



Technologia i rozwiązania

Spring MVC 4

Projektowanie zaawansowanych aplikacji WWW

Programuj jak mistrz — odkryj Spring MVC!



Geoffroy Warin

[PACKT] open source*
PUBLISHING community experience distilled

Tytuł oryginału: Mastering Spring MVC 4

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-2347-6

Copyright © 2015 Packt Publishing

First published in the English language under the title 'Mastering Spring MVC 4 – (9781783982387)'.

Polish edition copyright © 2016 by Helion S.A.

All rights reserved.

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

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

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

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

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/smvc4p.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/smvc4p>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

O autorze	11
O korektorach merytorycznych	12
Przedmowa	15
Rozdział 1. Błyskawiczne tworzenie aplikacji Spring	19
Rozpoczęcie pracy w środowisku Spring Tool Suite	20
Rozpoczęcie pracy w środowisku IntelliJ	25
Rozpoczęcie pracy w serwisie start.Spring.io	26
Rozpoczęcie pracy za pomocą wiersza poleceń	26
Pierwsze kroki	27
Kompilowanie kodu za pomocą narzędzia Gradle	28
Chcę zobaczyć kod!	32
Spring Boot od wewnątrz	34
Dyspozytor i konfiguracja elementów aplikacji	35
Interpreter widoków, zasoby statyczne i ustawienia regionalne	38
Konfiguracja obsługi błędów i kodowania znaków	40
Konfiguracja wbudowanego serwletu kontenera serwera (Tomcat)	42
Port HTTP	44
Konfiguracja protokołu SSL	44
Inne opcje konfiguracyjne	45
Podsumowanie	46
Rozdział 2. Tajniki architektury MVC	47
Architektura MVC	47
Krytyka architektury MVC i dobre praktyki	48
Anemiczny model domeny	48
Informacje ze źródeł	50
Platforma MVC 1-0-1	50

Szablony Thymeleaf	51
Twoja pierwsza strona	52
Architektura platformy Spring MVC	54
Servlet DispatcherServlet	54
Przekazywanie danych do widoku	55
Język Spring Expression Language	56
Użycie parametru przy odczytywaniu danych	56
Dosyć już „Witaj, świecie!”, odczytajmy tweety!	58
Rejestracja aplikacji	58
Zastosowanie projektu Spring Social	60
Dostęp do serwisu Twitter	60
Strumienie i funkcje lambda w Java 8	62
Styl material design i biblioteka WebJars	63
Układy stron	66
Poruszanie się po witrynie	67
Punkt kontrolny	71
Podsumowanie	72
Rozdział 3. Obsługa formularzy i złożonych adresów URL	73
Strona profilu — formularz	73
Weryfikacja danych	80
Dostosowanie komunikatów o błędach	82
Niestandardowe adnotacje do weryfikacji danych	85
Internacjonalizacja	85
Zmiana ustawień regionalnych	87
Tłumaczenie tekstów aplikacji	89
Lista w formularzu	91
Weryfikacja danych po stronie klienta	94
Punkt kontrolny	96
Podsumowanie	96
Rozdział 4. Ładowanie plików i obsługa błędów	99
Ładowanie plików	99
Umieszczanie obrazu w odpowiedzi na zapytanie	104
Zarządzanie konfiguracją ładowania plików	104
Wyświetlenie załadowanego obrazu	107
Obsługa błędów ładowania plików	108
Tłumaczenia komunikatów o błędach	112
Zapisywanie profilu użytkownika w sesji	112
Własne strony z komunikatami o błędach	116
Zmienne tablicowe w adresach URL	117
Wszystko razem	121
Punkt kontrolny	128
Podsumowanie	129

Rozdział 5. Tworzenie aplikacji w stylu REST	131
Czym jest styl REST?	131
Model dojrzałości Richardsona	132
Poziom 0 — HTTP	132
Poziom 1 — zasoby	132
Poziom 2 — metody HTTP	133
Poziom 3 — kontrolki hipermediów	134
Wersje interfejsu API	135
Przydatne kody HTTP	136
Klient jest królem	137
Diagnostyka interfejsu REST API	139
Rozszerzenia przeglądarek wyświetlające format JSON	139
Klient REST w przeglądarce	139
Narzędzie httpie	139
Dostosowanie odpowiedzi JSON	139
Interfejs API do zarządzania zasobami użytkowników	144
Kody stanu i obsługa wyjątków	147
Zwrot kodu stanu za pomocą obiektu ResponseEntity	148
Zwrot kodów stanu za pomocą wyjątków	149
Dokumentowanie interfejsu za pomocą platformy Swagger	153
Tworzenie odpowiedzi XML	154
Punkt kontrolny	156
Podsumowanie	157
Rozdział 6. Zabezpieczanie aplikacji	159
Podstawowe uwierzytelnienie	159
Upoważnieni użytkownicy	160
Uprawnione adresy URL	163
Znaczniki bezpieczeństwa w szablonie Thymeleaf	164
Formularz logowania	165
Uwierzytelnienie przez Twitter	170
Konfiguracja uwierzytelnienia społecznościowego	170
Objaśnienia do kodu	174
Rozproszone sesje	176
Protokół SSL	178
Generowanie certyfikatu z własnym podpisem	179
Jeden kanał	179
Dwa kanały	180
Za bezpiecznym serwerem	181
Punkt kontrolny	181
Podsumowanie	182
Rozdział 7. Zero ryzyka — testy jednostkowe i integracyjne	183
Dlaczego powinienem testować swój kod?	183
Jak powinienem testować swój kod?	184
Programowanie zorientowane na testy	185

