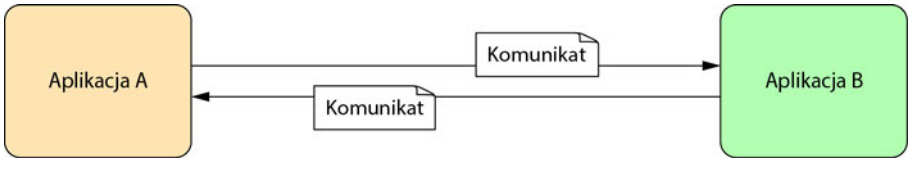
- interfejs API niskiego poziomu, służący do wysyłania i odbierania komunikatów;
- interfejs API wyższego poziomu, służący do obsługi komunikatów w kontrolerach MVC Springa;

- szablon służący do wysyłania komunikatów;
- wsparcie dla SockJS, pozwalające rozwiązać problem braku obsługi WebSocket w przeglądarkach, serwerach WWW oraz serwerach proxy.

W tym rozdziale dowiesz się, jak można zapewnić asynchroniczną komunikację pomiędzy serwerem oraz aplikacjami działającymi w przeglądarkach WWW przy użyciu mechanizmów obsługi WebSocket udostępnianych przez Springa. Zaczniemy od przedstawienia interfejsu API niskiego poziomu.

18.1. Korzystanie z API WebSocket niskiego poziomu

W swojej najprostszej postaci WebSocket jest jedynie kanałem komunikacyjnym łączącym dwie aplikacje. Aplikacja na jednym końcu tego kanału wysyła komunikat, a aplikacja na drugim końcu odbiera ten komunikat i go obsługuje. Ponieważ WebSocket jest protokołem dwukierunkowym, to każda z aplikacji może wysyłać komunikaty oraz je odbierać, jak pokazano na rysunku 18.1.



Rysunek 18.1. WebSocket jest dwukierunkowym kanałem komunikacyjnym łączącym dwie aplikacje

WebSocket może być wykorzystywany do prowadzenia komunikacji pomiędzy aplikacjami dowolnego rodzaju, jednak najczęściej stosuje się go, by ułatwić wymianę komunikatów pomiędzy serwerem oraz aplikacjami działającymi w przeglądarkach WWW. Klient JavaScript działający w przeglądarce otwiera połączenie z serwerem, a serwer, używając tego połączenia, przesyła do przeglądarki dane. Ogólnie rzecz biorąc, takie rozwiązanie jest wydajniejsze i naturalniejsze niż stosowane już od dawna odpytywanie serwera i pobieranie z niego aktualizacji przez przeglądarkę.

Aby zademonstrować API WebSocket niskiego poziomu dostępne w Springu, napiszemy prosty przykład, w którym klient napisany w języku JavaScript gra z serwerem w niekończącą się grę „Marco Polo". Aplikacja działająca na serwerze obsługuje komunikat tekstowy (`"Marco!"`), reagując na niego poprzez wysłanie innego komunikatu tekstowego (`"Polo!"`). Oba komunikaty są przesyłane przy użyciu tego samego połączenia. Żeby obsługiwać w Springu komunikaty przy użyciu API WebSocket niskiego poziomu, należy napisać klasę implementującą interfejs `WebSocketHandler`:

```
public interface WebSocketHandler {
  void afterConnectionEstablished(WebSocketSession session)
                                          throws Exception;
  void handleMessage(WebSocketSession session,
                     WebSocketMessage<?> message) throws Exception;
  void handleTransportError(WebSocketSession session,
                            Throwable exception) throws Exception;
```

```
  void afterConnectionClosed(WebSocketSession session,
                             CloseStatus closeStatus) throws Exception;
  boolean supportsPartialMessages();
}
```

Jak widać, w celu zaimplementowania interfejsu `WebSocketHandler` konieczne jest napisanie pięciu metod. Ale zamiast bezpośrednio implementować ten interfejs, łatwiej jest rozszerzyć klasę `AbstractWebSocketHandler` — abstrakcyjną implementację interfejsu `WebSocketHandler`. Listing 18.1 przedstawia klasę `MarcoHandler`; dziedziczy ona po klasie `AbstractWebSocketHandler` i będzie obsługiwać komunikaty po stronie serwera.

Listing 18.1. Klasa MarcoHandler obsługuje komunikaty tekstowe po stronie serwera

```
package marcopolo;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.handler.AbstractWebSocketHandler;

public class MarcoHandler extends AbstractWebSocketHandler {

  private static final Logger logger =
      LoggerFactory.getLogger(MarcoHandler.class);

  protected void handleTextMessage(                          // Obsługa komunikatu tekstowego
      WebSocketSession session, TextMessage message) throws Exception {
    logger.info("Odebrano komunikat: " + message.getPayload());
    Thread.sleep(2000);                                      // Symulacja opóźnienia
    session.sendMessage(new TextMessage("Polo!"));           // Wysłanie komunikatu tekstowego
  }

}
```

Choć `AbstractWebSocketHandler` jest klasą abstrakcyjną, to jednak nie wymaga od nas przesłaniania jakichkolwiek konkretnych metod. Zamiast tego mamy pełną swobodę co do wyboru metody, którą przesłonimy. Oprócz pięciu metod zdefiniowanych w interfejsie `WebSocketHandler` można także przesłonić trzy dodatkowe metody zdefiniowane w klasie `AbstractWebSocketHandler`:

- `handleBinaryMessage()`,
- `handlePongMessage()`,
- `handleTextMessage()`.

Stanowią one jedynie wyspecjalizowane wersje metody `hangleMessage()`, przeznaczone do obsługi komunikatów konkretnego typu.

Ponieważ nasza klasa `MarcoHandler` będzie obsługiwać tekstowe komunikaty `"Marco!"`, to sensownym rozwiązaniem będzie przesłonięcie metody `handleTextMessage()`. W momencie odebrania komunikatu tekstowego zostanie on zarejestrowany, a następnie, po zasymulowaniu dwusekundowego opóźnienia, przy użyciu tego samego połączenia zostanie wysłany inny komunikat tekstowy.

Wszystkie pozostałe metody, których klasa `MarcoHandler` nie przesłania, zostały zaimplementowane w klasie `AbstractWebSocketHandler` w formie pustych metod. Oznacza to, że nasza klasa `MarcoHandler` będzie także obsługiwać pozostałe dwa rodzaje komunikatów, choć nic z nimi nie będzie robić.

Ewentualnie zamiast rozszerzać klasę `AbstractWebSocketHandler`, możemy rozszerzyć klasę `TextWebSocketHandler`:

```
public class MarcoHandler extends TextWebSocketHandler {
  ...
}
```

`TextWebSocketHandler` jest klasą dziedziczącą po `AbstractWebSocketHandler`, która nie obsługuje komunikatów binarnych. Przesłania ona metodę `handleBinaryMessage()`, ale tylko po to, by zamknąć połączenie WebSocket, jeśli zostanie odebrany komunikat binarny. Analogicznie Spring udostępnia klasę `BinaryWebSocketHandler`, która też dziedziczy po `AbstractWebSocketHander`, i przesłania metodę `handleTextMessage()`, by zamknąć połączenie, jeżeli zostanie odebrany komunikat tekstowy.

Niezależnie od tego, czy obsługujemy komunikaty tekstowe, binarne, czy oba te rodzaje, to może nas również interesować obsługa nawiązania i zamknięcia połączenia. W takim przypadku można przesłonić metody `afterConnectionEstablished()` oraz `afterConnectionClosed()`:

```
public void afterConnectionEstablished(WebSocketSession session)
    throws Exception {
  logger.info("Nawiązano połączenie");
}

@Override
public void afterConnectionClosed(
    WebSocketSession session, CloseStatus status) throws Exception {
  logger.info("Zamknięto połączenie. Status: " + status);
}
```

Obsługę połączeń wspierają dwie metody: `afterConnectionEstablished()` oraz `after`
`↪ConnectionClosed()`. Pierwsza z nich, `afterConnectionEstablished()`, jest wywoływana po nawiązaniu nowego połączenia. Z kolei druga metoda, `afterConnectionClosed()`, jest wywoływana po zamknięciu połączenia. W powyższym przykładzie metody rejestrują jedynie fakt nawiązania i zamknięcia połączenia, ale można je z powodzeniem stosować do inicjowania lub zwalniania wszelkich zasobów używanych w trakcie istnienia połączenia.

Warto zwrócić uwagę, że nazwy obu tych metod zaczynają się od słowa „after”[1]. Oznacza ono, że obie metody zapewniają tylko możliwość reakcji na zdarzenie, lecz nie pozwalają zmieniać jego wyniku.

Skoro już dysponujemy klasą obsługującą komunikaty, musimy ją skonfigurować tak, by w momencie odebrania komunikatu Spring kierował go do instancji tej klasy.

[1] Ang. *po — przyp. tłum.*

W przypadku konfigurowania działania Springa w kodzie Javy oznacza to konieczność dodania do klasy konfiguracyjnej adnotacji `@EnableWebSocket` i zaimplementowania interfejsu `WebSocketConfigurer` w sposób przedstawiony na listingu 18.2.

Listing 18.2. Zastosowanie konfiguracji Javy do włączenie obsługi WebSocket i powiązania komponentu obsługującego komunikaty

```
package marcopolo;

import org.springframework.context.annotation.Bean;
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

  @Override
  public void registerWebSocketHandlers(
  WebSocketHandlerRegistry registry) {
    registry.addHandler(marcoHandler(), "/marco");
  }

  @Bean
  public MarcoHandler marcoHandler() {
    return new MarcoHandler();
  }
}
```

Skojarzenie MarcoHandler z adresem „/marco”

Deklaracja komponentu MarcoHandler

Kluczowe znaczenie dla zarejestrowania komponentu obsługującego komunikaty ma metoda `registerWebSocketHandlers()`. Przesłaniając ją, uzyskujemy obiekt `WebSocket` `↪HandlerRegistry`, który pozwala na wywołanie metody `addHandler()`, służącej do rejestrowania komponentów obsługujących komunikaty. W powyższym przykładzie rejestrujemy obiekt `MarcoHandler` (zadeklarowany jako komponent) i kojarzymy go ze ścieżką `/marco`.

Alternatywnym rozwiązaniem jest skonfigurowanie obsługi komunikatów w kodzie XML z użyciem przestrzeni nazw `websocket`. Przykład takiego pliku konfiguracyjnego przedstawia listing 18.3.

Listing 18.3. Przestrzeń nazw websocket pozwala na konfigurowanie obsługi komunikacji WebSocket w kodzie XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:websocket="http://www.springframework.org/schema/websocket"
  xsi:schemaLocation="
    http://www.springframework.org/schema/websocket
    http://www.springframework.org/schema/websocket/spring-websocket.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
  <websocket:handlers>
    <websocket:mapping handler="marcoHandler" path="/marco" />
  </websocket:handlers>
```

Powiązanie MarcoHandler ze ścieżką „/marco”

```
  <bean id="marcoHandler"
        class="marcopolo.MarcoHandler" /> <-- Deklaracja komponentu MarcoHandler
</beans>
```

Niezależnie od tego, czy obsługę komunikatów konfigurujemy w kodzie Javy, czy też w kodzie XML, to jest to jedyna konfiguracja, jakiej będziemy potrzebować.

Teraz możemy się skoncentrować na kliencie, który będzie wysyłał na serwer komunikaty tekstowe `"Marco!"` i czekał na nadesłanie komunikatu tekstowego z serwera. Poniżej, na listingu 18.4, zaprezentowany został klient JavaScript, który otwiera natywne połączenie WebSocket i używa go do wymiany komunikatów z serwerem.

Listing 18.4. Klient JavaScript wysyłający komunikaty „Marco!" przy użyciu WebSocket

```
var url = 'ws://' + window.location.host + '/websocket/marco';
var sock = new WebSocket(url); <-- Otworzenie połączenia WebSocket

sock.onopen = function() { <-- Obsługa zdarzenia nawiązania połączenia
  console.log('Otwieranie połączenia');
  sayMarco();
};

sock.onmessage = function(e) { <-- Obsługa otrzymanego komunikatu
  console.log('Odebrano komunikat: ', e.data);
  setTimeout(function(){sayMarco()}, 2000);
};

sock.onclose = function() { <-- Obsługa zdarzenia zamknięcia połączenia
  console.log('Zamykanie połączenia');
};

function sayMarco() {
  console.log('Wysyłanie komunikatu Marco!');
  sock.send("Marco!"); <-- Wysłanie komunikatu
}
```

Pierwszą czynnością wykonywaną przez kod przedstawiony na listingu 18.4 jest utworzenie instancji `WebSocket`. To natywny typ przeglądarki, służący do obsługi komunikacji WebSocket. Utworzenie instancji `WebSocket` oznacza w praktyce nawiązanie połączenia WebSocket z podanym adresem URL. W powyższym przykładzie adres URL zaczyna się od prefiksu protokołu `"ws://"`, co oznacza zwyczajne połączenie WebSocket. Gdyby konieczne było nawiązanie bezpiecznego połączenia, to prefiks protokołu powinien mieć postać `"wss://"`.

Po utworzeniu instancji `WebSocket` kilka kolejnych wierszy kodu przygotowuje funkcje służące do obsługi zdarzeń. Warto zwrócić uwagę na zdarzenia WebSocket: `onopen`, `onmessage` oraz `onclose`. Odpowiadają one metodom naszej klasy `MarcoHandler`: `afterConnectionEstablished()`, `handleTextMessage()` i `afterConnectionClosed()`. Do obsługi zdarzeń `onopen` będzie używana funkcja, która wywołuje funkcję `sayMarco()`. Z kolei funkcja `sayMarco()` wykorzystuje połączenie WebSocket, by wysłać komunikat tekstowy `"Marco!"`. Wysłanie tego komunikatu rozpoczyna niekończącą się grę, gdyż jego odebranie przez komponent `MarcoHandler` na serwerze spowoduje wysłanie w odpowiedzi

komunikatu `"Polo!"`. Kiedy klient odbierze komunikat z serwera, zostanie zgłoszone zdarzenie `onmessage`, a w efekcie jego obsługi klient wyśle na serwer kolejny komunikat `"Marco!"`.

Taka wymiana komunikatów będzie trwała aż do momentu zamknięcia połączenia. To szaleństwo można zakończyć wywołaniem `sock.close()`, które nie zostało pokazane na ostatnim listingu. Równie dobrze to serwer może zamknąć połączenie, ewentualnie w przeglądarce może zostać wyświetlona inna strona, co także spowoduje zamknięcie połączenia. W każdym z tych przypadków w wyniku zamknięcia połączenia zostanie zgłoszone zdarzenie `onclose`. W naszym przykładzie na zdarzenie to reagujemy bardzo prosto: wyświetlając stosowny komunikat.

W ten sposób udało się nam napisać wszystkie elementy związane z obsługą komunikacji WebSocket przy użyciu API niskiego poziomu — klasę odbierającą i obsługującą komunikaty na serwerze, jak również klienta, który to samo robi po stronie przeglądarki. Gdybyśmy spróbowali zbudować ten kod i wdrożyć go w kontenerze serwletów, to *mogłoby* się nawet okazać, że on działa.

Czy zwróciłeś uwagę na pewien pesymizm sugerowany przez zastosowanie słowa „mogłoby”? Wynika on z faktu, że nie jestem w stanie zagwarantować, iż to rozwiązanie faktycznie będzie działać. W rzeczywistości szanse na to, że nie będzie ono działać, są całkiem spore. Nawet jeśli my zrobimy wszystko prawidłowo, to okoliczności są przeciwko nam.

Przekonajmy się zatem, co sprawia, że kod WebSocket może nie działać prawidłowo, oraz co zrobić, by zwiększyć szanse na jego działanie.

18.2. Rozwiązanie problemu braku obsługi WebSocket

WebSocket jest stosunkowo młodą specjalizacją. Choć pod koniec 2011 roku został on uznany za standard, to jednak wciąż nie wszystkie przeglądarki i serwery aplikacji są w stanie go obsługiwać na jednakowym poziomie. Przeglądarki Firefox oraz Chrome już od jakiegoś czasu dysponują pełnym wsparciem WebSocket, ale pozostałe przeglądarki zaczęły go obsługiwać dopiero niedawno. Poniżej została przedstawiona lista najniższych wersji przeglądarek, które dysponują obsługą WebSocket:

- Internet Explorer: 10.0;
- Firefox: 4.0 (częściowo), 6.0 (w pełni);
- Chrome: 4.0 (częściowo), 13.0 (w pełni);
- Safari: 5.0 (częściowo), 6.0 (w pełni);
- Opera: 11.0 (częściowo), 12.0 (w pełni);
- iOS Safari: 4.2 (częściowo), 6.0 (w pełni);
- Android Browser: 4.4.

Niestety wielu użytkowników internetu nie zwraca uwagi na najnowsze możliwości przeglądarek ani ich nie rozumie, dlatego nie spieszą się z ich aktualizacją. Co więcej, spora liczba korporacji narzuca stosowanie konkretnej wersji wybranej przeglądarki, utrudniając (lub nawet całkowicie uniemożliwiając) swoim pracownikom zainstalowanie

jakiegokolwiek nowszego oprogramowania. Zważywszy na te okoliczności, prawdopodobieństwo, że osoby korzystające z naszej aplikacji nie będą w stanie jej używać, jeśli zastosujemy w niej WebSocket, jest całkiem wysokie.

Dokładnie to samo dotyczy drugiej strony medalu, czyli obsługi protokołu WebSocket przez serwery aplikacji. GlassFish obsługuje go już od kilku lat, jednak wiele innych serwerów udostępniło niezbędne możliwości dopiero w swoich najnowszych wersjach.

Nawet jeśli przeglądarka i serwer aplikacji będą dostatecznie nowe i wyposażone w obsługę WebSocket, to i tak nie ma gwarancji, że problemy nie wystąpią gdzieś pomiędzy nimi. Pośredniczące w ruchu sieciowym firewalle zazwyczaj blokują wszystko, co nie jest transmisją HTTP. Nie dysponują możliwością przekazywania połączeń WebSocket bądź (jeszcze) nie zostały skonfigurowane w odpowiedni sposób.

Zdaję sobie sprawę, że obraz obecnego stanu obsługi WebSocket maluje się raczej w czarnych barwach. Nie należy jednak pozwolić, by tak niewielka trudność, jaką jest brak wsparcia, zniechęciła nas do prób wykorzystania tego protokołu. WebSocket, gdy działa, sprawuje się rewelacyjnie. A jeśli nie działa, to jedyną rzeczą, jakiej potrzebujemy, jest plan ratunkowy.

Na szczęście istnieje takie rozwiązanie awaryjne, a jest nim SockJS. SockJS to emulator WebSocket, który od zewnątrz naśladuje API WebSocket tak wiernie, jak to tylko możliwe, a od wewnątrz jest na tyle mądry, że wybiera inną formę połączenia, jeżeli obsługa WebSocket nie jest dostępna. SockJS zawsze wybierze obsługę WebSocket, jeśli tylko będzie ona dostępna, natomiast w przeciwnym razie wybierze najlepsze z poniższych rozwiązań:

- przesyłanie strumieniowe z użyciem XHR;
- przesyłanie strumieniowe z użyciem XDR;
- zdarzenia generowane przez element `iframe`;
- pliki HTML wczytywane do elementu `iframe`;
- odpytywanie XHR;
- odpytywanie XDR;
- odpytywanie XHR z użyciem elementu `iframe`;
- odpytywanie JSONP.

Najważniejsze jest jednak to, że nie musimy rozumieć tych wszystkich rozwiązań, by korzystać z SockJS. Pozwala on bowiem stosować spójny model programowania, tak jakby obsługa WebSocket była wszechobecna, a w razie potrzeby, w niewidoczny dla nas sposób, wykorzysta rozwiązanie awaryjne.

Na przykład aby poprosić o obsługę komunikacji przy użyciu SockJS, wystarczy to zrobić w konfiguracji Springa. W tym celu należy tylko wprowadzić niewielką zmianę w metodzie `registerWebSocketHandlers()`, przedstawionej wcześniej na listingu 18.2.

```
@Override
public void registerWebSocketHandlers(
                        WebSocketHandlerRegistry registry) {
  registry.addHandler(marcoHandler(), "/marco").withSockJS();
}
```

Wywołując metodę `withSockJS()` obiektu `WebSocketHandlerRegistration`, zwróconego przez wywołanie metody `addHandler()`, informujemy, że chcemy włączyć korzystanie z SockJS i stosować jej rozwiązania awaryjne, jeśli używanie WebSocket nie będzie możliwe.

W przypadku konfigurowania aplikacji Springa za pomocą kodu XML zastosowanie SockJS sprowadza się do dodania do pliku konfiguracyjnego elementu `<websocket:sockjs>`:

```
<websocket:handlers>
  <websocket:mapping handler="marcoHandler" path="/marco" />
  <websocket:sockjs />
</websocket:handlers>
```

Aby korzystać z SockJS po stronie klienta, trzeba zadbać o dołączenie do strony klienckiej biblioteki SockJS. Sposób, w jaki należy to zrobić, w znacznej mierze zależy od tego, czy używamy jakiegoś mechanizmu do wczytywania bibliotek JavaScript (takich jak *require.js* lub *curl.js*), czy też stosujemy zwyczajne znaczniki `<script>`. Najprostszą metodą wczytania klienckiej biblioteki SockJS jest pobranie jej z sieci CDN SockJS przy wykorzystaniu poniższego znacznika `<script>`:

```
<script src="http://cdn.sockjs.org/sockjs-0.3.min.js"></script>
```

Odwoływanie się do zasobów sieciowych przy użyciu WebJars

W swoim przykładowym kodzie stosuję WebJars, by odwoływać się do bibliotek JavaScript jako elementów projektu utworzonego przy użyciu Maven lub Gradle, czyli tak samo jak w przypadku wszystkich innych zasobów projektu. W tym celu skonfigurowałem mechanizm obsługi zasobów w konfiguracji Spring MVC w taki sposób, aby żądania, w których ścieżka zaczyna się od */webjars/***, wyznaczał względem standardowej ścieżki WebJars:

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
  registry.addResourceHandler("/webjars/**")
      .addResourceLocations("classpath:/META-INF/resources/webjars/");
}
```

Po skonfigurowaniu tego mechanizmu obsługi zasobów mogę już wczytywać bibliotekę SockJS na stronie, używając następującego znacznika `<script>`:

```
<script th:src="@{/webjars/sockjs-client/0.3.4/sockjs.min.js}">
</script>
```

Warto zwrócić uwagę, że znacznik ten pochodzi z szablonu Thymeleaf, a do wyznaczenia pełnego, zależnego od kontekstu adresu URL do pliku JavaScript korzysta z wyrażenia `@{...}`.

Oprócz wczytywania klienckiej biblioteki SockJS stosowanie tego rozwiązania wymaga zmiany jedynie dwóch wierszy kodu:

```
var url = 'marco';
var sock = new SockJS(url);
```

Pierwszą rzeczą, którą możemy zmienić, jest adres URL. SockJS operuje na adresach rozpoczynających się od *http://* lub *https://*, a nie na tych zaczynających się od *ws://* lub *wss://*. Mimo to możemy używać adresów względnych, dzięki czemu unikniemy

konieczności podawania pełnych adresów URL. W tym przypadku, jeśli strona zawierająca kod JavaScript została pobrana przy użyciu adresu *http://localhost:8080/websocket*, zastosowana w przykładzie prosta ścieżka `marco` spowoduje nawiązanie połączenia z adresem *http://localhost:8080/websocket/margo*.

Jednak kluczową zmianą, którą musimy wprowadzić, jest utworzenie instancji `SockJS` zamiast `WebSocket`. Ponieważ `SockJS` naśladuje `WebSocket` w tak dużym stopniu, jak to tylko możliwe, reszta kodu z listingu 18.4 może pozostać bez zmian. Te same funkcje — `onopen`, `onmessage` oraz `onclose` — wciąż będą obsługiwały odpowiednie zdarzenia. A funkcja `send()` będzie przesyłać na serwer komunikat `"Marco!"`.

Choć zmiany wprowadzone w kodzie nie były zbyt rozległe, to jednak ich skutki, związane ze sposobem wymiany komunikatów pomiędzy klientem i serwerem, były ogromne. Teraz możemy mieć dosyć dużą pewność, że komunikacja, przypominająca wymianę komunikatów WebSocket, będzie działać prawidłowo, nawet jeżeli w rzeczywistości przeglądarka, serwer albo serwer proxy nie będą obsługiwać protokołu WebSocket.

WebSocket zapewnia możliwość komunikacji klient-serwer, a SockJS stanowi rozwiązanie awaryjne na wypadek, gdyby protokół WebSocket nie był obsługiwany. Wybór pomiędzy tymi rozwiązaniami nie ma jednak większego znaczenia, bo każde z nich operuje na zbyt niskim poziomie, by nadawało się do praktycznego zastosowania. Zobaczmy zatem, w jaki sposób możemy użyć protokołu STOMP (ang. *Simple Text Oriented Messaging Protocol*), korzystającego z WebSocket, aby zapewnić komunikacji klient-serwer prawidłową semantykę komunikatów tekstowych.

18.3. Wymiana komunikatów z użyciem STOMP

Gdybym miał Ci zasugerować napisanie aplikacji internetowej, to zapewne miałbyś już dosyć dobre pojęcie o bazowych technologiach i frameworkach, z których można by skorzystać, i to nawet przed jakąkolwiek dyskusją o wymaganiach wstępnych. Nawet w przypadku najprostszej aplikacji internetowej typu „Witaj, świecie!" mógłbyś pomyśleć o zastosowaniu kontrolera Spring MVC do obsługi żądań oraz JSP lub Thymeleaf do generacji odpowiedzi. W zupełnie najprostszym przypadku mógłbyś utworzyć statyczną stronę HTML i pozostawić serwerowi zadanie zwracania jej w odpowiedzi na odbierane żądania. Prawdopodobnie nie zawracałbyś sobie głowy dociekaniem, jak właściwie przeglądarka żąda tej strony, ani tym, w jaki sposób serwer ją dostarcza.

A teraz wyobraźmy sobie, że zasugerowałbym, iż nie ma protokołu HTTP i musimy napisać aplikację internetową, korzystając wyłącznie z gniazd TCP. Przypuszczalnie pomyślałbyś, że zupełnie straciłem rozum. Oczywiście można by wykonać takie zadanie, lecz wymagałoby to opracowania własnego protokołu warstwy połączenia, obsługiwanego zarówno przez klienta, jak i przez serwer, który ułatwiłby prowadzenie komunikacji. Delikatnie rzecz ujmując, nie byłoby to zadanie trywialne.

Na nasze szczęście protokół HTTP określa wszelkie szczegóły związane z generowaniem żądań przez przeglądarki WWW oraz sposobem ich obsługi przez serwery. W efekcie niemal nikt nie pisze już niskopoziomego kodu obsługującego komunikację przy użyciu gniazd TCP.

Bezpośrednie stosowanie WebSocket (lub SockJS) bardzo przypomina pisanie aplikacji internetowej przy użyciu wyłącznie gniazd TCP. Bez protokołu warstwy połączenia operującego na wyższym poziomie to na naszych barkach spoczywa obowiązek określenia semantyki komunikatów tekstowych przesyłanych pomiędzy aplikacjami. Co więcej, musimy mieć pewność, że komunikujące się ze sobą aplikacje zgadzają się co do tej semantyki.

Na szczęście okazuje się, że wcale nie musimy korzystać z połączeń WebSocket w sposób bezpośredni. Protokół HTTP przesłonił swoim modelem obsługi żądań i odpowiedzi gniazda TCP; analogicznie protokół STOMP stosuje swój format połączeniowy, bazujący na ramkach, przesłaniając nim połączenia WebSocket.

Na pierwszy rzut oka struktura ramek komunikatów STOMP wygląda podobnie jak struktura żądań HTTP. Tak jak żądania i odpowiedzi HTTP, także ramki STOMP składają się z polecenia, jednego lub kilku nagłówków oraz zawartości. Na przykład poniższa ramka STOMP służy do przesyłania danych:

```
SEND
destination:/app/marco
content-length:20

{\"message\":\"Marco!\"}
```

W powyższym prostym przykładzie zastosowanym poleceniem STOMP jest `SEND`. Oznacza ono, że coś jest przesyłane. Za poleceniem zostały umieszczone dwa nagłówki: pierwszy z nich określa miejsce docelowe, gdzie należy przesłać komunikat, a drugi określa jego zawartość. Za nagłówkami znajduje się pusty wiersz, a za nim zawartość ramki — w tym przypadku jest to komunikat w formacie JSON.

Prawdopodobnie najciekawszym elementem tej prostej ramki STOMP jest nagłówek `destination`. Stanowi on sugestię, że STOMP jest protokołem do przesyłu komunikatów, podobnie jak JMS i AMQP. Komunikaty są publikowanie i przesyłane do miejsc docelowych, które w rzeczywistości mogą być obsługiwane przez faktyczne brokery komunikatów. Po drugiej stronie odbiorcy komunikatów mogą obserwować te miejsca docelowe i odbierać przesyłane do nich komunikaty.

W kontekście komunikacji WebSocket aplikacja JavaScript działająca w przeglądarce może publikować komunikaty, przesyłając je do miejsca docelowego, gdzie będą one obsługiwane przez komponent działający na serwerze. Co więcej, rozwiązanie to działa także w przeciwnym kierunku: komponent na serwerze może opublikować w miejscu docelowym komunikat, który następnie zostanie odebrany przez klienta działającego w przeglądarce.

Spring obsługuje wymianę komunikatów STOMP, wykorzystując przy tym model programistyczny bazujący na Spring MVC. Jak się niebawem przekonasz, obsługa komunikatów STOMP w kontrolerach Spring MVC naprawdę niewiele różni się od obsługi żądań HTTP. Jednak zanim do tego dojdziemy, musimy skonfigurować Springa tak, by obsługiwał on komunikaty STOMP.

