

Craig Walls

SPRING

WYDANIE V

W AKCJI



Helion 

Tytuł oryginału: Spring in Action, 5th Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-5606-1

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

Original edition copyright © 2019 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2019 by Helion SA
All rights reserved.

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

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

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

Autor oraz Helion SA 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 Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/sprwa5.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/sprwa5>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

Wprowadzenie	13
Podziękowania	15
O książce	17

CZĘŚĆ I. PODSTAWY SPRINGA 21

Rozdział 1. Rozpoczęcie pracy ze Springiem 23

1.1.	Czym jest Spring?	24
1.2.	Inicjalizacja aplikacji Springa	26
1.2.1.	Inicjalizacja projektu Springa za pomocą Spring Tool Suite	27
1.2.2.	Analiza struktury projektu Springa	30
1.3.	Utworzenie aplikacji Springa	37
1.3.1.	Obsługa żądań internetowych	37
1.3.2.	Definiowanie widoku	39
1.3.3.	Testowanie kontrolera	40
1.3.4.	Kompilacja i uruchomienie aplikacji	41
1.3.5.	Poznajemy Spring Boot DevTools	43
1.3.6.	Przegląd	45
1.4.	Możliwości Springa	46
1.4.1.	Podstawowy framework Spring	47
1.4.2.	Spring Boot	47
1.4.3.	Spring Data	48
1.4.4.	Spring Security	48
1.4.5.	Spring Integration i Spring Batch	48
1.4.6.	Spring Cloud	48
	Podsumowanie	49

Rozdział 2. Tworzenie aplikacji internetowej 51

2.1.	Wyświetlanie informacji	52
2.1.1.	Zdefiniowanie domeny	52
2.1.2.	Utworzenie klasy kontrolera	55
2.1.3.	Przygotowanie widoku	58
2.2.	Przetwarzanie wysłanego formularza	62
2.3.	Weryfikacja danych wyjściowych formularza	68
2.3.1.	Deklarowanie reguł weryfikacji danych	68
2.3.2.	Przeprowadzanie weryfikacji danych podczas ich pobierania z formularza	70
2.3.3.	Wyświetlanie błędów operacji sprawdzania poprawności danych	72
2.4.	Praca z kontrolerem widoku	72
2.5.	Wybór biblioteki szablonów widoku	75
2.5.1.	Buforowanie szablonów	77
	Podsumowanie	78

Rozdział 3. Praca z danymi 79

- 3.1. Odczyt i zapis danych za pomocą JDBC 79
 - 3.1.1. Przygotowanie domeny do obsługi trwałego magazynu danych 82
 - 3.1.2. Praca z klasą `JdbcTemplate` 83
 - 3.1.3. Definiowanie schematu i wstępne przygotowanie danych 87
 - 3.1.4. Wstawianie danych 89
- 3.2. Implementacja trwałego magazynu danych za pomocą Spring Data JPA 98
 - 3.2.1. Dodawanie Spring Data JPA do projektu 99
 - 3.2.2. Określenie domeny jako encji 99
 - 3.2.3. Deklarowanie repozytoriów JPA 102
 - 3.2.4. Dostosowanie do własnych potrzeb repozytoriów JPA 104
- Podsumowanie 106

Rozdział 4. Spring Security 107

- 4.1. Włączenie obsługi Spring Security 108
- 4.2. Konfigurowanie Spring Security 110
 - 4.2.1. Istniejący w pamięci magazyn danych użytkownika 111
 - 4.2.2. Magazyn danych użytkownika oparty na JDBC 112
 - 4.2.3. Magazyn danych użytkownika oparty na LDAP 115
 - 4.2.4. Dostosowanie uwierzytelniania użytkownika do własnych potrzeb 118
- 4.3. Zabezpieczanie żądań internetowych 125
 - 4.3.1. Zabezpieczanie żądań 126
 - 4.3.2. Utworzenie własnej strony logowania 129
 - 4.3.3. Wylogowanie 131
 - 4.3.4. Ochrona przed atakami typu CSRF 131
- 4.4. Poznanie użytkownika 133
- Podsumowanie 135

Rozdział 5. Praca z właściwościami konfiguracyjnymi 137

- 5.1. Dostosowanie konfiguracji automatycznej do własnych potrzeb 138
 - 5.1.1. Poznajemy abstrakcję środowiska Springa 139
 - 5.1.2. Konfigurowanie źródła danych 140
 - 5.1.3. Konfigurowanie serwera osadzonego 142
 - 5.1.4. Konfigurowanie rejestrowania danych 143
 - 5.1.5. Używanie wartości właściwości specjalnych 144
- 5.2. Tworzenie własnych właściwości konfiguracyjnych 145
 - 5.2.1. Definiowanie komponentów przechowujących właściwości konfiguracyjne 147
 - 5.2.2. Deklarowanie metadanych właściwości konfiguracyjnej 149
- 5.3. Konfigurowanie za pomocą profili 152
 - 5.3.1. Definiowanie właściwości dla konkretnego profilu 153
 - 5.3.2. Aktywowanie profilu 154
 - 5.3.3. Warunkowe tworzenie komponentu bean z profilami 155
- Podsumowanie 156

CZĘŚĆ II. ZINTEGROWANY SPRING 157**Rozdział 6. Praca z właściwościami konfiguracyjnymi 159**

- 6.1. Utworzenie kontrolerów RESTful 160
 - 6.1.1. Pobieranie danych z serwera 162
 - 6.1.2. Przekazywanie danych do serwera 167
 - 6.1.3. Uaktualnienie danych w serwerze 168
 - 6.1.4. Usuwanie danych z serwera 170
- 6.2. Włączenie obsługi hipermediów 171
 - 6.2.1. Dodawanie hiperłączy 174
 - 6.2.2. Utworzenie komponentu asemblera zasobu 176
 - 6.2.3. Nazewnictwo osadzonych związków 180
- 6.3. Włączenie usług back-endu 181
 - 6.3.1. Dostosowanie nazw ścieżek dostępu zasobów i relacji 184
 - 6.3.2. Stronicowanie i sortowanie 186
 - 6.3.3. Dodawanie własnych punktów końcowych 187
 - 6.3.4. Dodawanie własnych hiperłączy do punktów końcowych Spring Data 189
- Podsumowanie 190

Rozdział 7. Używanie usług REST 191

- 7.1. Używanie punktów końcowych REST za pomocą RestTemplate 192
 - 7.1.1. Pobieranie zasobu 194
 - 7.1.2. Przekazywanie do serwera zasobów za pomocą metody HTTP PUT 195
 - 7.1.3. Usuwanie zasobu 196
 - 7.1.4. Przekazywanie do serwera zasobów za pomocą metody HTTP POST 196
- 7.2. Poruszanie się po API REST za pomocą Traverson 197
- Podsumowanie 199

Rozdział 8. Asynchroniczne wysyłanie komunikatów 201

- 8.1. Wysyłanie komunikatów za pomocą JMS 202
 - 8.1.1. Konfigurowanie JMS 202
 - 8.1.2. Wysyłanie komunikatów za pomocą JmsTemplate 204
 - 8.1.3. Otrzymywanie komunikatów JMS 211
- 8.2. Praca z RabbitMQ i AMQP 215
 - 8.2.1. Dodawanie obsługi brokera RabbitMQ do Springa 216
 - 8.2.2. Wysyłanie komunikatów za pomocą RabbitTemplate 217
 - 8.2.3. Pobieranie komunikatu z RabbitMQ 221
- 8.3. Obsługa komunikatów za pomocą Apache Kafki 225
 - 8.3.1. Konfigurowanie Springa do obsługi komunikatów Kafki 226
 - 8.3.2. Wysyłanie komunikatów za pomocą KafkaTemplate 227
 - 8.3.3. Utworzenie komponentu nasłuchującego Kafki 229
- Podsumowanie 231

Rozdział 9. Integracja Springa 233

- 9.1. Deklarowanie prostego przepływu integracji 234
 - 9.1.1. Definiowanie przepływu integracji za pomocą XML-a 235
 - 9.1.2. Konfigurowanie przepływu integracji za pomocą Javy 237
 - 9.1.3. Konfigurowanie Spring Integration za pomocą języka specjalizowanego 239

- 9.2. **Poznajemy Spring Integration** 241
 - 9.2.1. *Kanał komunikatu* 241
 - 9.2.2. *Filtr* 243
 - 9.2.3. *Przekształcenie* 244
 - 9.2.4. *Router* 245
 - 9.2.5. *Spliter* 247
 - 9.2.6. *Aktywator usługi* 249
 - 9.2.7. *Brama* 251
 - 9.2.8. *Adapter kanału* 252
 - 9.2.9. *Moduł punktu końcowego* 254
- 9.3. **Utworzenie własnego przepływu integracji dotyczącego poczty elektronicznej** 256
- Podsumowanie** 261

CZĘŚĆ III. REAKTYWNY SPRING 263

Rozdział 10. Wprowadzenie do projektu Reactor 265

- 10.1. **Wprowadzenie do programowania reaktywnego** 266
 - 10.1.1. *Definiowanie strumienia reaktywnego* 267
- 10.2. **Rozpoczęcie pracy z projektem Reactor** 269
 - 10.2.1. *Wykres przepływu reaktywnego* 270
 - 10.2.2. *Dodawanie zależności projektu Reactor* 271
- 10.3. **Najczęściej stosowane operacje reaktywne** 272
 - 10.3.1. *Tworzenie typu reaktywnego* 273
 - 10.3.2. *Łączenie typów reaktywnych* 277
 - 10.3.3. *Przekształcanie i filtrowanie strumienia reaktywnego* 281
 - 10.3.4. *Przeprowadzanie operacji logicznej na typie reaktywnym* 290
- Podsumowanie** 292

Rozdział 11. Tworzenie reaktywnego API 293

- 11.1. **Praca z frameworkiem WebFlux w Springu** 293
 - 11.1.1. *Wprowadzenie do Spring WebFlux* 295
 - 11.1.2. *Definiowanie kontrolera reaktywnego* 296
- 11.2. **Definiowanie funkcyjnych procedur obsługi żądania** 300
- 11.3. **Testowanie kontrolera reaktywnego** 303
 - 11.3.1. *Testowanie żądań HTTP GET* 303
 - 11.3.2. *Testowanie żądań POST* 306
 - 11.3.3. *Testowanie działającego serwera* 307
- 11.4. **Reaktywne używanie API REST** 308
 - 11.4.1. *Pobieranie zasobów* 309
 - 11.4.2. *Przekazywanie zasobu* 311
 - 11.4.3. *Usunięcie zasobu* 312
 - 11.4.4. *Obsługa błędów* 312
 - 11.4.5. *Wymiana żądań* 314
- 11.5. **Zabezpieczanie API reaktywnego** 316
 - 11.5.1. *Konfigurowanie zabezpieczeń reaktywnej aplikacji internetowej* 316
 - 11.5.2. *Konfigurowanie reaktywnej usługi szczegółów związanych z użytkownikiem* 318
- Podsumowanie** 319

Rozdział 12. Reaktywny trwały magazyn danych 321

- 12.1. Reaktywność i Spring Data 322
 - 12.1.1. *Reaktywny Spring Data* 323
 - 12.1.2. *Konwersja między typem reaktywnym a niereaktywnym* 323
 - 12.1.3. *Opracowanie repozytorium reaktywnego* 325
- 12.2. Praca z reaktywnymi repozytoriami bazy danych Cassandra 325
 - 12.2.1. *Włączenie obsługi Spring Data Cassandra* 326
 - 12.2.2. *Modelowanie danych w bazie danych Cassandra* 328
 - 12.2.3. *Mapowanie typów domeny pod kątem przechowywania informacji w bazie danych Cassandra* 329
 - 12.2.4. *Tworzenie reaktywnego repozytorium bazy danych Cassandra* 335
- 12.3. Tworzenie reaktywnych repozytoriów MongoDB 337
 - 12.3.1. *Dodawanie obsługi Spring Data MongoDB* 338
 - 12.3.2. *Mapowanie typu domeny na dokument MongoDB* 339
 - 12.3.3. *Tworzenie interfejsów repozytoriów reaktywnych MongoDB* 343
- Podsumowanie 345