Testy jednostkowe	186
Narzędzia odpowiednie do zadania	187
Testy integracyjne	187
Twój pierwszy test jednostkowy	188
Imitacje i atrapy	191
Imitowanie klas przy użyciu narzędzia Mockito	191
Tworzenie atrap klas podczas testów	193
Trzeba używać imitacji czy atrapy?	195
Testy jednostkowe kontrolerów REST	195
Testowanie uwierzytelnienia	201
Tworzenie testów integracyjnych	202
Konfiguracja systemu Gradle	202
Pierwszy test FluentLenium	204
Obiekty stron w bibliotece FluentLenium	209
Tworzenie testów w języku Groovy	212
Testy jednostkowe z wykorzystaniem biblioteki Spock	212
Testy integracyjne z wykorzystaniem biblioteki Geb	215
Obiekty stron w bibliotece Geb	217
Punkt kontrolny	220
Podsumowanie	221
Rozdział 8. Optymalizacja zapytań	223
Produkcyjny profil aplikacji	223
Kompresja gzip	224
Kontrola pamięci podręcznej	224
Pamięć podręczna aplikacji	226
Unieważnianie danych w pamięci podręcznej	231
Rozproszona pamięć podręczna	232
Metody asynchroniczne	233
Tagi ETag	237
Protokół WebSocket	241
Punkt kontrolny	244
Podsumowanie	244
Rozdział 9. Udostępnianie aplikacji w chmurze	245
Wybór operatora usług chmurowych	245
Cloud Foundry	246
OpenShift	246
Heroku	247
Udostępnienie aplikacji w usłudze Pivotal Web Services	247
Instalacja narzędzi konsolowych Cloud Foundry	247
Złożenie aplikacji	248
Aktywacja usługi Redis	252
Udostępnienie aplikacji w usłudze Heroku	253
Instalacja narzędzi	253
Konfiguracja aplikacji	254
Profil Heroku	255

Uruchomienie aplikacji	256
Aktywacja usługi Redis	258
Ulepszanie aplikacji	259
Podsumowanie	260
Rozdział 10. Nie tylko Spring Web	261
Platforma Spring	261
Core (rdzeń)	262
Execution (uruchamianie)	262
Data (dane)	262
Inne ciekawe projekty	263
Wdrożenie	263
Platforma Docker	263
Aplikacje jednostronicowe	264
Najważniejsi gracze	265
Przyszłość	265
Bezstanowość	266
Podsumowanie	266
Skorowidz	267

Ładowanie plików i obsługa błędów

W tym rozdziale umożliwisz użytkownikom ładowanie obrazów do profili. Dowiesz się również, jak obsługiwać błędy w platformie Spring MVC.

Ładowanie plików

Teraz umożliwisz użytkownikom ładowanie obrazów do profili. Opis, jak to zrobić, znajduje się w dalszej części rozdziału, ale teraz uprość nieco projekt i utwórz w katalogu *templates/profile* nową stronę *uploadPage.html*:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
  <head lang="pl">
    <title>Ładowanie obrazu</title>
  </head>
  <body>
    <div class="row" layout:fragment="content">
      <h2 class="indigo-text center">Ładowanie obrazu</h2>
      <form th:action="@{/upload}" method="post" enctype="multipart/form-data"
            class="col m8 s12 offset-m2">
        <div class="input-field col s6">
          <input type="file" id="file" name="file"/>
        </div>
        <div class="col s6 center">
```

```

        <button class="btn indigo waves-effect waves-light" type="submit"
            ↪name="save"
            th:text="#{submit}">Wyślij
            <i class="mdi-content-send right"></i>
        </button>
    </div>
</form>
</div>
</body>
</html>

```

Nie ma tu niczego ciekawego poza atrybutem `enctype`. Plik z obrazem będzie wysyłany metodą POST na adres URL *upload*. Teraz w pakiecie *profile* obok klasy `ProfileController` utwórz odpowiedni kontroler:

```

package masterSpringMvc.profile;

import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.multipart.MultipartFile;

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

@Controller
public class PictureUploadController {
    public static final Resource PICTURES_DIR = new
        ↪FileSystemResource("./pictures");

    @RequestMapping("upload")
    public String uploadPage() {
        return "profile/uploadPage";
    }

    @RequestMapping(value = "/upload", method = RequestMethod.POST)
    public String onUpload(MultipartFile file) throws IOException {
        String filename = file.getOriginalFilename();
        File tempFile = File.createTempFile("pic", getFileExtension(filename),
            PICTURES_DIR.getFile());

        try (InputStream in = file.getInputStream();
            OutputStream out = new FileOutputStream(tempFile)) {
            IOUtils.copy(in, out);
        }
    }
}

```

```

        return "profile/uploadPage";
    }

    private static String getFileExtension(String name) {
        return name.substring(name.lastIndexOf("."));
    }
}

```

Pierwszą operacją wykonywaną przez powyższy kod jest utworzenie tymczasowego pliku w katalogu *pictures* (obrazy), znajdującym się w głównym katalogu projektu. Sprawdź więc, czy ten katalog istnieje. W języku Java tymczasowy plik posiada unikatowy identyfikator w systemie plików. Usunięcie pliku zależy od użytkownika.

Utwórz w głównym katalogu projektu katalog *pictures*, a w nim pusty plik o nazwie *.gitkeep*, aby katalog był zapisywany w systemie Git.

Puste katalogi w systemie Git

System Git służy do zarządzania plikami i nie ma możliwości zapisywania w nim pustych katalogów. Powszechnie stosowaną metodą ominięcia tego ograniczenia jest stosowanie pustych plików, na przykład *.gitkeep*. W ten sposób zmusza się system do uwzględniania katalogu w procesie kontroli wersji projektu.

Plik ładowany przez użytkownika jest reprezentowany w kontrolerze przez interfejs `Multipart` ↪ `File`. Interfejs ten posiada kilka metod zwracających nazwę pliku, jego wielkość i zawartość.

Szczególnie interesuje nas metoda `getInputStream`. Reprezentowany przez nią strumień wejściowy zostanie skopiowany do metody `fileOutputStream` za pomocą metody `IOUtils.copy`. Pisanie kodu kopiującego strumień wejściowy do wyjściowego jest dość nudne, więc wygodniej będzie skorzystać z biblioteki Apache Utils, zawartej w pliku *tomcat-embed-core.jar*, umieszczonym w ścieżce *classpath*.

W tej części będziesz intensywnie wykorzystywał ciekawe funkcjonalności platformy Spring i biblioteki NIO wprowadzonej w wersji Java 7:

- pomocniczą klasę `String` reprezentującą zasób, który można przetwarzać na różne sposoby;
- blok instrukcji `try...with` automatycznie zamykający strumień nawet w przypadku wystąpienia wyjątku, dzięki czemu nie trzeba każdorazowo definiować bloku `finally`.

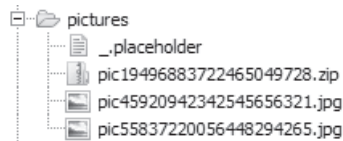
W powyższym kodzie każdy plik załadowany przez użytkownika zostanie skopiowany do katalogu *pictures*.

Platforma Spring Boot oferuje kilka właściwości umożliwiających dostosowanie procesu ładowania plików. Przyjrzyjmy się klasie `MultipartProperties`.

Jej najciekawsze właściwości to:

- `multipart.maxFileSize`, właściwość definiująca maksymalną wielkość ładowanego pliku. Przy próbie załadowania większego pliku jest zgłaszany wyjątek `MultipartException`. Domyślna wielkość pliku to 1 MB.
- `multipart.maxRequestSize`, właściwość definiująca maksymalną wielkość żądania wieloczęściowego. Domyślna wielkość to 10 MB.

Powyższe wartości domyślne są odpowiednie dla Twojej aplikacji. Po załadowaniu kilku plików katalog *pictures* będzie wyglądał następująco:



Zaczekaj! Ktoś załadował plik ZIP! Aż trudno w to uwierzyć. Trzeba w kodzie kontrolera sprawdzić, czy ładowane pliki faktycznie reprezentują obrazy:

```
package masterSpringMvc.profile;

import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import java.io.*;

@Controller
public class PictureUploadController {
    public static final Resource PICTURES_DIR = new
    FileSystemResource("./pictures");

    @RequestMapping("upload")
    public String uploadPage() {
        return "profile/uploadPage";
    }

    @RequestMapping(value = "/upload", method = RequestMethod.POST)
    public String onUpload(MultipartFile file, RedirectAttributes redirectAttrs)
    throws IOException {
        if (file.isEmpty() || !isImage(file)) {
            redirectAttrs.addFlashAttribute("error",
            "Niewłaściwy plik. Załaduj plik z obrazem.");
            return "redirect:/upload";
        }
    }
}
```

```

    }
    copyFileToPictures(file);
    return "profile/uploadPage";
}

private Resource copyFileToPictures(MultipartFile file) throws IOException {
    String fileExtension = getFileExtension(file.getOriginalFilename());
    File tempFile = File.createTempFile("pic", fileExtension, PICTURES_DIR.getFile());
    try (InputStream in = file.getInputStream();
        OutputStream out = new FileOutputStream(tempFile)) {
        IOUtils.copy(in, out);
    }
    return new FileSystemResource(tempFile);
}

private boolean isImage(MultipartFile file) {
    return file.getContentType().startsWith("image");
}

private static String getFileExtension(String name) {
    return name.substring(name.lastIndexOf("."));
}
}

```

Bardzo proste! Metoda `getContentType` zwraca informację o typie pliku w formacie **MIME** (ang. *Multipurpose Internet Mail Extensions* — uniwersalne rozszerzenie poczty internetowej), np. `image/png`, `image/jpg` itp. Wystarczy więc sprawdzić, czy ciąg MIME rozpoczyna się od słowa `image`.

W formularzu została zakodowana obsługa błędów, więc trzeba rozbudować stronę o jakieś elementy umożliwiające wyświetlanie komunikatów. Wpisz w pliku `uploadPage.html` następująco wiersze tuż poniżej tytułu strony:

```

<div class="col s12 center red-text" th:text="${error}" th:if="${error}">
    Błąd ładowania pliku
</div>

```

Przy następnej próbie załadowania pliku ZIP pojawi się komunikat o błędzie, jak na poniższym rysunku:



Umieszczanie obrazu w odpowiedzi na zapytanie

Ładowane obrazy nie są zapisywane w statycznych katalogach. Aby wyświetlić je na stronie trzeba przedsięwziąć specjalne środki.

W kodzie strony *uploadPage.html* tuż nad znacznikiem form wpisz poniższe wiersze:

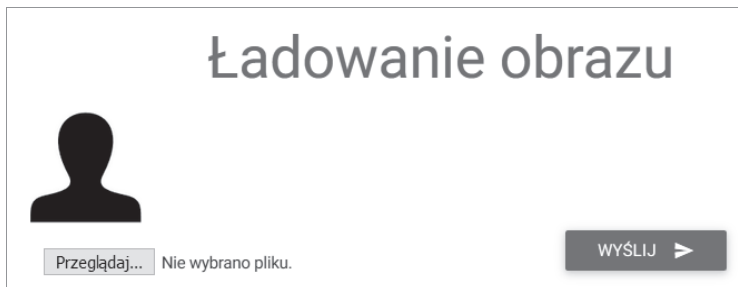
```
<div class="col m8 s12 offset-m2">
  
</div>
```

Powyzszy kod będzie próbował pobrać obraz z kontrolera. W klasie *PictureUploadController* utwórz odpowiednią metodę:

```
@RequestMapping(value = "/uploadedPicture")
public void getUploadedPicture(HttpServletRequest response) throws IOException {
    ClassPathResource classPathResource = new
    ↳ClassPathResource("/images/anonymous.png");
    response.setHeader("Content-Type",
        URLConnection.guessContentTypeFromName(classPathResource.getFilename()));
    IOUtils.copy(classPathResource.getInputStream(), response.getOutputStream());
}
```

Powyzszy kod będzie umieszczał bezpośrednio w odpowiedzi obraz z pliku *src/main/resources/images/anonymous.png*. To bardzo ciekawy sposób!

Jeżeli ponownie otworzysz stronę, pojawi się na niej następujący obraz:



Rysunek anonimowego użytkownika znalazłem na stronie <http://iconmonstr.com/user-icon> i zapisałem go jako obraz o wymiarach 128×128 pikseli w pliku PNG.

Zarządzanie konfiguracją ładowania plików

W tym momencie warto jest określić w pliku konfiguracyjnym *application.properties* katalog dla ładowanych obrazów i ścieżkę do rysunku anonimowego użytkownika.

Utwórz pakiet *config*, a w nim klasę *PicturesUploadProperties*:

```
package masterSpringMvc.config;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.core.io.DefaultResourceLoader;
import org.springframework.core.io.Resource;
import java.io.IOException;

@ConfigurationProperties(prefix = "upload.pictures")
public class PictureUploadProperties {
    private Resource uploadPath;
    private Resource anonymousPicture;

    public Resource getAnonymousPicture() {
        return anonymousPicture;
    }

    public void setAnonymousPicture(String anonymousPicture) {
        this.anonymousPicture = new DefaultResourceLoader().getResource
            ↪(anonymousPicture);
    }

    public Resource getUploadPath() {
        return uploadPath;
    }

    public void setUploadPath(String uploadPath) {
        this.uploadPath = new DefaultResourceLoader().getResource(uploadPath);
    }
}
```

W tej klasie wykorzystany został pakiet *ConfigurationProperties* platformy Spring Boot, umożliwiający automatyczne mapowanie właściwości znalezionych w plikach umieszczonych w ścieżce *classpath* (domyślnie w pliku *application.properties*), z uwzględnieniem typów tych właściwości.

Zwróć uwagę, że zdefiniowana jest metoda z argumentem typu *String*, ale nic nie stoi na przeszkodzie, aby inna metoda zwracała inny typ danych, który będzie najprzydatniejszy.

Teraz w pakiecie *config* zdefiniuj klasę *PicturesUploadProperties*:

```
@SpringBootApplication
@EnableConfigurationProperties({PictureUploadProperties.class})
public class MasterSpringMvc4Application extends WebMvcConfigurerAdapter {
    // kod pominięty
}
```

W pliku *application.properties* zdefiniuj następujące właściwości:

```
upload.pictures.uploadPath=file:./pictures
upload.pictures.anonymousPicture=classpath:/images/anonymous.png
```

Ponieważ użyta została klasa `DefaultResourceLoader` platformy Spring, można zastosować prefiksy, np. `file:` lub `classpath:` do określenia położenia zasobów.

Jest to sposób alternatywny do utworzenia klasy `FileSystemResource` lub `ClassPathResource`.

Zaletą powyższej metody jest możliwość dokumentowania kodu. Dzięki niej od razu widać, że katalog z obrazami znajduje się w głównym katalogu obiektu, natomiast obraz przedstawiający anonimowego użytkownika znajduje się w pliku zapisanym w ścieżce *classpath*.

To wszystko. Teraz można wykorzystać zdefiniowane właściwości w kontrolerze. Poniżej przedstawione są odpowiednie części klasy `PictureUploadController`:

```
package masterSpringMvc.profile;

import masterSpringMvc.config.PictureUploadProperties;
import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import javax.servlet.http.HttpServletResponse;
import java.io.*;
import java.net.URLConnection;

@Controller
public class PictureUploadController {
    private final Resource picturesDir;
    private final Resource anonymousPicture;

    @Autowired
    public PictureUploadController(PictureUploadProperties uploadProperties) {
        picturesDir = uploadProperties.getUploadPath();
        anonymousPicture = uploadProperties.getAnonymousPicture();
    }

    @RequestMapping(value = "/uploadedPicture")
    public void getUploadedPicture(HttpServletResponse response) throws
        IOException {
        response.setHeader("Content-Type",
            URLConnection.guessContentTypeFromName(anonymousPicture.getFilename()));
        IOUtils.copy(anonymousPicture.getInputStream(),
            response.getOutputStream());
    }

    private Resource copyFileToPictures(MultipartFile file) throws IOException {
```



```

String fileExtension = getFileExtension(file.getOriginalFilename());
File tempFile = File.createTempFile("pic", fileExtension,
    picturesDir.getFile());
try (InputStream in = file.getInputStream();
    OutputStream out = new FileOutputStream(tempFile)) {
    IOUtils.copy(in, out);
}
return new FileSystemResource(tempFile);
}
// Pozostała część kodu pozostaje bez zmian
}

```

Jeżeli uruchomisz aplikację ponownie, stwierdzisz, że strona się nie zmieniła. Obraz anonimowego użytkownika będzie dalej pokazywany, a obrazy ładowane przez użytkowników będą dalej zapisywane w podkatalogu *pictures* w głównym katalogu projektu.

Wyświetlenie załadowanego obrazu

Teraz dobrze byloby wyświetlić obraz załadowany przez użytkownika, prawda? W tym celu do klasy `PictureUploadController` dodaj atrybut modelu:

```

@ModelAttribute("picturePath")
public Resource picturePath() {
    return anonymousPicture;
}

```

Umieść powyższy atrybut w kodzie, aby odczytać jego wartość po załadowaniu obrazu:

```

@RequestMapping(value = "/uploadedPicture")
public void getUploadedPicture(HttpServletRequest response,
    @ModelAttribute("picturePath") Path picturePath) throws IOException {
    response.setHeader("Content-Type",
        URLConnection.guessContentTypeFromName(picturePath.toString()));
    Files.copy(picturePath, response.getOutputStream());
}

```

Adnotacja `@ModelAttribute` umożliwia wygodne tworzenie atrybutów modelu dla wybranych metod, które potem można umieszczać w metodzie kontrolera z tą samą adnotacją. W powyższym kodzie parametr `picturePath` będzie dostępny w modelu danych do czasu przekierowania użytkownika na inną stronę. Jego domyślną wartością jest zdefiniowana we właściwościach nazwa pliku z obrazem anonimowego użytkownika.

Po załadowaniu pliku trzeba zaktualizować wartość atrybutu. Zmień w tym celu metodę `onUpload`:

```

@RequestMapping(value = "/upload", method = RequestMethod.POST)
public String onUpload(MultipartFile file, RedirectAttributes redirectAttrs,
    Model model) throws IOException {
    if (file.isEmpty() || !isImage(file)) {

```

```

        redirectAttrs.addFlashAttribute("error",
            "Niewłaściwy plik. Załaduj plik z obrazem.");
        return "redirect:/upload";
    }
    Resource picturePath = copyFileToPictures(file);
    model.addAttribute("picturePath", picturePath);
    return "profile/uploadPage";
}

```

Dzięki umieszczeniu w kodzie odwołania do modelu można odczytać wartość parametru `picturePath` po załadowaniu obrazu.

Teraz problem polega na tym, że metody `onUpload` i `getUploadedPicture` są wywoływane w różnych zapytaniach. Niestety, wartości atrybutów modelu są usuwane pomiędzy kolejnymi zapytaniami.

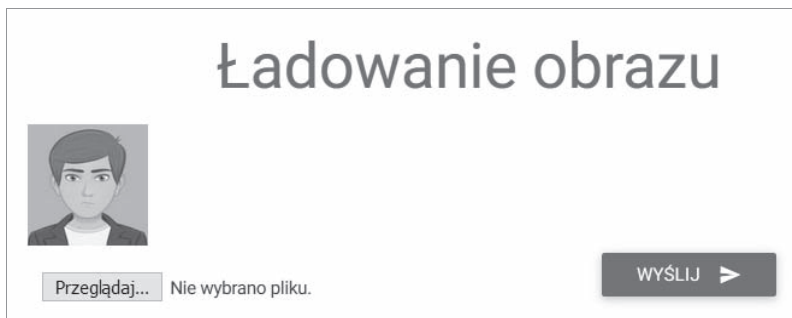
Dlatego parametr `picturePath` trzeba zdefiniować jako atrybut sesji. W tym celu klasę kontrolera należy opatrzyć dodatkową adnotacją:

```

@Controller
@SessionAttributes("picturePath")
public class PictureUploadController {
}

```

Uff! Sporo adnotacji jak na prostą obsługę atrybutu sesji. Teraz strona będzie wyglądała jak poniżej:



W ten sposób można naprawdę łatwo tworzyć kod. Ponadto nie trzeba bezpośrednio stosować interfejsu `HttpServletRequest` ani `HttpSession`. Co więcej, można również łatwo określać typ obiektu.

Obsługa błędów ładowania plików

Wnikliwy Czytelnik z pewnością zauważy, że powyższy kod może zgłaszać dwa rodzaje wyjątków:

- `IOException`: wyjątek zgłaszany w sytuacji, gdy coś złego wydarzy się podczas zapisywania pliku na dysku;
- `MultipartException`: wyjątek zgłaszany w sytuacji, gdy podczas ładowania pliku wystąpi błąd, na przykład przekroczona zostanie maksymalna wielkość pliku.

Teraz jest dobra okazja do zapoznania się z dwoma sposobami obsługi błędów w środowisku Spring:

- poprzez lokalne użycie adnotacji `@ExceptionHandler` w metodzie kontrolera,
- poprzez zdefiniowanie globalnego kontenera serwletów.

Obsłuż wyjątek `IOException` za pomocą poniższej metody z adnotacją `@ExceptionHandler`, utworzonej w klasie `PictureUploadController`:

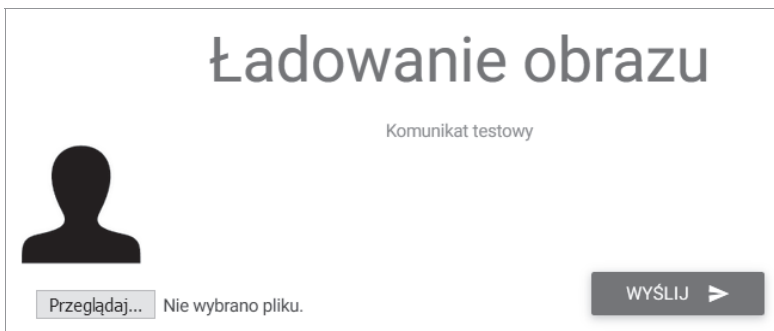
```
@ExceptionHandler(IOException.class)
public ModelAndView handleIOException(IOException exception) {
    ModelAndView modelAndView = new ModelAndView("profile/uploadPage");
    modelAndView.addObject("error", exception.getMessage());
    return modelAndView;
}
```

To prosty, ale skuteczny sposób. Powyższa metoda będzie wywoływana za każdym razem, gdy w kontrolerze zostanie zgłoszony wyjątek `IOException`.

Zgłaszanie wyjątków w kodzie Java jest dość trudne, więc w celu sprawdzenia działania metody obsługi wyjątku zmień na czas testów kod metody `onUpload` w następujący sposób:

```
@RequestMapping(value = "/upload", method = RequestMethod.POST)
public String onUpload(MultipartFile file, RedirectAttributes
    redirectAttrs, Model model) throws IOException {
    throw new IOException("Komunikat testowy");
}
```

Po wprowadzeniu powyższych zmian i załadowaniu obrazu na stronie pojawi się poniższy komunikat:



Teraz trzeba obsłużyć wyjątek `MultipartException`. Należy to zrobić na poziomie kontenera serwetów (czyli serwera Tomcat), ponieważ wyjątek ten nie jest zgłaszany bezpośrednio przez kontroler.

W tym celu musisz utworzyć nową metodę konfiguracyjną `EmbeddedServletContainerCustomizer` w klasie `WebConfiguration`:

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer() {
    EmbeddedServletContainerCustomizer embeddedServletContainerCustomizer =
        new EmbeddedServletContainerCustomizer() {
            @Override
            public void customize(ConfigurableEmbeddedServletContainer container) {
                container.addErrorPages(new ErrorPage(MultipartException.class,
                    ↪"/uploadError"));
            }
        };
    return embeddedServletContainerCustomizer;
}
```

Kod jest nieco rozwlekły. Zwróć uwagę, że `EmbeddedServletContainerCustomizer` to interfejs zawierający jedną metodę, którą można zastąpić wyrażeniem lambda:

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer() {
    EmbeddedServletContainerCustomizer embeddedServletContainerCustomizer =
        container -> container.addErrorPages(new ErrorPage(MultipartException.class,
            "/uploadError"));
    return embeddedServletContainerCustomizer;
}
```

Można więc po prostu napisać taki kod:

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer() {
    return container -> container.addErrorPages(new
        ErrorPage(MultipartException.class,
            "/uploadError"));
}
```

Powyższy kod tworzy nową stronę z komunikatem o błędzie, która będzie wyświetlana w momencie zgłoszenia wyjątku `MultipartException`. Będzie ona również powiązana z kodem HTTP. Interfejs `EmbeddedServletContainerCustomizer` posiada wiele innych funkcjonalności umożliwiających dostosowanie kontenera serwetów odpowiednio do działania aplikacji. Więcej informacji na ten temat jest dostępnych pod adresem <http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-developing-web-applications.html#boot-features-customizing-embedded-containers>.

Teraz w klasie `PictureUploadController` trzeba obsłużyć adres URL `uploadError`:

```
@RequestMapping("uploadError")
public ModelAndView onUploadError(HttpServletRequest request) {
    ModelAndView modelAndView = new ModelAndView("uploadPage");
    modelAndView.addObject("error",
        request.getAttribute(WebUtils.ERROR_MESSAGE_ATTRIBUTE));
    return modelAndView;
}
```

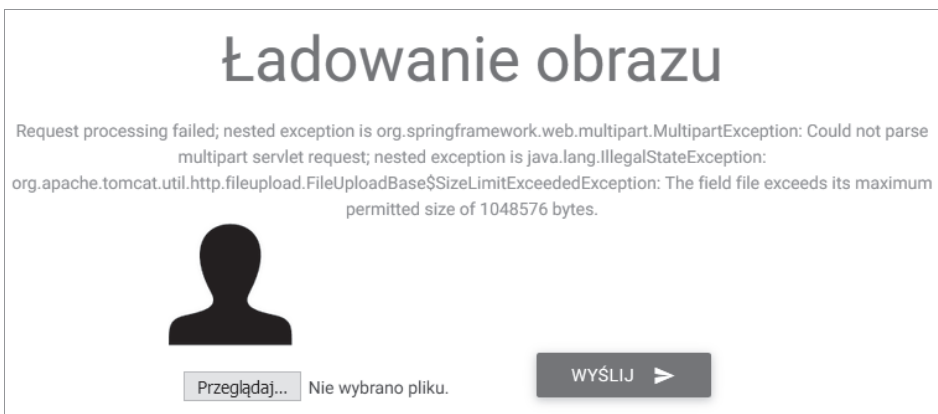
Strony z komunikatami o błędach zdefiniowane w kontenerze serwletów posiadają kilka ciekawych atrybutów umożliwiających diagnozowanie błędów:

Atrybut	Opis
<code>javax.servlet.error.status_code</code>	Kod HTTP błędu
<code>javax.servlet.error.exception_type</code>	Klasa wyjątku
<code>javax.servlet.error.message</code>	Komunikat o wyjątku
<code>javax.servlet.error.request_uri</code>	Adres URL, dla którego został zgłoszony wyjątek
<code>javax.servlet.error.exception</code>	Właściwy wyjątek
<code>javax.servlet.error.servlet_name</code>	Nazwa serwletu zgłaszającego wyjątek

Z powyższych atrybutów można wygodnie korzystać dzięki klasie `WebUtils` z biblioteki `Spring Web`.

Jeżeli użytkownik spróbuje załadować bardzo duży obraz, pojawi się czytelny komunikat.

Teraz możesz sprawdzić, czy błąd jest poprawnie obsługiwany, ładując duży plik (większy niż 1 MB) lub ustawiając właściwość `multipart.maxFileSize` na mniejszą wartość, na przykład 1 kB.



Tłumaczenia komunikatów o błędach

Dla programisty komunikaty o wyjątkach wyświetlane na stronie są naprawdę cenne, jednak dla użytkownika nie przedstawiają większej wartości. Dlatego musisz je przetłumaczyć. W tym celu w konstruktorze kontrolera należy zastosować klasę `MessageSource`:

```
private final MessageSource messageSource;
@Autowired
public PictureUploadController(PictureUploadProperties uploadProperties,
    MessageSource messageSource) {
    picturesDir = uploadProperties.getUploadPath();
    anonymousPicture = uploadProperties.getAnonymousPicture();
    this.messageSource = messageSource;
}
```

Teraz możesz odczytywać komunikaty zapisane w pliku:

```
@ExceptionHandler(IOException.class)
public ModelAndView handleIOException(Locale locale) {
    ModelAndView modelAndView = new ModelAndView("profile/uploadPage");
    modelAndView.addObject("error",
        messageSource.getMessage("upload.io.exception", null, locale));
    return modelAndView;
}

@RequestMapping("uploadError")
public ModelAndView onUploadError(Locale locale) {
    ModelAndView modelAndView = new ModelAndView("profile/uploadPage");
    modelAndView.addObject("error",
        messageSource.getMessage("upload.file.too.big", null, locale));
    return modelAndView;
}
```

Poniżej zdefiniowane są komunikaty:

```
upload.io.exception=Podczas ładowania pliku wystąpił błąd. Spróbuj jeszcze raz.
upload.file.too.big=Plik jest za duży.
```

I jeszcze komunikaty w języku angielskim:

```
upload.io.exception=An error occurred while uploading the file. Please try again.
upload.file.too.big=Your file is too big.
```

Zapisywanie profilu użytkownika w sesji

Kolejną niezbędną rzeczą jest zapisywanie profilu użytkownika w sesji, dzięki czemu informacje nie będą z niego usuwane przy każdorazowym wyświetleniu strony. Takie działanie aplikacji byłoby dla użytkowników irytujące, więc musisz to zmienić.

Sesje HTTP umożliwiają przechowywanie informacji przesyłanych w zapytaniach. Protokół HTTP jest protokołem bezstanowym, tzn. nie ma możliwości powiązania dwóch zapytań wysyłanych przez tego samego użytkownika. Większość kontenerów serwetów przypisuje każdemu użytkownikowi plik ciasteczka o nazwie JSESSIONID. Plik ten jest przesyłany w nagłówku zapytania i umożliwia zapisywanie dowolnych obiektów w mapie o nazwie HttpSession (sesja HTTP). Taka sesja zazwyczaj kończy się w chwili zamknięcia przeglądarki przez użytkownika lub po określonym czasie braku jego aktywności.

Wcześniej poznałeś sposób umieszczania obiektów w sesji za pomocą adnotacji @SessionAttribute ↪butes. Ta metoda dobrze sprawdza się wewnątrz jednego kontrolera, ale gdy jest ich kilka, wtedy pojawiają się problemy ze współdzieleniem danych. Aby móc identyfikować atrybut na podstawie jego nazwy, trzeba stosować ciąg znaków, co utrudnia modyfikację kodu. Z tego samego powodu nie można bezpośrednio manipulować obiektem HttpSession. Kolejnym argumentem przemawiającym przeciwko bezpośredniemu manipulowaniu sesją są trudnienia w wykonywaniu testów jednostkowych kontrolera.

W środowisku Spring stosuje się jednak inny popularny sposób zapisywania informacji w sesji, mianowicie opatrywanie metody adnotacją @Scope("session").

Dzięki temu można umieszczać metody w kontrolerach lub innych komponentach aplikacji Spring, aby odczytywać bądź zapisywać w nich dane.

Utwórz w pakiecie *profile* klasę *UserProfileSession*:

```
package masterSpringMvc.profile;

import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;
import java.io.Serializable;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class UserProfileSession implements Serializable {
    private String twitterHandle;
    private String email;
    private LocalDate birthDate;
    private List<String> tastes = new ArrayList<>();

    public void saveForm(ProfileForm profileForm) {
        this.twitterHandle = profileForm.getTwitterHandle();
        this.email = profileForm.getEmail();
        this.birthDate = profileForm.getBirthDate();
        this.tastes = profileForm.getTastes();
    }
}
```

```

public ProfileForm toForm() {
    ProfileForm profileForm = new ProfileForm();
    profileForm.setTwitterHandle(twitterHandle);
    profileForm.setEmail(email);
    profileForm.setBirthDate(birthDate);
    profileForm.setTastes(tastes);
    return profileForm;
}
}