18.3.1. Włączanie obsługi komunikatów STOMP

Już zaraz zobaczysz, jak należy dodawać do metod kontrolerów adnotacje `@MessageMapping`, aby zapewnić możliwość obsługi komunikatów STOMP w sposób bardzo podobny do tego, w jaki metody z adnotacją `@RequestMapping` pozwalają na obsługę żądań HTTP. Jednak w odróżnieniu od adnotacji `@RequestMapping` adnotacja `@MessageMapping` nie jest włączana przez użycie `@EnableWebMvc`. Obsługa komunikatów internetowych w Springu bazuje na brokerze komunikatów, dlatego skonfigurowanie jego działania wymaga nieco większego zachodu niż jedynie poinformowanie Springa o tym, że chcemy obsługiwać komunikaty. Dodatkowo konieczne jest skonfigurowanie brokera komunikatów oraz informacji o miejscu docelowym.

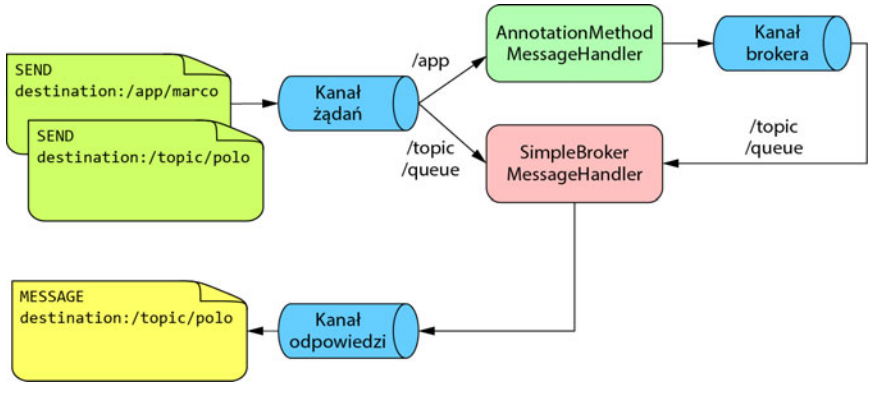
Listing 18 5 przedstawia kod Javy stanowiący niezbędną konfigurację obsługi komunikatów internetowych przy użyciu brokera.

Listing 18.5. Dyrektywa @EnableWebSocketMessageBroker włącza obsługę komunikatów STOMP przesyłanych przy użyciu protokołu WebSocket

```
package marcopolo;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.config.annotation.
                                              AbstractWebSocketMessageBrokerConfigurer;
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

@Configuration
@EnableWebSocketMessageBroker <-- Włącza obsługę komunikatów STOMP
public class WebSocketStompConfig
      extends AbstractWebSocketMessageBrokerConfigurer {

  @Override
  public void registerStompEndpoints(StompEndpointRegistry registry) {
    registry.addEndpoint("/marcopolo").withSockJS();   <-- Włącza wykorzystanie SockJS na adresie /marcopolo
  }

  @Override
  public void configureMessageBroker(MessageBrokerRegistry registry) {
    registry.enableSimpleBroker("/queue", "/topic");
    registry.setApplicationDestinationPrefixes("/app");
  }
}
```

W odróżnieniu od klasy konfiguracyjnej przedstawionej na listingu 18.2, do klasy `WebSocketStompConfig` została dodana adnotacja `@EnableWebSocketMessageBroker`. Informuje ona, że nie jest to jedynie klasa konfigurująca komunikację WebSocket, lecz że dodatkowo konfiguruje ona także obsługę komunikatów STOMP przy użyciu brokera komunikatów. Klasa ta przesłania metodę `registerStompEndpoints()`, aby zarejestrować adres `/marcopolo` jako punkt końcowy STOMP. Ta ścieżka odróżnia się od ścieżek wszelkich innych punktów docelowych, do których można wysyłać komunikatu lub z których można je odbierać. Określa ona punkt końcowy, z którym klient musi nawiązać połączenie, zanim będzie mógł subskrybować komunikaty albo publikować je w określonym miejscu docelowym.

Klasa `WebSocketStompConfig` konfiguruje też prosty broker komunikatów — w tym celu przesłania ona metodę `configureMessageBroker()`. Przesłanianie tej metody jest opcjonalne. Jeśli tego nie zrobimy, to uzyskamy prosty, działający w pamięci broker komunikatów, skonfigurowany tak, żeby obsługiwał komunikaty przesyłane na adres z prefiksem `/topic`. Jednak w powyższym przykładzie metoda ta została przesłonięta, a jej kod sprawia, że broker będzie odpowiedzialny za obsługę komunikatów przesyłanych na adresy z prefiksem `/topic` i `/queue`. Ponadto wszelkie komunikaty przeznaczone dla aplikacji będą przesyłane na adres z prefiksem `/app`. Przepływ komunikatów w przypadku zastosowania takiej konfiguracji został zilustrowany na rysunku 18.2.



Rysunek 18.2. Prosty broker STOMP Springa działa w pamięci i naśladuje kilka funkcji brokera STOMP

Po odebraniu komunikatu sposób jego obsługi zostanie określony na podstawie prefiksu miejsca docelowego. W przykładzie przedstawionym na rysunku 18.2 miejsca docelowe aplikacji mają prefiks `/app`, a miejsca docelowe brokera mają prefiksy `/topic` bądź `/queue`. Komunikaty skierowane do miejsca docelowego aplikacji trafiają bezpośrednio do metody kontrolera opatrzonej adnotacją `@MessageMapping`. Z kolei komunikaty przeznaczone dla brokera, w tym także wszelkie komunikaty stanowiące efekt wartości zwracanych przez metody z adnotacją `@MessageMapping`, są kierowane do brokera, a w rezultacie są przesyłane do klientów subskrybujących dane miejsca docelowe.

WŁĄCZENIE PRZEKAZYWANIA DO BROKERA STOMP

Prosty broker doskonale nadaje się na początek, ma jednak kilka ograniczeń. Choć naśladuje brokera komunikatów STOMP, to obsługuje jedynie podzbiór wszystkich poleceń tego protokołu. A ponieważ działa w pamięci komputera, to nie nadaje się do zastosowania w klastrach serwerów, w których każdy z węzłów korzystałby z własnego brokera i zbioru komunikatów.

W przypadku aplikacji produkcyjnych prawdopodobnie zdecydowalibyśmy się zatem na obsługę naszej aplikacji WebSocket przez brokera STOMP z prawdziwego zdarzenia, takiego jak RabbitMQ lub ActiveMQ. Brokery te są w stanie zaoferować znacznie bardziej skalowalne i niezawodne mechanizmy przesyłania komunikatów, nie wspominając nawet o obsłudze wszystkich poleceń STOMP. W przypadku użycia takich brokerów

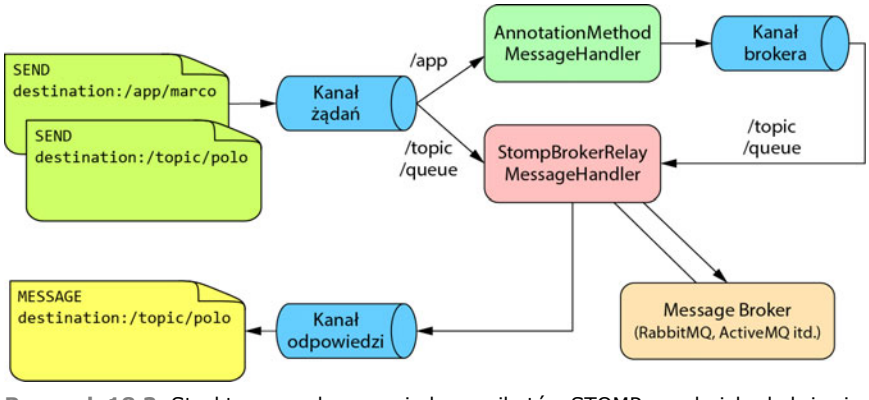
trzeba pamiętać, by skonfigurować je zgodnie z ich dokumentacją. Po skonfigurowania brokera będzie można zastąpić domyślnego brokera działającego w pamięci przekazywaniem komunikatów do brokera STOMP. W tym celu należy skorzystać z metody `configureMessageBroker()` o następującej postaci:

```
@Override
public void configureMessageBroker(MessageBrokerRegistry registry) {
  registry.enableStompBrokerRelay("/topic", "/queue");
  registry.setApplicationDestinationPrefixes("/app");
}
```

Pierwszy wiersz kodu tej metody włącza przekazywanie do brokera STOP i ustawia jego prefiksy na `/topic` i `/queue`. Stanowi to informację, że komunikaty, dla których miejsce docelowe rozpoczyna się od prefiksu `/topic` lub `/queue`, powinny zostać przekazane do brokera STOMP. W zależności od wybranego brokera STOMP wybór dostępnych prefiksów może być ograniczony. Na przykład RabbitMQ pozwala na stosowanie miejsc docelowych typów `/temp-queue`, `/exchange`, `/topic`, `/queue`, `/amq/queue` oraz `/reply-queue`. Dostępne typy miejsc docelowych oraz ich przeznaczenie należy sprawdzić w dokumentacji brokera.

Oprócz prefiksu miejsca docelowego drugi wiersz kodu metody `configureMessage➥Broker()` ustawia prefiks aplikacji na `/app`. Wszelkie komunikaty, których miejsce docelowe będzie się rozpoczynać od `/app`, zostaną skierowane do metody opatrzonej adnotacją `@MessageMapping`, a nie opublikowane w temacie czy kolejce brokera.

Rysunek 18.3 pokazuje, jak wygląda model obsługi komunikatów STOMP w Springu z włączonym przekazywaniem do brokera. Jak widać, główna różnica polega na tym, że zamiast naśladować funkcjonalność brokera STOMP, mechanizm przekazywania kieruje komunikaty do faktycznego brokera, w którym zostają one obsłużone.



Rysunek 18.3. Struktura przekazywania komunikatów STOMP w celu ich obsłużenia w prawdziwym brokerze komunikatów

Warto zwrócić uwagę, że zarówno metoda `enableStompBrokerRelay()`, jak i `setApplication➥DestinationPrefixes()` pozwalają na przekazywanie zmiennej liczby argumentów typu `String`, dzięki czemu za ich pomocą można skonfigurować wiele miejsc docelowych i prefiksów aplikacji. Oto przykład:

```
@Override
public void configureMessageBroker(MessageBrokerRegistry registry) {
  registry.enableStompBrokerRelay("/topic", "/queue");
  registry.setApplicationDestinationPrefixes("/app", "/foo");
}
```

Domyślnie mechanizm przekazywania do brokera STOMP zakłada, że oczekuje on na komunikaty na porcie 61613 komputera localhost oraz że nazwą i hasłem dostępu klienta jest `"guest"`. Jeśli broker STOMP działa na innym serwerze bądź wymaga innych danych uwierzytelniających, to informacje te można podać w następujący sposób:

```
@Override
public void configureMessageBroker(MessageBrokerRegistry registry) {
  registry.enableStompBrokerRelay("/topic", "/queue")
          .setRelayHost("rabbit.someotherserver")
          .setRelayPort(62623)
          .setClientLogin("marcopolo")
          .setClientPasscode("letmein01");
  registry.setApplicationDestinationPrefixes("/app", "/foo");
}
```

Powyższy fragment kodu konfiguracyjnego określa nazwę serwera, numer portu oraz dane uwierzytelniające klientów. Nic nie zmusza nas jednak do podawania wszystkich tych informacji. Na przykład jeśli musimy zmienić jedynie nazwę komputera, to wystarczy wywołać tylko metodę `setRelayHost()`, a wszystkie inne można pominąć.

Tak oto Spring został już skonfigurowany do obsługi komunikatów STOMP.

18.3.2. Obsługa komunikatów STOMP nadsyłanych przez klienty

Zgodnie z informacjami podanymi w rozdziale 5. Spring zapewnia możliwość obsługi żądań HTTP, korzystając z modelu programistycznego bazującego na stosowaniu adnotacji. `@RequestMapping`, jedna z kluczowych adnotacji Springa, kojarzy żądania HTTP z metodami, które będą służyć do ich obsługi. Dokładnie ten sam model programistyczny został użyty także do obsługi zasobów REST, o czym mogliśmy się przekonać w rozdziale 16.

STOMP oraz WebSocket są jednak nastawione bardziej na obsługę komunikatów synchronicznych niż na działanie w modelu żądanie-odpowiedź, stosowanym w protokole HTTP. Niemniej jednak Spring pozwala, by obsługa komunikatów STOMP w bardzo dużym stopniu przypominała korzystanie ze Spring MVC. W rzeczywistości oba rozwiązania są do siebie tak podobne, że metody obsługujące komunikaty STOMP mogą być składowymi klas, do których dodano adnotacje `@Controller`.

W wersji 4.0 Springa wprowadzono adnotację `@MessageMapping`, stanowiącą odpowiednik adnotacji `@RequestMapping` używanej w Spring MVC. Metody, do których dodano adnotację `@MessageMapping`, mogą obsługiwać komunikaty przesyłane do określonego miejsca docelowego. W ramach przykładu przeanalizujmy prostą klasę kontrolera, przedstawioną na listingu 18.6.

Na pierwszy rzut oka zaprezentowana klasa wygląda tak samo jak każda inna klasa kontrolera Spring MVC. Została do niej dodana adnotacja `@Controller`, zostanie zatem odnaleziona i zarejestrowana jako komponent. Ma również metodę obsługi, zupełnie tak samo jak każda inna klasa opatrzona adnotacją `@Controller`.

Listing 18.6. Metody z adnotacją @MessageMapping obsługują komunikaty STOMP w kontrolerze

```
package marcopolo;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.stereotype.Controller;

@Controller
public class MarcoController {

  private static final Logger logger =
      LoggerFactory.getLogger(MarcoController.class);

  @MessageMapping("/marco")
  public void handleShout(Shout incoming) {
    logger.info("Odebrano komunikat: " + incoming.getMessage());
  }

}
```

Obsługuje komunikaty kierowane do miejsca docelowego /app/marco

Jednak ta klasa różni się nieco od tych, z którymi spotkaliśmy się wcześniej. Zdefiniowana w niej metoda `handleShout()` została opatrzona adnotacją `@MessageMapping`, a nie `@RequestMapping`. Oznacza to, że metoda ta powinna obsługiwać wszelkie komunikaty nadesłane do określonego miejsca docelowego. W przedstawionym przykładzie tym miejscem docelowym jest `/app/marco` (prefiks `"/app"` został zastosowany domyślnie, gdyż skonfigurowaliśmy go wcześniej jako prefiks miejsc docelowych aplikacji).

Ponieważ metoda `handleShout()` pobiera parametr typu `Shout`, to zawartość komunikatu STOMP zostanie skonwertowana do tego typu przy wykorzystaniu konwertera komunikatów Springa. `Shout` jest zwyczajną klasą komponentu JavaBean i definiuje jedną właściwość służącą do przechowywania zawartości komunikatu:

```
package marcopolo;

public class Shout {

  private String message;

  public String getMessage() {
    return message;
  }

  public void setMessage(String message) {
    this.message = message;
  }
}
```

Jako że nie mamy tu do czynienia z protokołem HTTP, konwersja obiektów `Shout` nie będzie wykonywana przy użyciu jednej z implementacji interfejsu `HttpMessageConverter`. Zamiast tego, w ramach swojego API do obsługi komunikatów, Spring 4.0 udostępnia jedynie parę konwerterów komunikatów. Tabela 18.1 przedstawia konwertery komunikatów, które mogą być stosowane podczas obsługi komunikatów STOMP.

Tabela 18.1. Spring może konwertować zawartość komunikatów do obiektów Javy przy użyciu kilku rodzajów konwerterów

Konwerter komunikatów	Opis
`ByteArrayMessageConverter`	Konwertuje komunikaty typu MIME `application/octet-stream` do postaci `byte[]` i na odwrót.
`MappingJackson2MessageConverter`	Konwertuje komunikaty typu MIME `application/json` do postaci obiektów Javy i na odwrót.
`StringMessageConverter`	Konwertuje komunikaty typu MIME `text/plain` do postaci łańcucha znaków (`String`) i na odwrót.

Zakładając, że komunikat obsługiwany przez metodę `handleShout()` jest typu MIME `application/json` (co jest zapewne bezpiecznym założeniem, zważywszy na to, że `Shout` nie jest ani tablicą bajtów, ani łańcuchem znaków), zadanie konwersji tego komunikatu do postaci obiektu Javy zostanie zlecone konwerterowi `MappingJackson2MessageConverter`. Podobnie jak w przypadku konwertera `MappingJackson2HttpMessageConverter` stosowanego podczas obsługi żądań HTTP, konwerter `MappingJackson2MessageConverter` przekazuje wykonanie konwersji do procesora JSON biblioteki Jackson 2. Domyślnie Jackson wykona konwersję właściwości JSON na właściwości obiektu Javy, korzystając z techniki odzwierciedlania. Używając adnotacji biblioteki Jackson, możemy modyfikować sposób realizacji tej konwersji, choć w tym przypadku nie jest to konieczne.

PRZETWARZANIE SUBSKRYPCJI

Oprócz adnotacji `@MessageMapping` Spring udostępnia także adnotację `@SubscribeMapping`. Każda metoda, do której została dodana ta adnotacja, będzie wywoływana podobnie jak metody z adnotacją `@MessageMapping`, czyli po odebraniu komunikatu STOMP skierowanego do subskrybowanego miejsca docelowego.

Koniecznie trzeba zrozumieć, że metody z adnotacją `@SubscribeMapping`, tak samo jak metody z adnotacją `@MessageMapping`, otrzymują swoje komunikaty za pośrednictwem `AnnotationMethodMessageHandler` (jak pokazano na rysunkach 18.2 oraz 18.3). W przypadku zastosowania konfiguracji z listingu 18 5 oznacza to, że metody opatrzone adnotacją `@SubscribeMapping` mogą obsługiwać wyłącznie komunikaty, których miejsca docelowe mają prefiks `/app`.

Może się to wydawać nieco dziwne, skoro wychodzące komunikaty są kierowane do miejsc docelowych brokera mających prefiksy `/topis` i `/queue`. Klienty subskrybują właśnie te miejsca docelowe i zapewne nie będą zainteresowane subskrybowaniem miejsc docelowych z prefiksem `/app`. Jeśli klienty subskrybują miejsca docelowe z prefiksami `/topic` i `/queue`, to nie ma możliwości, by subskrypcje te były obsługiwane przez metody opatrzone adnotacją `@SubscribeMapping`. Jeżeli faktycznie tak jest, to do czego może nam się przydać ta adnotacja?

Podstawowym przypadkiem użycia adnotacji `@SubscribeMapping` jest implementacja wzorca żądanie-odpowiedź. Wzorzec ten zakłada, że klient subskrybuje miejsce docelowe, oczekując, że zostanie w nim udostępniona jedna odpowiedź.

W ramach przykładu przeanalizujmy następującą metodę, w której została zastosowana adnotacja `@SubscribeMapping`:

```
@SubscribeMapping({"/marco"})
public Shout handleSubscription() {
  Shout outgoing = new Shout();
  outgoing.setMessage("Polo!");
  return outgoing;
}
```

Jak widać, do metody `handleSubscription()` została dodana adnotacja `@SubscribeMapping`, dzięki czemu metoda ta będzie obsługiwać subskrypcje dotyczące adresu `/app/marco`. (Podobnie jak w przypadku adnotacji `@MessageMapping`, zostanie tutaj domyślnie użyty prefiks `"/app"`). W ramach obsługi subskrypcji metoda `handleSubscription()` tworzy i zwraca obiekt `Shout`. Obiekt ten jest następnie konwertowany do postaci odpowiedzi i przesyłany do klienta przy wykorzystaniu tego samego miejsca docelowego, które klient subskrybował.

Jeśli sądzisz, że wzorzec żądanie-odpowiedź nie różni się niczym szczególnym od wzorca żądania `GET` protokołu HTTP, to właściwie masz rację. Kluczowa różnica pomiędzy nimi polega jednak na tym, że w odróżnieniu od żądania `GET` protokołu HTTP, które jest synchroniczne, wzorzec żądanie-odpowiedź jest asynchroniczny i pozwala klientowi obsłużyć wiadomość w momencie, gdy się ona pojawi, bez zmuszania go do oczekiwania na jej odebranie.

PISANIE KLIENTA JAVASCRIPT

Metoda `handleShout()` jest już gotowa do obsługi nadsyłanych komunikatów. Jedyne, co nam zatem pozostaje, to napisanie klienta, który by je wysyłał.

Listing 18.7 przedstawia kod JavaScript klienta, który nawiązuje połączenie z punktem końcowym /marcopolo i przesyła komunikat `"Marco!"`.

Listing 18.7. W kodzie JavaScript komunikaty można wysyłać przy użyciu biblioteki STOMP

```
r url = 'http://' + window.location.host + '/stomp/marcopolo';
var sock = new SockJS(url); <-- Tworzy połączenie SockJS

var stomp = Stomp.over(sock); <-- Tworzy klienta STOMP

var payload = JSON.stringify({ 'message': 'Marco!' });

stomp.connect('guest', 'guest', function(frame) { <-- Nawiązuje połączenie z punktem końcowym STOMP
  stomp.send("/marco", {}, payload); <-- Wysyła komunikat
});
```

Ten klient JavaScript, podobnie jak poprzedni, rozpoczyna się od utworzenia instancji `SockJS`, łączącej się z podanym adresem URL. W tym przypadku adres URL odwołuje się do punktu końcowego STOMP, skonfigurowanego na listingu 18 5 (przy czym ścieżka kontekstu aplikacji, `/stomp`, nie jest uwzględniana).

Jednak ten klient różni się od poprzedniego tym, że w jego przypadku biblioteka SockJS nie jest używana bezpośrednio. Zamiast tego wywoływana jest funkcja `Stomp.over(sock)`, która tworzy instancję klienta STOMP. W efekcie w kliencie STOMP zostaje umieszczona instancja `SockJS`, która będzie wykorzystywana do wysyłania komunikatów STOMP przez połączenie WebSocket.

Następnie klient STOMP nawiązuje połączenie i wysyła komunikat z kodem JSON do miejsca docelowego o nazwie `/maro`, oczywiście przy założeniu, że połączenie zostało pomyślnie nawiązane. Drugim parametrem metody `send()` jest mapa nagłówków, które zostaną dodane do ramki STOMP — w naszym przykładzie żadne nagłówki nie są przesyłane, więc mapa ta jest pusta.

A zatem dysponujemy już klientem przesyłającym komunikaty na serwer oraz metodą obsługi, która jest gotowa, by je obsługiwać po stronie serwera. To dobry początek, choć być może zwróciłeś uwagę, że taka komunikacja jest właściwie jednokierunkowa. Udzielmy zatem głosu serwerowi i zobaczmy, w jaki sposób może on wysyłać komunikaty do klienta.

18.3.3. Wysyłanie komunikatów do klienta

Jak na razie to tylko klient wysyła komunikaty, a serwer jest zmuszony do oczekiwania na nie. Choć jest to całkowicie poprawne zastosowanie WebSocket i STOMP, to jednak najprawdopodobniej nie jest to przypadek użycia, który przychodzi nam do głowy, gdy myślimy o WebSocket. W powszechnej opinii traktuje się go jako sposób, dzięki któremu serwer może przesyłać informacje do przeglądarki z własnej inicjatywy, a nie w odpowiedzi na żądanie HTTP. Jak zatem możemy komunikować się z klientem JavaScript, działającym w przeglądarce, przy wykorzystaniu Springa i WebSocket oraz STOMP?

Spring udostępnia dwa sposoby wysyłania danych do klientów:

- jako efekt uboczny obsługi komunikatu lub subskrypcji;
- za pomocą szablonu obsługi komunikatów.

Znasz już kilka metod służących do obsługi komunikatów i subskrypcji, dlatego w pierwszej kolejności przyjrzymy się, jak wysyłać komunikaty do klientów jako efekt uboczny stosowania tych metod. Następnie zajmiemy się szablonem `SimpMessagingTemplate` Springa, umożliwiającym wysyłanie komunikatów z dowolnego miejsca aplikacji.

WYSYŁANIE KOMUNIKATU PO OBSŁUŻENIU INNEGO KOMUNIKATU

Metoda `handleShout()` przedstawiona na listingu 18.6 zwraca wartość `void`. Jej zadaniem jest obsługa komunikatu, a nie odpowiadanie klientowi.

Mimo to, gdybyśmy chcieli wysłać komunikat w odpowiedzi na komunikat nadesłany przez klienta, wystarczyłoby zwrócić jakąkolwiek wartość różną od `void`. Na przykład aby w odpowiedzi na komunikat `"Marco!"` wysłać odpowiedź `"Polo!"`, wystarczy zmienić metodę `handleShout()` w następujący sposób:

```
@MessageMapping("/marco")
public Shout handleShout(Shout incoming) {
  logger.info("Odebrano komunikat: " + incoming.getMessage());

  Shout outgoing = new Shout();
  outgoing.setMessage("Polo!");
  return outgoing;
}
```

Ta nowa wersja metody `handleShout()` zwraca nowy obiekt `Shout`. Dzięki zwyczajnemu zwróceniu obiektu metoda obsługi może się stać metodą wysyłającą komunikat. W przypadku gdy metoda opatrzona adnotacją `@MessageMapping` zwraca jakąś wartość, to zwracany przez nią obiekt zostanie skonwertowany (przy użyciu konwertera komunikatów) i umieszczony jako zawartość w ramce STOMP, która następnie zostanie przekazana do brokera.

Domyślnie ramka zostanie opublikowana w tym samym miejscu docelowym, które spowodowało wywołanie metody obsługi, choć będzie ono miało prefiks `/topic`. W przypadku metody `handleShout()` oznacza to, że zwrócony obiekt `Shout` zostanie umieszczony w ramce STOMP jako jej zawartość, a ramka ta zostanie opublikowana w miejscu docelowym `/topic/marco`. Istnieje jednak możliwość zmiany miejsca docelowego — wystarczy dodać do metody adnotację `@SentTo`:

```
@MessageMapping("/marco")
@SendTo("/topic/shout")
public Shout handleShout(Shout incoming) {
  logger.info("Odebrano komunikat: " + incoming.getMessage());

  Shout outgoing = new Shout();
  outgoing.setMessage("Polo!");
  return outgoing;
}
```

Po zastosowaniu takiej adnotacji `@SendTo` komunikat zostanie opublikowany w miejscu docelowym `/topic/shout`. Ten komunikat otrzyma każda aplikacja subskrybująca dany temat (na przykład klient).

Teraz, w odpowiedzi na odebranie komunikatu, metoda `handleShout()` wysyła jak odpowiedź inny komunikat. W analogiczny sposób wszystkie metody, do których dodano adnotację `@SubscribeMapping`, mogą wysyłać komunikaty w odpowiedzi na subskrypcję. Na przykład dodając do kontrolera poniższą metodę, moglibyśmy wysyłać komunikaty `Shout`, gdy klient rozpocznie subskrypcję:

```
@SubscribeMapping("/marco")
public Shout handleSubscription() {
  Shout outgoing = new Shout();
  outgoing.setMessage("Polo!");
  return outgoing;
}
```

Adnotacja `@SubscribeMapping` informuje, że za każdym razem, gdy klient subskrybuje miejsce docelowe `/app/marco` (gdzie `/app` jest prefiksem miejsca docelowego aplikacji), zostanie wywołana metoda `handleSubscription()`. Zwracany przez nią obiekt `Shout` zostanie skonwertowany i przesłany z powrotem do klienta.

Różnica, która pojawia się w przypadku użycia adnotacji `@SubscribeMapping`, polega na tym, że obiekt `Shout` jest przesyłany bezpośrednio do klienta, z pominięciem brokera. Jeśli metoda zostanie dodatkowo opatrzona adnotacją `@SendTo`, to komunikat zostanie przesłany do określonego w niej miejsca docelowego za pośrednictwem brokera.

WYSYŁANIE KOMUNIKATÓW Z DOWOLNEGO MIEJSCA

Adnotacje `@MessageMapping` oraz `@SubscribeMapping` zapewniają prosty sposób wysyłania komunikatów w odpowiedzi na odebranie komunikatu lub obsługę subskrypcji. Jednak szablon `SimpMessagingTemplate` Springa pozwala na wysyłanie komunikatów z dowolnego miejsca aplikacji, i to nawet w przypadku, gdy wcześniej nie odebrano żadnego innego komunikatu.

Najprostszą metodą użycia szablonu `SimpMessagingTemplate` jest jego (bądź jego interfejsu — `SimpMessageSendingOperations`) automatyczne dowiązanie do obiektu, który będzie go potrzebował.

Aby pokazać praktyczne zastosowanie tych możliwości, wróćmy do strony głównej naszej aplikacji Spittr i udostępnijmy na niej aktualizowany na bieżąco kanał obiektów `Spittle`. W obecnej postaci kontroler obsługujący żądania dotyczące strony głównej aplikacji pobiera listę najnowszych obiektów `Spittle` i umieszcza je w modelu, dzięki czemu zostaną one wyświetlone w przeglądarce użytkownika. Choć takie rozwiązanie działa doskonale, to jednak nie daje możliwości aktualizowania kanału „na żywo". Jeżeli użytkownik zechce zaktualizować zawartość kanału, będzie musiał odświeżyć stronę w przeglądarce.