CZĘŚĆ IV. NATYWNA CHMURA SPRINGA 347**Rozdział 13. Odkrywanie usług 349**

- 13.1. Poznajemy mikrousługi 350
- 13.2. Konfiguracja rejestru usług 352
 - 13.2.1. *Konfigurowanie Eureka* 356
 - 13.2.2. *Skalowanie serwera Eureka* 359
- 13.3. Rejestrowanie i odkrywanie usług 361
 - 13.3.1. *Konfigurowanie właściwości klienta Eureka* 362
 - 13.3.2. *Używanie usługi* 363
- Podsumowanie 368

Rozdział 14. Zarządzanie konfiguracją 371

- 14.1. Konfiguracja współdzielona 372
- 14.2. Uruchamianie serwera konfiguracji 373
 - 14.2.1. *Włączanie Config Server* 374
 - 14.2.2. *Umieszczanie właściwości w repozytorium konfiguracyjnym* 377
- 14.3. Używanie konfiguracji współdzielonej 380
- 14.4. Udostępnienie konfiguracji przeznaczonej dla konkretnej aplikacji i konkretnego profilu 382
 - 14.4.1. *Udostępnianie właściwości przeznaczonych dla konkretnej aplikacji* 382
 - 14.4.2. *Udostępnianie właściwości z profili* 383
- 14.5. Utajnienie właściwości konfiguracyjnych 385
 - 14.5.1. *Szyfrowanie właściwości w repozytorium Git* 386
 - 14.5.2. *Przechowywanie danych wrażliwych w magazynie Vault* 389
- 14.6. Odświeżanie właściwości konfiguracyjnych w locie 393
 - 14.6.1. *Ręczne odświeżanie właściwości konfiguracyjnych* 394
 - 14.6.2. *Automatyczne odświeżanie właściwości konfiguracyjnych* 396
- Podsumowanie 404

Rozdział 15. Obsługa awarii i opóźnień 405

- 15.1. Wprowadzenie do wzorca bezpiecznika 405
- 15.2. Deklarowanie bezpiecznika 408
 - 15.2.1. Łagodzenie opóźnienia 410
 - 15.2.2. Zarządzanie wartościami granicznymi bezpiecznika 411
- 15.3. Monitorowanie awarii 412
 - 15.3.1. Wprowadzenie do panelu kontrolnego biblioteki Hystrix 413
 - 15.3.2. Pula wątków biblioteki Hystrix 416
- 15.4. Agregowanie wielu strumieni biblioteki Hystrix 418
- Podsumowanie 419

CZĘŚĆ V. WDRAŻANIE APLIKACJI SPRINGA 421**Rozdział 16. Praca ze Spring Boot Actuator 423**

- 16.1. Wprowadzenie do Actuatora 424
 - 16.1.1. Konfigurowanie bazowej ścieżki dostępu Actuatora 425
 - 16.1.2. Włączanie i wyłączanie punktów końcowych Actuatora 426
- 16.2. Używanie punktów końcowych Actuatora 427
 - 16.2.1. Pobieranie podstawowych informacji o aplikacji 428
 - 16.2.2. Wyświetlanie konfiguracji aplikacji 431
 - 16.2.3. Wyświetlanie informacji o aktywności aplikacji 439
 - 16.2.4. Dane statystyczne dotyczące środowiska uruchomieniowego 442
- 16.3. Dostosowanie Actuatora do własnych potrzeb 444
 - 16.3.1. Dodawanie informacji dostarczanych później przez punkt końcowy /info 445
 - 16.3.2. Definiowanie własnych wskaźników informacji o stanie aplikacji 449
 - 16.3.3. Rejestrowanie niestandardowych danych statystycznych 450
 - 16.3.4. Tworzenie własnego punktu końcowego 452
- 16.4. Zabezpieczanie Actuatora 455
- Podsumowanie 457

Rozdział 17. Administrowanie Springiem 459

- 17.1. Używanie Spring Boot Admin 459
 - 17.1.1. Tworzenie serwera administracyjnego 460
 - 17.1.2. Rejestrowanie klientów serwera administracyjnego 462
- 17.2. Poznajemy serwer administracyjny 465
 - 17.2.1. Wyświetlanie ogólnych informacji o stanie aplikacji 466
 - 17.2.2. Obserwowanie kluczowych danych statystycznych 466
 - 17.2.3. Analiza zmiennych środowiskowych 467
 - 17.2.4. Wyświetlanie i zmiana poziomu rejestrowania danych 469
 - 17.2.5. Monitorowanie wątków 469
 - 17.2.6. Monitorowanie żądań HTTP 470
- 17.3. Zabezpieczanie serwera administracyjnego 471
 - 17.3.1. Włączanie logowania w serwerze administracyjnym 472
 - 17.3.2. Uwierzytelnianie z użyciem Actuatora 473
- Podsumowanie 474

Rozdział 18. Monitorowanie Springa za pomocą JMX 475

- 18.1. Praca z Actuatorem i MBean 476
- 18.2. Tworzenie własnego komponentu MBean 478
- 18.3. Wysyłanie powiadomień 480
- Podsumowanie 481

Rozdział 19. Wdrażanie aplikacji Springa 483

- 19.1. Opcje podczas wdrażania aplikacji 484
- 19.2. Kompilowanie i wdrażanie pliku WAR 485
- 19.3. Przekazanie pliku JAR do Cloud Foundry 487
- 19.4. Umieszczanie aplikacji Spring Boota w kontenerze Dockera 490
- 19.5. To nie koniec, ale dopiero początek 494
- Podsumowanie 494

DODATKI 495**Dodatek A. Tworzenie aplikacji Springa 497**

- A.1. Inicjalizacja projektu za pomocą Spring Tool Suite 497
- A.2. Tworzenie projektu za pomocą IntelliJ IDEA 500
- A.3. Tworzenie projektu za pomocą NetBeans 503
- A.4. Inicjalizacja projektu na stronie start.spring.io 506
- A.5. Inicjalizacja projektu z poziomu powłoki 510
 - A.5.1. Polecenie curl i API Inicjalizacji 510
 - A.5.2. Interfejs powłoki dla Spring Boota 512
- A.6. Tworzenie aplikacji Springa za pomocą metaframeworka 513
- A.7. Kompilowanie i uruchamianie projektu 513

Skorowidz 515

Praca z właściwościami konfiguracyjnymi

W tym rozdziale:

- dostosowanie skonfigurowanych automatycznie komponentów bean;
- zastosowanie właściwości konfiguracyjnych dla komponentów aplikacji;
- praca z profilami Springa.

Czy pamiętasz, jak pojawił się pierwszy iPhone? Niewielka metalowa płytka ze szkłem z trudem wpisywała się w to, co dotychczas było uznawane za telefon. Mimo tego iPhone był pionierem ery smartfonów i zmienił właściwie wszystko w zakresie komunikacji. Wprawdzie telefon z ekranem dotykowym jest pod wieloma względami łatwiejszy w obsłudze i oferuje znacznie potężniejsze możliwości od poprzedników, ale po pojawieniu się iPhone'a trudno było sobie wyobrazić, jak urządzenie zawierające tylko jeden przycisk może być używane do wykonywania połączeń.

Pod pewnymi względami konfiguracja automatyczna Spring Boota działa w taki właśnie sposób i znacznie ułatwia tworzenie aplikacji Springa. Jednak po dekadzie definiowania wartości właściwości w konfiguracji Spring XML i wywoływania metod w egzemplarzach komponentów bean nie od razu będzie jasne, jak należy definiować właściwości komponentów bean, dla których nie istnieje wyraźnie przygotowana konfiguracja.

Na szczęście Spring Boot dostarcza rozwiązanie w postaci właściwości konfiguracyjnych. To po prostu właściwości w komponencie bean kontekstu aplikacji Springa, które mogą być definiowane na podstawie jednego z wielu źródeł właściwości, m.in.: systemu JVM, argumentów powłoki i zmiennych środowiskowych.

W tym rozdziale nie będziesz zajmować się implementacją nowych funkcji aplikacji Taco Cloud, a zamiast tego poznasz właściwości konfiguracyjne. Zdobyta tutaj wiedza bez wątpienia będzie użyteczna w trakcie lektury kolejnych rozdziałów. Na początek zobaczysz, jak wykorzystać właściwości konfiguracyjne do dostosowania ustawień zdefiniowanych już przez konfigurację automatyczną Spring Boota.

5.1. Dostosowanie konfiguracji automatycznej do własnych potrzeb

Zanim zagłębisz się we właściwości konfiguracyjne, musisz dowiedzieć się o istnieniu dwóch odmiennych (choć powiązanych ze sobą) rodzajów konfiguracji w Springu.

- *Łączenie komponentów bean* — konfiguracja deklarująca utworzenie komponentów aplikacji jako komponentów bean w kontekście aplikacji Springa, a także określenie sposobu ich wstrzykiwania do siebie nawzajem.
- *Wstrzyknięcie właściwości* — konfiguracja przypisująca wartości komponentom bean w kontekście aplikacji Springa.

W konfiguracji Spring XML i opartej na Javie te dwa rodzaje konfiguracji są bardzo często zadeklarowane w tym samym miejscu. W konfiguracji Javy metoda oznaczona adnotacją `@Bean` prawdopodobnie powoduje utworzenie egzemplarza komponentu bean, a następnie przypisanie wartości jego właściwościom. Rozważ przedstawioną w kolejnym fragmencie kodu metodę z adnotacją `@Bean` deklarującą źródło danych dla osadzonej bazy danych H2.