```

Zaimplementowałeś wygodny sposób odczytywania i zapisywania danych w postaci obiektu ProfileForm. Dzięki temu będziesz mógł zapisywać i odczytywać dane z kontrolera Profile ↪ Controller. Musisz w nim umieścić zmienną typu UserProfileSession i zapisać ją jako pole klasy. Ponadto musisz udostępnić obiekt ProfileForm jako atrybut modelu, dzięki czemu nie trzeba będzie stosować go w metodzie displayProfile. Na koniec dane profilu po ich sprawdzeniu należy zapisać:

```

@Controller
public class ProfileController {
    private UserProfileSession userProfileSession;

    @Autowired
    public ProfileController(UserProfileSession userProfileSession) {
        this.userProfileSession = userProfileSession;
    }

    @ModelAttribute
    public ProfileForm getProfileForm() {
        return userProfileSession.toForm();
    }

    @RequestMapping(value = "/profile", params = {"save"}, method =
    ↪ RequestMethod.POST)
    public String saveProfile(@Valid ProfileForm profileForm,
        BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return "profile/profilePage";
        }
        userProfileSession.saveForm(profileForm);
        return "redirect:/profile";
    }
    // Pozostała część kodu bez zmian
}

```

To wszystko, co trzeba zrobić w aplikacji Spring, aby zapisywać dane w sesji.

Jeżeli teraz użytkownik wypełni formularz i odświeży stronę, wprowadzone dane będą zachowane.

Teraz zamierzam opisać kilka użytych wcześniej pojęć.

Pierwsze z nich to umieszczanie (wstrzykiwanie) obiektów w konstruktorze. Konstruktor kontrolera `ProfileController` jest opatrzony adnotacją `@Autowired`, oznaczającą, że platforma Spring będzie przetwarzała argumenty konstruktora przed utworzeniem instancji klasy. Alternatywnym rozwiązaniem, nieco mniej obszernym, jest zastosowanie wstrzykiwania pól:

```
@Controller
public class ProfileController {
    @Autowired
    private UserProfileSession userProfileSession;
}
```

Wstrzykiwanie obiektów w konstruktorze jest zdecydowanie lepszym sposobem, ponieważ ułatwia wykonywanie testów jednostkowych, gdyby trzeba było zrezygnować z testów oferowanych przez platformę Spring. Ponadto zależności między klasami są nieco ściślejsze.

Szczegółowy opis wstrzykiwania pól i konstruktorów jest zawarty w doskonałym wpisie na blogu Olivera Gierkego pod adresem <http://olivergierke.de/2013/11/why-field-injection-is-evil>.

Kolejną rzeczą wymagającą wyjaśnienia jest parametr `proxyMode` (tryb serwera proxy) w adnotacji `@Scope`:

```
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
```

W platformie Spring dostępne są trzy wartości parametru `proxyMode`, nie licząc domyślnego:

- `TARGET_CLASS` — oznaczający zastosowanie serwera CGI,
- `INTERFACES` — oznaczający zastosowanie serwera JDK,
- `NO` — oznaczający brak serwera proxy.

Zaletą serwera proxy zazwyczaj ujawnia się podczas wstrzykiwania obiektów do długotrwałych komponentów, na przykład singletonów. Ponieważ wstrzykiwanie odbywa się tylko raz, podczas tworzenia klasy Bean, więc w kolejnych wywołaniach tej klasy nie jest uwzględniany jej bieżący stan.

W Twoim przypadku aktualny stan klasy Bean jest zapisywany w sesji, a nie bezpośrednio w samej klasie. Z tego powodu platforma Spring musi utworzyć serwer proxy, który przechwytyje wywołania metod klasy i wykrywa ich mutacje. Dzięki temu stan klasy może być zapisywany i odczytywany z sesji HTTP.

W przypadku klasy sesji trzeba stosować tryb proxy. Serwer CGI dokonuje instrumentalizacji kodu bajtowego i współpracuje z każdą klasą, natomiast serwer JDK jest nieco mniej ingerencyjny, ale wymaga zaimplementowania interfejsu.

Ponadto klasa `UserProfileSession` została zdefiniowana jako implementacja interfejsu `Serializable`. Nie jest to konieczne, ponieważ sesja HTTP może przechowywać w pamięci dowolne obiekty, ale dobrą praktyką jest tworzenie obiektów, które można serializować.

W rzeczywistości można zmienić sposób przechowywania danych w sesji. W rozdziale 8., „Optymalizacja zapytań”, poznasz sposób zapisywania ich w bazie danych Redis, która współpracuje z obiektami typu `Serializable`. Najlepiej jest traktować sesję jako uniwersalny magazyn danych. Zadaniem programisty jest opracowanie sposobu zapisywania i odczytywania danych z tego magazynu.

Aby serializacja klasy przebiegała poprawnie, każde jej pole również musi umożliwiać serializację. W naszym przypadku ciągi znaków oraz daty spełniają ten warunek, więc nic nie stoi na przeszkodzie.

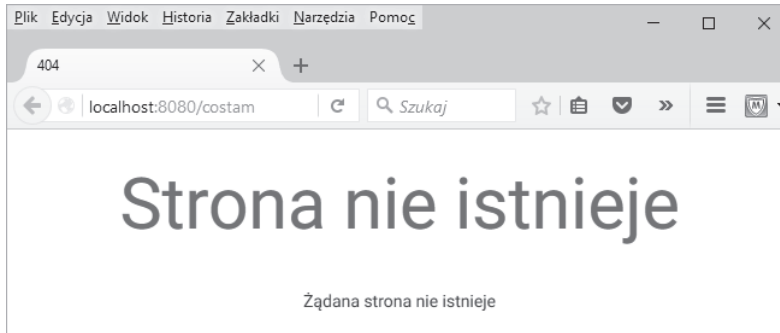
Własne strony z komunikatami o błędach

Platforma Spring Boot umożliwia definiowanie własnych widoków z komunikatami o błędach, wyświetlanych zamiast opisanej wcześniej strony z komunikatem *Whitelabel Error Page*. Widok musi mieć nazwę `error` (błąd). Jego przeznaczeniem jest obsługa wszystkich wyjątków. Domyślna klasa `BasicErrorController` udostępnia wiele przydatnych atrybutów modelu danych, które można wykorzystać na takiej stronie.