Ale zamiast zmuszać użytkowników do odświeżania strony, można subskrybować na niej temat STOMP i na bieżąco otrzymywać wszelkie pojawiające się aktualizacje obiektów `Spittle`. W tym celu do strony głównej należy dodać następujący fragment kodu JavaScript:

```
<script>
  var sock = new SockJS('spittr');
  var stomp = Stomp.over(sock);

  stomp.connect('guest', 'guest', function(frame) {
    console.log('Connected');
    stomp.subscribe("/topic/spittlefeed", handleSpittle);
  });

  function handleSpittle(incoming) {
    var spittle = JSON.parse(incoming.body);
    console.log('Received: ', spittle);
    var source = $("#spittle-template").html();
    var template = Handlebars.compile(source);
    var spittleHtml = template(spittle);
    $('.spittleList').prepend(spittleHtml);
  }
</script>
```

W tym przykładzie, podobnie jak w poprzednim, tworzymy instancję `SockJS`, a następnie używamy jej do utworzenia instancji `Stomp`. Po nawiązaniu połączenia z brokerem STOMP subskrybujemy miejsce docelowe `/topic/spittlefeed` i określamy, że wszelkie odbierane aktualizacje obiektów `Spittle` mają być obsługiwane przez funkcję `handle` ↪`Spittle()`. Funkcja ta przekształca zawartość odebranego komunikatu do prawidłowego obiektu JavaScript, a następnie używa biblioteki Handlebars, by wyświetlić

odebrany obiekt Spittle w formie kodu HTML, umieszczając go na samym początku listy. Szablon biblioteki Handlebars jest zdefiniowany w odrębnym elemencie `<script>` i ma następującą postać:

```
<script id="spittle-template" type="text/x-handlebars-template">
  <li id="preexist">
  <div class="spittleMessage">{{message}}</div>
  <div>
    <span class="spittleTime">{{time}}</span>
    <span class="spittleLocation">({{latitude}}, {{longitude}})</span>
  </div>
  </li>
</script>
```

Po stronie serwera możemy skorzystać z szablonu `SimpMessagingTemplate`, aby wszelkie nowe obiekty `Spittle` publikować w formie komunikatów w temacie `/topic/spittlefeed`. Właśnie to robi klasa `SpittleFeedServiceImpl`, przedstawiona na listingu 18.8.

Listing 18.8. SimpMessagingTemplate pozwala na publikowanie komunikatów z dowolnego miejsca kodu

```
package spittr;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.simp.SimpMessageSendingOperations;
import org.springframework.stereotype.Service;

@Service
public class SpittleFeedServiceImpl implements SpittleFeedService {

  private SimpMessageSendingOperations messaging;

  @Autowired
  public SpittleFeedServiceImpl(
  SimpMessageSendingOperations messaging) {          <-- Wstrzykuje szablon obsługi komunikatów
    this.messaging = messaging;
  }

  public void broadcastSpittle(Spittle spittle) {
    messaging.convertAndSend("/topic/spittlefeed", spittle);   <-- Wysyła komunikat
  }
}
```

Efektem ubocznym skonfigurowania obsługi STOMP w Springu jest utworzenie i umieszczenie w kontekście aplikacji komponentu `SimpMessageTemplate`. Dzięki temu nie musimy tworzyć jego nowej instancji. Zamiast tego do konstruktora klasy `Spittle` ↪`FeedServiceImpl` dodaliśmy adnotację `@Autowired`, która spowoduje wstrzyknięcie istniejącego komponentu `SimpMessagingTemple` (jako obiektu typu `SimpMessageSendingOperation`) do tworzonej instancji `SpittleFeedServiceImpl`.

Wysyłanie komunikatów `Spittle` odbywa się w metodzie `broadcastSpittle()`. Wywołuje ona metodę `convertAndSend()` wstrzykniętego komponentu `SimpMessageSending` ↪`Operations`, by skonwertować obiekt `Spittle` do postaci komunikatu i wysłać go do

tematu /topic/spittlefeed. Jeśli metoda convertAndSend() wygląda znajomo, to zapewne dlatego, że celowo naśladuje ona metody o tej samej nazwie udostępniane przez szablony JmsTemplate oraz RabbitTemplate.

Kiedy publikujemy komunikat w temacie STOMP, czy to przy użyciu metody convert
➥AndSend(), czy jako efekt wywołania metody obsługi, otrzyma go każdy klient subskrybujący dany temat. Jeśli chcemy, żeby klienty prezentowały kanał obiektów Spittle aktualizowany „na żywo", takie rozwiązanie jest wprost idealne. Jednak może się także zdarzyć, że będziemy chcieli przesyłać komunikaty przeznaczone dla konkretnego klienta, a nie do wszystkich klientów.

18.4. Komunikaty skierowane do konkretnego klienta

Aż do tej chwili wszystkie wysyłane i odbierane komunikaty były przekazywane pomiędzy klientem (działającym w przeglądarce WWW) a serwerem. Użytkownik korzystający z tego klienta nie był brany pod uwagę. Kiedy zostaje wywołana metoda opatrzona adnotacją @MessageMapping, wiemy, że nadesłano komunikat, ale nie wiemy, kto to zrobił. Podobnie jeżeli nie wiemy, kim jest użytkownik, to wszystkie wysyłane komunikaty będą trafiały do wszystkich klientów subskrybujących temat, do którego komunikat został wysłany; nie ma możliwości wysłania komunikatu do konkretnego użytkownika.

Jeśli jednak wiemy, kim jest użytkownik, to pojawia się możliwość stosowania komunikatów skojarzonych z użytkownikiem, a nie tylko z klientem. Na szczęście wiemy już również, jak można zidentyfikować użytkownika. Korzystając z mechanizmu uwierzytelniania przedstawionego w rozdziale 9., możemy zastosować mechanizm Spring Security, aby uwierzytelnić użytkownika i wykorzystywać komunikaty, które będą z nim skojarzone.

W przypadku wymiany komunikatów przy użyciu Springa i STOMP informacje o uwierzytelnionym użytkowniku można wykorzystać na trzy sposoby:

- Do metod opatrzonych adnotacjami @MessageMapping oraz @SubscribeMapping może być przekazywany obiekt Principal uwierzytelnionego użytkownika.
- Wartości zwracane przez metody opatrzone adnotacjami @MessageMapping, @SubscribeMapping oraz @MessageException mogą być przesyłane jako komunikaty do uwierzytelnionego użytkownika.
- Szablon SimpMessagingTemplate może przesyłać komunikaty do konkretnego użytkownika.

Zacznijmy od przeanalizowania dwóch pierwszych sposobów, które pozwalają na to, żeby metody kontrolera obsługujące komunikaty operowały na komunikatach skojarzonych z konkretnym użytkownikiem.

18.4.1. Obsługa komunikatów skojarzonych z użytkownikiem w kontrolerze

Jak już wspomniałem, dostępne są dwa sposoby na to, aby metody kontrolera opatrzone adnotacjami @MessageMapping oraz @SubscribeMapping obsługiwały komunikaty, uwzględniając przy tym użytkownika. Prosząc o parametr typu Principal, metoda obsługi może

uzyskać informacje o tym, kim jest użytkownik, i korzystać z nich, by skoncentrować się na obsłudze danych konkretnego użytkownika. Ponadto do metod obsługi można dodać adnotację `@SendToUser`, aby zaznaczyć, że zwrócona przez nią wartość powinna zostać przesłana jako komunikat do klienta uwierzytelnionego użytkownika (i tylko do tego klienta).

W ramach przykładu napiszmy metodę kontrolera, która na podstawie odebranego komunikatu będzie tworzyć nowy obiekt `Spittle` i przesyłać odpowiedź informującą o jego zapisaniu. Jeśli ta operacja wydaje się znajoma, to dlatego, że zaimplementowaliśmy ją już w rozdziale 16. jako punkt końcowy REST. Jednak żądania REST są synchroniczne, klient musi zatem czekać, aż serwer je obsłuży. Przesyłając obiekt `Spittle` jako komunikat STOMP, możemy w pełni wykorzystać asynchroniczny sposób działania STOMP.

Przeanalizujmy poniższą metodę `handleSpittle()`, która obsługuje odbierane komunikaty i zapisuje je w formie obiektów `Spittle`:

```
@MessageMapping("/spittle")
@SendToUser("/queue/notifications")
public Notification handleSpittle(
      Principal principal, SpittleForm form) {

  Spittle spittle = new Spittle(
      principal.getName(), form.getText(), new Date());

  spittleRepo.save(spittle);

  return new Notification("Zapisano spittle'a");
}
```

Jak widać, metoda `handleSpittle()` pobiera zarówno obiekt `Principal`, jak i `SpittleFrom`. Obiekty te są następnie stosowane podczas tworzenia instancji `Spittle`, która zostaje zapisana przy użyciu `SpittleRepository`. Na samym końcu metoda zwraca nowy obiekt, `Notification`, informujący o tym, że obiekt `Spittle` został zapisany.

Oczywiście to, co się dzieje w tej metodzie, nie jest aż tak interesujące jak to, co się dzieje poza nią. Ponieważ metodę opatrzono adnotacją `@MessageMapping`, będzie ona wywoływana za każdym razem, gdy do miejsca docelowego `/app/spittle` zostanie nadesłany jakiś komunikat. Na podstawie odebranego komunikatu zostanie utworzony obiekt `SpittleFrom`, a zakładając, że udało się uwierzytelnić użytkownika, to na podstawie nagłówków ramki STOMP zostanie także utworzony obiekt `Principal`.

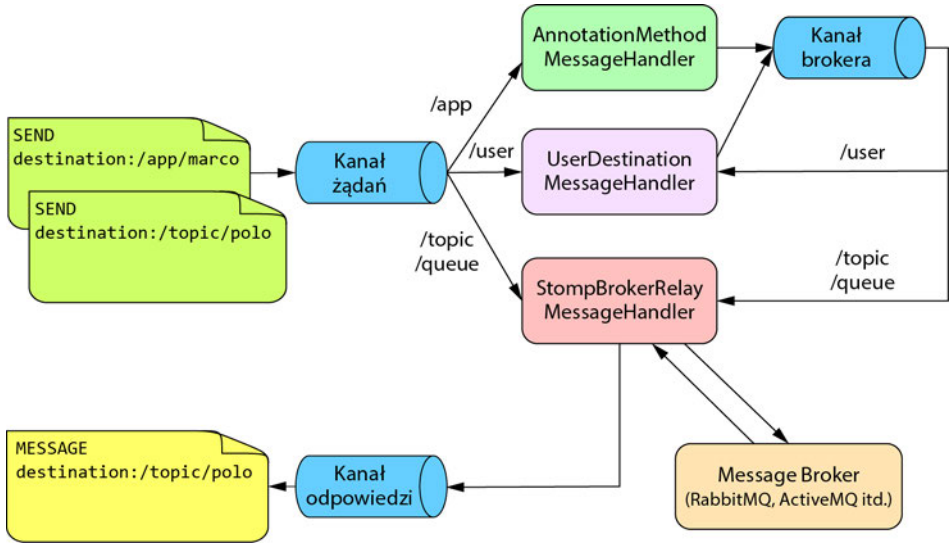
Jednak najważniejszą rzeczą, na którą należy zwrócić uwagę w powyższej metodzie, jest to, że zwraca ona obiekt `Notification`. Adnotacja `@SendToUser` określa, że zwrócony obiekt `Notification` powinien zostać wysłany jako komunikat do miejsca docelowego `/queue/notifications`. Można sądzić, że miejsce docelowe `/queue/notifications` nie jest powiązane z konkretnym użytkownikiem. Ale ze względu na to, że zostało ono określone w adnotacji `@SendToUser`, a nie `@SendTo`, sprawy wyglądają nieco inaczej.

Aby zrozumieć, w jaki sposób Spring opublikuje ten komunikat, cofnijmy się o krok i zobaczmy, jak klient subskrybuje miejsce docelowe, w którym przedstawiona metoda

obsługi będzie publikować obiekty `Notification`. Przeanalizujmy poniższy wiersz kodu JavaScript, który tworzy subskrypcję miejsca docelowego powiązanego z konkretnym użytkownikiem:

```
stomp.subscribe("/user/queue/notifications", handleNotifications);
```

Koniecznie należy zwrócić uwagę na prefiks `/user` poprzedzający adres miejsca docelowego. Spring obsługuje takie miejsca docelowe w szczególny sposób. Otóż kierowane do nich komunikaty nie są przekazywane przez `AnntotationMethodMessageHandler` (jak komunikaty aplikacji) ani przez `SimpleBrokerMessageHandler` lub `StompBrokerRelay` ↪`MessageHandler` (jak komunikaty obsługiwane przez brokery), lecz, jak pokazano na rysunku 18.4, trafiają one do obiektu `UserDestinationMessageHandler`.



Rysunek 18.4. Komunikaty powiązane z konkretnym użytkownikiem trafiają do UserDestinationMessageHandler

Podstawowym przeznaczeniem obiektu `UserDestinationMessageHandler` jest dalsze skierowanie komunikatu powiązanego z użytkownikiem do miejsca docelowego unikalnego dla danego użytkownika. W przypadku subskrypcji to ostateczne miejsce docelowe jest określane poprzez usunięcie prefiksu `/user` i dodanie końcówki zależnej od sesji użytkownika. Na przykład subskrypcja `/user/queue/notifications` może być w efekcie skierowana do miejsca docelowego o nazwie `/queue/notifications-user6hr83v6t`.

W naszym przykładzie metoda `handleSpittle()` została opatrzona adnotacją `@SendTo` ↪`User("/queue/notifications")`. To nowe miejsce docelowe ma prefiks `/queue`, będący jednym z prefiksów obsługiwanych przez nasz `StompBrokerRelayMessageHandler` (lub `SimpleBrokerMessageHandler`), a zatem komunikat zostanie skierowany właśnie tam. Jak się okazuje, klient subskrybował to miejsce docelowe, więc odbierze on komunikat `Notification`.

Adnotacja `@SendToUser` oraz parametr `Principal` są bardzo przydatne w przypadku korzystania z metod kontrolerów. Jednak na listingu 18.8 zobaczyliśmy, w jaki sposób można wysyłać komunikaty z dowolnego miejsca aplikacji, używając w tym celu szablonu obsługi komunikatów. Przekonajmy się zatem, jak można wykorzystać szablon `SimpMessagingTemplate`, aby wysyłać komunikaty do konkretnego użytkownika.

18.4.2. Wysyłanie komunikatów do konkretnego użytkownika

Oprócz metody `convertAndSend()` szablon `SimpMessagingTemplate` udostępnia także metodę `convertAndSendToUser()`. Zgodnie z tym, co sugeruje jej nazwa, pozwala ona na wysyłanie komunikatów przeznaczonych dla konkretnego użytkownika.

Żeby przedstawić działanie tej metody w praktyce, dodajmy do naszej aplikacji Spittr nowy mechanizm, pozwalający na powiadamianie użytkownika, kiedy ktoś w treści spittle'a wspomni o danym użytkowniku. Na przykład jeśli tekst umieszczony w spittle'u zawiera `"@jbauer"`, to należy wysłać komunikat do klienta, którego użytkownik zalogowany jest jako „jbauer". Metoda `broadcastSpittle()`, przedstawiona na listingu 18.9, używa metody `convertAndSendToUser()`, aby poinformować użytkownika, że coś o nim napisano.

Listing 18.9. Metoda convertAndSentDoUser() pozwala wysyłać komunikaty do konkretnego użytkownika

```
package spittr;

import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.stereotype.Service;

@Service
public class SpittleFeedServiceImpl implements SpittleFeedService {

  private SimpMessagingTemplate messaging;
  private Pattern pattern = Pattern.compile("\\@(\\S+)");   <-- Wyrażenie regularne wykrywające nawiązanie do użytkownika

  @Autowired
  public SpittleFeedServiceImpl(SimpMessagingTemplate messaging) {
    this.messaging = messaging;
  }

  public void broadcastSpittle(Spittle spittle) {
    messaging.convertAndSend("/topic/spittlefeed", spittle);
    Matcher matcher = pattern.matcher(spittle.getMessage());
    if (matcher.find()) {
      String username = matcher.group(1);
      messaging.convertAndSendToUser(   <-- Wysyła powiadomienie do użytkownika
          username, "/queue/notifications",
          new Notification("Właśnie o Tobie wspomniano!"));
    }
  }
}
```

Dzięki metodzie `broadcastSpittle()`, jeśli komunikat zamieszczony w danym obiekcie `Spittle` będzie zawierać fragment przypominający nazwę użytkownika (czyli tekst rozpoczynający się od znaku „@"), do miejsca docelowego o nazwie `/queue/notifications` zostanie wysłany komunikat `Notification`. A zatem jeżeli spittle będzie zawierać tekst `"@jbauer"`, komunikat zostanie wysłany do miejsca docelowego o nazwie `/jbauer/queue/notifications`.

18.5. Obsługa wyjątków komunikatów

Czasami aplikacja nie będzie działać tak, jak byśmy tego oczekiwali. Podczas obsługi komunikatów coś może pójść nie tak, jak powinno, i może zostać zgłoszony wyjątek. Ze względu na asynchroniczny charakter obsługi komunikatów STOMP nadawca komunikatu może się nigdy nie dowiedzieć, że podczas jego obsługi pojawiły się problemy. Pomijając zarejestrowanie przez Spring faktu wystąpienia takich problemów, taki wyjątek może całkowicie przepaść, bez szans na reakcję czy możliwości rozwiązania problemu.

W Spring MVC, jeśli podczas obsługi żądania wystąpi wyjątek, można skorzystać z metody opatrzonej adnotacją `@ExceptionHandler`, która zapewnia możliwość obsłużenia wyjątku. Analogicznie, jeżeli do metody kontrolera dodamy adnotację `@MessageExceptionHandler`, to będzie ona używana do obsługi wyjątków zgłaszanych w metodach opatrzonych adnotacją `@MessageMapping`.

W ramach przykładu przeanalizujmy poniższą metodę, służącą do obsługi wyjątków zgłaszanych w metodach obsługi komunikatów:

```
@MessageExceptionHandler
public void handleExceptions(Throwable t) {
  logger.error("Błąd obsługi komunikatu: " + t.getMessage());
}
```

Metody z adnotacją `@MessageExceptionHandler`, w swojej najprostszej postaci, będą obsługiwały wszystkie wyjątki zgłaszane w metodach obsługi komunikatów. Jednak dodając do adnotacji parametr, można określić konkretny typ wyjątku, który ma być obsługiwany przez daną metodę:

```
@MessageExceptionHandler(SpittleException.class)
public void handleExceptions(Throwable t) {
  logger.error("Błąd obsługi komunikatu: " + t.getMessage());
}
```

Można także podać kilka typów wyjątków, które będą obsługiwane przez daną metodę — wystarczy wymienić je w formie parametru tablicowego:

```
@MessageExceptionHandler(
        {SpittleException.class, DatabaseException.class})
public void handleExceptions(Throwable t) {
  logger.error("Błąd obsługi komunikatu: " + t.getMessage());
}
```

Choć przedstawiona metoda rejestruje jedynie fakt wystąpienia błędu, to w praktyce mogłaby robić znacznie więcej. Na przykład mogłaby odpowiadać wysłaniem komunikatu z informacją o problemie:

```
@MessageExceptionHandler(SpittleException.class)
@SendToUser("/queue/errors")
public SpittleException handleExceptions(SpittleException e) {
  logger.error("Błąd obsługi komunikatu: " + e.getMessage());
  return e;
}
```

W tym przypadku, kiedy zostanie zgłoszony wyjątek `SpittleException`, metoda zarejestruje jego wystąpienie, a następnie zwróci obiekt wyjątku. Zgodnie z informacjami podanymi w punkcie 18.4.1, `UserDestinationMessageHandler` przekieruje taki komunikat do miejsca docelowego skojarzonego z użytkownikiem.

18.6. Podsumowanie

Protokół WebSocekt jest fascynującym sposobem przekazywania komunikatów pomiędzy aplikacjami, zwłaszcza w przypadkach, gdy jedna z tych aplikacji działa w przeglądarce WWW. Ma on kluczowe znaczenie dla pisania wysoce interaktywnych aplikacji internetowych, które płynnie przekazują dane pomiędzy klientem a serwerem.

Mechanizmy obsługi WebSocket dostępne w Springu obejmują interfejs API niskiego poziomu, który pozwala na bezpośrednie korzystanie z połączeń WebSocket. Niestety nie wszystkie przeglądarki, serwery WWW oraz serwery proxy zapewniają możliwość używania WebSocket. Dlatego Spring udostępnia także SockJS, protokół, który awaryjnie stosuje inne formy komunikacji, w razie gdy obsługa WebSocket nie jest dostępna.

Spring oferuje również model programowania wyższego poziomu, pozwalający na obsługę komunikatów WebSocket przy użyciu STOMP — protokołu warstwy połączenia. W tym przypadku komunikaty WebSocket są obsługiwane przez kontrolery Spring MVC, podobnie jak żądania HTTP.

W kilku ostatnich rozdziałach przedstawione zostały sposoby asynchronicznego przesyłania komunikatów pomiędzy aplikacjami. Jednak okazuje się, że Spring udostępnia jeszcze jedną metodę asynchronicznego przekazywania komunikatów. W następnym rozdziale dowiesz się, jak można używać Springa do wysyłania wiadomości e-mail.

19 Wysyłanie poczty elektronicznej w Springu

W tym rozdziale omówimy:

- Konfigurowanie warstwy abstrakcji komunikacji e-mail
- Wysyłanie rozbudowanych wiadomości e-mail
- Tworzenie wiadomości e-mail w oparciu o szablony

Nie jest żadną tajemnicą, że poczta elektroniczna stała się bardzo popularną formą komunikacji, która zastąpiła wiele tradycyjnych sposobów, takich jak zwyczajne listy papierowe, rozmowy telefoniczne, a nawet, do pewnego stopnia, bezpośrednie kontakty z innymi osobami. Wiadomości e-mail zapewniają większość tych samych korzyści, które oferują asynchroniczne komunikaty opisane w rozdziale 17., choć w przypadku e-maili ich nadawcami i odbiorcami są ludzie. Gdy tylko naciśniemy przycisk *Wyślij* w kliencie pocztowym, możemy zająć się innymi sprawami, mając pewność, że odbiorca kiedyś dostanie i (ewentualnie) przeczyta wysłaną wiadomość.

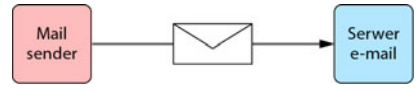
Jednak nie zawsze to ludzie są nadawcami wiadomości e-mail. Bardzo często zdarza się, że są one generowane przez aplikacje. Przykładami takich wiadomości mogą być potwierdzenia wysyłane przez internetowe sklepy albo automatyczne powiadomienia o operacjach wykonywanych na koncie bankowym. Niezależnie do konkretnego tematu istnieje całkiem spore prawdopodobieństwo, że tworzona aplikacja będzie musiała wysyłać wiadomości e-mail. Na szczęście Spring może nam w tym pomóc.

W rozdziale 17. skorzystaliśmy z mechanizmów obsługi komunikatów dostępnych w Springu, by asynchronicznie kolejkować zadania wysyłania powiadomień do innych użytkowników aplikacji Spittr. Nie udało się nam jednak dokończyć tego zadania, gdyż

aplikacja nie wysyłała żadnych wiadomości e-mail. Spróbujemy to zatem zrobić w tym rozdziale — w tym celu najpierw zobaczymy, jak w Springu wyodrębniono problem wysyłania wiadomości e-mail, a następnie skorzystamy z możliwości Springa, aby wysyłać powiadomienia o spittle'ach w formie e-maili.

19.1. Konfigurowanie Springa do wysyłania wiadomości e-mail

Kluczowym elementem warstwy abstrakcji komunikacji e-mail jest interfejs `MailSender`. Jak widać na rysunku 19.1, zadanie implementacji `MailSender` polega na nawiązywaniu połączenia z serwerem pocztowym i wysyłaniu wiadomości e-mail.



Rysunek 19.1. Interfejs Springa MailSender jest głównym komponentem API Springa do obsługi wiadomości e-mail. Wysyła wiadomość na serwer pocztowy, skąd jest ona dostarczana odbiorcy

Spring dostarcza jedną implementację interfejsu `MailSender` — `JavaMailSenderImpl`, która wysyła wiadomości e-mail, korzystając z JavaMail API. Zanim jednak będziemy mogli wysyłać wiadomości e-mail z aplikacji Springa, konieczne będzie powiązanie instancji `JavaMailSenderImpl` w jej kontekście jako komponentu.

19.1.1. Konfigurowanie komponentu wysyłającego

W najprostszym przypadku `JavaMailSenderImpl` można skonfigurować, używając zaledwie kilku wierszy kodu umieszczonych w metodzie z adnotacją `@Bean`:

```
@Bean
public MailSender mailSender(Environment env) {
  JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
  mailSender.setHost(env.getProperty("mailserver.host"));
  return mailSender;
}
```

Właściwość `host` jest opcjonalna (domyślnie odpowiada ona nazwie hosta stosowanej w używanej sesji JavaMail), choć najprawdopodobniej będziemy chcieli ją podać. Określa ona nazwę hosta dla serwera pocztowego, z którego skorzystamy do wysłania wiadomości. W powyższym przykładzie zostaje ona określona poprzez pobranie wartości ze wstrzykniętego komponentu `Environment`, dzięki czemu konfigurowanie wysyłania poczty elektronicznej będzie można podawać poza Springiem (na przykład w plikach właściwości).

Domyślnie `JavaMailSenderImpl` zakłada, że serwer pocztowy działa na porcie 25 (czyli standardowym porcie SMTP). Jeśli nasz serwer pocztowy działa na innym porcie, to jego numer należy podać, używając właściwości `port`. Oto przykład:

```
@Bean
public MailSender mailSender(Environment env) {
  JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
  mailSender.setHost(env.getProperty("mailserver.host"));
  mailSender.setPort(env.getProperty("mailserver.port"));
  return mailSender;
}
```

Jeśli z kolei serwer pocztowy wymaga uwierzytelnienia, potrzebne będzie podanie danych dostępowych we właściwościach `username` i `password`:

```
@Bean
public MailSender mailSender(Environment env) {
  JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
  mailSender.setHost(env.getProperty("mailserver.host"));
  mailSender.setPort(env.getProperty("mailserver.port"));
  mailSender.setUsername(env.getProperty("mailserver.username"));
  mailSender.setPassword(env.getProperty("mailserver.password"));
  return mailSender;
}
```

Jak na razie `JavaMailSenderImpl` był skonfigurowany w taki sposób, by tworzył własną sesję pocztową. Może się jednak zdarzyć, że już będziemy dysponowali sesją skonfigurowaną w JNDI (bądź umieszczoną tam przez serwer aplikacji). W takim przypadku nie ma sensu konfigurować `JavaMailSencerImpl` z użyciem wszystkich szczegółowych informacji dotyczących serwera pocztowego. Zamiast tego można go skonfigurować tak, aby zastosował sesję `MailSession` gotową do użycia i dostępną w JNDI.

Korzystając z `JndiObjectFactoryBean`, można skonfigurować komponent w taki sposób, żeby odszukał w JNDI obiekt `MailSession`. Wystarczy w tym celu zastosować poniższą metodę z adnotacją `@Bean`.

```
@Bean
public JndiObjectFactoryBean mailSession() {
  JndiObjectFactoryBean jndi = new JndiObjectFactoryBean();
  jndi.setJndiName("mail/Session");
  jndi.setProxyInterface(MailSession.class);
  jndi.setResourceRef(true);
  return jndi;
}
```

Wiemy już także, jak pobierać obiekty z JNDI, używając elementu Springa `<jee:jndi-lookup>`. Spróbujmy uzyskać za jego pomocą komponent odwołujący się do sesji pocztowej z JNDI:

```
<jee:jndi-lookup id="mailSession"
  jndi-name="mail/Session" resource-ref="true" />
```

Dysponując już skonfigurowanym komponentem sesji pocztowej, możemy go dowiązać do komponentu `mailSender` w następujący sposób:

```
@Bean
public MailSender mailSender(MailSession mailSession) {
  JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
  mailSender.setSession(mailSession);
  return mailSender;
}
```

Dowiązując sesję pocztową do właściwości `session` komponentu `JavaMailSenderImpl`, nadpisaliśmy wcześniejszą konfigurację serwera (i nazwy użytkownika oraz hasła). Sesja pocztowa jest teraz w całości konfigurowana i zarządzana w JNDI. Obiekt `JavaMailSenderImpl` może się skoncentrować na wysyłaniu wiadomości e-mail, zamiast zajmować się serwerem.

19.1.2. Dowiązanie komponentu wysyłającego pocztę do komponentu usługi

Teraz, gdy komponent wysyłający pocztę został skonfigurowany, czas go dowiązać do komponentu, przez który będzie używany. W aplikacji Spitter najodpowiedniejszym miejscem do wysyłania wiadomości e-mail jest klasa `SpitterEmailServiceImpl`. Klasa definiuje właściwość `mailSender` oznaczoną adnotacją `@Autowired`:

```
@Autowired
JavaMailSender mailSender;
```

Podczas tworzenia komponentu `SpitterEmailServiceImpl` Spring będzie próbował znaleźć komponent implementujący interfejs `MailSender`, który będzie mógł dowiązać do właściwości `mailSender`. Powinien odnaleźć nasz komponent `mailSender` i go użyć. Z dowiązanym komponentem `mailSender` jesteśmy gotowi do tworzenia i wysyłania wiadomości e-mail.

Ponieważ chcemy wysłać użytkownikowi Spittera wiadomość e-mail informującą o nowych spittle'ach napisanych przez jego przyjaciół, potrzebujemy metody, która wyśle taką wiadomość na podstawie adresu e-mail i obiektu typu `Spittle`. Metoda `sendSim`
`↪pleSpitterEmail()` z listingu 19.1 wykorzystuje komponent `mailSender` właśnie do tego:

Listing 19.1. Wysyłanie wiadomości e-mail w Springu z wykorzystaniem implementacji MailSender

```
public void sendSimpleSpittleEmail(String to, Spittle spittle) {
  SimpleMailMessage message = new SimpleMailMessage();          <- Utwórz wiadomość
  String spitterName = spittle.getSpitter().getFullName();
  message.setFrom("noreply@spitter.com");                       <- Zaadresuj wiadomość
  message.setTo(to);
  message.setSubject("Nowy spittle od " + spitterName);
  message.setText(spitterName + " pisze: " +                    <- Określ tekst wiadomości
                  spittle.getText());
  mailSender.send(message);                                     <- Wyślij wiadomość
}
```

Pierwszą czynnością metody `sendSimpleSpittleEmail()` jest utworzenie instancji klasy `SimpleMailMessage`. Ten obiekt wiadomości doskonale nada się do wysyłania praktycznych wiadomości e-mail.

W dalszej kolejności ustawiane są pozostałe detale wiadomości. Nadawca i odbiorca określani są przy użyciu metod `setFrom()` i `setTo()`, wywoływanych dla obiektu wiadomości. Określając temat poprzez wywołanie `setSubject()`, kończymy adresowanie naszej wirtualnej koperty. Pozostaje już tylko wywołać `setText()`, aby ustawić zawartość wiadomości.