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDataSourceBuilder()
        .setType(H2)
        .addScript("taco_schema.sql")
        .addScripts("user_data.sql", "ingredient_data.sql")
        .build();
}
```

W tym przykładzie metody `addScript()` i `addScripts()` definiują pewne właściwości typu ciągu tekstowego z nazwami skryptów SQL, które powinny zostać wykonane w bazie danych, gdy źródło danych będzie gotowe. Wprawdzie w taki sposób możesz konfigurować komponent bean źródła danych, jeśli nie używasz Spring Boota, ale konfiguracja automatyczna powoduje, że ta metoda jest zupełnie zbędna.

Jeżeli zależność H2 jest dostępna w ścieżce dostępu klas w trakcie działania aplikacji, wówczas Spring Boot w kontekście aplikacji Springa automatycznie tworzy odpowiedni komponent bean `DataSource`. Ten komponent następnie stosuje skrypty SQL `schema.sql` i `data.sql`.

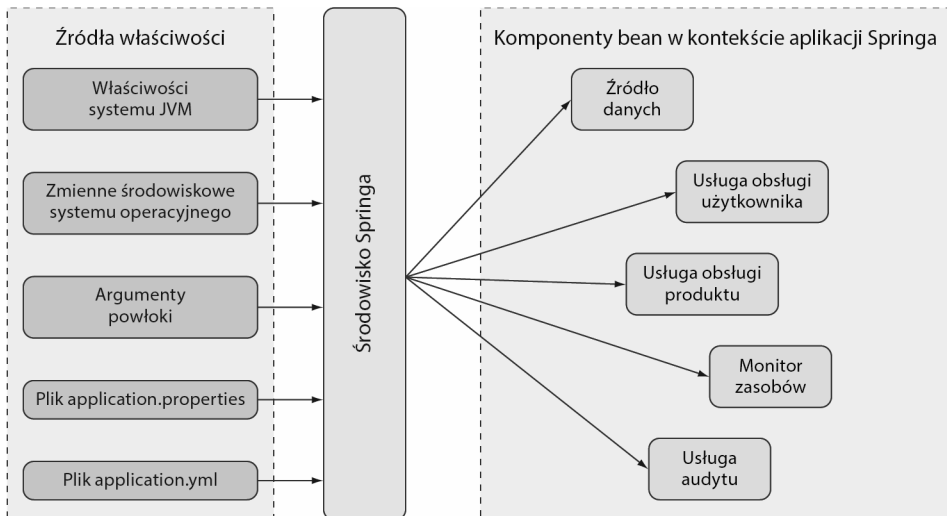
Co w sytuacji, jeśli tym skryptom SQL chcesz nadać inne nazwy lub jeśli chcesz skorzystać z więcej niż tylko dwóch skryptów SQL? W takim przypadku do gry wchodzi właściwości konfiguracyjne. Jednak zanim przystąpisz do ich używania, najpierw powinieneś się dowiedzieć, skąd one pochodzą.

5.1.1. Poznajemy abstrakcję środowiska Springa

Abstrakcja środowiska Springa to miejsce dla wszystkich właściwości konfiguracyjnych. Jest to źródło właściwości, więc jeśli będą one wymagane przez komponent bean, wówczas mogą pochodzić z samego Springa. Środowisko Springa zawiera wiele źródeł właściwości, m.in.:

- właściwości systemu JVM,
- zmienne środowiskowe systemu operacyjnego,
- argumenty powłoki,
- pliki konfiguracyjne aplikacji.

Następnie te właściwości są agregowane w pojedynczym źródle, z którego mogą być wstrzykiwane do komponentów bean. Na rysunku 5.1 pokazałem, jak właściwości pochodzące z różnych źródeł przechodzą przez abstrakcję środowiska Springa i trafiają do komponentów bean.



Rysunek 5.1. Środowisko Springa pobiera właściwości z różnych źródeł, a następnie udostępnia je komponentom bean w kontekście aplikacji Springa

Komponenty bean konfigurowane automatycznie przez Spring Boota zostają skonfigurowane za pomocą właściwości pobranych ze środowiska Springa. Oto prosty przykład. Przyjmując założenie, że chcesz, aby serwet kontenera nasłuchiwał żądań na porcie innym niż domyślny 8080. Wymaga to podania innego numeru portu we właściwości `server.port` zdefiniowanej w pliku `src/main/resources/application.properties`:

```
server.port=9090
```

Podczas przypisywania wartości właściwościom preferuję używanie formatu YAML. Dlatego też do zdefiniowania wartości `server.port` można zamiast pliku *application.properties* użyć pliku *src/main/resources/application.yml*:

```
server:
  port: 9090
```

Jeżeli wolisz zewnętrzne konfigurowanie właściwości, wówczas numer portu możesz zdefiniować podczas uruchamiania aplikacji, używając do tego argumentu powłoki:

```
$ java -jar tacocloud-0.0.5-SNAPSHOT.jar --server.port=9090
```

Natomiast jeśli chcesz, aby aplikacja zawsze nasłuchiwała na określonym porcie, możesz go zdefiniować za pomocą zmiennej środowiskowej systemu operacyjnego:

```
$ export SERVER_PORT=9090
```

Zwróć uwagę na to, że podczas definiowania właściwości jako zmiennych środowiskowych konwencja nadawania nazw jest nieco inna i uwzględnia ograniczenia dla nazw zmiennych środowiskowych narzucane przez system operacyjny. To całkowicie zrozumiałe. Spring potrafi sobie z tym poradzić i bez problemów interpretuje `SERVER_PORT` jako `server.port`.

Jak wcześniej wspomniałem, istnieje wiele sposobów definiowania właściwości konfiguracyjnych. W rozdziale 14. poznasz jeszcze inny sposób na skonfigurowanie właściwości w scentralizowanym serwerze konfiguracyjnym. Do dyspozycji masz setki właściwości konfiguracyjnych, które można wykorzystać podczas dostosowywania do własnych potrzeb sposobu zachowania komponentów bean w Springu. Kilka już poznałeś: `server.port` w tym rozdziale oraz `security.user.name` i `security.user.password` w poprzednim.

Nie mam możliwości przedstawienia w tym rozdziale wszystkich dostępnych właściwości konfiguracyjnych. Ograniczę się do wymienienia paru najużyteczniejszych, które będziesz najczęściej spotykać. Na początek przedstawię kilka właściwości pozwalających na dostosowanie do własnych potrzeb automatycznie skonfigurowanego źródła danych.

5.1.2. Konfigurowanie źródła danych

W obecnej postaci aplikacja Taco Cloud nadal jest nieukończona. Pozostało jeszcze sporo do zrobienia, zanim stanie się gotowa do wdrożenia. Dlatego też osadzona baza danych H2 używana obecnie jako źródło danych doskonale sprawdza się w tej roli, przynajmniej na razie. Jednak podczas wdrażania aplikacji w środowisku produkcyjnym prawdopodobnie będziesz chciał rozważyć użycie bardziej trwałego rozwiązania opartego na bazie danych.

Wprawdzie istnieje możliwość wyraźnego skonfigurowania własnego komponentu bean `DataSource`, jednak najczęściej jest to niepotrzebne. Zamiast tego prostszym rozwiązaniem jest skonfigurowanie adresu URL i danych uwierzytelniających do bazy danych, wykorzystując do tego właściwości konfiguracyjne. Na przykład jeśli zaczniesz używać bazy danych MySQL, być może do pliku *application.yml* będziesz chciał dodać następujące właściwości:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacodb
    password: tacopassword
```

Wprowadzie do pliku specyfikacji kompilacji należy dodać odpowiedni sterownik JDBC, ale zwykle nie trzeba określać klasy tego sterownika — Spring Boot może ją ustalić na podstawie struktury adresu URL bazy danych. Jeżeli pojawi się problem, można spróbować ustawić wartość właściwości `spring.datasource.driver-class-name`, jak pokazałem w kolejnym fragmencie kodu.

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacodb
    password: tacopassword
    driver-class-name: com.mysql.jdbc.Driver
```

Spring Boot używa tego połączenia danych podczas konfiguracji automatycznej komponentu bean `DataSource`. Jeżeli w ścieżce dostępu klas znajduje się pula połączeń Tomcat JDBC, to komponent bean z niej skorzysta. W przeciwnym razie Spring Boot (po odszukaniu) użyje którejś z innych dostępnych w ścieżce dostępu klas implementacji puli połączeń:

- HikariCP,
- Commons DBCP 2.

Choć są to jedyne pule połączeń dostępne za pomocą konfiguracji automatycznej, zawsze możesz wyraźnie skonfigurować komponent bean `DataSource` i zdecydować się na użycie dowolnie wybranej implementacji puli połączeń.

Wcześniej w rozdziale wspomniałem o możliwości wskazania skryptów inicjalizacyjnych bazy danych przeznaczonych do wykonania podczas uruchamiania aplikacji. W takim przypadku użyteczne okazują się właściwości `spring.datasource.schema` i `spring.datasource.data`.

```
spring:
  datasource:
    schema:
      - order-schema.sql
      - ingredient-schema.sql
      - taco-schema.sql
      - user-schema.sql
    data:
      - ingredients.sql
```

Możliwe, że wyraźna konfiguracja źródła danych nie jest w Twoim stylu. Zamiast tego może preferujesz skonfigurowanie źródła danych w JNDI, skąd zostanie pobrane przez Springa. W takim przypadku do przygotowania źródła danych możesz skorzystać z właściwości `spring.datasource.jndi-name`.

```
spring:
  datasource:
    jndi-name: java:/comp/env/jdbc/tacoCloudDS
```

Jeżeli zdefiniujesz właściwość `spring.datasource.jndi-name`, pozostałe właściwości źródła danych (o ile istnieją) zostaną zignorowane.

5.1.3. Konfigurowanie serwera osadzonego

Wcześniej dowiedziałeś się, jak zdefiniować port serwletu kontenera za pomocą właściwości `server.port`. Natomiast nie pokazałem tego, co się stanie po przypisaniu tej właściwości wartości 0.

```
server:
  port: 0
```

Wprawdzie wyraźnie przypisałeś wartość 0 właściwości `server.port`, mimo to serwer nie będzie nasłuchiwał na porcie o tym numerze. Zamiast tego skorzysta z portu o dowolnie wybranym numerze. Takie rozwiązanie jest użyteczne podczas wykonywania zautomatyzowanych testów integracji, co ma zagwarantować, że jednocześnie wykonywane testy nie będą kolidowały ze sobą w na stałe zdefiniowanym numerze portu. Jak zobaczysz w rozdziale 13., takie rozwiązanie okazuje się użyteczne także wtedy, gdy nie przejmujesz się numerem portu z powodu istnienia mikrousługi wyszukującej go w rejestrze usługi.

Jednak serwer to znacznie więcej niż tylko numer portu, na którym nasłuchuje żądań. Jednym z najczęściej wykonywanych zadań związanych z kontenerem jest zdefiniowanie obsługi żądań HTTPS. Wymaga ono przede wszystkim utworzenia magazynu kluczy za pomocą działającego z poziomu powłoki narzędzia JDK o nazwie `keytool`.

```
$ keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

To narzędzie wyświetli wiele pytań dotyczących Ciebie i organizacji, do której należysz — większość tych pytań jest bez znaczenia. Musisz natomiast koniecznie zapamiętać podane hasło. W celu zachowania prostoty przykładu zdecydowałem się na użycie hasła `letmein`.

Kolejnym krokiem jest zdefiniowanie kilku właściwości włączających HTTPS w osadzonym serwerze. Wszystkie można podać w powłoce, choć to szalenie niewygodne rozwiązanie. Zamiast tego możesz je umieścić w pliku *application.properties* lub *application.yml*, np.:

```
server:
  port: 8443
  ssl:
    key-store: file:///ścieżka/dostępu/do/pliku/mykeys.jks
    key-store-password: letmein
    key-password: letmein
```

W omawianym przykładzie właściwość `server.port` ma przypisaną wartość 8443, czyli często stosowany numer portu dla programistycznych serwerów HTTPS. Właściwość `server.ssl.key-store` powinna wskazywać ścieżkę dostępu do pliku będącego magazynem kluczy. Tutaj jest to adres URL `file://` wskazujący plik do wczytania z systemu plików. Jeżeli będzie się on znajdował w pliku aplikacji JAR, wówczas aby się do niego

odwołać, trzeba będzie użyć adresu URL `classpath:`. Właściwości `server.ssl.key-store-password` i `server.ssl.key-password` mają przypisane hasło, które zostało zdefiniowane podczas tworzenia magazynu kluczy.

Po zdefiniowaniu tych właściwości aplikacja powinna nasłuchiwać żądań HTTPS na porcie 8443. W zależności od używanej przeglądarki WWW możesz otrzymać ostrzeżenie, że serwer nie potrafi zweryfikować tożsamości. Jednak nie powinieneś się tym przejmować podczas testowania aplikacji w komputerze lokalnym.

5.1.4. Konfigurowanie rejestrowania danych

Większość aplikacji zapewnia pewną formę rejestrowania danych. Nawet jeśli aplikacja bezpośrednio nie rejestruje żadnych danych, używane przez nią biblioteki zdecydowanie będą wykazywały pewną aktywność pod tym względem.

