Craig Walls

SPRING W AKCJI

Kompendium wiedzy na temat
Spring Framework!

Wydanie IV

Helion

Tytuł oryginału: Spring in Action, Fourth Edition

Tłumaczenie: Mirosław Gołda (wstęp, rozdz. 1 – 14),
Piotr Rajca (rozdz. 15 – 21)

ISBN: 978-83-283-0849-7

Original edition copyright © 2015 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2015 by HELION SA.
All rights reserved.

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

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

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

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

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

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/sprwa4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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

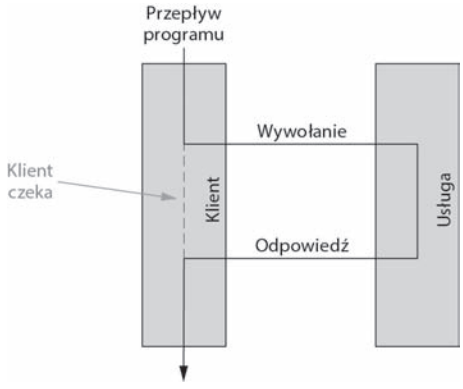
Podobnie jak w przypadku mechanizmów zdalnego dostępu i interfejsów REST, którymi zajmowaliśmy się wcześniej w tej części książki, asynchroniczna wymiana komunikatów służy do nawiązywania komunikacji pomiędzy aplikacjami. Jednak różni się ona od przedstawionych wcześniej mechanizmów sposobem przekazywania 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.

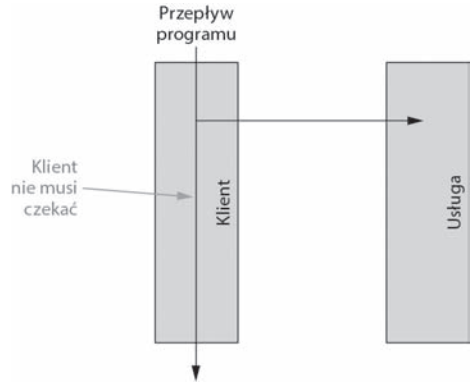
Z drugiej strony, kiedy komunikaty są przesyłane asynchronicznie, jak pokazano na rysunku 17.2, klient nie musi czekać, aż usługa przetworzy komunikat, ani nawet aż zostanie on dostarczony. Klient wysyła komunikat i kontynuuje działanie, zakładając, że prędzej czy później dotrze on do usługi i zostanie 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.

Kup książkę Poleć książkę



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".

Kup książkę Poleć książkę

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

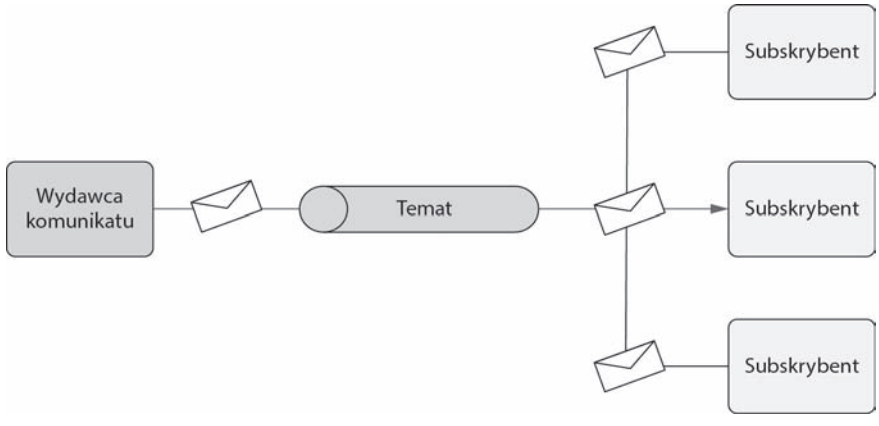
W modelu obsługi komunikatów publikacja-subskrypcja komunikaty są wysyłane do tematu. Tak jak w przypadku kolejek, wielu odbiorców nasłuchuje komunikatów z tematu. Ale w przeciwieństwie do kolejek, gdzie dany komunikat jest doręczany tylko i wyłącznie jednemu odbiorcy, wszyscy subskrybenci tematu otrzymają kopię komunikatu (rysunek 17.4).

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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.*

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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) {
    this.jmsOperations = jmsOperations;
```

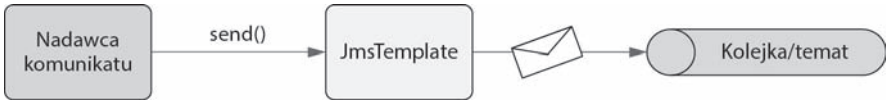
← Wstrzykuje szablon JMS

Kup książkę Poleć książkę

```
  }

  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()` `JmsTemplate` 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 `MessageCreator`, 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 `JMSException` 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" />
```

Kup książkę Poleć książkę

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)
```