Ostatnim krokiem jest przekazanie wiadomości do metody `send()` obiektu `mailSender`, po czym jest ona już w drodze.

A zatem udało nam się skonfigurować komponent do wysyłania wiadomości e-mail i wysłać przy jego użyciu prostą wiadomość. Przekonaliśmy się także, że korzystanie z warstwy abstrakcji obsługi wiadomości e-mail w Springu jest całkiem łatwe. Moglibyśmy na tym poprzestać i przejść do następnego rozdziału. Jednak w ten sposób

ominęłoby nas to, co w udostępnianym przez Springa mechanizmie obsługi poczty elektronicznej jest najzabawniejsze. Spróbujmy więc uatrakcyjnić nasze e-maile i zobaczmy, jak można dodawać do nich załączniki.

19.2. Tworzenie e-maili z załącznikami

Proste tekstowe wiadomości e-mail świetnie nadają się do prostych zastosowań, takich jak przekazanie przyjaciołom zaproszenia na mecz. Nie przydadzą nam się jednak za bardzo w sytuacjach, gdy musimy wysłać zdjęcia lub jakieś dokumenty. Poza tym nie na wiele się zdadzą, kiedy będziemy musieli przyciągnąć uwagę odbiorcy, jak to jest w przypadku wiadomości marketingowych.

Na szczęście możliwości Springa w zakresie tworzenia i wysyłania wiadomości poczty elektronicznej nie kończą się na prostych wiadomościach tekstowych. Spring pozwala także na dodawanie załączników, a nawet przystrajanie treści wiadomości kodem HTML. Zacznijmy od prostego zadania, jakim jest dodawanie załączników. Później pójdziemy o krok dalej i używając kodu HTML, zadbamy o odpowiedni wygląd naszych wiadomości.

19.2.1. Dodawanie załączników

Cała sztuczka związana z wysyłaniem e-maili z załącznikami sprowadza się do tworzenia wiadomości wieloczęściowych, czyli składających się z wielu części — jedna z nich jest jej treścią, a pozostałe zawierają załączniki.

Prosta klasa `SimpleMailMessage` nie pozwala na dodawanie załączników do wiadomości. Do wysyłania wiadomości wieloczęściowych potrzebujemy wiadomości MIME (*Multipurpose Internet Mail Extensions*). Metoda `createMimeMessage()` obiektu `mailSender` pomoże nam ją utworzyć:

```
MimeMessage message = mailSender.createMimeMessage();
```

Otrzymaliśmy tym samym wiadomość MIME, na której możemy operować. Mogłoby się wydawać, że wystarczy jej tylko podać adres, temat, przekazać jakiś tekst i załącznik. Istotnie, choć nie do końca jest to takie proste. API klasy `javax.mail.internet.Mime` ↪`Message` jest bardzo niewygodne w użyciu. Na szczęście, Spring dostarcza dużo bardziej przyjazną klasę `MimeMessageHelper`.

Aby skorzystać z klasy `MimeMessageHelper`, utwórz jej instancję, przekazując wiadomość typu `MimeMessage` w konstruktorze:

```
MimeMessageHelper helper = new MimeMessageHelper(message, true);
```

Drugi parametr konstruktora, wartość logiczna (w tym przypadku `true`), wskazuje, że wiadomość będzie wieloczęściowa.

Mając do dyspozycji obiekt pomocniczy w postaci instancji `MimeMessageHelper`, jesteśmy gotowi na przygotowanie naszej wiadomości e-mail. Procedura przypomina wyżej opisaną. Jedyna różnica polega na tym, że zamiast wywoływać metody dla samej wiadomości, wywołujemy je dla obiektu pomocniczego:

```
String spitterName = spittle.getSpitter().getFullName();
helper.setFrom("noreply@spitter.com");
helper.setTo(to);
helper.setSubject("Nowy spittle od " + spitterName);
```

```
helper.setText(spitterName + " pisze: " + spittle.getText());
```

Teraz jedyną rzeczą, jaką musimy jeszcze zrobić przed wysłaniem wiadomości, jest dołączenie załącznika — obrazu z kuponem. W tym celu należy załadować obraz jako zasób, a następnie przekazać ten zasób, wywołując metodę `addAttachment()` obiektu pomocniczego:

```
FileSystemResource couponImage =
    new FileSystemResource("/collateral/coupon.png");
helper.addAttachment("Coupon.png", couponImage);
```

Korzystamy tu z klasy `FileSystemResource`, aby załadować plik *coupon.png* ze ścieżki do klas. Następnie wywołujemy metodę `addAttachmenet()`. Pierwszym parametrem jest nazwa załącznika w wiadomości. Drugi to zasób obrazka.

Wieloczęściowa wiadomość została zbudowana. Jest już więc gotowa do wysłania. Kompletną metodę `sendSpittleEmailWithAttachment()` pokazano na listingu 19.2.

Listing 19.2. Wysyłanie wiadomości e-mail z załącznikami przy użyciu klasy MimeMessageHelper

```
public void sendSpittleEmailWithAttachment(
          String to, Spittle spittle) throws MessagingException {
  MimeMessage message = mailSender.createMimeMessage();
  MimeMessageHelper helper =
        new MimeMessageHelper(message, true); <-- Tworzy obiekt pomocniczy
  String spitterName = spittle.getSpitter().getFullName();
  helper.setFrom("noreply@spitter.com");
  helper.setTo(to);
  helper.setSubject("Nowy spittle od " + spitterName);
  helper.setText(spitterName + " pisze: " + spittle.getText());
  FileSystemResource couponImage =
      new FileSystemResource("/collateral/coupon.png");
  helper.addAttachment("Coupon.png", couponImage); <-- Dodaje załącznik
  mailSender.send(message);
}
```

Dodawanie załączników to tylko jedna z możliwości oferowanych przez wieloczęściowe wiadomości e-mail. Określając treść wiadomości jako HTML, możemy stworzyć wiadomość, która będzie wyglądała dużo lepiej niż zwykły tekst. Zobaczmy, jak wysyłać ciekawie wyglądające wiadomości e-mail za pomocą klasy Springa `MimeMessageHelper`.

19.2.2. Wysyłanie wiadomości e-mail z bogatą zawartością

Wysyłanie wiadomości w formacie HTML niewiele się różni od wysyłania zwykłego tekstu. Kluczem jest ustawienie tekstu wiadomości jako HTML. Wystarczy do tego przekazanie metodzie `setText()` obiektu pomocniczego łańcucha HTML oraz wartości `true` w drugim parametrze:

```
helper.setText("<html><body><img src='cid:spitterLogo'>" +
  "<h4>" + spittle.getSpitter().getFullName() + " pisze...</h4>" +
  "<i>" + spittle.getText() + "</i>" +
  "</body></html>", true);
```

Drugi parametr wskazuje, że przekazywany tekst ma format HTML, co spowoduje ustawienie odpowiedniego typu zawartości części wiadomości.

Zwróć uwagę na znacznik `<img>` w przekazywanej treści HTML, który ma za zadanie wyświetlić logo aplikacji Spitter w wiadomości. Atrybut `src` mógłby przyjąć wartość standardowego URL typu `http:`, aby pobrać logo aplikacji z sieci. Tu jednak osadziliśmy obraz logo w samej wiadomości. Wartość `cid:spitterLogo` wskazuje, że jedna z części wiadomości zawierać będzie obraz, identyfikowany jako `spitterLogo`.

Dodanie osadzonego obrazka do wiadomości odbywa się na podobnej zasadzie, co dodawanie załącznika. Zamiast wywołania metody `addAttachment()` obiektu pomocniczego trzeba wywołać metodę `addInline()`:

```
ClassPathResource image =
    new ClassPathResource("spitter_logo_50.png");
helper.addInline("spitterLogo", image);
```

Pierwszy parametr metody `addInline()` identyfikuje obraz wewnętrzny (ang. *inline image*) jako ten sam, który został wcześniej podany w atrybucie `src` znacznika `<img>`. Drugi parametr jest referencją do zasobu obrazka, tutaj utworzonego za pomocą instancji `ClassPathResource`, która pobiera obraz ze ścieżki do klas aplikacji.

Poza nieco innym wywołaniem `setText()` i użyciem metody `addInline()` wysyłanie wiadomości e-mail z bogatą zawartością niewiele różni się od wysyłania wiadomości ze zwykłym tekstem i załącznikami. Oto nowa metoda `sendRichSpitterEmail()`, dla porównania:

```
public void sendRichSpitterEmail(String to, Spittle spittle)
                                 throws MessagingException {
  MimeMessage message = mailSender.createMimeMessage();
  MimeMessageHelper helper = new MimeMessageHelper(message, true);
  helper.setFrom("noreply@spitter.com");
  helper.setTo("craig@habuma.com");
  helper.setSubject("Nowy spittle od " +
          spittle.getSpitter().getFullName());
  helper.setText("<html><body><img src='cid:spitterLogo'>" +
    "<h4>" + spittle.getSpitter().getFullName() + " pisze...</h4>" +
    "<i>" + spittle.getText() + "</i>" +
    "</body></html>", true);
  ClassPathResource image =
      new ClassPathResource("spitter_logo_50.png");
  helper.addInline("spitterLogo", image);
  mailSender.send(message);
}
```

Określa kod HTML będący treścią wiadomości

Dodaje obraz wewnętrzny

Potrafimy już wysyłać wiadomości e-mail z bogatą zawartością i z osadzonymi obrazkami! Można by na tym poprzestać i zostawić kod w jego obecnej formie. Fakt, że treść wiadomości powstaje w wyniku łączenia łańcuchów, nie daje mi jednak spokoju. Zanim zamkniemy temat poczty elektronicznej, zobaczmy, jak zamiast łączenia łańcuchów wykorzystać do tworzenia wiadomości szablon.

19.3. Tworzenie wiadomości e-mail przy użyciu szablonów

Problem z łączeniem łańcuchów przy tworzeniu wiadomości polega na tym, że trudno przewidzieć, jak taka wiadomość będzie wyglądać. Już sam kod HTML może być trudny do zwizualizowania, a wymieszanie go dodatkowo z kodem Javy na pewno w tym nie pomaga. Poza tym warto wyodrębnić układ wiadomości do szablonu, nad którym może pracować grafik (z awersją do kodu Javy).

Potrzebujemy sposobu, by wyrazić układ wiadomości e-mail w formie HTML, a następnie przekształcić tak otrzymany szablon w łańcuch, który zostanie przekazany do metody `setText()` obiektu pomocniczego. Jeśli chodzi o przekształcanie szablonów na łańcuchy znaków, dostępnych jest kilka rozwiązań, w tym Apache Velocity oraz Thymeleaf. Zobaczmy zatem, jak tworzyć wiadomości e-mail z bogatą zawartością przy użyciu obu tych rozwiązań, a zaczniemy od Velocity.

19.3.1. Tworzenie wiadomości e-mail przy użyciu Velocity

Apache Velocity jest mechanizmem przetwarzania szablonów o ogólnym przeznaczeniu, opracowanym przez fundację Apache. Jest on dostępny już od długiego czasu i był używany do przeróżnych zadań, zaczynając od generowania kodu, a kończąc na rozwiązaniu do generowania kodu HTML, stanowiącym alternatywę dla JSP. Velocity można także wykorzystywać do generowania wiadomości e-mail o bogatej zawartości, co niebawem zrobimy.

Aby użyć Velocity do planowania układu wiadomości e-mail, musimy najpierw dowiązać `VelocityEngine` do `SpitterEmailServiceImpl`. Spring udostępnia wygodny komponent fabryki nazwany `VelocityEngineFactoryBean`, który wyprodukuje obiekt `Velocity ↪Engine` w kontekście aplikacji Springa. Deklaracja `VelocityEngineFactoryBean` wygląda następująco:

```
@Bean
public VelocityEngineFactoryBean velocityEngine() {
  VelocityEngineFactoryBean velocityEngine =
      new VelocityEngineFactoryBean();

  Properties props = new Properties();
  props.setProperty("resource.loader", "class");
  props.setProperty("class.resource.loader.class",
          ClasspathResourceLoader.class.getName());
  velocityEngine.setVelocityProperties(props);
  return velocityEngine;
}
```

Jedyną właściwością, którą musimy ustawić, deklarując komponent `VelocityEngineFac ↪toryBean`, jest `velocityProperties`. W tym przypadku skonfigurowaliśmy ją do ładowania szablonów ze ścieżki do klas (więcej szczegółów na temat konfiguracji Velocity znajdziesz w dokumentacji projektu).

Musimy teraz dowiązać silnik `VelocityEngine` do `SpitterEmailServiceImpl`. Ponieważ komponent `SpitterEmailServiceImpl` jest automatycznie rejestrowany przez skaner komponentów, możemy wykorzystać adnotację `@Autowired`, aby automatycznie dowiązać właściwość `velocityEngine`:

```
@Autowired
VelocityEngine velocityEngine;
```

Mając do dyspozycji właściwość `velocityEngine`, możemy jej użyć do przekształcenia szablonu Velocity w łańcuch, który posłuży za tekst naszej wiadomości e-mail. Pomoże nam w tym klasa Springa `VelocityEngineUtils`, dzięki której bez trudu zjednoczymy szablon Velocity i dane modelu w obiekt `String`, na przykład w taki sposób:

```
Map<String, String> model = new HashMap<String, String>();
model.put("spitterName", spitterName);
model.put("spittleText", spittle.getText());
String emailText = VelocityEngineUtils.mergeTemplateIntoString(
        velocityEngine, "emailTemplate.vm", model );
```

Przygotowanie do przetworzenia szablonu rozpoczynamy od utworzenia mapy przechowującej wykorzystywane przez szablon dane modelu. We wcześniejszym przykładzie potrzebowaliśmy pełnej nazwy spittera oraz tekstu jego spittle'a, podobnie będzie tym razem. Do wygenerowania tekstu wiadomości e-mail wystarczy nam wywołanie metody `mergeTemplateIntoString()` klasy `VelocityEngineUtils`. Jako argumenty przekazujemy jej silnik Velocity, ścieżkę do szablonu (względem katalogu głównego ścieżki do klas) i mapę modelu.

Jeśli chodzi o kod Javy, pozostaje tylko przekazanie scalonego tekstu wiadomości e-mail do metody `setText()` obiektu pomocniczego:

```
helper.setText(emailText, true);
```

Sam szablon znajduje się natomiast w katalogu głównym ścieżki do klas w pliku o nazwie *emailTemplate.vm* i następującej zawartości:

```
<html>
<body>
  <img src='cid:spitterLogo'>
  <h4>${spitterName} pisze...</h4>
  <i>${spittleText}</i>
</body>
</html>
```

Jak widać, kod szablonu jest dużo bardziej czytelny niż jego wcześniejsza wersja, wykorzystująca łączenie łańcuchów. Ułatwia to konserwację i edycję pliku. Na rysunku 19.2 pokazano przykładową wiadomość e-mail, powstałą na skutek użycia szablonu.

Patrząc na rysunek 19.2, widzimy, że można by nad szablonem jeszcze trochę popracować, aby poprawić nieco wygląd wiadomości. To zadanie, jak zwykle w takich sytuacjach, pozostawiam do wykonania przez czytelnika.

Velocity jest używany od wielu lat i stanowi preferowane rozwiązanie w wielu różnych sytuacjach. Jednak w rozdziale 6. poznaliśmy jeszcze inny mechanizm obsługi szablonów, który staje się coraz popularniejszy. Zobaczmy zatem, jak utworzyć wiadomość e-mail za pomocą Thymeleaf.

Rysunek 19.2. Szablon Velocity i parę osadzonych obrazków pozwalają ozdobić niezbyt atrakcyjną wiadomość e-mail

19.3.2. Stosowanie Thymeleaf do tworzenia wiadomości e-mail

Jak już wspominałem w rozdziale 6., Thymeleaf jest bardzo atrakcyjnym mechanizmem obsługi szablonów w przypadku tworzenia kodu HTML, gdyż pozwala na jego edycję w trybie WYSIWYG[1]. W odróżnieniu od JSP oraz Velocity, szablony Thymeleaf nie korzystają z żadnych specjalnych bibliotek znaczników ani specjalnego kodu. Ułatwia to projektantom tworzenie tych szablonów w dowolnych programach bez obawy o to, że nie będą one w stanie poradzić sobie ze specjalnymi znacznikami.

Kiedy skonwertujemy szablon wiadomości e-mail do postaci szablonu Thymeleaf, łatwo zauważymy, dlaczego nadaje się on do edycji w trybie WYSIWYG:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
  <img src="spitterLogo.png" th:src='cid:spitterLogo'>
  <h4><span th:text="${spitterName}">Craig Walls</span> pisze...</h4>
  <i><span th:text="${spittleText}">Witam serdecznie!</span></i>
</body>
</html>
```

Warto zwrócić uwagę, że w tym szablonie nie ma żadnych specjalnych znaczników (które występują w kodzie JSP). I choć odwołania do atrybutów modelu mają postać `${}`, to jednak pojawiają się one wyłącznie w atrybutach, a nie, jak to jest w przypadku Velocity, w zawartości elementów. Taki szablon bez trudu można otworzyć w dowolnej przeglądarce i zobaczyć, jak będzie wyglądał w swojej ostatecznej postaci, i to bez wcześniejszego przetwarzania go przy użyciu silnika Thymeleaf.

---

[1] Ang. *What You See Is What You Get* —— otrzymujesz to, co widzisz; kod HTML tworzony w takim trybie jest prezentowany w edytorze w takiej postaci, jaką przyjmie po wyświetleniu w przeglądarce — *przyp. tłum.*

Stosowanie Thymeleaf do generowania i wysyłania wiadomości e-mail przypomina rozwiązanie wykorzystujące Velocity:

```
Context ctx = new Context();
ctx.setVariable("spitterName", spitterName);
ctx.setVariable("spittleText", spittle.getText());
String emailText = thymeleaf.process("emailTemplate.html", ctx);
...
helper.setText(emailText, true);
mailSender.send(message);
```

Pierwszą czynnością, jaką należy wykonać, jest utworzenie instancji klasy `Context` i zapisanie w niej danych modelu. Odpowiada to przygotowaniu i wypełnieniu danymi obiektu `Map`, którego używaliśmy w poprzednim przykładzie podczas korzystania z Velocity. Następnie przetwarzamy szablon za pomocą silnika Thymeleaf, wywołując w tym celu metodę `process()`, która scala z szablonem dane modelu zapisane w kontekście. Tak uzyskany tekst zapisujemy jako treść wiadomości e-mail przy użyciu obiektu pomocniczego, po czym wysyłamy wiadomość za pomocą komponentu `mailSender`.

Jak widać, to rozwiązanie jest całkiem proste. Pozostaje jednak pytanie, skąd bierze się silnik Thymeleaf (reprezentowany przez zmienną `thymeleaf`).

Otóż jest on tym samym komponentem `SpringTemplateEngine`, który w rozdziale 6. skonfigurowaliśmy w celu generowania widoków HTML. Jednakże w tym przypadku wstrzykujemy go do komponentu `SpitterEmailServiceImpl` przy użyciu wstrzykiwania przez konstruktor:

```
@Autowired
private SpringTemplateEngine thymeleaf;

@Autowired
public SpitterEmailServiceImpl(SpringTemplateEngine thymeleaf) {
  this.thymeleaf = thymeleaf;
}
```

Niemniej jednak w komponencie `SpringTemplateEngine` musimy wprowadzić jedną niewielką zmianę. W takiej postaci, w jakiej przygotowaliśmy go w rozdziale 6., pozwala on wyłącznie na przetwarzanie szablonów określanych względem kontekstu serwletu. Nasze szablony wiadomości e-mail muszą być natomiast lokalizowane względem ścieżki klas. Dlatego oprócz producenta szablonów `ServletContextTemplateResolver` będziemy także potrzebowali producenta `ClassLoaderTemplateResolver`:

```
@Bean
public ClassLoaderTemplateResolver emailTemplateResolver() {
  ClassLoaderTemplateResolver resolver =
        new ClassLoaderTemplateResolver();
  resolver.setPrefix("mail/");
  resolver.setTemplateMode("HTML5");
  resolver.setCharacterEncoding("UTF-8");
  setOrder(1);
  return resolver;
}
```

Konfiguracja tego producenta w znacznej mierze przypomina konfigurowanie producenta `ServletContextTemplateResolver`. Warto jednak zwrócić uwagę, że właściwości `prefix` nadano wartość `mail/`, co oznacza, że szablony Thymeleaf powinny być umieszczone w katalogu *mail*, znajdującym się w głównym katalogu ścieżki klas. A zatem nasz plik szablonu wiadomości musi mieć nazwę *emailTemplate.html* i znajdować się w katalogu *mail* umieszczonym gdzieś wewnątrz głównego katalogu ścieżki klas.

Co więcej, ponieważ obecnie mamy do dyspozycji dwóch producentów szablonów, musimy określić, który z nich będzie ważniejszy. Służy do tego właściwość `order`. W producencie `ClassLoaderTemplateResolver` właściwości tej przypisaliśmy wartość `1`, natomiast w konfiguracji producenta `ServletContextTemplateResolver` zmienimy ją na wartość `2`:

```
@Bean
public ServletContextTemplateResolver webTemplateResolver() {
  ServletContextTemplateResolver resolver =
          new ServletContextTemplateResolver();
  resolver.setPrefix("/WEB-INF/templates/");
  resolver.setTemplateMode("HTML5");
  resolver.setCharacterEncoding("UTF-8");
  setOrder(2);
  return resolver;
}
```

Jedyną rzeczą, jaka pozostaje nam jeszcze do zrobienia, jest zmiana konfiguracji komponentu `SpringTemplateEngine`, tak by używał on obu producentów szablonów:

```
@Bean
public SpringTemplateEngine templateEngine(
    Set<ITemplateResolver> resolvers) {
  SpringTemplateEngine engine = new SpringTemplateEngine();
  engine.setTemplateResolvers(resolvers);
  return engine;
}
```

Wcześniej dysponowaliśmy tylko jednym producentem szablonów, mogliśmy go zatem wstrzyknąć do właściwości `templateResolver` komponentu `SpringTemplateEngine`. Jednak teraz dysponujemy dwoma takimi producentami, dlatego musimy ich wstrzykiwać jako elementy zbioru (obiektu typu `Set`) do właściwości o nazwie `templateResolvers` (zwróć uwagę na końcówkę „s”, oznaczającą liczbę mnogą).

19.4. Podsumowanie

Wiadomości e-mail stanowią ważną formę komunikacji międzyludzkiej, a bardzo często są również istotnym sposobem komunikacji pomiędzy aplikacjami a ludźmi. Spring bazuje na możliwościach obsługi poczty elektronicznej, udostępnianych przez Javę; ukrywa przy tym API JavaMail, by ułatwić konfigurowanie i tworzenie wiadomości e-mail w aplikacjach Spring.

W tym rozdziale dowiedziałeś się, jak używać warstwy abstrakcji komunikacji e-mail w Springu, aby wysyłać proste wiadomości. Następnie poszliśmy o krok dalej i nauczyłeś się wysyłać wiadomości zawierające załączniki i sformatowane przy wykorzystaniu kodu HTML. Na końcu omówiłem sposoby generowania wiadomości e-mail o bogatej zawartości przy użyciu silników do obsługi szablonów, takich jak Velocity i Thymeleaf, które pozwoliły nam uniknąć tworzenia kodu HTML poprzez konkatenację łańcuchów znaków.

W następnym rozdziale dowiesz się, jak stosować Java Management Extensions (JMX) do wyposażania naszych komponentów w możliwości zarządzania i obsługi powiadomień.

20 Zarządzanie komponentami Springa za pomocą JMX

W tym rozdziale omówimy:

- Udostępnianie komponentów Springa jako komponentów zarządzanych
- Zdalne zarządzanie komponentami Springa
- Obsługę powiadomień JMX

Obsługa wstrzykiwania zależności przez Springa pozwala na konfigurację właściwości komponentu w aplikacji. Jednak po wdrożeniu i uruchomieniu aplikacji samo wstrzykiwanie zależności niewiele może pomóc w zmianie konfiguracji. Przypuśćmy, że chcemy zajrzeć do aplikacji głębiej i skonfigurować ją „w locie". Z pomocą przyjdzie nam wtedy technologia JMX (ang. *Java Management Extensions*).

JMX jest technologią, która wyposaża aplikacje w narzędzia do zarządzania, monitorowania i konfiguracji. Pierwotnie dostępna jako osobne rozszerzenie Javy, dziś jest standardową częścią dystrybucji Java 5.

Kluczowym komponentem aplikacji, wyposażonej przez JMX w narzędzia do zarządzania, jest **komponent zarządzany** (ang. *managed bean*, w skrócie *MBean*). MBean jest komponentem JavaBean, udostępniającym pewne metody, które definiują interfejs zarządzania. Specyfikacja JMX wyróżnia cztery typy komponentów zarządzanych:

- **Standardowe** (ang. *standard MBeans*) — Komponenty zarządzane, których interfejs zarządzania jest wyznaczany za pomocą refleksji ustalonego interfejsu Javy, implementowanego przez klasę komponentu.

- **Dynamiczne** (ang. *dynamic MBeans*) — Komponenty zarządzane, których interfejs zarządzania wyznaczany jest w trakcie wykonania, poprzez wywołanie metod interfejsu `DynamicMBean`. Ponieważ interfejs zarządzania nie jest określony statycznym interfejsem, może różnić się w trakcie wykonania.
- **Otwarte** (ang. *open MBeans*) — Specjalny rodzaj dynamicznych komponentów zarządzanych, których atrybuty i operacje ograniczają się do typów podstawowych, opakowań klas typów podstawowych i wszystkich typów, które mogą zostać rozłożone na typy podstawowe lub opakowania typów podstawowych.
- **Modelowe** (ang. *model MBeans*) — Specjalny rodzaj dynamicznych komponentów zarządzanych, który łączy interfejs zarządzania z zarządzanym zasobem. Modelowe komponenty zarządzane są nie tyle tworzone, co deklarowane. Najczęściej produkowane są przez fabrykę, która używa metadanych do konstrukcji interfejsu zarządzania.

Moduł JMX Springa pozwala na eksport komponentów Springa w postaci modelowych komponentów zarządzanych. Dzięki temu można zajrzeć w głąb aplikacji i manipulować konfiguracją — nawet w trakcie działania aplikacji. Zobaczmy, jak włączyć JMX dla naszej aplikacji Springa, aby móc zarządzać komponentami w kontekście aplikacji Springa.

20.1. Eksportowanie komponentów Springa w formie MBean

JMX może być wykorzystany do zarządzania komponentami naszej aplikacji Spitter na kilka sposobów. Aby nie komplikować zbytnio, zacznijmy od drobnej zmiany kontrolera strony głównej `SpittleController`, przedstawionego na listingu 5.10, polegającej na dodaniu nowej właściwości `spittlesPerPage`, która odpowiedzialna będzie za liczbę spittle'ów na stronie:

```
public static final int DEFAULT_SPITTLES_PER_PAGE = 25;

private int spittlesPerPage = DEFAULT_SPITTLES_PER_PAGE;

public void setSpittlesPerPage(int spittlesPerPage) {
   this.spittlesPerPage = spittlesPerPage;
}

public int getSpittlesPerPage() {
   return spittlesPerPage;
}
```

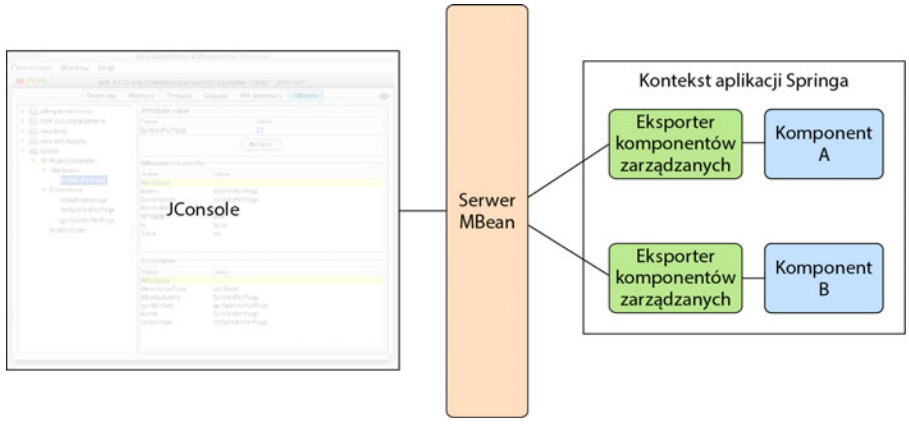
Wcześniej, kiedy kontroler strony głównej wywoływał metodę `getRecentSpittles()` usługi `SpitterService`, przekazywał on domyślną wartość `20`, co powodowało wyświetlenie na stronie nie więcej niż 20 spittle'ów. Teraz, zamiast ostatecznie podejmować tę decyzję w trakcie kompilacji przy wykorzystaniu wartości podanej na stałe w kodzie, używając JMX, pozostawiamy ją otwartą na ewentualne zmiany w czasie wykonania. Nowa właściwość `spittlesPerPage` jest pierwszym krokiem do tego celu.

Sama właściwość `spittlesPerPage` nie umożliwi zewnętrznej konfiguracji liczby spittle'ów wyświetlanych na stronie głównej. Jest taką samą właściwością komponentu,

jak każda inna. Potrzebujemy czegoś więcej i dlatego udostępnimy komponent kontrolera strony głównej w formie MBean. Właściwość `spittlesPerPage` zostanie wtedy udostępniona jako **zarządzany atrybut** (ang. *managed attribute*) komponentu MBean, dzięki czemu można będzie zmienić jej wartość w trakcie wykonania.

Eksporter `MBeanExporter` Springa jest kluczem do wzbogacenia komponentów o mechanizm JMX w Springu. `MBeanExporter` jest komponentem, który eksportuje jeden lub więcej komponentów Springa w formie modelowych komponentów zarządzanych do **serwera MBean**. Serwer MBean (zwany również **agentem MBean**) jest kontenerem zawierającym komponenty zarządzane i umożliwiającym dostęp do nich.

Jak pokazano na rysunku 20.1, eksport komponentów Springa w postaci komponentów zarządzanych JMX umożliwia narzędziom do zarządzania opartym na JMX, jak na przykład JConsole czy VisualVM, wgląd w uruchomioną aplikację: podejrzenie właściwości komponentów i wywołanie ich metod.



Rysunek 20.1. MBeanExporter eksportuje właściwości i metody komponentów Springa jako atrybuty i operacje JMX do serwera MBean. Dzięki temu ostatniemu narzędzia do zarządzania JMX, jak na przykład JConsole, mogą zajrzeć do uruchomionej aplikacji

Poniższa metoda z adnotacją `@Bean` deklaruje `MBeanExporter` w Springu celem eksportu komponentu `spittleController` w formie modelowego MBean:

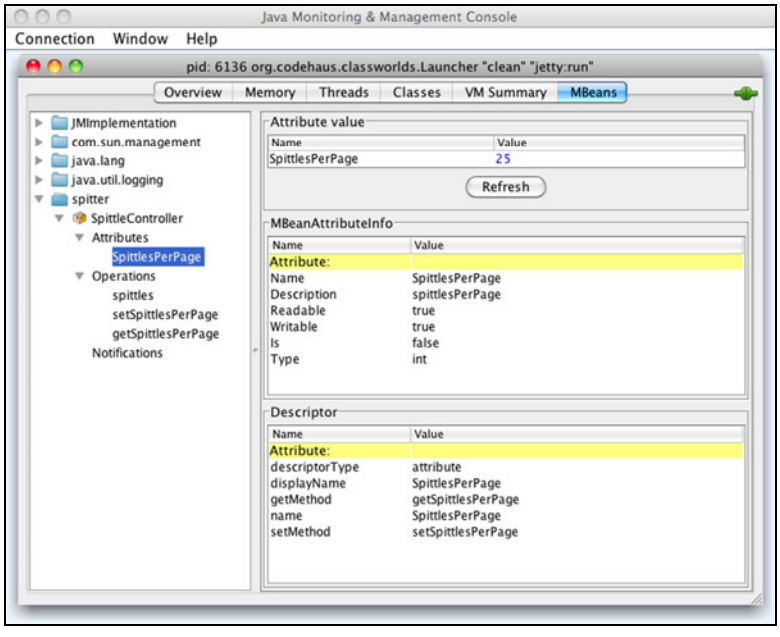
```
@Bean
public MBeanExporter mbeanExporter(SpittleController spittleController) {
  MBeanExporter exporter = new MBeanExporter();
  Map<String, Object> beans = new HashMap<String, Object>();
  beans.put("spitter:name=SpittleController", spittleController);
  exporter.setBeans(beans);
  return exporter;
}
```

W swojej najprostszej formie `MBeanExporter` może zostać skonfigurowany za pomocą swojej właściwości `beans`, na podstawie wstrzykniętej mapy, złożonej z jednego lub więcej komponentów, które chcielibyśmy udostępnić jako modelowe komponenty zarządzane w JMX. Kluczem każdego elementu `<entry>` jest nazwa, która zostanie nadana

komponentowi zarządzanemu (złożona z nazwy domeny zarządzania i pary klucz-wartość — `spitter:name=SpittleController` w przypadku komponentu MBean `Spittle➥Controller`). Jako wartość elementu `<entry>` podana została referencja do komponentu Springa, który ma zostać wyeksportowany. Tutaj eksportujemy komponent `spittle➥Controller`, aby uzyskać dostęp do jego właściwości w trakcie wykonania poprzez JMX.

Dzięki eksporterowi `MBeanExporter` komponent `spittleController` zostanie wyeksportowany jako modelowy komponent zarządzany do serwera MBean, gdzie będzie mógł być zarządzany pod nazwą `SpittleController`. Na rysunku 20.2 pokazano komponent zarządzany `SpittleController`, oglądany za pomocą narzędzia JConsole.

Jak widać po lewej stronie rysunku 20.2, wszystkie publiczne składowe komponentu `SpittleController` eksportowane są w postaci operacji i atrybutów komponentu zarządzanego. To o wiele więcej, niżbyśmy chcieli. Tak naprawdę, chcemy skonfigurować tylko właściwość `spittlesPerPage`. Nie musimy wywoływać metody `spittles()` ani dotykać żadnej innej części kontrolera `SpittleController`. Trzeba zatem jakoś określić, które atrybuty i operacje powinny być dostępne.



Rysunek 20.2. SpittleController wyeksportowany jako komponent zarządzany i oglądany za pomocą narzędzia JConsole

W uzyskaniu dokładniejszej kontroli nad atrybutami i operacjami komponentu zarządzanego Spring może nam pomóc na kilka sposobów. Oto niektóre z nich:

- Imienna deklaracja metod komponentu do udostępnienia/zignorowania.
- Ukrycie komponentu za interfejsem, pozwalające wybrać metody do udostępnienia.
- Określenie zarządzanych atrybutów i operacji poprzez oznaczenie komponentu adnotacją.

Wypróbujmy każdy z tych sposobów i zobaczmy, który najlepiej nadaje się do naszego komponentu SpittleController. Zaczniemy od wybrania metod do udostępnienia na podstawie nazwy.

Skąd wziąć serwer MBean?

MBeanExporter przyjmuje, że działa w środowisku serwera aplikacji (na przykład Tomcat) lub w innym kontekście, który oferuje serwer MBean. Jeśli jednak aplikacja uruchamiana jest niezależnie lub w kontenerze niedostarczającym serwera MBean, musi on zostać skonfigurowany w kontekście Springa.

W przypadku konfigurowania Springa przy użyciu kodu XML można wykorzystać element <context:mbean-erver>. Przy konfigurowaniu w kodzie Javy będziemy musieli zastosować bardziej bezpośrednie rozwiązanie i skonfigurować komponent typu MBeanServerFactoryBean() (co w kodzie XML robi za nas element <context:mbean-server>).

Komponent MBeanServerFactoryBean tworzy serwer MBean jako komponent dostępny w kontekście aplikacji Springa. Domyślnie identyfikatorem tego komponentu będzie mbeanServer. Dysponując tą informacją, możemy już dowiązać serwer MBean do właściwości server eksportera MBeanExporter i określić w ten sposób, którego serwera MBean należy używać do udostępniania komponentu zarządzanego.

20.1.1. Udostępnianie metod na podstawie nazwy

Kluczem do ograniczenia zakresu operacji i atrybutów, które będą wyeksportowane w komponencie zarządzanym, są **asemblery informacji MBean**. Jednym z takich asemblerów jest MethodNameBasedMBeanInfoAssembler. Należy mu dostarczyć listę nazw metod, które mają zostać wyeksportowane jako operacje komponentu zarządzanego. W przypadku komponentu SpittleController chcemy wyeksportować jako zarządzany atrybut tylko właściwość spittlesPerPage. Czy asembler bazujący na nazwach metod może nam pomóc w eksporcie zarządzanego atrybutu?

Przypomnij sobie, że zgodnie z regułami komponentów JavaBean (choć niekoniecznie z regułami komponentów Springa) spittlesPerPage jest właściwością, ponieważ posiada dwie odpowiadające jej metody dostępowe: setSpittlesPerPage() i getSpittlesPerPage(). Aby nie udostępniać całego komponentu zarządzanego, musimy nakazać naszemu asemblerowi umieszczenie tylko tych dwóch metod w interfejsie komponentu zarządzanego. Poniższa deklaracja MethodNameBasedMBeanInfoAssembler wymienia tylko te metody:

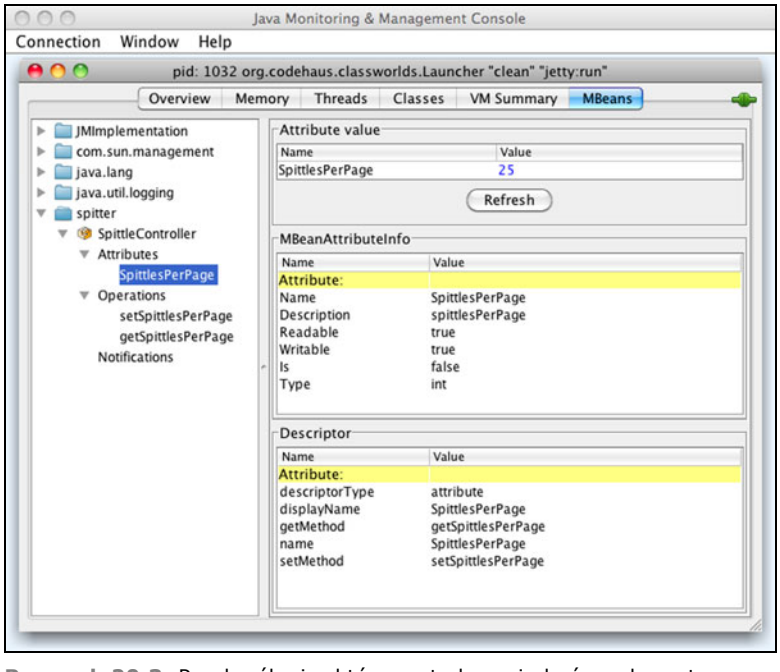
```
@Bean
public MethodNameBasedMBeanInfoAssembler assembler() {
  MethodNameBasedMBeanInfoAssembler assembler =
        new MethodNameBasedMBeanInfoAssembler();
  assembler.setManagedMethods(new String[] {
    "getSpittlesPerPage", "setSpittlesPerPage"
  });
  return assembler;
}
```

Właściwość managedMethods przyjmuje listę nazw metod. Metody te zostaną udostępnione jako zarządzane operacje komponentu MBean. Ponieważ są one metodami dostępowymi, komponent zarządzany zawierał będzie również przypisaną im właściwość spittlesPerPage.

Aby skorzystać z asemblera, musimy dowiązać go do eksportera `MBeanExporter`:

```
@Bean
public MBeanExporter mbeanExporter(
    SpittleController spittleController,
    MBeanInfoAssembler assembler) {
  MBeanExporter exporter = new MBeanExporter();
  Map<String, Object> beans = new HashMap<String, Object>();
  beans.put("spitter:name=SpittleController", spittleController);
  exporter.setBeans(beans);
  exporter.setAssembler(assembler);
  return exporter;
}
```

Jeśli uruchomimy teraz aplikację, właściwość `spittlesPerPage` będzie dostępna jako zarządzany atrybut `SpittleController`, metoda `spittles()` nie zostanie natomiast udostępniona jako zarządzana operacja. Rysunek 20.3 pokazuje, jak to wygląda w JConsole.



Rysunek 20.3. Po określeniu, które metody mają być wyeksportowane w komponencie zarządzanym SpittleController, metody spittles() nie ma już wśród zarządzanych operacji

Kolejnym asemblerem bazującym na nazwach metod jest `MethodExclusionMBeanInfo➥Assembler`. Działanie tego asemblera jest odwróceniem działania `MethodNameBased➥MBeanInfoAssembler`. Zamiast określać, które metody mają być udostępnione jako zarządzane operacje, `MethodExclusionMBeanInfoAssembler` przyjmuje listę metod, które *nie* powinny być udostępnione. Poniżej pokazano przykładowe użycie `MethodExclusion➥MBeanInfoAssembler` do wykluczenia metody `spittles()` z grona metod udostępnianych jako zarządzane operacje:

```
@Bean
public MethodExclusionMBeanInfoAssembler assembler() {
  MethodExclusionMBeanInfoAssembler assembler =
        new MethodExclusionMBeanInfoAssembler();
  assembler.setIgnoredMethods(new String[] {
      "spittles"
  });
  return assembler;
}
```

Asemblery bazujące na nazwach metod są zrozumiałe i bardzo proste w użyciu. Ale wyobraź sobie, co by się stało, gdybyśmy chcieli wyeksportować kilka komponentów Springa w postaci MBean. Po pewnym czasie lista metod przekazywanych asemblerowi bardzo by się rozrosła. Może też się zdarzyć, że chcemy wyeksportować metodę jednego komponentu, podczas gdy inny komponent zawiera metodę o tej samej nazwie, której nie chcemy eksportować.

Wygląda na to, że podejście oparte na nazwach metod nie sprawdza się najlepiej przy eksportowaniu wielu komponentów zarządzanych. Sprawdźmy, czy użycie interfejsów okaże się do tego celu lepsze.

20.1.2. Użycie interfejsów do definicji operacji i atrybutów komponentu zarządzanego

`InterfaceBasedMBeanInfoAssembler` jest kolejnym asemblerem informacji MBean, pozwalającym na wykorzystanie interfejsów do wybrania metod komponentu, które mają być wyeksportowane jako operacje zarządzane przez komponent MBean. Od asemblerów bazujących na nazwach metod różni się tylko tym, że zamiast wymieniać nazwy metod do wyeksportowania, wymieniasz interfejsy, które definiują metody do wyeksportowania.

Załóżmy na przykład, że mamy interfejs `SpittleControllerManagedOperations`, zdefiniowany następująco:

```
package com.habuma.spittr.jmx;

public interface SpittleControllerManagedOperations {
  int getSpittlesPerPage();
  void setSpittlesPerPage(int spittlesPerPage);
}
```

Jako operacje do wyeksportowania wybraliśmy tu metody `setSpittlesPerPage()` i `get➥SpittlesPerPage()`. Podobnie jak wcześniej, te dwie metody dostępowe spowodują niejawny eksport związanej z nimi właściwości `spittlesPerPage` jako zarządzanego atrybutu. Aby użyć tego asemblera, wystarczy zastąpić asemblery bazujące na nazwach metod następującym komponentem:

```
@Bean
public InterfaceBasedMBeanInfoAssembler assembler() {
  InterfaceBasedMBeanInfoAssembler assembler =
        new InterfaceBasedMBeanInfoAssembler();
  assembler.setManagedInterfaces(
      new Class<?>[] { SpittleControllerManagedOperations.class }
  );
  return assembler;
}
```

Właściwość `managedInterfaces` przyjmuje listę interfejsów, które służą jako interfejsy operacji zarządzanych przez komponent MBean. W tym przykładzie —— interfejs `SpittleControllerManagedOperations`.

Nie jest to oczywiste, dlatego warto wspomnieć, że `SpittleController` nie musi jawnie implementować `SpittleControllerManagedOperations`. Interfejs ten jest potrzebny eksporterowi, nie musimy jednak implementować go bezpośrednio w kodzie. Kontroler `SpittleController` zapewne powinien implementować ten interfejs, choćby po to, by zapewnić spójność kontraktu pomiędzy komponentem MBean a implementującą go klasą.

Jedną z zalet korzystania z interfejsów do wybierania zarządzanych operacji jest możliwość zgrupowania wielu metod w kilku interfejsach, dzięki czemu konfiguracja asemblera `InterfaceBasedMBeanInfoAssembler` pozostaje czysta. Pozwala to zachować zwięzłość konfiguracji Springa, nawet przy eksporcie wielu komponentów zarządzanych.

Ostatecznie, zarządzane operacje muszą być gdzieś zadeklarowane — albo w konfiguracji Springa, albo w jakimś interfejsie. Co więcej, deklaracja zarządzanych operacji jest powieleniem istniejącego już kodu. Nazwy metod pojawiają się w formie deklaracji zarówno w interfejsie lub kontekście Springa, jak i w samej implementacji. Powtórzenie to jest konieczne ze względu na eksporter `MBeanExporter`.

Pozbycie się powielonego kodu to jedna z rzeczy, do których świetnie nadają się adnotacje Javy. Zobaczmy, jak oznaczyć komponent Springa, aby mógł zostać wyeksportowany jako komponent zarządzany.

20.1.3. Praca z komponentami MBean sterowanymi adnotacjami

Oprócz pokazanych dotychczas asemblerów informacji MBean, Spring posiada jeszcze asembler `MetadataMBeanInfoAssembler`. Może on zostać skonfigurowany, aby użycie adnotacji do nadania metodom komponentów statusu zarządzanych operacji i atrybutów było możliwe. Nie będziemy się nim jednak zajmować. Jego dowiązanie jest zbyt uciążliwe i niewarte zachodu dla samej możliwości użycia adnotacji.

Zamiast tego użyjemy elementu `<context:mbean-export>` z konfiguracyjnej przestrzeni nazw `context` Springa. Ten wygodny element wiąże eksporter komponentów zarządzanych i odpowiednie asemblery, włączając komponenty MBean sterowane adnotacjami w Springu. Wystarczy tylko użyć go zamiast wykorzystywanego przez nas wcześniej komponentu `MBeanExporter`:

```
<context:mbean-export server="mbeanServer" />
```

Teraz, aby przekształcić dowolny komponent Springa w komponent zarządzany, musimy go tylko oznaczyć adnotacją `@ManagedResource`, a jego metody adnotacjami `@Managed`
`↪Operation` lub `@ManagedAttribute`. Na listingu 20.1 pokazano przykład modyfikacji klasy `SpittleController`, przeprowadzonej, by umożliwić eksport komponentu w formie komponentu zarządzanego, z wykorzystaniem adnotacji.

Zastosowana na poziomie klasy adnotacja `@ManagedResource` wskazuje, że komponent powinien zostać wyeksportowany jako komponent zarządzany. Atrybut `objectName` określa domenę (`spitter`) i nazwę (`SpittleController`) komponentu zarządzanego.

Listing 20.1. Przekształcanie klasy SpittleController w komponent zarządzany za pomocą adnotacji

```
package com.habuma.spittr.mvc;

import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spittr.service.SpittrService;

@Controller
@ManagedResource(objectName="spitter:name=SpittleController") //
public class SpittleController {
  ...

  @ManagedAttribute //
  public void setSpittlesPerPage(int spittlesPerPage) {
    this.spittlesPerPage = spittlesPerPage;
  }

  @ManagedAttribute //
  public int getSpittlesPerPage() {
    return spittlesPerPage;
  }
}
```

Eksport SpittleController w formie MBean

Udostępnienie właściwości spittlesPerPage jako zarządzanego atrybutu

Obie metody dostępowe właściwości `spittlesPerPage` oznaczono adnotacją `@Managed`
`↪Attribute`, która wskazuje, że właściwość powinna zostać udostępniona jako zarządzany atrybut. Zauważ, że wcale nie musisz oznaczać tą adnotacją obu metod dostępowych. Jeśli opatrzysz nią tylko metodę `setSpittlesPerPage()`, ustawienie właściwości poprzez JMX będzie możliwe, natomiast odczyt jej wartości — nie. I odwrotnie — umieszczenie adnotacji przy metodzie `getSpittlesPerPage()` spowoduje, że właściwość będzie dostępna przez JMX w trybie tylko do odczytu.

Warto też zwrócić uwagę, że metody dostępowe, zamiast `@ManagedAttribute`, oznaczać można również adnotacją `@ManagedOperation`. Na przykład:

```
@ManagedOperation
public void setSpittlesPerPage(int spittlesPerPage) {
   this.spittlesPerPage = spittlesPerPage;
}

@ManagedOperation
public int getSpittlesPerPage() {
   return spittlesPerPage;
}
```

Zabieg ten spowoduje udostępnienie metod przez JMX, właściwość `spittlesPerPage` nie zostanie natomiast udostępniona jako zarządzany atrybut. Dzieje się tak, ponieważ przy udostępnianiu komponentu MBean metody z adnotacją `@ManagedOperation` traktowane są jak metody sensu stricte, nie jako metody dostępowe JavaBean. Dlatego

adnotacja `@ManagedOperations` powinna być zarezerwowana do udostępniania metod jako operacji komponentu MBean, `@ManagedAttribute` natomiast powinna służyć wyłącznie do udostępniania zarządzanych atrybutów.

20.1.4. Postępowanie przy konfliktach nazw komponentów zarządzanych

Omówiliśmy dotychczas kilka sposobów publikacji komponentu zarządzanego na serwerze MBean. W każdym z przypadków nadawaliśmy komponentowi zarządzanemu nazwę obiektu złożoną z domeny zarządzania i pary klucz-wartość. Zakładając, że nie istnieje komponent o identycznej nazwie, nie powinniśmy mieć problemów z jego publikacją. Co jednak zrobić w razie konfliktu nazw?

`MBeanExporter` zgłosi domyślnie wyjątek `InstanceAlreadyExistsException` przy próbie eksportu komponentu zarządzanego o takiej samej nazwie jak komponent znajdujący się już na serwerze MBean. To zachowanie można jednak zmienić, ustalając postępowanie w razie konfliktu za pomocą właściwości `registrationBehaviorName` eksportera `MBeanExporter` lub atrybutu `registration` elementu `<context:mbean-export>`.

Istnieją trzy sposoby rozwiązywania konfliktów nazw komponentów zarządzanych przy użyciu właściwości `registrationPolicy`:

- `FAIL_ON_EXISTING` — pozwala na wystąpienie błędu, jeśli istnieje komponent zarządzany o takiej samej nazwie (zachowanie domyślne).
- `IGNORE_EXISTING` — pozwala zignorować konflikt i nie rejestrować nowego komponentu zarządzanego.
- `REPLACE_EXISTING` — pozwala zastąpić istniejący komponent zarządzany nowym.

Jeżeli używasz eksportera `MBeanExporter`, możesz na przykład wybrać ignorowanie konfliktów, ustawiając właściwość `registrationPolicy` na `RegistrationPolicy.IGNORE_EXI` ➥`STING` w następujący sposób:

```
@Bean
public MBeanExporter mbeanExporter(
      SpittleController spittleController,
      MBeanInfoAssembler assembler) {
  MBeanExporter exporter = new MBeanExporter();
  Map<String, Object> beans = new HashMap<String, Object>();
  beans.put("spitter:name=SpittleController", spittleController);
  exporter.setBeans(beans);
  exporter.setAssembler(assembler);
  exporter.setRegistrationPolicy(RegistrationPolicy.IGNORE_EXISTING);
  return exporter;
}
```

Właściwość `registrationPolicy` akceptuje wartości typu wyliczeniowego `Registration` ➥`Policy`, reprezentujące trzy strategie postępowania przy konflikcie nazw.

Teraz, kiedy zarejestrowaliśmy już nasze komponenty MBean przy użyciu `MBeanEx` ➥`porter`, musimy mieć do nich dostęp, aby móc nimi zarządzać. Wiemy już, że w dostępie do lokalnego serwera MBean celem operowania na komponentach pomóc nam mogą narzędzia, takie jak JConsole. Ale narzędzia te nie umożliwiają programowego zarządzania komponentami MBean. Jak przetwarzać komponenty zarządzane w jednej

aplikacji z poziomu innej? Na szczęście, istnieje inny sposób dostępu do komponentów zarządzanych jako do zdalnych obiektów. Zbadajmy, jak obsługa Springa zdalnych komponentów zarządzanych pozwoli nam uzyskać standardowy dostęp do tych komponentów poprzez zdalny interfejs.

20.2. Zdalny dostęp do komponentów zarządzanych

Chociaż oryginalna specyfikacja JMX odniosła się do zdalnego zarządzania aplikacjami poprzez komponenty MBean, nie określiła konkretnego zdalnego protokołu ani API. Zadanie stworzenia własnych, często niestandardowych rozwiązań zdalnego dostępu spadło na dostawców JMX.

Potrzeba stworzenia standardu zdalnego dostępu JMX zaowocowała stworzeniem *JSR-160* (Java Management Extensions Remote API) przez Java Community Process. Specyfikacja ta definiuje standard zdalnego dostępu JMX, który wymaga przynajmniej wiązania RMI i opcjonalnie protokołu *JMXMP* (*JMX Messaging Protocol*).

W tym podrozdziale pokażemy, jak można pracować ze zdalnymi komponentami MBean w Springu. Zaczniemy od konfiguracji Springa, pozwalającej wyeksportować komponent zarządzany `SpittleController` jako zdalny komponent MBean. Potem zobaczymy, jak przetworzyć ten komponent zdalnie za pomocą Springa.

20.2.1. Udostępnianie zdalnych komponentów MBean

Najprostszym sposobem na udostępnienie naszych komponentów zarządzanych jako zdalnych obiektów jest konfiguracja fabryki serwerów połączeń Springa `ConnectorSer`
`↪verFactoryBean`:

```
@Bean
public ConnectorServerFactoryBean connectorServerFactoryBean() {
  return new ConnectorServerFactoryBean();
}
```

`ConnectorServerFactoryBean` tworzy i uruchamia serwer połączenia `JMXConnectorServer` według specyfikacji JSR-160. Domyślnie serwer nasłuchuje protokołu JMXMP na porcie 9875 — jest więc związany z adresem `service:jmx:jmxmp://localhost:9875`. Eksport komponentów zarządzanych za pomocą JMXMP to tylko jedna z możliwości.

W zależności od implementacji JMX można wybierać z różnych opcji zdalnego dostępu, w tym RMI, SOAP, Hessian/Burlap, a nawet Internet InterORB Protocol (IIOP). Aby zdefiniować inne zdalne wiązanie dla naszych komponentów zarządzanych, musimy ustawić właściwość `serviceUrl` komponentu `ConnectorServerFactoryBean`. Jeżeli na przykład do zdalnego dostępu do komponentów zarządzanych chcielibyśmy użyć RMI, `serviceUrl` przyjęłaby wartość:

```
@Bean
public ConnectorServerFactoryBean connectorServerFactoryBean() {
  ConnectorServerFactoryBean csfb = new ConnectorServerFactoryBean();
  csfb.setServiceUrl(
      "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/spitter");
  return csfb;
}
```

Dowiązujemy tu komponent `ConnectorServerFactoryBean` do rejestru RMI, który nasłuchuje na porcie 1099 lokalnego komputera. Oznacza to, że będziemy potrzebować również uruchomionego rejestru RMI, nasłuchującego na tym porcie. Jak być może pamiętasz z rozdziału 15., `RmiServiceExporter` może uruchomić rejestr RMI automatycznie. W tym przypadku nie używamy jednak `RmiServiceExporter`, rejestr RMI będziemy musieli zatem uruchomić, deklarując w Springu komponent `RmiRegistryFactoryBean` przy wykorzystaniu następującej metody z adnotacją `@Bean`:

```
@Bean
public RmiRegistryFactoryBean rmiRegistryFB() {
  RmiRegistryFactoryBean rmiRegistryFB = new RmiRegistryFactoryBean();
  rmiRegistryFB.setPort(1099);
  return rmiRegistryFB;
}
```

I to już wszystko! Nasze komponenty zarządzane są już dostępne przez RMI. Nic nam to jednak nie da, o ile dostęp ten nie będzie wykorzystywany. Zajmijmy się więc teraz stroną klienta zdalnego dostępu JMX i zobaczmy, jak dowiązać zdalny komponent MBean w Springu.

20.2.2. Dostęp do zdalnego komponentu MBean

Dostęp do zdalnego serwera MBean wymaga między innymi konfiguracji komponentu `MBeanServerConnectionFactoryBean` w kontekście Springa. Poniżej zadeklarowano komponent `MBeanServerConnectionFactoryBean`, który może zostać użyty do dostępu do zdalnego serwera opartego na RMI stworzonego przez nas w poprzednim punkcie:

```
@Bean
public MBeanServerConnectionFactoryBean connectionFactoryBean() {
  MBeanServerConnectionFactoryBean mbscfb =
        new MBeanServerConnectionFactoryBean();
  mbscfb.setServiceUrl(
      "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/spitter");
  return mbscfb;
}
```

Zgodnie z tym, co sugeruje jego nazwa, `MBeanServerConnectionFactoryBean` jest komponentem fabryki, który tworzy `MBeanServerConnection`. `MBeanServerConnection` funkcjonuje jako lokalny serwer pośredniczący dla zdalnego serwera MBean. Może on zostać dowiązany do właściwości komponentu, jak komponent `MBeanServerConnection`:

```
@Bean
public JmxClient jmxClient(MBeanServerConnection connection) {
  JmxClient jmxClient = new JmxClient();
  jmxClient.setMbeanServerConnection(connection);
  return jmxClient;
}
```

`MBeanServerConnection` oferuje kilka metod pozwalających na wysłanie zapytania do zdalnego serwera MBean i wywołania metod komponentów zarządzanych, które zawiera. Powiedzmy na przykład, że chcemy poznać liczbę komponentów MBean zarejestrowanych na zdalnym serwerze MBean. Poniższy fragment kodu pozwoli nam uzyskać tę informację:

```
int mbeanCount = mbeanServerConnection.getMBeanCount();
System.out.println("Znaleziono " + mbeanCount + " komponentów zarządzanych");
```

Możemy też wysłać do zdalnego serwera zapytanie o nazwy wszystkich komponentów zarządzanych, używając metody `queryNames()`:

```
java.util.Set mbeanNames = mbeanServerConnection.queryNames(null, null);
```

Dwa parametry metody `queryNames()` służą do zawężania rezultatów. Przekazując w obu wartość `null`, uzyskujemy pełną listę nazw wszystkich zarejestrowanych komponentów MBean.

Zapytania do zdalnego serwera MBean o liczbę komponentów i ich nazwy cieszą, ale nie posuwają naszej pracy ani trochę do przodu. Prawdziwa wartość zdalnego dostępu do serwera MBean wiąże się z operowaniem na atrybutach i wywoływaniem operacji komponentów MBean, zarejestrowanych na zdalnym serwerze.

Dostęp do atrybutów komponentu zarządzanego uzyskamy za pomocą metod `get➥Attribute()` i `setAttribute()`. Aby pobrać wartość atrybutu na przykład metodę `getAt➥tribute()` możemy wywołać w ten sposób:

```
String cronExpression = mbeanServerConnection.getAttribute(
    new ObjectName("spitter:name=SpittleController"), "spittlesPerPage");
```

Zmienimy ją natomiast, używając metody `setAttribute()`:

```
mbeanServerConnection.setAttribute(
    new ObjectName("spitter:name=SpittleController"),
    new Attribute("spittlesPerPage", 10));
```

Do wywołania operacji komponentu zarządzanego służy metoda `invoke()`. Poniższy przykład pokazuje wywołanie metody `setSpittlesPerPage()` komponentu zarządzanego `SpittleController`:

```
mbeanServerConnection.invoke(
    new ObjectName("spitter:name=SpittleController"),
    "setSpittlesPerPage",
    new Object[] { 100 },
    new String[] {"int"});
```

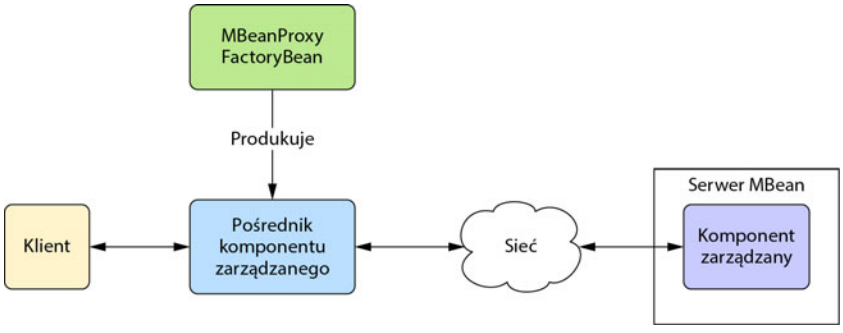
To tylko niektóre z rzeczy możliwych przy zdalnych komponentach MBean, gdy używasz metod dostępnych przez `MBeanServerConnection`. Zachęcam Cię do eksperymentowania.

Ale wywoływanie metod i ustawianie wartości atrybutów zdalnego komponentu zarządzanego jest jednak trochę niewygodne, jeśli wykorzystujemy do tego `MBeanSer➥verConnection`. Tak prosta czynność jak wywołanie metody `setSpittlesPerPage()` wiąże się z utworzeniem instancji `ObjectName` i przekazaniem kilku innych parametrów do metody `invoke()`. Jest to daleko mniej intuicyjne niż tradycyjne wywołanie metody. Bardziej bezpośrednie podejście wymagało będzie pośrednika dla zdalnego komponentu zarządzanego.

20.2.3. Obiekty pośredniczące komponentów zarządzanych

`MBeanProxyFactoryBean` jest komponentem fabryki obiektów pośredniczących, podobnym do tych omawianych w rozdziale 15. Zamiast jednak pośredniczyć w dostępie do zdalnych komponentów Springa, `MBeanProxyFactoryBean` pozwala na bezpośredni dostęp

do komponentów zarządzanych (taki, jak do każdych innych skonfigurowanych lokalnie komponentów). Rysunek 20.4 pokazuje ten mechanizm.



Rysunek 20.4. MBeanProxyFactoryBean tworzy obiekt pośrednika zdalnego komponentu MBean. Klient pośrednika może się wtedy komunikować ze zdalnym komponentem MBean tak, jakby był on lokalnie skonfigurowanym obiektem POJO

Rozważmy na przykład następującą deklarację `MBeanProxyFactoryBean`:

```
@Bean
public MBeanProxyFactoryBean remoteSpittleControllerMBean(
        MBeanServerConnection mbeanServerClient) {
  MBeanProxyFactoryBean proxy = new MBeanProxyFactoryBean();
  proxy.setObjectName("");
  proxy.setServer(mbeanServerClient);
  proxy.setProxyInterface(SpittleControllerManagedOperations.class);
  return proxy;
}
```

Właściwość `objectName` określa nazwę obiektu zdalnego komponentu MBean. W tym przypadku odnosi się ona do komponentu zarządzanego `SpittleController`, który wyeksportowaliśmy wcześniej.

Właściwość `server` odnosi się do serwera `MBeanServerConnection`, przez który przechodzi cała komunikacja z komponentem zarządzanym. W przykładzie dowiązaliśmy skonfigurowany przez nas wcześniej komponent `MBeanServerConnectionFactoryBean`.

Wreszcie, właściwość `proxyInterface` określa interfejs implementowany przez pośrednika. Używamy tutaj interfejsu `SpittleControllerManagedOperations`, który zdefiniowaliśmy w punkcie 20.1.2.

Zadeklarowany komponent `remoteSpittleControllerMBean` możemy teraz dowiązać do każdej właściwości komponentu o typie `SpittleControllerManagedOperations` i wykorzystać go do komunikacji ze zdalnym komponentem MBean. Z tego miejsca będziemy mogli wywołać metody `setSpittlesPerPage()` i `getSpittlesPerPage()`.

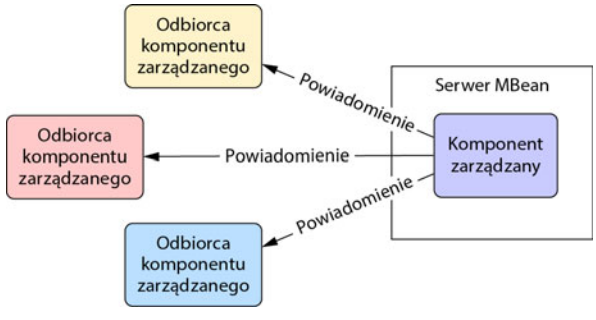
Poznaliśmy już kilka sposobów komunikacji z komponentami zarządzanymi. Wiemy, jak odczytywać i zmieniać konfigurację naszego komponentu Springa w czasie działania aplikacji. Do tej pory była to jednak jednostronna konwersacja. Dużo mówiliśmy do komponentów zarządzanych, nie dając im dojść do słowa. Czas ich wreszcie posłuchać, odbierając wysyłane przez nie powiadomienia.

20.3. Obsługa powiadomień

Uzyskiwanie informacji z komponentu zarządzanego jest tylko jednym ze sposobów kontrolowania stanu aplikacji. Nie jest to jednak najefektywniejszy sposób trzymania ręki na pulsie.

Załóżmy na przykład, że aplikacja Spittr liczy wysłane spittle'e. Powiedzmy, że chcemy być informowani o każdym osiągniętym milionie (spittle nr 1 000 000, 2 000 000, 3 000 000 i tak dalej). Jednym z rozwiązań będzie kod, który okresowo wyśle zapytanie do bazy danych i policzy spittle'e. Ciągłe sprawdzanie liczby spittle'ów byłoby jednak pewnym obciążeniem dla aplikacji i bazy danych.

Dużo lepszym rozwiązaniem od bezustannego wysyłania zapytań do bazy danych w celu uzyskania tej informacji byłoby powiadomienie wysyłane przez komponent zarządzany w momencie osiągnięcia okrągłej liczby. Jak widać na rysunku 20 5, powiadomienia JMX pozwalają komponentom zarządzanym na aktywną komunikację ze światem zewnętrznym, bez potrzeby czekania, aż zewnętrzna aplikacja zapyta je o informacje.



Rysunek 20.5. Powiadomienia JMX pozwalają komponentom zarządzanym na aktywną komunikację ze światem zewnętrznym

Spring obsługuje wysyłanie powiadomień, dostarczając interfejs `NotificationPublis ↪herAware`. Każdy komponent, który stał się komponentem zarządzanym i który chce wysyłać powiadomienia, powinien go implementować. Rozważmy na przykład implementację `SpittleNotifierImpl` z listingu 20.2.

Listing 20.2. Wykorzystanie publikatora NotifcationPublisher do wysyłania powiadomień JMX

```
package com.habuma.spittr.jmx;
import javax.management.Notification;
import org.springframework.jmx.export.annotation.ManagedNotification;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.stereotype.Component;

@Component
@ManagedResource("spitter:name=SpitterNotifier")
@ManagedNotification(
        notificationTypes="SpittleNotifier.OneMillionSpittles",
        name="TODO")
public class SpittleNotifierImpl
    implements NotificationPublisherAware, SpittleNotifier {
```

Implementacja NotificationPublisherAware

```
  private NotificationPublisher notificationPublisher;

  public void setNotificationPublisher(                    <-- Wstrzyknięcie publikatora powiadomień
          NotificationPublisher notificationPublisher) {
    this.notificationPublisher = notificationPublisher;
  }

  public void millionthSpittlePosted() {
    notificationPublisher.sendNotification(                <-- Wysłanie powiadomienia
          new Notification(
                  "SpittleNotifier.OneMillionSpittles", this, 0));
  }

}
```

Jak widać, klasa `SpittleNotifierImpl` implementuje `NotificationPublisherAware`. Nie jest to specjalnie wymagający interfejs. Wymaga on implementacji tylko jednej metody: `setNotificationPublisher`.

`SpittleNotifierImpl` implementuje także jedyną metodę interfejsu `SpittleNotifier` — `millionthSpittlePosted()`. Metoda ta wykorzystuje publikator komunikatów, wstrzykiwany automatycznie przez metodę `setNotificationPublisher()`, aby wysłać powiadomienie o każdym milionowym spittle'u.

Po wywołaniu metody `sendNotification()` powiadomienie jest już w drodze do… no właśnie… wygląda na to, że jeszcze nie zdecydowaliśmy, kto będzie odbiorcą. Ustawimy teraz odbiorcę, który będzie czekał na powiadomienia i na nie reagował.

20.3.1. Odbieranie powiadomień

Standardowym sposobem odbioru powiadomień komponentów zarządzanych jest implementacja interfejsu `javax.management.NotificationListener`. Rozważmy na przykład klasę `PagingNotificationListener`:

```
package com.habuma.spittr.jmx;

import javax.management.Notification;
import javax.management.NotificationListener;

public class PagingNotificationListener
        implements NotificationListener {
  public void handleNotification(
          Notification notification, Object handback) {
    //...
  }
}
```

`PagingNotificationListener` jest typowym odbiorcą powiadomień JMX. Po otrzymaniu powiadomienia wywołana zostanie metoda `handleNotification()`, będąca reakcją na powiadomienie. Przypuszczalnie wyśle ona na pager lub telefon komórkowy powiadomienie o wysłaniu milionowego spittle'a (rzeczywistą implementację pozostawiam czytelnikowi).

Pozostaje tylko rejestracja `PagingNotificationListerner` w eksporterze `MBeanExporter`:

```
@Bean
public MBeanExporter mbeanExporter() {
  MBeanExporter exporter = new MBeanExporter();
  Map<?, NotificationListener> mappings =
          new HashMap<?, NotificationListener>();
  mappings.put("Spitter:name=PagingNotificationListener",
               new PagingNotificationListener());
  exporter.setNotificationListenerMappings(mappings);
  return exporter;
}
```

Właściwość `notificationListenerMappings` eksportera wykorzystywana jest do przypisania odbiorców powiadomień do komponentów, które będą te powiadomienia wysyłać. W tym przypadku postanowiliśmy, że `PagingNotificationListener` będzie odbierać powiadomienia publikowane przez komponent zarządzany `SpittleNotifier`.

20.4. Podsumowanie

JMX jest oknem do wnętrza Twojej aplikacji. W tym rozdziale dowiedzieliśmy się, jak skonfigurować Springa, aby automatycznie eksportować komponenty Springa jako komponenty zarządzane JMX, co umożliwi ich odczyt i przetwarzanie za pomocą odpowiednich narzędzi do zarządzania JMX. Pokazaliśmy też, jak używać zdalnych komponentów zarządzanych w sytuacjach, gdy są od tych narzędzi oddalone. Na koniec omówiliśmy użycie Springa do publikacji i odbierania powiadomień JMX.

Nie da się ukryć, że do końca książki pozostało już niewiele stron. Nasza podróż przez krainę Springa dobiega powoli końca. Zanim jednak zakończymy, musimy się jeszcze raz zatrzymać. W następnym rozdziale poznasz Spring Boot, nowy i fascynujący sposób tworzenia aplikacji Spring przy wykorzystaniu bardzo niewielkiej, lub niemal żadnej, konfiguracji.

21 Upraszczanie tworzenia aplikacji przy użyciu Spring Boot

W tym rozdziale omówimy:

- Dodawanie zależności projektu przy wykorzystaniu starterów Spring Boot
- Automatyczną konfigurację komponentów
- Groovy oraz Spring Boot CLI
- Aktuator Spring Boot

Pamiętam kilka początkowych dni swojego pierwszego kursu analizy matematycznej, w trakcie których poznałem pochodne funkcji. Wykonywaliśmy raczej karkołomne obliczenia z wykorzystaniem granic, by wyliczyć pochodne kilku funkcji. Choć funkcje były całkiem proste, to obliczenia związane z wyliczeniem pochodnych były koszmarne.

Po kilku pracach domowych, konsultacjach oraz egzaminie prawie każdy był jednak w stanie to zrobić. Mimo wszystko uciążliwość tych operacji była niemal nie do zniesienia. Skoro to było pierwszą rzeczą, której uczyliśmy się na początku roku na kursie „Analiza matematyczna I", to jakie matematyczne okropności czekały na nas w połowie semestru na kursie „Analiza matematyczna II"?

Ale później wykładowca zaskoczył nas pewną sztuczką. Wystarczyło zastosować pewien wzór, aby znacznie skrócić obliczanie pochodnych (jeśli kiedykolwiek miałeś

styczność z analizą matematyczną, to będziesz wiedział, co mam na myśli). Dzięki tej nowej sztuczce byliśmy w stanie obliczać pochodne kilkunastu funkcji w takim czasie, jaki wcześniej musieliśmy poświęcić na obliczenie jednej pochodnej.

W tym momencie jeden z kolegów zapytał wykładowcę, dlaczego nie nauczył nas tej sztuczki już pierwszego dnia.

Instruktor odpowiedział, że dzięki temu mogliśmy docenić pochodne za to, czym one są, że to wzmocniło nasz charakter, a potem wspomniał jeszcze coś o wyrywaniu włosów z głowy.

Teraz, kiedy mamy już za sobą całą książkę o Springu, znalazłem się w tej samej sytuacji, w której wtedy znajdował się nasz wykładowca analizy matematycznej. Choć podstawowym zadaniem Springa jest ułatwianie tworzenia aplikacji w Javie, to w tym rozdziale przekonasz się, że Spring Boot potrafi sprawić, iż używanie Springa może być jeszcze łatwiejsze. Nie ulega wątpliwości, że Spring Boot to najbardziej fascynująca rzecz, jaka pojawiła się w Springu od chwili jego stworzenia. Spring Boot udostępnia całkowicie nowy model programowania, który przesłania Springa i w znacznym stopniu eliminuje to wszystko, co w pisaniu aplikacji Springa jest nudne.

Zacznę od przedstawienia sztuczek wykorzystywanych przez Spring Boot w celu uproszczenia tworzenia aplikacji Springa. Zanim ten rozdział dobiegnie końca, zdążysz napisać kompletną (chociaż prostą) aplikację przy użyciu Spring Boot.

21.1. Prezentacja Spring Boot

Spring Boot jest nowym (choć nie wiem, czy śmiem napisać: zmieniającym reguły gry) projektem, wchodzącym w skład rodziny Springa. Udostępnia on cztery podstawowe możliwości, które zmienią nasz sposób tworzenia aplikacji Spring:

- *Startery Spring Boot* — startery te, nazywane także zależnościami początkowymi, łączą często stosowane grupy zależności w pojedyncze zależności, które można dodawać do procesu budowania aplikacji przy użyciu takich narzędzi jak Maven lub Gradle.
- *Automatyczna konfiguracja* — mechanizm automatycznej konfiguracji Spring Boot rozszerza dostępne w Springu 4 wsparcie dla konfiguracji warunkowej, pozwalając na snucie sensownych przypuszczeń odnośnie do komponentów, których aplikacja może potrzebować, i automatyczne ich konfigurowanie.
- *Interfejs wiersza poleceń* (ang. *Command-line interface*, w skrócie CLI) — interfejs CLI Spring Boot korzysta z języka Groovy oraz mechanizmu automatycznej konfiguracji, by w jeszcze większym stopniu uprościć tworzenie aplikacji Spring.
- *Aktuator* — dodaje do aplikacji Spring Boot nowe możliwości zarządzania.

W tym rozdziale napiszesz niewielką aplikację, korzystając przy tym ze wszystkich możliwości udostępnianych przez Spring Boot. Jednak zanim zaczniemy, przyjrzymy się każdej z nich, aby lepiej zrozumieć, jak przyczyniają się one do uproszczenia modelu programowania w Springu.

21.1.1. Dodawanie zależności początkowych

Ciasto można upiec na dwa sposoby. Ambitniejszy cukiernik wymiesza mąkę, jajka, cukier, proszek do pieczenia, sól, masło, wanilię i mleko. Można jednak także kupić gotowe ciasto w proszku i dodać do niego kilka dodatków, takich jak woda, jajka bądź olej roślinny.

Analogicznie do ciasta w proszku, które łączy w sobie większość składników przepisu na ciasto, startery Spring Boot łączą w sobie różne zależności występujące w aplikacji, tworząc z nich jedną zależność.

Żeby to zilustrować, załóżmy, że chcemy od podstaw utworzyć nowy projekt aplikacji Spring. Będzie to aplikacja internetowa, dlatego skorzystamy ze Spring MVC. Aplikacja ma także posiadać REST API i udostępniać swoje zasoby w formie kodu JSON, dlatego będzie musiała korzystać z biblioteki Jackson JSON.

Ponieważ aplikacja będzie używać JDBC do zapisywania i pobierania danych z relacyjnej bazy danych, musimy się również upewnić, że zostanie do niej dołączony moduł JDBC Springa (niezbędny do korzystania z `JdbcTemplate`) oraz moduł transakcji (niezbędny dla deklaratywnego stosowania transakcji). Jeśli natomiast chodzi o samą bazę danych, to baza H2 doskonale spełni nasze oczekiwania.

Ponadto chcemy także używać Thymeleaf do generowania widoków Spring MVC.

Gdyby nasz projekt był budowany przy wykorzystaniu Gradle, to w pliku *build.gradle* musiałyby się znaleźć przynajmniej następujące zależności:

```
dependencies {
  compile("org.springframework:spring-web:4.0.6.RELEASE")
  compile("org.springframework:spring-webmvc:4.0.6.RELEASE")
  compile("com.fasterxml.jackson.core:jackson-databind:2.2.2")
  compile("org.springframework:spring-jdbc:4.0.6.RELEASE")
  compile("org.springframework:spring-tx:4.0.6.RELEASE")
  compile("com.h2database:h2:1.3.174")
  compile("org.thymeleaf:thymeleaf-spring4:2.1.2.RELEASE")
}
```

Na szczęście Gradle pozwala zapisywać zależności w zwartej formie. (By oszczędzić trochę papieru, nie będę tu pokazywał, jak wyglądałaby lista zależności w pliku *pom.xml* używanym przez projekty budowane z zastosowaniem programu Maven). Mimo wszystko utworzenie tej listy wymagało sporo pracy, a jeszcze więcej zachodu będzie wymagało jej utrzymanie. Skąd możemy wiedzieć, czy te zależności będą ze sobą dobrze współdziałać? A kiedy aplikacja będzie się powiększać i zmieniać, zarządzanie tymi zależnościami stanie się jeszcze większym wyzwaniem.

Jednak kiedy skorzystamy z przygotowanych zależności udostępnianych przez startery Spring Boot, lista zależności Gradle może się stać nieco krótsza:

```
dependencies {
  compile("org.springframework.boot:spring-boot-starter-web:
           1.1.4.RELEASE")
  compile("org.springframework.boot:spring-boot-starter-jdbc:
           1.1.4.RELEASE")
  compile("com.h2database:h2:1.3.174")
  compile("org.thymeleaf:thymeleaf-spring4:2.1.2.RELEASE")
}
```

Jak widać w powyższym przykładzie, startery aplikacji internetowej (spring-boot-star
↪ter-web) oraz JDBC (spring-boot-starter-web) zastąpiły kilka bardziej szczegółowych zależności. Choć wciąż musimy podawać zależności dotyczące H2 oraz Thymeleaf, to jednak wszystkie pozostałe weszły w skład zależności starterów. To rozwiązanie daje nam, oprócz skrócenia listy, pewność, że wersje zależności określone w starterach będą ze sobą zgodne.

Startery aplikacji internetowych oraz JDBC są jedynie dwoma spośród wielu starterów udostępnianych przez Spring Boot. Wszystkie startery dostępne w czasie powstawania tego rozdziału zostały przedstawione w tabeli 21.1.

Tabela 21.1. Startery Spring Boot łączą w sobie często występujące zależności, grupując je w formie pojedynczych zależności projektu

Starter	Udostępnia
spring-boot-starter-actuator	spring-boot-starter, spring-bootactuator, spring-core
spring-boot-starter-amqp	spring-boot-starter, spring-bootrabbit, spring-core, spring-tx
spring-boot-starter-aop	spring-boot-starter, spring-aop, AspectJ Runtime, AspectJ Weaver, springcore
spring-boot-starter-batch	spring-boot-starter, HSQLDB, springjdbc, spring-batch-core, spring-core
spring-boot-starter-elasticsearch	spring-boot-starter, spring-dataelasticsearch, spring-core, spring-tx
spring-boot-starter-gemfire	spring-boot-starter, Gemfire, springcore, spring-tx, spring-context, spring-context-support, spring-datagemfire
spring-boot-starter-data-jpa	spring-boot-starter, spring-bootstarter-jdbc, spring-boot-starteraop, spring-core, Hibernate EntityManager, spring-orm, spring-data-jpa, springaspects
spring-boot-starter-data-mongodb	spring-boot-starter, sterownik Java MongoDB, spring-core, spring-tx, spring-datamongodb
spring-boot-starter-data-rest	spring-boot-starter, spring-bootstarter-web, adnotacje Jackson, mechanizm wiązania danych Jackson, spring-core, spring-tx, spring-data-rest-webmvc
spring-boot-starter-data-solr	spring-boot-starter, Solrj, spring-core, spring-tx, spring-data-solr, Apache HTTP Mime
spring-boot-starter-freemarker	spring-boot-starter, spring-bootstarter-web, Freemarker, spring-core, spring-context-support
spring-boot-starter-groovy-templates	spring-boot-starter, spring-bootstarter-web, Groovy, Groovy Templates, spring-core
spring-boot-starter-hornetq	spring-boot-starter, spring-core, spring-jms, Hornet JMS Client

Tabela 21.1. Startery Spring Boot łączą w sobie często występujące zależności, grupując je w formie pojedynczych zależności projektu — *ciąg dalszy*

Starter	Udostępnia
spring-boot-starter-integration	spring-boot-starter, spring-aop, spring-tx, spring-web, spring-webmvc, spring-integration-core, spring-integration-file, spring-integration-http, spring-integration-ip, spring-integration-stream
spring-boot-starter-jdbc	spring-boot-starter, spring-jdbc, tomcat-jdbc, spring-tx
spring-boot-starter-jetty	jetty-webapp, jetty-jsp
spring-boot-starter-log4j	jcl-over-slf4j, jul-to-slf4j, slf4j-log4j12, log4j
spring-boot-starter-logging	jcl-over-slf4j, jul-to-slf4j, log4j-over-slf4j, logback-classic
spring-boot-starter-mobile	spring-boot-starter, spring-boot-starter-web, spring-mobile-device
spring-boot-starter-redis	spring-boot-starter, spring-data-redis, lettuce
spring-boot-starter-remote-shell	spring-boot-starter-actuator, spring-context, org.crashub.**
spring-boot-starter-security	spring-boot-starter, spring-security-config, spring-security-web, spring-aop, spring-beans, spring-context, spring-core, spring-expression, spring-web
spring-boot-starter-social-facebook	spring-boot-starter, spring-boot-starter-web, spring-core, spring-social-config, spring-social-core, spring-social-web, spring-social-facebook
spring-boot-starter-social-twitter	spring-boot-starter, spring-boot-starter-web, spring-core, spring-social-config, spring-social-core, spring-social-web, spring-social-twitter
spring-boot-starter-social-linkedin	spring-boot-starter, spring-boot-starter-web, spring-core, spring-social-config, spring-social-core, spring-social-web, spring-social-linkedin

Tabela 21.1. Startery Spring Boot łączą w sobie często występujące zależności, grupując je w formie pojedynczych zależności projektu — *ciąg dalszy*

Starter	Udostępnia
`spring-boot-starter`	`spring-boot`, `spring-boot-autoconfigure`, `spring-boot-starter-logging`
`spring-boot-starter-test`	`spring-boot-starter-logging`, `spring-boot`, `junit`, `mockito-core`, `hamcrest-library`, `spring-test`
`spring-boot-starter-thymeleaf`	`spring-boot-starter`, `spring-boot-starter-web`, `spring-core`, `thymeleaf-spring4`, `thymeleaf-layout-dialect`
`spring-boot-starter-tomcat`	`tomcat-embed-core`, `tomcat-embed-logging-juli`
`spring-boot-starter-web`	`spring-boot-starter`, `spring-boot-starter-tomcat`, `jackson-databind`, `spring-web`, `spring-webmvc`
`spring-boot-starter-websocket`	`spring-boot-starter-web`, `spring-websocket`, `tomcat-embed-core`, `tomcat-embed-logging-juli`
`spring-boot-starter-ws`	`spring-boot-starter`, `spring-boot-starter-web`, `spring-core`, `spring-jms`, `spring-oxm`, `spring-ws-core`, `spring-ws-support`

Gdybyśmy zajrzeli za kulisy tych starterów, okazałoby się, że w ich działaniu nie ma nic tajemniczego. Korzystając z mechanizmu przechodniego określania zależności stosowanego w programach Maven i Gradle, startery te definiują kilka zależności w swoich własnych plikach *pom.xml*. Kiedy taki starter dodamy do własnego pliku budowy programu Maven lub Gradle, zostaną użyte określane przez niego zależności. A zależności te mogą definiować dalsze, własne zależności. A zatem starter może powodować pobranie dziesiątek zależności.

Warto zwrócić uwagę, że wiele spośród tych starterów odwołuje się do innych starterów Spring Boot. Na przykład starter `spring-boot-starter-mobile` odwołuje się do startera `spring-boot-starter-web`, który z kolei odwołuje się do startera serwera Tomcat. Przeważająca większość starterów odwołuje się także do startera `spring-boot-starter`, który w zasadzie pełni funkcję startera bazowego (choć sam odwołuje się do startera `spring-boot-starter-logging`). Wszystkie te zależności są uwzględniane w sposób przechodni, czyli dodanie startera `spring-boot-starter-mobile` spowoduje uwzględnienie wszystkich kolejnych zależności, do których starter ten się odwołuje.

21.1.2. Automatyczna konfiguracja

Startery Spring Boot ograniczają wielkość listy zależności aplikacji, natomiast mechanizm automatycznej konfiguracji Spring Boot ogranicza wielkość informacji konfiguracyjnych. Jego działanie bazuje na uwzględnianiu innych czynników występujących

w aplikacji i dokonywaniu założeń dotyczących ustawień konfiguracyjnych, których aplikacja będzie potrzebować.

Na przykład przypomnijmy sobie z rozdziału 6. (listing 6.4), że stosowanie szablonów Thymeleaf jako widoków Spring MVC wymaga użycia przynajmniej trzech komponentów: `ThymeleafeViewResolver`, `SpringTemplateEngine` oraz `TemplateResolver`. Jednak w przypadku skorzystania z automatycznej konfiguracji Spring Boot jedyną czynnością, jaką będziemy musieli wykonać, będzie dodanie biblioteki Thymeleaf do ścieżki klas. Kiedy Spring Boot wykryje ją w ścieżce klas, przyjmie, że chcemy jej używać do generowania widoków w Spring MVC, i automatycznie skonfiguruje wszystkie trzy niezbędne komponenty.

Startery Spring Boot mogą powodować uruchomienie mechanizmu automatycznej konfiguracji. Na przykład aby skorzystać ze Spring MVC w aplikacji Spring Boot, wystarczy dodać do pliku budowy odpowiedni starter. Kiedy mechanizm automatycznej konfiguracji wykryje Spring MVC w ścieżce klas, automatycznie skonfiguruje kilka komponentów związanych ze Spring MVC, takich jak producent widoków, komponenty obsługi zasobów oraz konwertery komunikatów. Nam pozostanie jedynie napisanie klas kontrolerów obsługujących żądania.

21.1.3. Spring Boot CLI

Interfejs Spring Boot CLI łączy magię, jaką zapewniają startery oraz mechanizm automatycznej konfiguracji Spring Boot, i doprawia ją odrobiną języka Groovy. Sprowadza on pisanie aplikacji w Springu do postaci, w której możemy uruchomić jeden lub kilka skryptów Groovy z poziomu wiersza poleceń i cieszyć się działającą aplikacją. W trakcie jej działania CLI będzie automatycznie importować typy Spring i rozwiązywać wszelkie zależności.

Jednym z najbardziej interesujących przykładów ilustrujących Spring Boot CLI jest poniższy skrypt Groovy:

```
@RestController
class Hi {
  @RequestMapping("/")
  String hi() {
      "Cześć!"
  }
}
```

Możesz mi wierzyć albo nie, lecz ten kod reprezentuje kompletną (choć prostą) aplikację Spring, którą można wykonać, używając Spring Boot CLI. Włącznie ze wszystkimi białymi znakami kod ma długość 85 znaków. Możesz wkleić tę aplikację do klienta Twittera i rozesłać do swoich znajomych.

Gdyby usunąć z niej wszystkie niepotrzebne znaki odstępu i nowego wiersza, to zmieściłaby się w jednym wierszu kodu o długości 67 znaków:

```
@RestControllerclass Hi{@RequestMapping("/")String hi() {"Cześć!"}}
```

W tej postaci aplikacja jest tak krótka, że w tweecie można by ją zmieścić *dwa razy*. A jest to przecież kompletna i działająca (chociaż może nieco uboga, jeśli chodzi

o możliwości) aplikacja Spring. Jeżeli zainstalowaliśmy Spring Boot CLI, to powyższą aplikację możemy uruchomić przy użyciu następującego polecenia:

```
$ spring run Hi.groovy
```

Choć przedstawienie możliwości Spring Boot CLI w formie aplikacji, której cały kod zmieści się w jednym tweecie, było zabawne, to jednak rozwiązanie to oferuje znacznie więcej możliwości. W podrozdziale 21.3 zobaczysz, jak można napisać trochę bardziej złożoną aplikację przy wykorzystaniu Groovy i Spring Boot CLI.

21.1.4. Aktuator

Aktuator Spring Boot rozszerza projekt Springa o kilka użytecznych możliwości, takich jak:

- punkty końcowe do zarządzania;
- sensowna obsługa błędów oraz domyślne powiązanie punktu końcowego `/error`;
- punkt końcowy `/info`, który może przekazywać informacje dotyczące aplikacji;
- framework zdarzeń inspekcyjnych, dostępnych w przypadku stosowania Spring Security.

Wszystkie te możliwości są użyteczne, lecz punkty końcowe służące do zarządzania są najbardziej interesujące i najłatwiej można z nich skorzystać. W podrozdziale 21.4 przeanalizujemy kilka przykładów pokazujących, jak aktuator Spring Boot zapewnia nam wgląd w tajniki działania naszych aplikacji Spring.

Skoro poznaliśmy już pobieżnie wszystkie cztery podstawowe możliwości Spring Boot, spróbujmy je zastosować w praktyce, pisząc w tym celu niewielką, ale kompletną aplikację.

21.2. Pisanie aplikacji korzystającej ze Spring Boot

W pozostałej części tego rozdziału spróbuję pokazać, jak można stworzyć kompletną i nadającą się do praktycznego użycia aplikację, korzystając przy tym z możliwości, jakie daje Spring Boot. Oczywiście to, jakie cechy powinna posiadać taka aplikacja „nadająca się do praktycznego zastosowania", pozostaje kwestią dyskusyjną, której wyjaśnienie zapewne przekracza ramy tematyczne tego rozdziału. Dlatego zamiast naprawdę tworzyć taką aplikację, ograniczymy się trochę i napiszemy coś nieco mniej praktycznego, lecz dobrze reprezentującego cechy faktycznych aplikacji, które moglibyśmy tworzyć przy użyciu Spring Boot.

Nasza aplikacja będzie prostym programem do zarządzania listą kontaktów. Zapewni ona użytkownikom możliwość podawania informacji o kontakcie (imienia, numeru telefony i adresu e-mail) oraz wyświetlania listy wszystkich wpisanych kontaktów.

Możemy określić, czy aplikacja będzie budowana przy wykorzystaniu programu Maven, czy Gradle. Osobiście wolę Gradle, jednak pokażę także, jak to robić przy użyciu programu Maven, na wypadek gdyby ktoś go preferował. Przedstawiony poniżej listing 21.1 prezentuje początkową postać pliku *build.gradle*. Jak widać, na początku blok zależności jest pusty, ale w trakcie prac nad aplikacją powoli będziemy go wypełniać.

Listing 21.1. Plik budowy Gradle naszej aplikacji do obsługi kontaktów

```
buildscript {
  repositories {
    mavenLocal()
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:
              1.1.4.RELEASE")
  }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

jar {
  baseName = 'contacts'
  version = '0.1.0'
}

repositories {
  mavenCentral()
}

dependencies {
}

task wrapper(type: Wrapper) {
  gradleVersion = '1.8'
}
```

Aplikacja będzie używać Spring Boot

Należy utworzyć plik JAR

Tu będą umieszczane zależności

Warto zauważyć, że plik zawiera zależność `buildscript`, związaną z wtyczką Spring Boot Gradle. Jak się przekonasz w dalszej części rozdziału, w ten sposób łatwiej będzie wygenerować wykonywalny plik JAR zawierający wszystkie zależności aplikacji.

Jeśli ktoś woli używać programu Maven, to poniżej, na listingu 21.2, zamieściłem pełny kod pliku *pom.xml*.

Listing 21.2. Plik budowy Maven naszej aplikacji do obsługi listy kontaktów

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.habuma</groupId>
  <artifactId>contacts</artifactId>
  <version>0.1.0</version>
  <packaging>jar</packaging>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.1.4.RELEASE</version>
  </parent>
```

Należy utworzyć plik JAR

Należy dziedziczyć po starterze Spring Boot

```
  <dependencies>                                    Tu będziemy dodawać zależności

  </dependencies>

  <build>
    <plugins>
      <plugin>                                      Należy zastosować wtyczkę Spring Boot
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

Podobnie jak przedstawiony wcześniej plik Gradle, także i ten plik *pom.xml* korzysta z wtyczki Spring Boot programu Maven. Stanowi ona odpowiednik wtyczki Spring Boot używanej przez program Gradle i pozwala na wygenerowanie wykonywalnego pliku JAR.

Warto również zwrócić uwagę, że w odróżnieniu od pliku Gradle, w tym pliku *pom.xml* został określony projekt nadrzędny. Tworząc projekt Maven w oparciu o projekt nadrzędny, którym jest starter Spring Boot `spring-boot-starter-parent`, zyskamy możliwość korzystania z zarządzania zależnościami przez program Maven i nie będziemy musieli jawnie deklarować numerów wersji w niektórych zależnościach. Numery te zostaną odziedziczone po projekcie nadrzędnym.

Zgodnie ze standardową strukturą projektów Maven i Gradle po zakończeniu prac nad naszą aplikacją będzie ona miała następującą strukturę:

```
$ tree
.
├── build.gradle
├── pom.xml
└── src
    └── main
        ├── java
        │   └── contacts
        │       ├── Application.java
        │       ├── Contact.java
        │       ├── ContactController.java
        │       └── ContactRepository.java
        └── resources
            ├── schema.sql
            ├── static
            │   └── style.css
            └── templates
                └── home.html
```

Na razie nie przejmuj się tymi wszystkimi brakującymi plikami. Napiszemy je w kilku następnych punktach rozdziału, podczas prac nad naszą aplikacją. Zaczniemy już teraz — od napisania jej warstwy internetowej.

21.2.1. Obsługa żądań

Ponieważ warstwę internetową aplikacji mamy zamiar zaimplementować przy użyciu Spring MVC, będziemy musieli dodać odpowiednią zależność do pliku budowy. Zgodnie z podanymi wcześniej informacjami starter Spring Boot o nazwie `spring-boot-star ↪ter-web` jest doskonałym sposobem na szybkie i proste dodanie do pliku budowy wszystkiego, co jest niezbędne do tworzenia aplikacji korzystającej ze Spring MVC. Poniżej przedstawiłem stosowną zależność, którą trzeba będzie dodać do pliku budowy programu Gradle:

```
compile("org.springframework.boot:spring-boot-starter-web")
```

Osoby stosujące Maven będą natomiast musiały skorzystać z następujących zależności:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Warto zauważyć, że projekt nadrzędny Spring Boot określa numer wersji zależności `spring-boot-starter-web`, dlatego nie trzeba tego numeru podawać jawnie w plikach zależności *build.gradle* lub *pom.xml*.

Po użyciu tego startera wszystkie zależności niezbędne do korzystania ze Spring MVC zostaną już dodane do projektu. Teraz jesteśmy gotowi, by skoncentrować się na pisaniu klasy kontrolera dla naszej aplikacji.

Kontroler będzie stosunkowo prosty: jego działanie sprowadzi się do wyświetlania listy kontaktów w przypadku odebrania żądania HTTP `GET` oraz do zapisania kontaktu przesłanego w żądaniu `POST`. Sam kontroler nie będzie wykonywał żadnych faktycznych operacji na danych — użyje do tego komponentu `ContactRepository` (nim zajmiemy się już niebawem), który będzie odpowiadał za trwałe przechowywanie danych o kontaktach. Klasa `ContactControllers` przedstawiona na listingu 21.3 spełnia te wymagania.

Listing 21.3. Klasa ContactController obsługuje żądania w naszej aplikacji internetowej

```
package contacts;

import java.util.List;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/")
public class ContactController {

private ContactRepository contactRepo;

  @Autowired
  public ContactController(ContactRepository contactRepo) {
    this.contactRepo = contactRepo;
  }
```

Wstrzykuje komponent ContactRepository

```
  @RequestMapping(method=RequestMethod.GET)
  public String home(Map<String,Object> model) {
    List<Contact> contacts = contactRepo.findAll();
    model.put("contacts", contacts);
    return "home";
  }

  @RequestMapping(method=RequestMethod.POST)
  public String submit(Contact contact) {
    contactRepo.save(contact);
    return "redirect:/";
  }
}
```

Obsługuje żądania GET

Obsługuje żądania POST

Pierwszą rzeczą, na którą należy zwrócić uwagę, jest to, że klasa `ContactController` jest typowym kontrolerem Spring MVC. Choć Spring Boot pomaga nam w kwestii zarządzania zależnościami lub minimalizacji kodu konfiguracyjnego, to jednak jeśli chodzi o logikę aplikacji, stosowany model programowania jest taki sam.

W naszej aplikacji kontroler `ContactController` działa zgodnie ze standardowym modelem kontrolera Spring MVC, który wyświetla dane i akceptuje dane przesyłane z formularza. Metoda `home()` używa wstrzykniętego komponentu `ContactRepository` do pobrania listy obiektów `Contact`, po czym umieszcza ją w modelu i przekazuje dalszą obsługę do widoku `home`. Ten widok wyświetli listę kontaktów oraz formularz pozwalający na dodanie nowego. Z kolei metoda `submit()` będzie obsługiwać żądania `POST`, zawierające dane przesłane z formularza, a konkretnie będzie zapisywać nowy obiekt `Contact` i generować przekierowanie do strony głównej.

Ponieważ do kontrolera `ContactController` została dodana adnotacja `@Controller`, będzie on wykrywany podczas skanowania komponentów. Oznacza to także, że nie będziemy musieli jawnie konfigurować go w kontekście aplikacji Springa.

Jeśli chodzi o klasę modelu, `Contact`, jest to zwyczajny obiekt POJO zawierający kilka właściwości i metod dostępowych. Klasa ta została przedstawiona na listingu 21.4.

Listing 21.4. Contact jest prostym typem domeny

```
package contacts;

public class Contact {
  private Long id;
  private String firstName;
  private String lastName;
  private String phoneNumber;
  private String emailAddress;

  public void setId(Long id) {
    this.id = id;
  }

  public Long getId() {
    return id;
  }
```

Właściwości

Metody dostępowe

```
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getFirstName() {
    return firstName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setPhoneNumber(String phoneNumber) {
    this.phoneNumber = phoneNumber;
  }

  public String getPhoneNumber() {
    return phoneNumber;
  }

  public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
  }

  public String getEmailAddress() {
    return emailAddress;
  }
}
```

Warstwa internetowa naszej aplikacji jest już niemal gotowa. Pozostało nam jedynie utworzenie szablonu Thymeleaf definiującego widok `home`.

21.2.2. Tworzenie widoku

Przyjęło się, że internetowe aplikacje pisane w Javie tworzą warstwę widoku przy wykorzystaniu technologii JSP. Jednak jak już wspomniałem w rozdziale 6., na scenie pojawił się nowy gracz. Stosowanie naturalnych szablonów Thymeleaf jest znacznie przyjemniejsze niż tworzenie szablonów JSP i pozwala na tworzenie ich jako kodu HTML. Właśnie z tego powodu użyjemy Thymeleaf do zdefiniowania widoku `home` naszej aplikacji do obsługi szablonów.

W pierwszej kolejności musimy dodać Thymeleaf do pliku budowy aplikacji. W tym przykładzie korzystam ze Springa 4, dlatego do pliku budowy dodam moduł Thymeleaf Spring 4. Jeżeli używamy programu Gradle, to taka zależność przyjmie następującą postać:

```
compile("org.thymeleaf:thymeleaf-spring4")
```

W przypadku stosowania programu Maven ta sama zależność wygląda tak:

```
<dependency>
  <groupId>org.thymeleaf</groupId>
  <artifactId>thymeleaf-spring4</artifactId>
</dependency>
```

Trzeba pamiętać, że samo dodanie biblioteki Thymeleaf do ścieżki klas projektu spowoduje uruchomienie mechanizmu automatycznej konfiguracji Spring Boot. Po uruchomieniu aplikacji Spring Boot wykryje bibliotekę Thymeleaf w ścieżce klas i automatycznie skonfiguruje producenta widoków oraz komponenty obsługi szablonów, niezbędne do korzystania z widoków Thymeleaf w Spring MVC. Właśnie dlatego używanie Thymeleaf w naszej aplikacji nie wymaga stosowania żadnej jawnej konfiguracji.

Oprócz dodania zależności dla Thymeleaf do pliku budowy jedyną rzeczą, jaką musimy zrobić, jest zdefiniowanie szablonu widoku. Listing 21.5 przedstawia plik home.html — szablon Thymeleaf definiujący widok `home`.

Listing 21.5. Widok home wyświetla formularz do dodawania nowego kontaktu oraz wyświetla listę już zapisanych kontaktów

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Spring Boot - Lista kontaktów</title>
    <link rel="stylesheet" th:href="@{/style.css}" /> <-- Wczytuje arkusz stylów
  </head>
  <body>
    <h2>Spring Boot - Kontakty</h2>

    <form method="POST"> <-- Formularz nowego kontaktu
      <label for="firstName">Imię:</label>
        <input type="text" name="firstName"></input><br/>
      <label for="lastName">Nazwisko:</label>
        <input type="text" name="lastName"></input><br/>
      <label for="phoneNumber">Numer telefonu:</label>
        <input type="text" name="phoneNumber"></input><br/>
      <label for="emailAddress">E-mail:</label>
        <input type="text" name="emailAddress"></input><br/>
      <input type="submit"></input>
    </form>

    <ul th:each="contact : ${contacts}"> <-- Wyświetla listę kontaktów
      <li>
        <span th:text="${contact.firstName}">Imię</span>
        <span th:text="${contact.lastName}">nazwisko</span> :
        <span th:text="${contact.phoneNumber}">telefon</span>,
        <span th:text="${contact.emailAddress}">e-mail</span>
      </li>
    </ul>
  </body>
</html>
```

To stosunkowo prosty szablon Thymeleaf. Składa się on z dwóch części: formularza oraz listy kontaktów. Formularz przesyła dane metodą `POST` na serwer, gdzie są one obsługiwane przez metodę `submit()` kontrolera `ContactController`, która tworzy nowy obiekt `Contact`. Lista z kolei wyświetla wszystkie obiekty `Contacts` dostępne w modelu.

Aby móc używać tego szablonu, musimy bardzo uważnie określić jego nazwę i umieścić go w odpowiednim miejscu projektu. Ponieważ logiczną nazwą widoku zwracaną przez metodę `home()` kontrolera `ContactController` jest `home`, szablon ten powinien być zapisany w pliku *home.html*. A ponieważ automatycznie skonfigurowany producent szablonów będzie szukał szablonów Thymeleaf w katalogu *templates* umieszczonym wewnątrz katalogu głównego ścieżki klas, będziemy musieli umieścić plik *home.html* w projekcie Maven lub Gradle, w katalogu *src/main/resources/templates*.

Jest jeszcze tylko jedna rzecz związana z szablonem Thymeleaf, o którą musimy zadbać. Otóż generowany przez niego kod HTML odwołuje się do arkusza stylów CSS o nazwie *style.css*. Musimy go zatem dodać do naszego projektu.

21.2.3. Dodawanie statycznych artefaktów

Zazwyczaj arkusze i obrazy są zasobami, o których staram się nie pisać w kontekście tworzenia aplikacji Spring. Oczywiście zasoby tego typu są nieodzownymi elementami pozwalającymi nadać atrakcyjniejszy wygląd tworzonym aplikacjom internetowym (w tym także aplikacjom Spring). Niemniej jednak nie mają one krytycznego znaczenia podczas opisywania sposobów pisania kodu aplikacji Spring.

Ale w kontekście Spring Boot warto wspomnieć o tym, jak obsługuje on zasoby statyczne. Kiedy mechanizm automatycznej konfiguracji Spring Boot automatycznie konfiguruje komponenty używane przez Spring MVC, komponenty te będą zawierać obiekt obsługi zasobów odwzorowujący /** na kilka katalogów z zasobami. Katalogi te przedstawiłem na poniższej liście (są one wyznaczane względem katalogu głównego ścieżki klas):

- */META-INF/resources*,
- */resources*,
- */static*,
- */public*.

W standardowej aplikacji tworzonej przy użyciu programów Maven lub Gradle takie statyczne zasoby byłyby zazwyczaj umieszczane w katalogu */src/main/webapp*, tak by mogły zostać umieszczone w katalogu głównym generowanego pliku WAR. W razie generowania pliku WAR przy wykorzystaniu Spring Boot takie rozwiązanie też jest możliwe. Jednak oprócz niego istnieje także możliwość umieszczania treści statycznych w jednym z czterech katalogów obsługiwanych przez obiekt obsługi zasobów.

A zatem aby odwołanie szablonu Thymeleaf do pliku */style.css* mogło zostać prawidłowo obsłużone, musimy utworzyć plik o nazwie *style.css* w jednym z poniższych katalogów:

- */META-INF/resources/style.css*,
- */resources/style.css*,
- */static/style.css*,
- */public/style.css*.

Wybór jednej z tych lokalizacji zależy wyłącznie od nas. Ja preferuję umieszczanie zasobów statycznych w katalogu */public*, jednak każdy z nich jest tak samo dobry.

Choć zawartość pliku *style.css* nie ma żadnego znaczenia dla naszych rozważań, to i tak przedstawiłem ją poniżej. Jest to prosty arkusz stylów, który sprawi, że nasza aplikacja będzie wyglądać nieco lepiej:

```
body {
  background-color: #eeeeee;
  font-family: sans-serif;
}

label {
  display: inline-block;
  width: 120px;
  text-align: right;
}
```

Możesz mi wierzyć lub nie, lecz to już więcej niż połowa prac nad aplikacją! Zakończyliśmy tworzenie warstwy internetowej, która już jest gotowa. Pozostało nam napisanie klasy `ContactRepository`, która będzie obsługiwać trwałe zapisywanie obiektów `Contact`.

21.2.4. Trwałe zapisywanie danych

W Springu możemy używać baz danych na wiele różnych sposobów. Możemy w tym celu skorzystać z JPA lub Hibernate, by odwzorowywać obiekty na tabele i kolumny relacyjnych baz danych. Jednak możemy także całkowicie porzucić model relacyjnych baz danych i zastosować bazy danych innego typu, takie jak Mongo lub Noe4j.

W przypadku naszej aplikacji do zarządzania kontaktami użycie relacyjnej bazy danych będzie dobrym rozwiązaniem. Aby ułatwić sobie zadanie, zastosujemy bazę H2 oraz JDBC (korzystając przy tym z szablonu `JdbcTemplate` Springa).

Rozwiązanie to zmusza nas do dodania kolejnych zależności do pliku budowy aplikacji. Starter JDBC (`spring-boot-starter-jdbc`) doda do projektu wszystko, co jest niezbędne do używania szablonu `JdbcTemplate`. Jednak by korzystać z bazy H2, będziemy musieli dodać do projektu odpowiednią zależność. W przypadku stosowania programu Gradle wszystko sprowadza się do dodania do bloku `dependencies` dwóch poniższych wierszy kodu:

```
compile("org.springframework.boot:spring-boot-starter-jdbc")
compile("com.h2database:h2")
```

Osoby korzystające z programu Maven będą musiały dodać do pliku budowy dwa poniższe elementy `<dependency>`:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

Po dodaniu tych dwóch zależności możemy już napisać klasę repozytorium. Klasa `ContactRepository` przedstawiona na listingu 21.6 odczytuje i zapisuje obiekty `Contact` w bazie danych, używając wstrzykniętego szablonu `JdbcTemplate`.

Listing 21.6. Klasa ContactRepository zapisuje i pobiera obiekty Contact z bazy danych

```
package contacts;

import java.util.List;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

@Repository
public class ContactRepository {
  private JdbcTemplate jdbc;

  @Autowired
  public ContactRepository(JdbcTemplate jdbc) {          <-- Wstrzykuje JdbcTemplate
    this.jdbc = jdbc;
  }

  public List<Contact> findAll() {
    return jdbc.query(                                   <-- Pobiera listę kontaktów
      "select id, firstName, lastName, phoneNumber, emailAddress " +
      "from contacts order by lastName",
      new RowMapper<Contact>() {
        public Contact mapRow(ResultSet rs, int rowNum)
            throws SQLException {
          Contact contact = new Contact();
          contact.setId(rs.getLong(1));
          contact.setFirstName(rs.getString(2));
          contact.setLastName(rs.getString(3));
          contact.setPhoneNumber(rs.getString(4));
          contact.setEmailAddress(rs.getString(5));
          return contact;
        }
    });
  }

  public void save(Contact contact) {
    jdbc.update(                                         <-- Zapisuje kontakt w bazie
      "insert into contacts " +
      "(firstName, lastName, phoneNumber, emailAddress) " +
      "values (?, ?, ?, ?)",
    contact.getFirstName(), contact.getLastName(),
    contact.getPhoneNumber(), contact.getEmailAddress());
  }
}
```

Ta klasa, podobnie jak przedstawiony wcześniej kontroler `ContactController`, jest naprawdę prosta. Niczym się nie różni od klas, które mogłyby być używane w tradycyjnych aplikacjach Spring. W jej kodzie nie ma niczego, co mogłoby sugerować, że

wchodzi ona w skład aplikacji korzystającej ze Spring Boot. Metoda `findAll()` używa wstrzykniętego szablonu `JdbcTemplate`, aby pobierać obiekty `Contact` z bazy danych. Z kolei metoda `save()` korzysta z tego samego szablonu do zapisania w bazie nowego obiektu `Contact`. A ponieważ do klasy dodaliśmy adnotację `@Repository`, zostanie ona automatycznie wykryta podczas skanowania komponentów, a w kontekście aplikacji zostanie utworzony komponent tego typu.

A co z `JdbcTemplate`? Czy nie musimy zadeklarować komponentu `JdbcTemplate` w kontekście aplikacji? I skoro już o tym mowa, to czy nie musimy zadeklarować komponentu H2 `DataSource`?

Otóż krótka odpowiedź na oba pytania brzmi: nie. Kiedy Spring Boot wykryje moduł JDBC oraz bazę H2 w ścieżce klas, zostanie uruchomiony mechanizm automatycznej konfiguracji, który automatycznie skonfiguruje zarówno komponent `JdbcTemplate`, jak i `DataSource`. Zatem także tym razem Spring Boot zadbał za nas o przeprowadzenie niezbędnej konfiguracji.

A co ze schematem bazy danych? Nie ulega wątpliwości, że musimy stworzyć odpowiedni schemat dla tabeli `contacts`, prawda?

Oczywiście. Spring Boot nie jest w stanie odgadnąć, jak powinna wyglądać tabela `contacts`. Będziemy więc musieli stworzyć jej schemat, taki jak ten przedstawiony poniżej:

```
create table contacts (
  id identity,
  firstName varchar(30) not null,
  lastName varchar(50) not null,
  phoneNumber varchar(13),
  emailAddress varchar(30)
);
```

Teraz musimy jeszcze w jakiś sposób wczytać to polecenie SQL i wykonać je w bazie danych H2. Na szczęście Spring Boot może się tym zająć. Jeśli ten kod SQL zapiszemy w pliku *schema.sql* i umieścimy w katalogu głównym ścieżki klas (czyli w katalogu *src/main/resources* projektu Maven lub Gradle), to zostanie on automatycznie odszukany i wykonany podczas uruchamiania aplikacji.

21.2.5. Próba aplikacji

Nasza aplikacja do zarządzania listą kontaktów jest raczej prosta, niemniej jednak można ją uznać za realistyczną aplikację Spring. Dysponuje ona warstwą internetową zdefiniowaną przy użyciu kontrolera Spring MVC oraz szablonu Thymeleaf, jak również warstwą trwałości danych, zdefiniowaną przez repozytorium i szablon `JdbcTemplate` Springa.

Na obecnym etapie prac napisaliśmy już wszystkie niezbędne elementy naszej aplikacji. Ale jest coś, czego nie napisaliśmy — jakakolwiek forma konfiguracji. Nie napisaliśmy nawet wiersza kodu konfiguracyjnego, nie skonfigurowaliśmy ani serwletu `DispatcherServlet` w pliku *web.xml*, ani jego klasy inicjującej.

Czy uwierzyłbyś, gdybym oznajmił, że nie musisz pisać żadnego kodu konfiguracyjnego?

To przecież nie może być prawda. W końcu, zgodnie z wypowiedziami krytyków Springa, chodzi w nim właśnie o konfigurację. Na pewno jest jakiś plik XML bądź plik konfiguracyjny Javy, który przeoczyliśmy. Przecież nie można napisać aplikacji Spring bez żadnej konfiguracji — a może jednak można?

Ogólnie rzecz biorąc, mechanizm automatycznej konfiguracji Spring Boot eliminuje przeważającą większość kodu konfiguracyjnego, o ile nie cały. Dzięki temu można napisać kompletną aplikację Spring bez choćby wiersza kodu konfiguracyjnego. Oczywiście mechanizm ten nie może obsłużyć wszystkich potencjalnych scenariuszy, dlatego typowe aplikacje Spring Boot będą wymagać napisania jakiegoś kodu konfiguracyjnego.

W przypadku naszej aplikacji do zarządzania kontaktami pisanie kodu konfiguracyjnego nie będzie jednak konieczne. O wszelkie szczegóły zadbał za nas mechanizm automatycznej konfiguracji Springa.

Niemniej będziemy potrzebować specjalnej klasy, która wczyta i uruchomi naszą aplikację. Sam Spring nic nie wie o mechanizmie automatycznej konfiguracji. Listing 21.7 przedstawia typową klasę służącą do uruchamiania aplikacji Spring Boot.

Listing 21.7. Prosta klasa uruchomieniowa inicjująca mechanizm automatycznej konfiguracji Spring Boot

```
package contacts;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan;

@ComponentScan
@EnableAutoConfiguration <-------------------- Włącza automatyczną konfigurację
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args); <-------------- Uruchamia aplikację
  }
}
```

No dobrze, muszę przyznać, że klasa `Application` zawiera fragment kodu konfiguracyjnego. Dodaliśmy do niej adnotację `@ComponentScan`, która uruchamia skanowanie komponentów. Poza tym dodaliśmy do niej adnotację `@EnableAutoConfiguration`, która uruchamia mechanizm automatycznej konfiguracji. Ale to wszystko! Cały kod konfiguracyjny naszej aplikacji sprowadza się do tych dwóch wierszy.

Tym, co wyróżnia klasę `Application`, jest obecność metody `main()`. Jak się niebawem przekonamy, aplikacje Spring Boot mogą być uruchamiane w unikalny sposób, a metoda `main()` umożliwia jego stosowanie. Wewnątrz metody `main()` umieszczony został jeden wiersz kodu, który informuje Spring Boot (za pośrednictwem klasy `SpringApplication`), by uruchomić aplikację przy użyciu konfiguracji dostępnej w samej klasie `Application` oraz argumentów podanych w wierszu poleceń.

Teraz jesteśmy już niemal gotowi do uruchomienia aplikacji. Pozostaje ją jedynie zbudować. W przypadku stosowania programu Gradle poniższe polecenie spowoduje zbudowanie projektu i zapisanie go w pliku *build/libss/contacts-0.1.0.jar*:

```
$ gradle build
```

Jeśli ktoś jest fanem programu Maven, może zrobić to samo za pomocą następującego polecenia:

```
$ mvn package
```

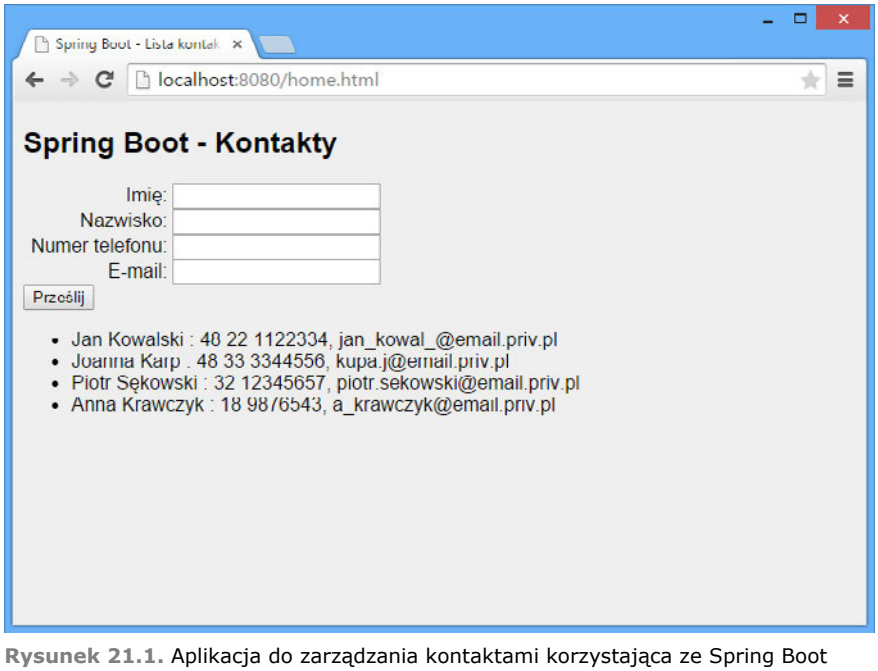
Po wykonaniu tego polecenia wygenerowany artefakt będzie można znaleźć w katalogu *target*.

Teraz jesteśmy już gotowi do uruchomienia aplikacji. Tradycyjnie oznaczałoby to konieczność umieszczenia jej pliku WAR w kontenerze serwletów, takim jak Tomcat lub WebSphere. Ale my nawet nie dysponujemy plikiem WAR — proces budowy wygenerował tylko plik JAR.

To jednak żaden problem. Naszą aplikację możemy wykonać z poziomu wiersza poleceń — wystarczy wydać następujące polecenie, odwołując się w nim do pliku JAR wygenerowanego w procesie budowy:

```
$ java -jar build/libs/contacts-0.1.0.jar
```

Po kilku sekundach aplikacja powinna zostać uruchomiona i być gotowa do działania. Wystarczy wyświetlić w przeglądarce stronę *http://localhost:8080* i jesteśmy gotowi do wpisywania naszych kontaktów. Po wpisaniu kilku z nich strona wyświetlona w przeglądarce może przypominać tę pokazaną na rysunku 21.1.