Domyślnie Spring Boot konfiguruje rejestrowanie danych za pomocą Logback (<https://logback.qos.ch/>), aby generować w konsoli komunikaty na poziomie INFO. Prawdopodobnie widziałeś już wiele komunikatów tego rodzaju w dziennikach zdarzeń po uruchomieniu aplikacji Taco Cloud i innych przykładów.

Jeżeli chcesz zachować pełną kontrolę nad konfiguracją rejestrowania danych, możesz utworzyć plik `logback.xml` w katalogu głównym ścieżki dostępu klas (`src/main/resources`). Spójrz na przykład prostego pliku `logback.xml`, z którego możesz skorzystać.

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </pattern>
    </encoder>
  </appender>
  <logger name="root" level="INFO"/>
  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

Pomijając wzorzec użyty do rejestrowania danych, ta konfiguracja Logback jest mniej więcej odpowiednikiem domyślnego zachowania aplikacji, gdy nie istnieje plik `logback.xml`. Jednak przeprowadzając edycję tego pliku, można zyskać pełną kontrolę nad plikami zdarzeń aplikacji.

UWAGA Specyfika tego, co można umieścić w pliku `logback.xml`, wykracza poza zakres tematyczny tej książki. Więcej informacji na ten temat znajdziesz w dokumentacji Logback.

Najczęściej wprowadzane zmiany w konfiguracji rejestrowania danych dotyczą poziomu rejestrowania informacji i być może położenia pliku dziennika zdarzeń. Dzięki właściwościom konfiguracyjnym Spring Boota te zmiany można wprowadzić bez konieczności tworzenia pliku `logback.xml`.

Aby zdefiniować poziom rejestrowania danych, należy utworzyć właściwości z prefiksem `logging.level` i nazwą mechanizmu rejestrowania danych, dla którego dana właściwość jest przeznaczona. Przyjmuję założenie, że chcesz dla rejestrowania da-

nych zdefiniować poziom WARN, a Spring Security rejestruje dane na poziomie DEBUG. Przedstawiona tutaj konfiguracja w pliku *application.yml* stanowi rozwiązanie w omawianej sytuacji.

```
logging:
  level:
    root: WARN
    org:
      springframework:
        security: DEBUG
```

Opcjonalnie nazwę pakietu Spring Security można zwinąć do pojedynczego wiersza, co powinno ułatwić odczyt.

```
logging:
  level:
    root: WARN
    org.springframework.security: DEBUG
```

Teraz przyjmuję założenie, że dane mają zostać zapisywane w pliku *TacoCloud.log* w katalogu */var/logs/*. Można to zdefiniować za pomocą właściwości *logging.path* i *logging.file*, jak pokazałem w kolejnym fragmencie kodu.

```
logging:
  path: /var/logs/
  file: TacoCloud.log
  level:
    root: WARN
    org:
      springframework:
        security: DEBUG
```

Jeżeli aplikacja ma uprawnienia zapisu w katalogu */var/logs/*, informacje mechanizmu rejestrowania danych zostaną umieszczone w pliku */var/logs/TacoCloud.log*. Domyślnie rotacja pliku odbywa się po osiągnięciu przez niego wielkości 10 MB.

5.1.5. Używanie wartości właściwości specjalnych

Podczas przypisywania wartości właściwościom nie jesteś ograniczony jedynie do deklarowania ich wartości jako na stałe zdefiniowanych w postaci ciągu tekstowego lub liczby. Zamiast tego te wartości mogą pochodzić z innych właściwości konfiguracyjnych.

Przyjmuję założenie, że (z jakiegokolwiek powodu) chcesz właściwości o nazwie *greeting.welcome* przypisać wartość innej właściwości, *spring.application.name*. Jest to możliwe dzięki użyciu miejsca zarezerwowanego *\${}* podczas definiowania właściwości *greeting.welcome*.

```
greeting:
  welcome: ${spring.application.name}
```

Istnieje również możliwość osadzenia tego miejsca zarezerwowanego w innym tekście.

```
greeting:
  welcome: Używasz aplikacji ${spring.application.name}.
```

Jak widać, konfigurowanie komponentów Springa za pomocą właściwości konfiguracyjnych niezwykle ułatwia wstrzykiwanie wartości do właściwości tych komponentów i tym samym pozwala na dostosowanie do własnych potrzeb konfiguracji automatycznej. Właściwości konfiguracyjne nie są przeznaczone tylko dla komponentów bean tworzonych przez Springa. Wkładając niewiele wysiłku, można we własnych komponentach bean wykorzystać możliwości oferowane przez właściwości konfiguracyjne. W następnym podrozdziale zobaczysz, jak to zrobić.

5.2. Tworzenie własnych właściwości konfiguracyjnych

Wezśniej dowiedziałeś się, że właściwości konfiguracyjne to po prostu zwykle właściwości komponentu bean, które zostały zaprojektowane do akceptowania konfiguracji pochodzącej z abstrakcji środowiska Springa. Natomiast nie przedstawiłem jeszcze tego, jak te komponenty bean są projektowane do wykorzystania wspomnianych konfiguracji.

Aby zapewnić obsługę wstrzykiwania właściwości konfiguracyjnych, Spring Boot dostarcza adnotację `@ConfigurationProperties`. Po jej zastosowaniu dla dowolnego komponentu bean właściwości tego komponentu mogą być wstrzykiwane z właściwości środowiska Springa.

Teraz pokażę sposób działania adnotacji `@ConfigurationProperties`. Przyjmuję założenie, że do kontrolera `OrderController` dodałeś przedstawioną tutaj metodę odpowiedzialną za wyświetlenie wcześniejszych zamówień uwierzytelnionego użytkownika.

```
@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

    model.addAttribute("orders",
        orderRepo.findByUserOrderByPlacedAtDesc(user));

    return "orderList";
}
```

Poza tym w `OrderRepository` dodałeś niezbędną metodę `findByUser()`.

```
List<Order> findByUserOrderByPlacedAtDesc(User user);
```

Zwróć uwagę na to, że ta metoda repozytorium zawiera klauzulę `OrderByPlacedAtDesc`. Człon `OrderBy` określa właściwość, według której będą uporządkowane wyniki — w omawianym przykładzie jest to `placedAt`. Słowo `Desc` na końcu oznacza ułożenie w kolejności malejącej. Dlatego też zwrócona zostanie lista zamówień ułożonych w kolejności od najnowszego do najstarszego.

Ta metoda kontrolera może być użyteczna po złożeniu przez użytkownika wielu zamówień. Okazuje się jednak nieporęczna dla wielu gorliwych koneserów taco. Kilka zamówień wyświetlonych w przeglądarce WWW będzie miało pewną wartość, ale niekończąca się lista setek zamówień to po prostu kłopot. Przyjmuję założenie, że chcesz ograniczyć liczbę wyświetlanych zamówień do 20 najnowszych. W takim przypadku możesz zmienić metodę `ordersForUser()` w następujący sposób:

```

@GetMapping
public String ordersForUser(@AuthenticationPrincipal User user, Model model) {
    Pageable pageable = PageRequest.of(0, 20);
    model.addAttribute("orders",
        orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

    return "orderList";
}

```

I oczywiście wprowadzić odpowiednią zmianę w OrderRepository:

```
List<Order> findByUserOrderByPlacedAtDesc(User user, Pageable pageable);
```

Tutaj po prostu zmieniłem sygnaturę metody `findByUserOrderByPlacedAtDesc()`, aby akceptowała parametr `Pageable`. Ten parametr to oferowany przez Spring Data sposób na wybór podzbioru wyników przez określenie numeru strony i jej wielkości. W metodzie kontrolera `ordersForUser()` następuje utworzenie obiektu `PageRequest` implementującego `Pageable` z żądaniem pierwszej strony (o numerze zero) o wielkości 20 zawierającej 20 ostatnich zamówień złożonych przez użytkownika.

Wprawdzie takie rozwiązanie działa fantastycznie, ale może pozostawiać pewien niesmak ze względu na zdefiniowaną na stałe w kodzie wielkość strony. Co bowiem zrobić w sytuacji, gdy później uznasz, że 20 to jednak zbyt duża liczba zamówień na stronie, i będziesz chciał zmienić tę wartość na np. 10? Skoro wartość została na stałe zdefiniowana w kodzie, trzeba zmodyfikować kod, a następnie ponownie skompilować i wdrożyć aplikację.

Zamiast na stałe definiować w kodzie wielkość strony, lepiej określić ją za pomocą niestandardowej właściwości konfiguracyjnej. Przede wszystkim trzeba dodać do `OrderController` nową właściwość o nazwie `pageSize` i oznaczyć `OrderController` adnotacją `@ConfigurationProperties`, jak pokazałem na listingu 5.1.

Listing 5.1. Włączenie właściwości konfiguracyjnych w klasie `OrderController`

```

@Controller
@RequestMapping("/orders")
@SessionAttributes("order")
@ConfigurationProperties(prefix="taco.orders")
public class OrderController {
    private int pageSize = 20;

    public void setPageSize(int pageSize) {
        this.pageSize = pageSize;
    }

    ...

    @GetMapping
    public String ordersForUser(@AuthenticationPrincipal User user, Model model) {
        Pageable pageable = PageRequest.of(0, pageSize);
        model.addAttribute("orders",
            orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

        return "orderList";
    }
}

```

Najważniejsza zmiana wprowadzona na listingu 5.1 polega na dodaniu adnotacji `@ConfigurationProperties`. Jej atrybut `prefix` ma wartość `taco.orders`, co oznacza, że podczas definiowania właściwości `pageSize` konieczne jest użycie właściwości konfiguracyjnej o nazwie `taco.orders.pageSize`.

Wartością domyślną nowej właściwości `pageSize` jest 20. Bardzo łatwo można ją zmienić przez przypisanie innej wartości `taco.orders.pageSize`. Oto, jak przypisać wartość tej właściwości w pliku *application.yml*:

```
taco:
  orders:
    pageSize: 10
```

Jeżeli musisz wprowadzić szybką zmianę do aplikacji działającej w środowisku produkcyjnym, możesz to zrobić bez konieczności jej ponownej kompilacji i wdrażania. Wystarczy przypisać wartość właściwości `taco.orders.pageSize` z użyciem zmiennej środowiskowej.

```
$ export TACO_ORDERS_PAGESIZE=10
```

Zmianę liczby ostatnio złożonych zamówień można wprowadzić na różne sposoby. W kolejnej sekcji dowiesz się, jak zmieniać dane konfiguracyjne w komponentach przechowujących właściwości.

5.2.1. Definiowanie komponentów przechowujących właściwości konfiguracyjne

Nic nie wskazuje, że adnotacja `@ConfigurationProperties` musi być użyta względem kontrolera lub innego rodzaju komponentu `bean`. Ta adnotacja jest często stosowana dla komponentów `bean`, których jedynym zadaniem w aplikacji jest przechowywanie danych konfiguracyjnych. Dzięki temu szczegóły związane z konfiguracją będą przechowywane poza kontrolerami i innymi klasami aplikacji. Ponadto takie rozwiązanie znacznie ułatwia współdzielenie właściwości konfiguracyjnych między różnymi komponentami `bean`, które mogą potrzebować tych informacji.

W przypadku właściwości `pageSize` w `OrderController` tę właściwość można wyodrębnić do oddzielnej klasy. Na listingu 5.2 przedstawiłem przykładowy sposób użycia klasy `OrderProps`.