Kup książkę Poleć książkę

```
                    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?

Kup książkę Poleć książkę

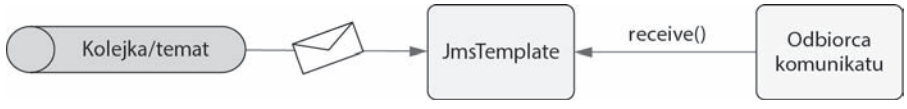
Owszem. I jest to nawet prostsze niż wysyłanie komunikatów. Wystarczy tylko wywołać metodę `receive()` szablonu JMS w sposób pokazany na listingu 17.4.

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()`:

Kup książkę Poleć książkę

```
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.*

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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" />
```

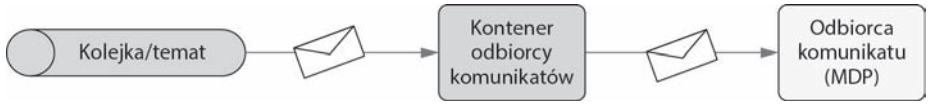
Następnie, aby zmienić `SpittleAlertHandler` w sterowany komunikatami obiekt POJO, możemy zadeklarować, że komponent ten jest odbiorcą komunikatów:

```
<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

Kup książkę Poleć książkę



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 `connectionFactory`.

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 `spittleHandler` (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 `JmsInvokerServiceExporter` 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 `HttpInvokerServiceExporter`, 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: `JmsInvokerServiceExporter` oraz `HttpInvokerServiceExporter`. Skoro `HttpInvokerServiceExporter` eksportuje usługi komunikujące się przy użyciu protokołu HTTP, oznacza to, że `JmsInvokerServiceExporter` musi eksportować usługi porozumiewające się za pomocą JMS.

Kup książkę Poleć książkę

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:

Kup książkę Poleć książkę

```
<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

Kup książkę Poleć książkę

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.

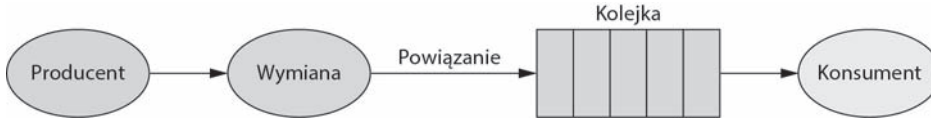
Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

- 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.

Kup książkę Poleć książkę

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}" />
```

Kup książkę Poleć książkę

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

Kup książkę Poleć książkę

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`:

Kup książkę Poleć książkę

```
<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

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

Skorowidz

A

Kup książkę Poleć książkę

B

C

D

Kup książkę Poleć książkę

E

F

G

H

I

Kup książkę Poleć książkę

J

K

Kup książkę Poleć książkę

L

Ł

M

Kup książkę Poleć książkę

N

O

P

Kup książkę Poleć książkę

R

S

Kup książkę Poleć książkę

T

U

V

W

Kup książkę Poleć książkę

X

Z

Kup książkę Poleć książkę

Ź

Ż

Kup książkę Poleć książkę

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

Spring jest odpowiedzią na problemy trapiące programistów, którzy tworzą oprogramowanie przy użyciu EJB 2.x. Dzień, w którym został udostępniony szerokiemu gronu użytkowników, był punktem zwrotnym w historii języka Java. Od tej pory życie deweloperów jest prostsze, a tworzenie nawet skomplikowanych aplikacji — zdecydowanie przyjemniejsze. Od tamtego czasu Spring jest wciąż rozwijany i oferuje coraz lepsze narzędzia programistom na całym świecie.

Kolejne wydanie tej książki, w całości poświęconej frameworkowi Spring, zostało poprawione, zaktualizowane i uzupełnione o nowe informacje. W trakcie lektury przekonasz się, jakie nowości zostały wprowadzone w czwartej wersji Springa, oraz zaznajomisz się z zaawansowanymi metodami wiązania komponentów. Ponadto zdobędziesz doświadczenie w stosowaniu aspektów, zobaczysz, jak działają Spring MVC i Spring WebFlow, oraz nauczysz się uzyskiwać dostęp do baz danych — zarówno SQL, jak i NoSQL. Osobny rozdział został poświęcony bezpieczeństwu aplikacji tworzonych z wykorzystaniem Springa. Spring Security to potężne narzędzie, które pozwoli Ci bezboleśnie wprowadzić zaawansowane mechanizmy bezpieczeństwa w Twoich programach. Na sam koniec poznasz techniki obsługi komunikatów oraz możliwości modułu Spring Boot. Książka ta jest doskonałą lekturą dla programistów chcących w pełni wykorzystać potencjał Springa!

Dzięki tej książce:

- poznasz komponenty składające się na Spring Framework
- zabezpieczysz aplikację za pomocą Spring Security
- błyskawicznie uruchomisz projekt ze Spring Boot
- z łatwością skorzystasz z baz danych SQL i NoSQL
- wykorzystasz potencjał najnowszej wersji Springa

Poznaj potencjał Springa!

Craig Walls — starszy programista w firmie Pivotal. Zaangażowany w promocję frameworka Spring, często występuje jako prelegent na konferencjach i swoje wystąpienia poświęca właśnie Springowi. Jego książki cieszą się dużą popularnością.

sięgnij po WIĘCEJ

KOD KORZYŚCI

Helion

35462 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

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

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

ISBN 978-83-283-0849-7

9 788328 308497

Informatyka w najlepszym wydaniu

cena: 89,00 zł