Rysunek 21.1. Aplikacja do zarządzania kontaktami korzystająca ze Spring Boot

Pewnie myślisz, że właśnie w taki sposób powinno się uruchamiać aplikacje internetowe. Możliwość uruchamiania aplikacji internetowych z poziomu wiersza poleceń jest przydatna i wygodna, jednak tak się tego nie robi. Normalnie w firmie lub w środowisku

produkcyjnym aplikacje internetowe są wdrażane w formie plików WAR w kontenerach serwletów. Nasza firmowa policja wdrożeniowa nie byłaby zachwycona, gdybyśmy przekazali jej cokolwiek innego niż plik WAR.

No cóż, niech i tak będzie.

Choć uruchamianie aplikacji z poziomu wiersza poleceń jest akceptowanym rozwiązaniem, i to nawet w przypadku aplikacji produkcyjnych, to zdaję sobie sprawę z tego, że zapewne musisz dostosować się do firmowych wytycznych określających stosowaną procedurę wdrażania. A to zazwyczaj oznacza tworzenie i wdrażanie plików WAR.

Na szczęście wymóg posługiwania się plikiem WAR nie oznacza, że będziemy musieli zrezygnować z prostoty, jaką zapewnia nam Spring Boot. W takim przypadku wystarczy nam niewielka zmiana w pliku budowy aplikacji. Jeśli używamy programu Gradle, będziemy musieli zastosować wtyczkę „war", dodając do pliku budowy poniższy wiersz:

```
apply plugin: 'war'
```

Ponadto konfigurację „jar" będziemy musieli zmienić na konfigurację „war". Sprowadza się to właściwie do zmiany litery „j" na „w":

```
war {
  baseName = 'contacts'
  version = '0.1.0'
}
```

W przypadku korzystania z programu Maven niezbędne zmiany są jeszcze prostsze — wystarczy zmienić sposób pakowania generowanej aplikacji z „jar" na „war":

```
<packaging>war</packaging>
```

Teraz pozostało już jedynie ponownie zbudować projekt i odszukać w katalogu *build* plik *contacts-0.1.0.war*. Ten plik można wdrożyć na dowolnym kontenerze sieciowym zgodnym ze specyfikacją Servlet 3.0. Co więcej, wciąż będziemy mogli wykonywać naszą aplikację z poziomu wiersza poleceń:

```
$ java -jar -build/libs/contacts-0.1.0.war
```

Dokładnie — to jest wykonywalny plik WAR! Czyli połączenie najlepszych cech obu rozwiązań!

Jak widać, Sprint Boot posunął się naprawdę daleko, by w jak największym stopniu ułatwić nam tworzenie aplikacji w Javie. Startery Spring Boot upraszczają określanie zależności w plikach budowy, a mechanizm automatycznej konfiguracji eliminuje konieczność stosowania większości kodu konfiguracyjnego. Jak się jednak niebawem przekonamy, dodanie do tej mieszaniny języka Groovy pozwala uprościć wszystko jeszcze bardziej.

21.3. Stosowanie Groovy i Spring Boot CLI

Groovy jest znacznie łatwiejszym językiem programowania niż Java. Jego składnia pozwala na takie uproszczenia jak pomijanie średników oraz słowa kluczowego `public`. Co więcej, właściwości klas w Groovy nie wymagają implementacji metod ustawiających

i odczytujących, jak to jest w Javie. A jeszcze nawet nie wspomniałem o wielu innych cechach tego języka, które sprawiają, że pisanie w nim programów wymaga dużo mniej zachodu niż w Javie.

Jeśli zechcemy napisać naszą aplikację w Groovy i uruchamiać ją przy użyciu Spring Boot CLI, to Spring Boot będzie w stanie wykorzystać prostotę języka Groovy, by w jeszcze większym stopniu uprościć tworzenie aplikacji Spring. Aby to pokazać, spróbujmy przepisać naszą aplikację do zarządzania listą kontaktów w języku Groovy.

Czemu by nie? Oryginalną aplikację tworzyło tylko kilka klas Javy, dlatego przygotowując jej odpowiednik w Groovy, nie będziemy musieli przepisywać wiele kodu. Możemy także wykorzystać dokładnie ten sam szablon Thymeleaf oraz plik *schema.sql*. A jeżeli moje twierdzenia dotyczące możliwości uproszczenia kodu aplikacji Spring w przypadku pisania jej w Groovy są prawdziwe, to przepisanie aplikacji nie będzie wielkim problemem.

Podczas tworzenia tej nowej wersji aplikacji będziemy się mogli pozbyć kilku plików. Spring Boot CLI jest w stanie sam zainicjować swoje działanie, dlatego nie będziemy potrzebowali klasy `Application`. Możemy się także pozbyć plików budowy Gradle i Maven, gdyż będziemy uruchamiali nieskompilowane pliki Groovy za pomocą CLI. A dzięki odrzuceniu programów Gradle i Maven całą strukturę projektu można znacząco uprościć. W tym przypadku będzie ona wyglądać następująco:

```
$ tree
.
├── Contact.groovy
├── ContactController.groovy
├── ContactRepository.groovy
├── schema.sql
├── static
│   └── style.css
└── templates
    └── home.html
```

Choć pliki *schema.sql*, *style.css* oraz *home.html* pozostają niezmienione, to będziemy musieli skonwertować trzy klasy Javy do kodu w języku Groovy. Zaczniemy od przerobienia warstwy internetowej.

21.3.1. Pisanie kontrolera w języku Groovy

Jak już wspominałem, pisanie w języku Groovy wymaga znacznie mniej zachodu niż pisanie w Javie. Oznacza to, że kod w Groovy może się obejść bez:

- średników;
- modyfikatorów, takich jak `public` i `private`;
- metod odczytujących i ustawiających we właściwościach;
- instrukcji `return` zwracającej wartość wynikową z metod.

Korzystając z uproszczonej składni języka Groovy (jak również z odrobiny magii dostarczanej przez Spring Boot), możemy przepisać w języku Groovy kod klasy `ContactCon`
`↪troller` w sposób przedstawiony na listingu 21.8.

Listing 21.8. ContactController jest prostszy w Groovy niż w Javie

```
@Grab("thymeleaf-spring4") <-- Pobiera zależność do Thymeleaf

@Controller
@RequestMapping("/")
class ContactController {

  @Autowired
  ContactRepository contactRepo <-- Wstrzykuje ContactRepository

  @RequestMapping(method=RequestMethod.GET) <-- Obsługuje żądania GET
  String home(Map<String,Object> model) {
    List<Contact> contacts = contactRepo.findAll()
    model.putAll([contacts: contacts])
    "home"
  }

  @RequestMapping(method=RequestMethod.POST) <-- Obsługuje żądania POST
  String submit(Contact contact) {
    contactRepo.save(contact)
    "redirect:/"
  }
}
```

Jak widać, ta wersja kontrolera `ContactController` jest znacznie prostsza od jej odpowiednika napisanego w Javie. Dzięki pozbyciu się wszystkich rzeczy, których Groovy nie potrzebuje, kod kontrolera stał się krótszy i prawdopodobnie jest łatwiejszy do zrozumienia.

Poza tym w kodzie z listingu 21.8 czegoś brakuje. Być może zwróciłeś uwagę, że nie ma w nim żadnych instrukcji importu, tak typowych dla klas Javy. Groovy domyślnie importuje kilka pakietów i klas, a należą do nich między innymi:

- `java.io.*`,
- `java.lang.*`,
- `java.math.BigDecimal`,
- `java.math.BigInteger`,
- `java.net.*`,
- `java.util.*`,
- `groovy.lang.*`,
- `groovy.util.*`.

Dzięki tym domyślnym instrukcjom importu nie musimy importować klasy `List` używanej w kontrolerze `ContactController`. Klasa ta należy do pakietu `java.util`, jest więc importowana domyślnie.

A co z typami Spring, takimi jak `@Controller`, `@RequestMapping`, `@Autowired` czy `@RequestMethod`? Nie należą one do żadnych pakietów importowanych domyślnie, dlaczego zatem nie musimy ich importować?

Później, kiedy uruchomimy aplikację, Spring Boot CLI spróbuje skompilować te klasy Groovy przy użyciu kompilatora Groovy. A ponieważ te typy nie zostały zaimportowane, proces kompilacji zakończy się niepowodzeniem.

Jednak Spring Boot CLI nie poddaje się tak łatwo. Właśnie w takich przypadkach CLI przenosi mechanizm automatycznej konfiguracji na zupełnie nowy poziom. Otóż CLI zorientuje się, że problemy z kompilacją wynikały z braku typów Spring, i przedsięweźmie kroki mające na celu rozwiązanie tego problemu. W pierwszej kolejności zastosuje starter `spring-boot-starter-web` oraz wszystkie jego zależności i doda je do ścieżki klas. (Dokładnie — wszystkie niezbędne pliki JAR zostaną pobrane i dodane do ścieżki klas). Następnie wszystkie niezbędne pakiety zostaną dodane do listy domyślnie importowanych pakietów przechowywanej przez kompilator Groovy. Na końcu kompilator Groovy spróbuje ponownie skompilować kod aplikacji.

To właśnie w efekcie tego mechanizmu autozależności i autoimportowania CLI w kodzie naszej klasy kontrolera nie musieliśmy umieszczać żadnych instrukcji importu. Nie trzeba będzie także dodawać żadnych bibliotek — ani ręcznie, ani przy użyciu programu Maven bądź Gradle. Spring Boot CLI zadba również o to.

A teraz cofnijmy się o krok i spróbujmy zobaczyć, co się dzieje w naszym kontrolerze. Samo zastosowanie któregoś z typów Spring MVC, takich jak `@Controller` lub `@RequestMapping` sprawi, że CLI automatycznie użyje startera `spring-boot-starter-web`. Po dodaniu do ścieżki klas wszystkich określanych przez niego zależności zostanie uruchomiony mechanizm automatycznej konfiguracji Spring Boot, który automatycznie skonfiguruje wszystkie komponenty niezbędne do działania Spring MVC. Także w tym przypadku wszystko, co musieliśmy zrobić, sprowadziło się do wykorzystania tych typów. Całą resztą zajął się Spring Boot.

Oczywiście możliwości CLI mają swoje granice. Choć CLI wie, jak rozwiązywać wiele zależności Springa i automatycznie importować wiele jego typów (jak również kilka innych bibliotek), to jednak CLI nie będzie w stanie automatycznie określić zależności i zaimportować wszystkiego. Na przykład sami zdecydowaliśmy się na używanie szablonów Thymeleaf, dlatego musieliśmy jawnie o to poprosić, umieszczając w kodzie adnotację `@Grab`.

Warto zwrócić uwagę, że w przypadku wielu zależności nie trzeba podawać ani identyfikatora grupy, ani numeru wersji. Spring Boot potrafi zająć się określaniem zależności dodawanych za pomocą adnotacji `@Grab` i uzupełnić brakujące identyfikatory grup oraz numery wersji.

Ponadto dodanie adnotacji `@Grab` i poproszenie o bibliotekę Thymeleaf spowodowało uruchomienie mechanizmu automatycznej konfiguracji w celu skonfigurowania komponentów niezbędnych do obsługi szablonów Thymeleaf w Spring MVC.

By prezentowana tu wersja aplikacji do obsługi kontaktów napisana w Groovy była kompletna, należy także pokazać kod klasy `Contact`, chociaż nie ma on wiele wspólnego ze Spring Boot. Oto kod tej klasy:

```
class Contact {
  long id
  String firstName
  String lastName
  String phoneNumber
  String emailAddress
}
```

Jak widać, także ta klasa jest znacznie prostsza: w jej kodzie nie ma żadnych średników, żadnych metod dostępowych ani modyfikatorów `private` czy `public`. Zawdzięczamy to w całości nieskomplikowanej składni języka Groovy. W tym przypadku uproszczenie kodu klasy nie ma nic wspólnego ze Spring Boot.

A teraz zobaczmy, jak można uprościć klasę repozytorium, korzystając z możliwości Spring Boot CLI oraz języka Groovy.

21.3.2. Zapewnianie trwałości danych przy użyciu repozytorium Groovy

Wszystkie sztuczki związane z Groovy oraz Spring Boot CLI, których wcześniej użyliśmy w klasie kontrolera `ContactCntroller`, możemy też zastosować w klasie `Contact ↪Repository`. Poniżej, na listingu 21.9, przedstawiony został kod tej klasy napisany w języku Groovy.

Listing 21.9. Klasa ContactRepository napisana w języku Groovy jest znacznie bardziej zwięzła

```
@Grab("h2") <-- Pobranie zależności od bazy danych H2

import java.sql.ResultSet

class ContactRepository {

  @Autowired
  JdbcTemplate jdbc <-- Wstrzyknięcie szablonu JdbcTemplate

  List<Contact> findAll() { <-- Pobranie listy kontaktów
    jdbc.query(
      "select id, firstName, lastName, phoneNumber, emailAddress " +
      "from contacts order by lastName",
      new RowMapper<Contact>() {
        Contact mapRow(ResultSet rs, int rowNum) {
          new Contact(id: rs.getLong(1), firstName: rs.getString(2),
                lastName: rs.getString(3), phoneNumber: rs.getString(4),
                emailAddress: rs.getString(5))
        }
    })
  }

  void save(Contact contact) { <-- Zapisanie kontaktu
    jdbc.update(
      "insert into contacts " +
        "(firstName, lastName, phoneNumber, emailAddress) " +
        "values (?, ?, ?, ?)",
      contact.firstName, contact.lastName,
      contact.phoneNumber, contact.emailAddress)
  }
}
```

Oprócz oczywistych usprawnień wynikających z zastosowania składni języka Groovy ta nowa wersja klasy `ContactRepository` wykorzystuje także mechanizm automatycznego importu Spring Boot CLI, aby automatycznie zaimportować klasy `JdbcTemplate` oraz

`RowMapper`. Co więcej, kiedy CLI zauważy, że klasy te zostały użyte w kodzie, to automatycznie zastosuje starter `spring-boot-starter-jdbc`.

Istnieje tylko kilka przypadków, w których mechanizmy automatycznego importu oraz automatycznego rozwiązywania zależności CLI nie będą nam w stanie pomóc. Jak widać na ostatnim listingu, wciąż musimy jawnie importować klasę `ResultSet`. A ponieważ Spring Boot nie wie, której bazy danych chcemy używać, to aby skorzystać z bazy H2, musimy zastosować adnotację `@Grab`.

W ten sposób udało nam się skonwertować wszystkie klasy Javy na język Groovy, a przy okazji skorzystać z odrobiny magii, jaką zapewnia Spring Boot. Teraz już możemy uruchomić nową wersję aplikacji.

21.3.3. Uruchamianie Spring Boot CLI

Po skompilowaniu aplikacji napisanej w Javie można ją uruchomić na dwa sposoby. Pierwszym jest uruchomienie wykonywalnego pliku JAR lub WAR z poziomu wiersza poleceń, a drugim umieszczenie pliku WAR w kontenerze serwletów. W przypadku stosowania Spring Boot CLI mamy jednak do dyspozycji jeszcze trzecie rozwiązanie.

Nazwa „Spring Boot CLI" sugeruje, że framework ten pozwala na uruchamianie aplikacji z poziomu wiersza poleceń. Ale w przypadku CLI nie ma potrzeby budowania aplikacji w postaci plików JAR bądź WAR. Aplikację można uruchomić bezpośrednio, przekazując do CLI kod źródłowy napisany w Groovy.

INSTALACJA CLI

Aby móc skorzystać ze Spring Boot CLI, trzeba go zainstalować. Można to zrobić na kilka sposobów:

- używając programu Groovy Environment Manager (GVM);
- wykorzystując Homebrew;
- instalując CLI ręcznie.

Żeby zainstalować Spring Boot CLI za pomocą GVM, należy wykonać następujące polecenie:

```
$ gvm install springboot
```

Użytkownicy systemu OS X mogą zainstalować Spring Boot CLI, korzystając z Homebrew:

```
$ brew tap pivotal/tap
$ brew install springboot
```

Jeśli ktoś woli instalować Spring Boot ręcznie, to może to zrobić, postępując zgodnie z instrukcjami opublikowanymi na stronie *http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/*.

Po zainstalowaniu CLI można sprawdzić poprawność instalacji oraz numer używanej wersji, wykonując następujące polecenie:

```
$ spring --version
```

Zakładając, że wszystko udało nam się prawidłowo zainstalować, jesteśmy gotowi do uruchomienia aplikacji.

URUCHAMIANIE APLIKACJI PRZY UŻYCIU CLI

Aby uruchomić aplikacje przy użyciu Spring Boot CLI, w wierszu poleceń należy wykonać polecenie `spring run`, podając w nim także nazwę jednego lub kilku plików Groovy, które CLI ma wykonać. Na przykład jeżeli nasza aplikacja składa się tylko z jednej klasy Groovy, to będziemy mogli ją uruchomić w następujący sposób:

```
$ spring run Hello.groovy
```

Powyższe polecenie powoduje wykonanie klasy Groovy o nazwie `Hello.groovy` przy wykorzystaniu Spring Boot CLI.

Jeśli aplikacja składa się z kilku klas, zapisanych w kilku plikach, to możemy skorzystać ze znaków wieloznacznych:

```
$ spring run *.groovy
```

Jeżeli pliki z klasami Groovy zostały umieszczone w jednym lub kilku podkatalogach, to można odszukać je wszystkie rekurencyjnie, używając w tym celu znaków wieloznacznych przypominających te stosowane w programie Ant:

```
$ sprig run **/*.groovy
```

Ponieważ nasza aplikacja składa się z trzech klas Groovy, które trzeba wczytać, i ponieważ wszystkie one znajdują się w głównym katalogu projektu, możemy ją uruchomić, korzystając z dwóch ostatnich rozwiązań. Po uruchomieniu aplikacji powinniśmy móc wyświetlić w przeglądarce stronę *http://localhost:8080/* i zobaczyć tę samą aplikację, którą wcześniej napisaliśmy w Javie.

A zatem udało nam się już dwa razy napisać tę samą aplikację Spring Boot: za pierwszym razem w Javie, a za drugim w Groovy. W obu przypadkach Spring Boot w znacznym stopniu wspomógł nas swoją magią, pozwalając minimalizować wtórny kod konfiguracyjny oraz określać i rozwiązywać zależności. Jednak Spring Boot ma w zanadrzu jeszcze jedną sztuczkę. Przekonajmy się, jak można skorzystać z aktuatora Spring Boot, by wzbogacić naszą aplikację o punkty końcowe służące do zarządzania.

21.4. Pozyskiwanie informacji o aplikacji z użyciem aktuatora

Podstawowym przeznaczeniem aktuatora Spring Boot jest dodawanie do aplikacji Spring Boot kilku pomocnych punktów końcowych służących do zarządzania nimi. Poniżej przedstawiłem listę tych punktów końcowych:

- `GET /autoconfig` — wyjaśnia podjęte przez Spring Boot decyzje związane z automatyczną konfiguracją.
- `GET /beans` — kataloguje wszystkie komponenty skonfigurowane w działającej aplikacji.
- `GET /configprops` — wyświetla listę wszystkich właściwości, które można skonfigurować w komponentach używanych w działającej aplikacji wraz z ich aktualnymi wartościami.
- `GET /dump` — wyświetla listę wątków aplikacji oraz postać stosu wywołań każdego z nich.

- `GET /env` — wyświetla całą zawartość środowiska i właściwości systemowych dostępnych w kontekście aplikacji.
- `GET /env/{nazwa}` — wyświetla wartość konkretnej zmiennej środowiskowej lub właściwości.
- `GET /health` — wyświetla informacje o bieżącym stanie aplikacji.
- `GET /info` — wyświetla informacje dotyczące aplikacji.
- `GET /metrics` — wyświetla metryki dotyczące aplikacji, w tym sumaryczną liczbę żądań obsłużonych przez poszczególne punkty końcowe.
- `GET / metrics/{nazwa}` — wyświetla metryki dla konkretnego klucza metrycznego aplikacji.
- `GET /shutdown` — wymusza zamknięcie aplikacji.
- `GET /trace` — wyświetla listę metadanych związanych z żądaniami obsłużonymi ostatnio przez aplikację, w tym także nagłówki żądań i odpowiedzi.

Aby włączyć aktuator, wystarczy dodać do projektu starter `spring-boot-starter-actuator`. Jeśli aplikacja jest napisana w języku Groovy, a uruchamiamy ją przy użyciu Spring Boot CLI, to starter aktuatora można dodać, umieszczając w kodzie poniższą adnotację `@Grab`:

```
@Grab("spring-boot-starter-actuator");
```

Jeżeli budujemy aplikację z wykorzystaniem programu Gradle, to do bloku `dependencies` w pliku *build.gradle* wystarczy dodać następującą zależność:

```
compile("org.springframework.boot:spring-boot-starter-actuator")
```

Alternatywnie, w razie stosowania programu Maven, do pliku *pom.xml* projektu należy dodać następujący element `<dependency>`:

```
<dependency>
  <groupId> org.springframework.boot</groupId>
  <artifactId>spring-boot-actuator</carlsbad>
</dependency>
```

Po dodaniu aktuatora Spring Boot można ponownie zbudować i uruchomić aplikację, a następnie wyświetlić w przeglądarce jeden z udostępnianych przez aktuator punktów końcowych. Na przykład chcąc zobaczyć wszystkie komponenty dostępne w kontekście aplikacji, wystarczy wyświetlić w przeglądarce stronę *http://localhost:8080/beans*. Poniżej zaprezentowałem postać zwracanych informacji, pobranych przy użyciu programu narzędziowego CURL (dla poprawy czytelności nieco je sformatowałem i skróciłem):

```
$ curl http://localhost:8080/beans
[
  {
  "beans": [
    {
      "bean": "contactController",
      "dependencies": [
        "contactRepository"
      ],
      "resource": "null",
      "scope": "singleton",
```

```
      "type": "ContactController"
    },
    {
      "bean": "contactRepository",
      "dependencies": [
        "jdbcTemplate"
      ],
      "resource": "null",
      "scope": "singleton",
      "type": "ContactRepository"
    },

    ...

    {
      "bean": "jdbcTemplate",
      "dependencies": [],
      "resource": "class path resource [...]",
      "scope": "singleton",
      "type": "org.springframework.jdbc.core.JdbcTemplate"
    },

    ...
    ]
  }
]
```

Jak widać, w kontekście aplikacji istnieje komponent o identyfikatorze `contactController`, który jest zależny od innego komponentu — o identyfikatorze `contactRepository`. Z kolei kontroler `contactRepository` zależy od komponentu `jdbcTemplate`.

Powyższe dane zostały nieco skrócone; w rzeczywistości w aplikacji dostępnych jest kilkanaście innych komponentów, które normalnie zobaczylibyśmy w kodzie JSON zwracanym przez punkt końcowy. Dzięki nim możemy zdobyć trochę informacji o tajemniczych wynikach działania mechanizmów automatycznego wiązania i automatycznej konfiguracji.

Kolejnym punktem końcowym, który pozwala zdobyć informacje o działaniu mechanizmu automatycznej konfiguracji Spring Boot, jest punkt końcowy */autoconfig*. Zwracany przez niego kod JSON wyjaśnia decyzje podejmowane przez Spring Boot podczas konfigurowania komponentów. Na przykład poniżej przedstawiłem skrócony (i sformatowany) kod JSON zwrócony po odwołaniu się do tego punktu końcowego w naszej aplikacji do zarządzania listą kontaktów:

```
$ curl http://localhost:8080/autoconfig
{
"negativeMatches": {
  "AopAutoConfiguration": [
    {
      "condition": "OnClassCondition",
      "message": "required @ConditionalOnClass classes not found:
          org.aspectj.lang.annotation.Aspect,
          org.aspectj.lang.reflect.Advice"
    }
  ],
```

```
    "BatchAutoConfiguration": [
      {
        "condition": "OnClassCondition",
        "message": "required @ConditionalOnClass classes not found:
              org.springframework.batch.core.launch.JobLauncher"
      }
    ],

  ...
  },
  "positiveMatches": {
    "ThymeleafAutoConfiguration": [
      {
        "condition": "OnClassCondition",
        "message": "@ConditionalOnClass classes found:
              org.thymeleaf.spring4.SpringTemplateEngine"
      }
    ],
    "ThymeleafAutoConfiguration.DefaultTemplateResolverConfiguration":[
      {
        "condition": "OnBeanCondition",
        "message": "@ConditionalOnMissingBean
              (names: defaultTemplateResolver; SearchStrategy: all)
              found no beans"
      }
    ],
    "ThymeleafAutoConfiguration.ThymeleafDefaultConfiguration": [
      {
        "condition": "OnBeanCondition",
        "message": "@ConditionalOnMissingBean (types:
              org.thymeleaf.spring4.SpringTemplateEngine;
              SearchStrategy: all) found no beans"
      }
    ],
    "ThymeleafAutoConfiguration.ThymeleafViewResolverConfiguration": [
      {
        "condition": "OnClassCondition",
        "message": "@ConditionalOnClass classes found:
              javax.servlet.Servlet"
      }
    ],
    "ThymeleafAutoConfiguration.ThymeleafViewResolverConfiguration
           #thymeleafViewResolver": [
      {
        "condition": "OnBeanCondition",
        "message": "@ConditionalOnMissingBean (names:
              thymeleafViewResolver; SearchStrategy: all)
              found no beans"
      }
    ],

    ...
  }
  }
```

Jak widać, zwrócony raport składa się z dwóch sekcji: jednej dla dopasowań negatywnych i drugiej dla pozytywnych. Sekcja dopasowań negatywnych zaprezentowana w powyższym przykładzie informuje, że nie została zastosowana automatyczna konfiguracja AOP oraz Spring Batch, gdyż w ścieżce klas nie udało się znaleźć odpowiednich klas. Analizując drugą sekcję, możemy się przekonać, że dzięki odnalezieniu w ścieżce klas klasy `SpringTemplateEngine` można było przeprowadzić automatyczną konfigurację Thymeleaf. Poza tym znajdziemy tam informację o automatycznym skonfigurowaniu domyślnego producenta szablonów, producenta widoków oraz silnika obsługi szablonów (nie będą one wyświetlane, jeśli sami skonfigurowaliśmy te komponenty). Co więcej, komponent domyślnego producenta widoków zostanie automatycznie skonfigurowany wyłącznie w przypadku, gdy w ścieżce klas uda się odnaleźć klasę `Servlet`.

Punkty końcowe */bean* oraz */autoconfig* są jedynie dwoma przykładami pokazującymi, jakie informacje udostępnia aktuator Spring Boot. Niestety nie jestem w stanie zamieścić w tym rozdziale szczegółowych opisów wszystkich punktów końcowych aktuatora, lecz zachęcam do samodzielnego ich wypróbowania i przekonania się, co aktuator ma do powiedzenia na temat naszych aplikacji.

21.5. Podsumowanie

Spring Boot jest fascynującym, nowym dodatkiem do rodziny projektów związanych z frameworkiem Spring. O ile sam Spring stara się ułatwić tworzenie programów w języku Java, Spring Boot stara się ułatwić stosowanie frameworka Spring.

Spring Boot próbuje wyeliminować konieczność tworzenia wtórnego kodu konfiguracyjnego, wykorzystując dwie główne sztuczki: startery Spring Boot oraz automatyczną konfigurację.

Startery Spring Boot mogą zastąpić kilka często używanych zależności umieszczanych w plikach budowy programów Maven lub Gradle. Na przykład dodanie do projektu tylko jednego startera `spring-boot-starter-web` spowoduje dodanie modułu Web oraz Spring MVC, jak również modułu Jackson 2.

Mechanizm automatycznej konfiguracji korzysta z konfiguracji warunkowej wprowadzonej w Springu 4.0 i pozwala automatycznie konfigurować niektóre komponenty, aby udostępniać wybrane możliwości. Na przykład Spring Boot potrafi wykryć, że w ścieżce klas jest dostępna biblioteka Thymeleaf, i automatycznie skonfigurować komponenty niezbędne do stosowania szablonów Thymeleaf jako widoków w Spring MVC.

Interfejs wiersza poleceń Spring Boot (określany skrótowo jako „Spring Boot CLI") pozwala na jeszcze większe upraszczanie projektów Spring dzięki wykorzystaniu języka Groovy. Upraszczając odwołania do komponentów Spring w kodzie Groovy, możemy używać CLI do automatycznego dodawania niezbędnych zależności początkowych (co z kolei może uruchomić mechanizm automatycznej konfiguracji). Co więcej, w przypadku uruchamiania kodu Groovy za pomocą Spring Boot CLI wiele typów Spring używanych w tym kodzie nie wymaga jawnego stosowania instrukcji `import`.

I w końcu — aktuator Spring Boot udostępnia dodatkowe możliwości zarządzania aplikacjami internetowymi korzystającymi ze Spring Boot. Należy do nich na przykład wyświetlanie informacji o aktualnym stanie wątków, historii obsłużonych żądań czy też komponentach dostępnych w kontekście aplikacji.

Po przeczytaniu tego rozdziału możesz się zastanawiać, dlaczego prezentację tak przydatnego projektu jak Spring Boot pozostawiłem na sam koniec książki. Mogłeś nawet pomyśleć, że gdybym przedstawił Spring Boot wcześniej, to przyswojenie wielu zagadnień opisanych na początku książki byłoby znacznie łatwiejsze. Prawdą jest, że Spring Boot udostępnia bardzo atrakcyjny model programowania, który przesłania tradycyjny sposób pisania aplikacji w Springu, a kiedy już się do niego przyzwyczaimy, to trudno będzie sobie wyobrazić pisanie aplikacji bez Spring Boot.

Mógłbym napisać, że zostawiając prezentację Spring Boot na sam koniec, chciałem, byś bardziej docenił Springa (a może także chciałem, żebyś zahartował swój charakter i trochę dorósł). Choć to mogłoby być prawdą, to jednak rzeczywistym powodem było to, że w momencie, kiedy udostępniano Spring Boot, przeważająca większość tej książki była już gotowa. Dlatego część dotyczącą Spring Boot umieściłem w jedynym miejscu, w którym mogłem to zrobić bez całkowitej reorganizacji zawartości książki — na jej końcu.

Kto wie — może następne wydanie tej książki zacznie się od opisu Spring Boot?

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

T

U

V

W

X

Z

Ź

Ż