Listing 5.2. Wyodrębnienie właściwości `pageSize` do klasy przechowującej właściwości

```
package tacos.web;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;
import lombok.Data;

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
public class OrderProps {
    private int pageSize = 20;
}
```

Podobnie jak w `OrderController`, także tutaj właściwość `pageSize` ma wartość domyślną 20. Klasa `OrderProps` została oznaczona adnotacją `@ConfigurationProperties`, której atrybut `prefix` ma wartość `taco.orders`.

Klasa przechowująca właściwości została oznaczona również adnotacją `@Component`, aby podczas operacji skanowania komponentu następowało jej automatyczne odkrycie i utworzenie egzemplarza klasy jako komponentu bean w kontekście aplikacji Springa. To bardzo ważne, ponieważ następnym krokiem jest wstrzyknięcie komponentu `OrderProps` do `OrderController`.

Nie ma niczego specjalnego w komponentach przechowujących właściwości konfiguracyjne. Są to po prostu komponenty bean, których właściwości zostały wstrzyknięte ze środowiska Springa. Te komponenty mogą być wstrzykiwane do innych komponentów wymagających przechowywanych w nich właściwości. W przypadku `OrderController` można usunąć z klasy właściwość `pageSize` i zamiast tego wstrzyknąć komponent bean `OrderProps`, jak pokazałem w kolejnym fragmencie kodu.

```
@Controller
@RequestMapping("/orders")
@SessionAttributes("order")
public class OrderController {
    private OrderRepository orderRepo;

    private OrderProps props;

    public OrderController(OrderRepository orderRepo,
        OrderProps props) {
        this.orderRepo = orderRepo;
        this.props = props;
    }

    ...

    @GetMapping
    public String ordersForUser(@AuthenticationPrincipal User user, Model model) {
        Pageable pageable = PageRequest.of(0, props.getPageSize());
        model.addAttribute("orders",
            orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

        return "orderList";
    }

    ...
}
```

Teraz klasa `OrderController` nie jest już odpowiedzialna za obsługę własnych właściwości konfiguracyjnych. Dzięki temu kod tej klasy jest bardziej elegancki, a właściwości zdefiniowane w `OrderProps` mogą być używane także w innych wymagających ich komponentach bean. Co więcej, dotyczące zamówień właściwości konfiguracyjne zostają zebrane w jednym miejscu: klasie `OrderProps`. Jeżeli zachodzi potrzeba dodania, usunięcia, zmiany nazwy lub przeprowadzenia innej modyfikacji właściwości, wówczas trzeba to zrobić jedynie w klasie `OrderProps`.

Przyjmuję założenie, że właściwość `pageSize` jest używana w wielu innych komponentach bean i zdecydowałeś o zastosowaniu pewnej weryfikacji danych, aby ograni-

czyć wartość właściwości do przedziału od 5 do 25. W przypadku braku komponentu bean przechowującego tę właściwość musiałbyś dodać odpowiednie adnotacje do klasy `OrderController`, właściwości `pageSize` i wszystkich pozostałych klas używających tej właściwości. Jednak dzięki wyodrębnieniu `pageSize` do `OrderProps` zmianę trzeba wprowadzić tylko w klasie `OrderProps`.

```
package tacos.web;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;
import org.springframework.validation.annotation.Validated;

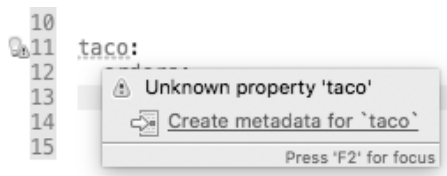
import lombok.Data;

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
@Validated
public class OrderProps {
    @Min(value=5, message="Wartość musi mieścić się w przedziale od 5 do 25.")
    @Max(value=25, message="Wartość musi mieścić się w przedziale od 5 do 25.")
    private int pageSize = 20;
}
//end::validated//
```

Wprawdzie adnotacje `@Validated`, `@Min` i `@Max` można łatwo zastosować w klasie `OrderController` (i wielu innych komponentach bean możliwych do wstrzyknięcia z `OrderProps`), ale spowodowałoby to zaśmieszenie kodu kontrolera. Dzięki komponentowi bean przechowującemu właściwości konfiguracyjne w jednym miejscu korzystające z nich klasy pozostają względnie czyste.

5.2.2. Deklarowanie metadanych właściwości konfiguracyjnej

W zależności od używanego środowiska IDE możesz zauważyć, że wpis `taco.orders.pageSize` w pliku `application.yml` (lub `application.properties`) powoduje wyświetlenie komunikatu ostrzeżenia informującego o nieznanym właściwości `taco`. Takie ostrzeżenie pojawia się ze względu na brak metadanych związanych z utworzoną wcześniej właściwością konfiguracyjną. Na rysunku 5.2 pokazałem przykład takiego komunikatu wyświetlanego po umieszczeniu kursora myszy nad słowem `taco` w środowisku IDE Spring Tool Suite.



Rysunek 5.2. Komunikat ostrzeżenia informujący o brakujących metadanych właściwości konfiguracyjnej

Metadane właściwości konfiguracyjnej są całkowicie opcjonalne i niewymagane do jej prawidłowego działania. Jednak te metadane okazują się użyteczne podczas dostarczania minimalnej dokumentacji dotyczącej właściwości konfiguracyjnej, zwłaszcza w środowisku IDE.

Na przykład po umieszczeniu kursora myszy nad właściwością `spring.security.user.password` zobaczysz minidokumentację, którą pokazałem na rysunku 5.3. Wprawdzie wyświetlona dokumentacja jest minimalna, ale może się okazać wystarczająca do zrozumienia przeznaczenia danej właściwości i sposobu jej użycia.



Rysunek 5.3. Dokumentacja wyświetlona po umieszczeniu kursora myszy nad nazwą właściwości w Spring Tool Suite

Aby pomóc tym, którzy mogą używać zdefiniowanych przez Ciebie właściwości konfiguracyjnych — możesz to być nawet Ty sam w przyszłości — dobrym rozwiązaniem jest utworzenie pewnych metadanych dla tych właściwości. W ten sposób przynajmniej pozbędziesz się irytujących żółtych ikon ostrzegawczych w środowisku IDE.

Dodawanie metadanych dla niestandardowych właściwości konfiguracyjnych trzeba zacząć od utworzenia w katalogu *META-INF* (np. `src/main/resources/META-INF`) pliku o nazwie `additional-spring-configuration-metadata.json`.

SZYBKIE ROZWIĄZANIE PROBLEMU DOTYCZĄCEGO BRAKUJĄCYCH METADANYCH

Jeżeli używasz środowiska IDE Spring Tool Suite, masz opcję pozwalającą na szybkie rozwiązanie problemu dotyczącego brakujących metadanych właściwości. Umieść kursor w wierszu zawierającym ostrzeżenie o brakujących metadanych, a następnie wyświetl okno dialogowe *Quick Fix* (rysunek 5.4) przez naciśnięcie klawiszy `Cmd+I` (macOS) lub `Ctrl+I` (Windows i Linux).

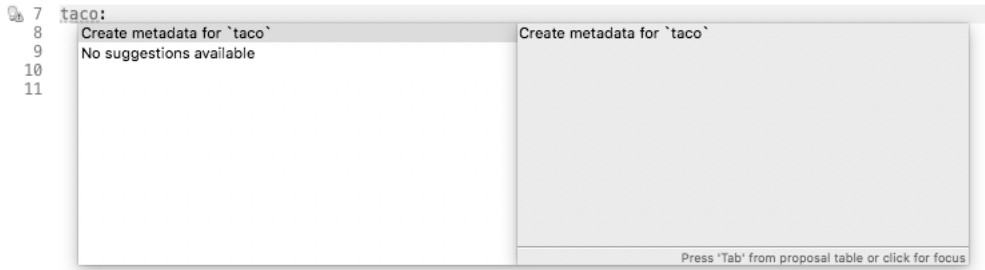
Teraz wybierz opcję *Create metadata for...* i dodaj pewne metadane dla właściwości w utworzonym wcześniej pliku `additional-spring-configuration-metadata.json`.

W przypadku właściwości `taco.orders.pageSize` metadane mogą mieć postać następującego kodu w formacie JSON:

```

{
  "properties": [
    {
      "name": "taco.orders.page-size",
      "type": "java.lang.String",

```

Rysunek 5.4. Utworzenie metadanych właściwości konfiguracyjnej za pomocą okna dialogowego Quick Fix w Spring Tool Suite

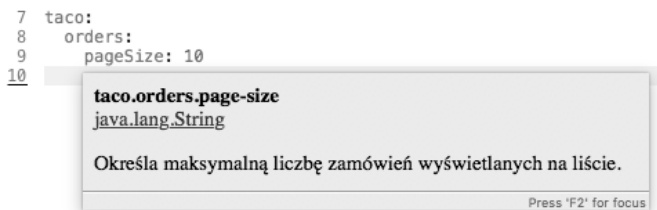
```

    "description":
      "Określa maksymalną liczbę zamówień wyświetlanych na liście."
  }
]
}

```

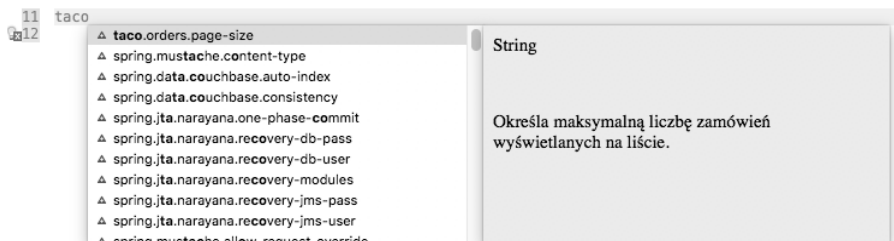
Zwróć uwagę na to, że nazwą właściwości używaną w metadanych jest `taco.orders.page-size`. Elastyczna konwencja nazw w Spring Boocie pozwala na używanie różnych wariantów nazw właściwości, więc np. `taco.orders.page-size` jest odpowiednikiem dla `taco.orders.pageSize`.

Po zdefiniowaniu metadanych ikona i komunikat ostrzeżenia powinny zniknąć. Co więcej, po umieszczeniu kursora myszy nad właściwością `taco.orders.pageSize` zobaczysz zdefiniowany wcześniej opis, jak pokazałem na rysunku 5.5.



Rysunek 5.5. Minidokumentacja dla niestandardowej właściwości konfiguracyjnej

Natomiast na rysunku 5.6 pokazałem, że środowisko IDE pomaga, oferując dla niestandardowych właściwości konfiguracyjnych możliwość ich automatycznego uzupełniania, podobnie jak w przypadku właściwości konfiguracyjnych dostarczanych przez Springa.



Rysunek 5.6. Metadane właściwości konfiguracyjnej pozwalają na użycie automatycznego uzupełniania kodu

Jak widać, właściwości konfiguracyjne są użyteczne podczas dostosowywania do własnych potrzeb zarówno automatycznie skonfigurowanych komponentów, jak i szczegółów związanych z obiektami wstrzykiwanymi do komponentów bean aplikacji. Co jednak zrobić w sytuacji, gdy zachodzi potrzeba skonfigurowania innych właściwości dla odmiennych środowisk wdrożenia? W następnym podrozdziale dowiesz się, jak wykorzystać profile Springa do zdefiniowania konfiguracji przeznaczonej dla konkretnego środowiska.

5.3. Konfigurowanie za pomocą profili

Gdy aplikacja jest wdrażana w różnych środowiskach, z reguły pewne aspekty konfiguracji będą w nich inne. Na przykład szczegóły dotyczące nawiązania połączenia z bazą danych prawdopodobnie będą różne w środowiskach programistycznym i produkcyjnym. Jednym ze sposobów pozwalających na unikatową konfigurację właściwości dla odmiennych środowisk jest wykorzystanie zmiennych środowiskowych dostarczających właściwości konfiguracyjne zamiast ich definiowania w plikach *application.properties* lub *application.yml*.

W trakcie pracy nad aplikacją możesz wykorzystać automatycznie skonfigurowaną bazę danych H2. Jednak w produkcji właściwości konfiguracyjne bazy danych mogą mieć postać zmiennych środowiskowych, jak pokazałem w kolejnym fragmencie kodu.

```
% export SPRING_DATASOURCE_URL=jdbc:mysql://localhost/tacocloud
% export SPRING_DATASOURCE_USERNAME=tacouser
% export SPRING_DATASOURCE_PASSWORD=tacopassword
```

Wprawdzie takie rozwiązanie działa, ale konieczność określania dwóch lub więcej właściwości konfiguracyjnych jako zmiennych środowiskowych szybko staje się uciążliwa. Ponadto nie istnieje dobry sposób na monitorowanie zmian wprowadzonych w takich zmiennych lub łatwego ich wycofywania, jeśli zostały błędnie zdefiniowane.

Dlatego też wolę pracować z profilami Springa. Profil to nic innego jak konfiguracja warunkowa, w której różne komponenty bean, klasy i właściwości konfiguracyjne są stosowane lub ignorowane na podstawie profilu aktywnego podczas uruchamiania aplikacji.

Przyjmuję założenie, że podczas tworzenia aplikacji i jej debugowania jest używana osadzona baza danych H2 i chcesz dla aplikacji Taco Cloud rejestrować dane na poziomie DEBUG. Jednak w środowisku produkcyjnym ma zostać użyta zewnętrzna baza danych MySQL, a dane mają być rejestrowane na poziomie WARN. W środowisku programistycznym można bardzo łatwo zrezygnować z definiowania jakichkolwiek właściwości źródła danych i tym samym mieć do dyspozycji automatycznie skonfigurowaną bazę H2. Aby rejestrować dane na poziomie DEBUG, właściwości `logging.level.↳tacos` w pakiecie `tacos` trzeba przypisać wartość `DEBUG`.

```
logging:
  level:
    tacos: DEBUG
```

To dokładnie takie wymaganie, jakie powinno być spełnione w środowisku programistycznym. Jednak po wdrożeniu aplikacji w produkcji i niewprowadzeniu żadnych zmian w pliku *application.yml* nadal będziesz mógł rejestrować dane na poziomie DEBUG i obsługiwać osadzoną bazę danych H2. Musisz więc zdefiniować profil z właściwościami odpowiednimi dla środowiska produkcyjnego.

5.3.1. Definiowanie właściwości dla konkretnego profilu

Jednym ze sposobów na utworzenie właściwości dla konkretnego profilu jest przygotowanie kolejnego pliku YAML-a lub właściwości z ustawieniami dla np. środowiska produkcyjnego. Nazwa pliku powinna stosować następującą konwencję: *application-{nazwa profilu}.yml* lub *application-{nazwa profilu}.properties*. Następnie w nowym pliku można zdefiniować ustawienia konfiguracyjne odpowiednie dla tego profilu. Na przykład możesz utworzyć plik o nazwie *application-prod.yml* zawierający przedstawione tutaj właściwości.

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword
  logging:
    level:
      tacos: WARN
```

Inny sposób zdefiniowania właściwości dla konkretnego profilu działa tylko w plikach konfiguracyjnych typu YAML. Polega na umieszczeniu właściwości charakterystycznych dla profilu w oddzielnej sekcji rozpoczynającej się od trzech znaków -. Jeżeli ustawienia dla środowiska produkcyjnego umieścisz w pliku *application.yml*, jego zawartość może przedstawiać się następująco:

```
logging:
  level:
    tacos: DEBUG

---
spring:
  profiles: prod

  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword

  logging:
    level:
      tacos: WARN
```

Jak widać, znaki --- dzielą profil na dwie sekcje. Druga sekcja zawiera wartość dla `spring.profiles` wskazującą, że znajdujące się dalej właściwości są przeznaczone dla profilu `prod`. W sekcji pierwszej nie została użyta wartość dla `spring.profiles`, dlatego też znajdujące się w niej właściwości są przeznaczone dla wszystkich profili i używane

domyślnie, jeśli aktywny profil nie ma aktualnie zdefiniowanej danej właściwości konfiguracyjnej.

Niezależnie od profilu aktywowanego podczas uruchamiania aplikacji, poziom rejestrowania danych dla pakietu `tacos` to `DEBUG`, na co wskazuje właściwość zdefiniowana w profilu domyślnym. Natomiast jeśli zostanie aktywowany profil `prod`, wartość właściwości `logging.level.tacos` będzie nadpisana i wyniesie `WARN`. Ponadto w przypadku aktywnego profilu `prod` właściwości źródła danych wskazują na używanie zewnętrznej bazy danych `MySQL`.

Istnieje możliwość zdefiniowania właściwości dla dowolnej liczby profili przez utworzenie oddzielnych plików `YAML`-a `application-{nazwa profilu}.yaml` lub właściwości `application-{nazwa profilu}.properties`. Jeżeli wolisz inne rozwiązanie, wpisz trzy znaki - w pliku `application.yaml` z kolejną właściwością `spring.profiles` i podaj nazwę profilu, a następnie dodaj wszystkie właściwości niezbędne dla danego profilu.

5.3.2. Aktywowanie profilu

Właściwości przeznaczone dla konkretnego profilu będą użyteczne dopiero wtedy, gdy dany profil stanie się aktywny. Mógłbyś w tym miejscu zapytać: jak się odbywa aktywowanie profilu? Aby profil stał się aktywny, trzeba go dołączyć na liście nazw profili przekazywanej właściwości `spring.profiles.active`. W pliku `application.yaml` można to zrobić następująco:

```
spring:
  profiles:
    active:
      - prod
```

To jednak prawdopodobnie najgorszy z możliwych sposobów na wybór aktywnego profilu. Jeżeli wybierzesz go w pliku `application.yaml`, dany profil stanie się domyślny i nie skorzystasz z zalety stosowania profili, jaką jest możliwość oddzielenia ustawień dla poszczególnych środowisk. Dlatego też zachęcam do wyboru aktywnego profilu za pomocą zmiennej środowiskowej. W środowisku produkcyjnym aktywny profil możesz wybrać za pomocą przedstawionego tutaj polecenia.

```
% export SPRING_PROFILES_ACTIVE=prod
```

Jeżeli którakolwiek z wdrożonych aplikacji w danym środowisku będzie miała aktywny profil `prod`, znajdujące się w nim właściwości konfiguracyjne będą miały pierwszeństwo przed tymi w profilu domyślnym.

W przypadku uruchamiania aplikacji za pomocą pliku wykonywalnego `JAR` aktywny profil można wskazać za pomocą argumentu powłoki, np.:

```
% java -jar taco-cloud.jar --spring.profiles.active=prod
```

Zwróć uwagę na to, że nazwa właściwości `spring.profiles.active` zawiera angielskie słowo *profile* w liczbie mnogiej: *profiles*. Daje to możliwość wskazania więcej niż tylko jednego aktywnego profilu. Bardzo często ma to postać rozdzielonej przecinkami listy definiowanej ze zmienną środowiskową.

```
% export SPRING_PROFILES_ACTIVE=prod,audit,ha
```

W przypadku pliku YAML-a lista jest definiowana, jak pokazałem w kolejnym fragmencie kodu.

```
spring:
  profiles:
    active:
      - prod
      - audit
      - ha
```

Warto w tym miejscu wspomnieć jeszcze o jednej kwestii. Gdy aplikacja Springa jest wdrażana w Cloud Foundry, automatycznie będzie aktywowany profil o nazwie `cloud`. Jeżeli Cloud Foundry to Twoje środowisko produkcyjne, właściwości przeznaczone do zastosowania w produkcji powinieneś umieścić w profilu `cloud`.

Jak się okazuje, profile są użyteczne nie tylko podczas warunkowego przypisywania wartości właściwościom konfiguracyjnym w aplikacji Springa. W następnej sekcji dowiesz się, jak zadeklarować komponenty bean przeznaczone dla aktywnego profilu.

5.3.3. Warunkowe tworzenie komponentu bean z profilami

Czasami użyteczne jest dostarczanie unikatowego zbioru komponentów bean dla poszczególnych profili. Standardowo każdy komponent bean zadeklarowany w klasie konfiguracyjnej Javy zostaje utworzony niezależnie od tego, który profil jest aktywny. W takim przypadku adnotacja `@Profile` może wskazać komponent bean jako odpowiedni tylko dla konkretnego profilu.

Na przykład w aplikacji Taco Cloud masz komponent bean `CommandLineRunner` używany do wczytania osadzonej bazy danych z danymi składników podczas uruchamiania aplikacji. Takie rozwiązanie doskonale się sprawdza w środowisku programistycznym, ale jest niepotrzebne (i niepożądane) po wdrożeniu aplikacji w produkcji. Aby uniemożliwić wczytywanie danych składników po każdym uruchomieniu aplikacji wdrożonej w środowisku produkcyjnym, metodę komponentu bean `CommandLineRunner` można oznaczyć adnotacją `@Profile`, jak pokazałem w kolejnym fragmencie kodu.

```
@Bean
@Profile("dev")
public CommandLineRunner dataLoader(IngredientRepository repo,
    UserRepository userRepo, PasswordEncoder encoder) {
    ...
}
```

Przyjmuję założenie, że komponent `CommandLineRunner` ma zostać utworzony tylko wtedy, gdy aktywny jest profil `dev` lub `qa`. W takim przypadku lista profili, dla których dany komponent powinien być utworzony, jest zdefiniowana następująco:

```
@Bean
@Profile({"dev", "qa"})
public CommandLineRunner dataLoader(IngredientRepository repo,
    UserRepository userRepo, PasswordEncoder encoder) {
    ...
}
```

Po wprowadzeniu tej zmiany dane składników będą wczytywane tylko wtedy, gdy aktywny jest profil `dev` lub `qa`. Oznacza to konieczność aktywowania profilu `dev` podczas uruchamiania aplikacji w środowisku programistycznym. Znacznie wygodniejsze rozwiązanie polega na tym, że komponent bean `CommandLineRunner` zawsze będzie tworzony, pod warunkiem że nie został aktywowany profil `prod`. W takim przypadku adnotację `@Profile` można zastosować nieco inaczej, jak pokazałem w kolejnym fragmencie kodu.

```
@Bean
@Profile("!prod")
public CommandLineRunner dataLoader(IngredientRepository repo,
    UserRepository userRepo, PasswordEncoder encoder) {
    ...
}
```

Tutaj znak wykrzyknika oznacza negację nazwy profilu. W efekcie komponent `CommandLineRunner` zostanie utworzony, jeśli aktywny profil jest inny niż `prod`.

Istnieje również możliwość użycia adnotacji `@Profile` dla całej klasy oznaczonej adnotacją `@Configuration`. Przyjmuję założenie, że chcesz wyodrębnić komponent bean `CommandLineRunner` do oddzielnej klasy konfiguracyjnej o nazwie `DevelopmentConfig`. W takim przypadku można ją oznaczyć adnotacją `@Profile`.

```
@Profile({"!prod", "!qa"})
@Configuration
public class DevelopmentConfig {
    @Bean
    public CommandLineRunner dataLoader(IngredientRepository repo,
        UserRepository userRepo, PasswordEncoder encoder) {
        ...
    }
}
```

Tutaj komponent bean `CommandLineRunner` (a także każdy inny komponent bean zdefiniowany w klasie `DevelopmentConfig`) będzie utworzony tylko wtedy, gdy aktywny profil jest inny niż `prod` lub `qa`.

Podsumowanie

- Komponent bean w Springu może być oznaczony adnotacją `@Configuration` `↳Properties`, aby w ten sposób umożliwić wstrzykiwanie wartości z jednego z wielu źródeł właściwości.
- Właściwości konfiguracyjne mogą być definiowane w argumentach powłoki, zmiennych środowiskowych, we właściwościach systemu JVM, w plikach właściwości, plikach YAML-a itd.
- Właściwości konfiguracyjne mogą być używane do nadpisywania ustawień zdefiniowanych przez konfigurację automatyczną; m.in. mogą określać adresy URL źródeł danych, a także poziomy rejestrowania danych.
- Profile w Springu można wykorzystać wraz ze źródłami właściwości do warunkowego definiowania właściwości konfiguracyjnych na podstawie aktywnego profilu.

Część II

Zintegrowany Spring

Rozdziały z drugiej części książki poruszają tematy pomagające w integrowaniu aplikacji Springa z innymi aplikacjami.

Rozdział 6. zawiera rozszerzenie materiału o Spring MVC przedstawionego w rozdziale 2. i pokazuje, jak utworzyć API REST w Springu. Nauczysz się definiować punkty końcowe REST w Spring MVC, włączać zasoby REST z hiperłączami, a także korzystać za pomocą Spring Data REST z automatycznie generowanych i opartych na repozytorium punktach końcowych REST. W rozdziale 7. zmienisz nieco perspektywę i zobaczysz, jak aplikacja Springa może konsumować API REST. W rozdziale 8. przedstawię komunikację asynchroniczną pozwalającą aplikacji Springa na wysyłanie i otrzymywanie komunikatów za pomocą brokerów JMS (ang. *java message service*), RabbitMQ i Kafka. Natomiast w rozdziale 9. przedstawię deklaratywną integrację aplikacji za pomocą projektu Spring Integration. Zapoznasz się z przetwarzaniem danych w czasie rzeczywistym, definiowaniem przepływu integracji, a także integrowaniem z systemami zewnętrznymi, np. pocztą elektroniczną i systemem plików.

Skorowidz

A

abstrakcja środowiska, 139
adapter kanału, 252
administrowanie, 459
adnotacje Spring MVC, 161
adres URI, 310
agregowanie strumieni, 418
aktywator usługi, 249
aktywowanie profilu, 154
AMQP, 215
analiza
 pliku specyfikacji
 kompilacji, 32
 struktury projektu, 30
 zmiennych środowiskowych, 467
Apache Kafka
 obsługa komunikatów, 225
API Initializr, 510
 parametry żądania, 511
API reaktywne, 293, 316
 konfigurowanie
 zabezpieczeń, 316
API REST, 197, 308
aplikacja internetowa, 51
asynchroniczne wysyłanie komunikatów, 201
asynchroniczny framework internetowy, 294
ataki typu CSRF, 131
automatyczne
 odświeżanie
 właściwości, 403
 przeglądarki, 44
 ponowne uruchomienie aplikacji, 44
awarie, 405
 monitorowanie, 412

B

bezpiecznik, 408
 zarządzanie wartością graniczną, 411
biblioteka
 Hystrix, 407
 agregowanie strumieni, 418
 panel kontrolny, 413
 pula wątków, 416
 Lombok, 82
 Traverson, 197
biblioteki szablonów widoku, 75
błędy weryfikacji danych, 73
brama, 251
broker Artemis, 203
buforowanie
 danych strumienia reaktywnego, 287
 szablonów, 44, 77

C

Cassandra, 325
 mapowanie typów domeny, 329
 modelowanie danych, 328
 tworzenie reaktywnego repozytorium, 335
chmura natywna, 347
Cloud Foundry, 487
Config Server
 definiowanie tokena Vaulta, 393
obsługa magazynu Vault, 391
uaktualnień zaczepu, 400

odświeżanie właściwości konfiguracyjnych, 393
zapisywanie danych wrażliwych, 393

D

dane, 79
 statystyczne, 451, 466
 uwierzytelniające RabbitMQ, 217
 wrażliwe magazyn Vault, 389
definiowanie domeny użytkownika, 119
 encji użytkownika, 119
 interfejsu klienta, 366
 kontrolera reaktywnego, 296
 przepływu integracji, 235
 schematu, 87
 strumienia reaktywnego, 267
 widoku, 39
 właściwości komunikatu, 220
deklarowanie bezpiecznika, 408
 komponentu nasłuchującego, 214
 metadanych, 149
diagram koralikowy, 271
Docker
 aplikacje Spring Boota, 490
 rozpowszechnianie aplikacji, 490
dodawanie hiperłączy, 174
 obsługi brokera, 216
 własnych hiperłączy, 189
domena, 52
 jako encja, 99
dostosowanie nazw ścieżek, 184

- E**
- encja, 99
 - Eureka, 352
 - konfigurowanie, 356
 - właściwości klienta, 362
 - panel kontrolny, 355
 - skalowanie serwera, 359
 - tryb utrzymania
 - egzemplarza, 358
 - używanie usługi, 363
- F**
- filtr, 243
 - filtrowanie
 - danych, 281
 - strumienia reaktywnego, 281
 - formularz, 62
 - weryfikacja danych, 68, 70
 - framework
 - Spring, 47
 - WebFlux, 293
 - funkcyjne procedury obsługi
 - żądania, 300
- G**
- generowanie danych
 - strumienia, 275
- H**
- hasła, 114
 - HATEOAS, 172
 - hiperłącza, 174, 189
 - hipermedia, 171
- I**
- implementacja magazynu
 - danych, 98
 - informacje
 - o aktywności aplikacji, 439
 - o aplikacji, 428
 - o komponencie Bean, 431
 - o stanie aplikacji, 449, 466
 - o użytkownika, 121
- inicjalizacja projektu, 26, 497, 506, 510
- na stronie start.spring.io, 506
 - w Spring Tool Suite, 497
 - z poziomu powłoki, 510
- integracja, *Patrz także* Spring Integration
- definiowanie przepływu, 235
 - deklarowanie przepływu, 234
 - konfigurowanie
 - przepływu, 237
 - Spring Integration, 239
 - tworzenie własnego
 - przepływu, 256
- IntelliJ IDEA, 500
- tworzenie projektu, 500
- interfejsy
- powłoki, 512
 - repozytoriów reaktywnych, 343
- J**
- JConsole, 477
 - JDBC, 79, 112
 - JMS
 - konfigurowanie, 202
 - otrzymywanie komunikatów, 211
 - wysyłanie komunikatów, 202
 - JmsTemplate
 - otrzymywanie komunikatów, 212
 - wysyłanie komunikatów, 204
 - JMX, java management extensions, 475
- K**
- KafkaTemplate
 - wysyłanie komunikatów, 227
 - kanal komunikatu, 241
 - klasa
 - JdbcTacoRepository, 90
 - JdbcTemplate, 80, 83, 84
 - początkowa aplikacji, 35
 - kompilowanie, 41, 494, 513
 - pliku WAR, 485
- komponent
- asemblera zasobu, 176
 - bean z profilami, 155
 - nasłuchujący, 214
 - Kafka, 229
 - wyodrębniający
 - powiadomienie, 402
- komponenty
- Bean, 434
 - MBean, 478
- komunikacja synchroniczna, 201
- komunikaty
- Apache Kafka, 225
 - definiowanie właściwości, 220
 - JMS, 202
 - JmsTemplate, 204, 212
 - KafkaTemplate, 227
 - konwertowanie, 208
 - otrzymywanie, 212
 - pobieranie z RabbitMQ, 221
 - przetwarzanie, 210
 - RabbitMQ, 216, 224
 - RabbitTemplate, 217
- konfiguracja
- automatyczna, 138, 432
 - udostępnianie, 378–382
 - uruchamianie serwera, 373
 - współdzielona, 372, 380
- konfigurowanie
- danych uwierzytelniających, 203
 - Eureki, 356
 - JMS, 202
 - konwertera komunikatu, 208, 220
 - osadzonego serwera LDAP, 117
 - przepływu integracji, 237
 - reaktywnej usługi, 318
 - rejestrowania danych, 143
 - rejestr usług, 352
 - serwera osadzonego, 142
 - Spring
 - Cloud Config Server, 374
 - Integration, 239
 - Security, 110
 - za pomocą profili, 152
 - źródła danych, 140

konsola H2, 45
 kontener Dockera, 490
 kontroler
 DesignTacoController, 86
 OrderController, 97
 reaktywny, 296
 RESTful, 160, 163
 strony głównej, 40
 widoku, 72
 konwertowanie komunikatu,
 208, 220

L

LDAP, 115, 116
 lista hiperłączy, 176
 logowanie, 472

Ł

łączenie typów reaktywnych, 277

M

magazyn
 danych, 82, 98, 321
 danych użytkownika, 111
 oparty na JDBC, 112
 oparty na LDAP, 115
 Vault, 389
 dane wrażliwe, 389
 w Config Server, 391
 zapisywanie danych
 wrażliwych, 390
 mapowanie
 danych reaktywnych, 284
 domeny, 339
 typów domeny, 329
 żądań HTTP, 436
 MBean
 własne komponenty, 478
 wysyłanie powiadomień, 480
 mechanizm równoważenia
 obciążenia, 354
 menedżer testów, 37
 metadane
 brakujące, 150
 właściwości konfiguracyjnej,
 149, 151
 metaframework
 tworzenie aplikacji, 513

metoda
 POST, 196
 PUT, 195
 patchOrder(), 170
 metody HTTP, 195, 306
 mikrouslugi, 350, 487
 minidokumentacja, 151
 moduł punktu końcowego, 254
 MongoDB, 325
 interfejsy repozytoriów
 reaktywnych, 343
 mapowanie typu domeny, 339
 tworzenie reaktywnych
 repozytoriów, 337
 monitor bezpiecznika, 416
 monitorowanie
 aktywności HTTP, 439
 awarii, 412
 Springa, 475
 strumienia biblioteki
 Hystrix, 415
 wątków, 440, 469
 żądań HTTP, 470

N

narzędzie HashiCorp Vault, 389
 natywna chmura Springa, 347
 nazewnictwo osadzonych
 związków, 180
 NetBeans
 tworzenie projektu, 503

O

obsługa
 awarii, 405
 błędów, 312
 brokera RabbitMQ, 216
 hipermediów, 171
 komunikatów
 Kafka, 226
 RabbitMQ, 224
 magazynu
 danych, 82
 Vault, 391
 opóźnień, 405
 Spring Data
 Cassandra, 326
 MongoDB, 338

uaktualnień zaczepu, 400
 żądań HTTP, 37, 56, 161
 odczyt danych, 79
 odkrywanie usług, 361
 opcje wdrażania, 484
 operacja
 buffer(), 287
 take(), 282
 zebrania
 elementów, 289
 mapy, 289
 operacje
 logiczne, 290
 reaktywne, 272
 opóźnienia, 405
 łagodzenie, 410

P

panel
 kontrolny biblioteki Hystrix,
 413
 kontrolny Eureki, 355
 plik
 JAR, 487, 494
 WAR, 485, 494
 pobieranie
 danych z serwera, 162
 zasobów, 309
 polecenie
 curl, 510
 spring init, 512
 porównywanie hasel, 115
 potok, 269
 profile, 152
 aktywowanie, 154
 programowanie
 funkcyjne, 319
 reaktywne, 263, 266
 projekt
 analiza struktury, 30
 HATEOAS, 174
 Reactor, 265
 przechowywanie danych
 wrażliwych, 389
 przekazywanie
 danych do serwera, 167
 zasobu, 311
 przekształcenie, 244

- przepływ
 Flux, 271
 integracji, 234, 235
 reaktywny, 270
- przetwarzanie
 formularza, 62
 komunikatu, 210
- pula wątków, 416
- punkty końcowe, 187
 Actuatora, 427, 476
 REST, 192
- ## R
- RabbitMQ, 215, 216, 221
- RabbitTemplate, 222
 pobieranie komunikatów, 222
 wysyłanie komunikatów, 217
- Reactor, 265, 269
 buforowanie danych, 287
 dodawanie zależności, 271
 filtrowanie strumienia, 281
 generowanie danych
 strumienia, 275
 łączenie typów reaktywnych,
 277
 mapowanie danych, 284
 operacje logiczne, 290
 przekształcanie strumienia,
 281
 tworzenie typu reaktywnego,
 274
- reaktywna obsługa danych, 299
- reaktywne
 aplikacje internetowe, 295
 repozytoria, 325, 335, 337
 usługi, 318
 używanie API REST, 293, 308
- reaktywny
 framework internetowy, 298
 magazyn danych, 321
- Spring
 Data, 323
 MVC, 296
- reguły weryfikacji danych, 68
- rejestrowanie
 danych, 143, 437, 469
 usług, 361
 użytkownika, 123
- repozytorium
 JDBC, 83
 JPA, 102
 GIT, 380
 konfiguracyjne, 377
 reaktywne, 323, 325, 335,
 337
- REST, 191
- RestTemplate, 192, 364
 pobieranie zasobu, 194
 usuwanie zasobu, 196
 używanie usługi, 364
- router, 245
- ## S
- serwer
 administracyjny, 460, 465
 konfiguracji, 373
 LDAP, 116
 przekazywanie danych, 167
 uaktualnienie danych, 168
 usuwanie danych, 170
 Vault, 389
- skalowanie serwera Eureka, 359
- sortowanie, 186
- SPA, single-page application, 161
- specyfikacja kompilacji
 Mavena, 32
- spliter, 247
- sprawdzanie
 poprawności danych, 72
- Spring, 24
- Spring Batch, 48
- Spring Boot, 47
- Spring Boot Actuator, 423
 analiza właściwości, 434
 dane statystyczne, 442
 niestandardowe, 450
 dostosowywanie, 444
- informacje
 o aktywności aplikacji, 439
 o aplikacji, 428
 o komponencie Bean, 431
 o stanie aplikacji, 429
- konfiguracja
 aplikacji, 431
 ścieżki dostępu, 425
 mapowania żądań HTTP, 436
- monitorowanie
 aktywności HTTP, 439
 wątków, 440
- punkty końcowe, 424
- rejestrowanie danych, 437
- udostępnianie informacji, 447
- uwierzytelnianie, 473
- używanie punktów
 końcowych, 427
- własne wskaźniki, 449
- wstrzykiwanie informacji, 446
- wyłączanie punktów
 końcowych, 426
- zabezpieczanie, 455
- Spring Boot Admin, 459
 analiza zmiennych
 środowiskowych, 467
 dane statystyczne, 466
 informacje o stanie aplikacji,
 466
 interfejs użytkownika, 463,
 464
 konfigurowanie aplikacji, 462
- monitorowanie
 wątków, 469
 żądań HTTP, 470
- odkrywanie usługi, 463
- rejestrowanie
 danych, 469
 klientów, 462
- serwer administracyjny, 465
- tworzenie serwera, 460
- uwierzytelnianie, 473
- włączanie logowania, 472
- zabezpieczanie serwera, 471
- Spring Boot
 Dashboard, 42
 DevTools, 43
- Spring Cloud, 48, 352
 Config Server, 374
 Services, 359
- Spring Data, 48, 189, 322
 Cassandra, 326
 JPA, 98
 MongoDB, 338
 repozytorium reaktywne, 323
 REST, 189
- Spring Expression Language, 127

Spring Integration, 48, 233, 234, 239, 241
 adapter kanału, 252
 aktywator usługi, 249
 brama, 251
 filtr, 243
 kanał komunikatu, 241
 moduł punktu końcowego, 254
 przekształcenie, 244
 router, 245
 splitter, 247
 Spring MVC, 78, 296
 Spring Security, 48, 107, 127
 konfigurowanie, 110
 rozszerzenia, 127
 włączenie obsługi, 108
 Spring Tool Suite, 27, 497
 inicjalizacja projektu, 497
 Spring WebFlux, 295
 strona logowania, 129
 stronicowanie, 186
 struktura projektu, 30
 strumienie reaktywne
 Processor, 268
 Publisher, 268
 Subscriber, 268
 Subscription, 268
 strumień
 Javy, 268
 reaktywny, 268
 system kontroli wersji, 447
 szablon widoku, 75
 szyfrowanie właściwości, 386

Ś

ścieżka dostępu
 Actuatora, 425
 relacji, 184
 zasobów, 184
 żądania, 127
 środowisko
 Springa, 139
 uruchomieniowe, 442

T

testowanie
 aplikacji, 36
 kontrolera, 40
 reaktywnego, 303

serwera, 307
 żądań
 GET, 303
 POST, 306
 Thymeleaf, 38
 Traverson, 197
 tryb utrzymania egzemplarza, 358
 tworzenie
 aplikacji, 37
 internetowej, 51
 interfejsów repozytoriów
 reaktywnych, 343
 klasy kontrolera, 55
 komponentu MBean, 478
 metadanych, 151
 pliku WAR, 494
 projektu
 przepływu integracji, 256
 w IntelliJ IDEA, 500
 w NetBeans, 503
 za pomocą metaframeworka, 513
 reaktywnego
 API, 293
 repozytorium, 335, 337
 serwera administracyjnego, 460
 typu reaktywnego, 273
 na podstawie kolekcji, 274
 na podstawie obiektu, 273
 własnego punktu
 końcowego, 452
 zaczepu, 398
 typ RxJava, 299

U

uaktualnienie danych
 w serwerze, 168
 udostępnianie
 konfiguracji, 378, 379, 382
 właściwości, 382, 383
 uruchamianie
 projektu, 41, 513
 serwera konfiguracji, 373
 usługi, 349
 back-endu, 181
 informacje o użytkowniku, 121
 Eureka, 352
 odkrywanie, 361
 rejestrowanie, 361
 REST, 191
 usunięcie
 danych z serwera, 170
 zasobu, 312
 uwierzytlanianie, 114, 380, 473
 użytkownika, 118
 użytkownik, 133
 używanie
 konfiguracji współdzielonej, 380
 usługi
 przez egzemplarz
 WebClient, 365
 za pomocą RestTemplate, 364

W

wartości właściwości
 specjalnych, 144
 wdrażanie, 421, 483
 pliku WAR, 485
 WebClient
 używanie usługi, 365
 weryfikacja danych, 70
 formularza, 68
 widok, 39, 58
 biblioteki szablonów, 75
 własna implementacja
 InfoContributor, 445
 własne
 hiperłącza, 189
 punkty końcowe, 187
 właściwości
 konfiguracyjnych, 145
 wskaźniki informacji, 449
 własny
 komponent MBean, 478
 przepływ integracji, 256
 punkt końcowy, 452
 właściwości
 dla konkretnego profilu, 153
 komunikatu, 220
 konfiguracyjne, 137, 145, 147, 159, 385
 odświeżanie
 automatyczne, 396
 odświeżanie ręczne, 394
 odświeżanie w locie, 393
 szyfrowanie, 386
 środowiskowe, 434
 udostępnianie, 382, 383

- właściwości
 - utajnienie, 385
 - w repozytorium, 377
- włączanie
 - Config Server, 374
 - obsługi Spring Data Cassandra, 326
 - usług back-endu, 181
- wstawianie
 - danych, 89, 94
 - rekordu, 85
- wstępne przygotowanie danych, 87
- wstrzykiwanie
 - informacji, 446
 - repozytorium, 86
- wykres przepływu reaktywnego, 270
- wylogowanie, 131
- wymiana żądań, 314
- wysyłanie komunikatów
 - za pomocą JMS, 202
 - za pomocą JmsTemplate, 204
 - powiadomień, 480
- wyświetlanie
 - błędów, 72
 - informacji, 52
 - konfiguracji aplikacji, 431
- wzorzec bezpiecznika, 405
- Z**
- zabezpieczanie
 - Actuatora, 455
 - API reaktywnego, 316
 - reaktywnej aplikacji, 316
 - serwera administracyjnego, 471
 - żądań internetowych, 125
- zaczep, 398
- zapisywanie
 - danych, 79, 90
 - projektów, 93
- zarządzanie konfiguracją, 371
- zmiennie środowiskowe, 467
- Ż**
- źródła danych, 140
- Ź**
- żądania
 - długo wykonywane, 311
 - HTTP GET, 56, 163, 303

PROGRAM PARTNERSKI

— GRUPY 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
Helion 

Spring jest frameworkiem ułatwiającym tworzenie nowoczesnych aplikacji w Javie. To narzędzie, które dynamicznie się rozwija i z każdym nowym wydaniem zapewnia programistom kolejne, ekscytujące możliwości. Piąta wersja Springa okazała się być krokiem milowym. Obecnie projektanci mogą tworzyć mikrousługi, korzystać z funkcji programowania reaktywnego i doskonalić budowanie aplikacji zgodnie z modelem MVC. Dzięki pełnej integracji Springa ze Spring Boot nawet najbardziej złożone projekty wymagają minimalnej ilości kodu konfiguracyjnego. W efekcie tworzone aplikacje internetowe są w większym stopniu skalowalne i efektywniejsze w wykorzystywaniu wątków.

To kolejne, uzupełnione i zaktualizowane wydanie przewodnika po frameworku Spring. Książka jest napisana zwięzłym, przejrzystym i jasnym stylem, dzięki czemu szybko zrozumiesz zasady pracy ze Springiem i zbudujesz nowoczesną aplikację internetową współpracującą z bazą danych. Pokazane są w niej techniki programowania reaktywnego, pisanie mikrousług, wykrywania usług, wyjaśniono również pracę z API RESTful i zasady wdrażania aplikacji. Nauczysz się stosować najlepsze praktyki programowania w Springu. Książka jest znakomitą pomocą dla programistów Javy, którzy dopiero zaczynają pracę z tym frameworkiem, a także dla tych, którzy chcą opanować nowe rozwiązania oferowane przez kolejne wersje ekosystemu Springa.

W tej książce między innymi:

- solidne wprowadzenie do frameworków Spring i Spring Boot
- integracja aplikacji Springa z innymi aplikacjami
- programowanie reaktywne w tworzeniu aplikacji internetowych
- tworzenie mikrousług i praca ze Spring Cloud
- wdrażanie aplikacji w środowisku produkcyjnym i korzystanie ze Spring Boot Admin

Craig Walls jest inżynierem i programistą. Z ogromnym zaangażowaniem działa na rzecz rozpowszechniania wiedzy o Springu. Często bierze udział w spotkaniach lokalnych grup użytkowników i konferencjach poświęconych temu frameworkowi. Uwielbia pracę nad kodem źródłowym. W wolnych chwilach planuje kolejną wycieczkę do parku Walt Disney World Resort lub Disneylandu.

Nowoczesna aplikacja w Javie? Sprawdź Springa!

 Helion	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI Sięgnij po więcej! ▶	
 helion.pl	 SZKOLENIA		
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	AKADEMIA IT & BUSINESS		
WWW.SZKOLENIA.HELION.PL		ISBN 978-83-283-5606-1	
INFORMATYKA W NAJLEPSZYM WYDANIU		9 788328 356061	Cena: 89,00 zł