Utwórz w katalogu `src/main/resources/templates` własną stronę `error.html` do wyświetlania komunikatów o błędach:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head lang="en">
    <meta charset="UTF-8"/>
    <title th:text="{status}">404</title>
    <link href="materialize/css/materialize.css"
          type="text/css" rel="stylesheet"
          media="screen,projection"/>
  </head>
  <body>
    <div class="row">
      <h1 class="indigo-text center" th:text="{error}">Strona nie istnieje</h1>
      <p class="col s12 center" th:text="{message}">
        Żądana strona nie istnieje
      </p>
    </div>
  </body>
</html>
```

Jeżeli teraz użytkownik otworzy adres URL, który nie będzie obsługiwany przez aplikację, pojawi się własna strona z komunikatem:



Bardziej zaawansowanym sposobem obsługi błędów jest zaimplementowanie klasy `ErrorController`, czyli kontrolera do obsługi wyjątków na poziomie globalnym. Przyjrzyj się klasom `ErrorMvcAutoConfiguration` i `BasicErrorController`, będącym standardowymi implementacjami tego kontrolera.

Zmienne tablicowe w adresach URL

Teraz wiesz już, jakimi tematami jest zainteresowany użytkownik. Warto byłoby ulepszyć kontroler tak, aby wyszukiwał tweety zawierające słowa kluczowe z zadanej listy.

Jednym z ciekawych sposobów przekazywania w adresie URL par danych klucz-wartość jest zastosowanie zmiennych tablicowych. Można to łatwo osiągnąć za pomocą parametrów zapytań. Przyjrzyj się poniższemu kodowi:

```
jakiśUrl/parametr?zmienna1=wartość1&zmienna2=wartość2
```

Zamiast parametru można zastosować tablicę zmiennych, jak poniżej:

```
jakiśUrl/parametr;zmienna1=wartość1;zmienna2=wartość2
```

Każda wartość może być również listą:

```
jakiśUrl/parametr;zmienna1=wartość1,wartość2;zmienna2=wartość3,wartość4
```

Zmienna tablicowa może zostać skojarzona w kontrolerze z obiektami różnych typów, na przykład:

- `Map<String, List<?>>` — typ obsługujący wiele zmiennych z wieloma wartościami;
- `Map<String, ?>` — typ obsługujący zmienne z pojedynczymi wartościami;
- `List<?>` — typ stosowany w przypadku, gdy potrzebna jest jedna zmienna, której nazwę można konfigurować.

W Twoim przypadku wymagana jest obsługa adresu jak poniżej:

```
http://localhost:8080/search/popular;keywords=spring,java
```

Pierwszy parametr, popular, oznacza typ wyniku stosowany w interfejsie API do przeszukiwania serwisu Twitter. Parametr ten może przyjmować wartości mixed (mieszane), recent (najnowsze) lub popular (popularne).

Pozostała część adresu URL zawiera listę słów kluczowych, którą można skojarzyć z obiektem typu `List<String>`.

Domyślnie platforma Spring MVC usuwa z adresu URL wszystkie znaki za średnikiem. Zatem pierwszą rzeczą, jaką trzeba zrobić, aby umożliwić zastosowanie zmiennych tablicowych, jest wyłączenie tej funkcjonalności.

Wpisz w definicji klasy `WebConfiguration` poniższy kod:

```
@Override
public void configurePathMatch(PathMatchConfigurer configurer) {
    UrlPathHelper urlPathHelper = new UrlPathHelper();
    urlPathHelper.setRemoveSemicolonContent(false);
    configurer.setUrlPathHelper(urlPathHelper);
}
```

Utwórz w pakiecie `search` nowy kontroler o nazwie `SearchController`. Jego zadaniem będzie obsługa następującego zapytania:

```
package masterSpringMvc.search;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.MatrixVariable;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
import java.util.List;

@Controller
public class SearchController {
    private SearchService searchService;

    @Autowired
    public SearchController(SearchService searchService) {
        this.searchService = searchService;
    }

    @RequestMapping("/search/{searchType}")
    public ModelAndView search(@PathVariable String searchType,
        @MatrixVariable List<String> keywords) {
        List<Tweet> tweets = searchService.search(searchType, keywords);
        ModelAndView modelAndView = new ModelAndView("resultPage");
        modelAndView.addObject("tweets", tweets);
    }
}
```

```

        modelAndView.addObject("search", String.join(", ", keywords));
        return modelAndView;
    }
}

```

Jak widać, do wyświetlenia tweetów można wykorzystać istniejącą stronę z wynikami wyszukiwania. Ponadto wyszukiwanie będzie realizowane przez inną klasę, o nazwie `SearchService` (usługa wyszukiwania). Usługę tę zdefiniuj w tym samym pakiecie, w którym znajduje się kontroler `SearchController`:

```

package masterSpringMvc.search;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.social.twitter.api.Twitter;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class SearchService {
    private Twitter twitter;
    @Autowired
    public SearchService(Twitter twitter) {
        this.twitter = twitter;
    }
    public List<Tweet> search(String searchType, List<String> keywords) {
        return null;
    }
}

```

Teraz musisz zaimplementować metodę `search`. Argumentem metody `twitter.searchOperations().search(parametry)` jest obiekt typu `searchParameters`, umożliwiający zaawansowane wyszukiwanie tweetów według kilkunastu kryteriów. Ciebie interesują atrybuty `query` (zapytanie), `resultType` (typ wyniku) i `count` (liczba).

Najpierw utwórz konstruktor typu `ResultType` z parametrem `searchType`. `ResultType` jest typem wyliczeniowym, zatem można przeglądać różne elementy obiektu i wyszukiwać te, które spełniają zadane warunki, nie uwzględniając wielkości znaków:

```

private SearchParameters.ResultType getResultType(String searchType) {
    for (SearchParameters.ResultType knownType :
        SearchParameters.ResultType.values()) {
        if (knownType.name().equalsIgnoreCase(searchType)) {
            return knownType;
        }
    }
    return SearchParameters.ResultType.RECENT;
}

```

Teraz utwórz następującą metodę typu SearchParameters:

```
private SearchParameters createSearchParam(String searchType, String taste) {
    SearchParameters.ResultType resultType = getResultType(searchType);
    SearchParameters searchParameters = new SearchParameters(taste);
    searchParameters.setResultType(resultType);
    searchParameters.count(3);
    return searchParameters;
}
```

Teraz utworzenie konstruktora typu lista obiektów SearchParameters jest proste i polega jedynie na wykonaniu operacji mapowania (pobrania listy słów kluczowych i zwrócenia dla każdego z nich konstruktora typu SearchParameters):

```
List<SearchParameters> searches = keywords.stream()
    .map(taste -> createSearchParam(searchType, taste))
    .collect(Collectors.toList());
```

Teraz trzeba odczytać tweety z każdego konstruktora SearchParameters. Można to zrobić w następujący sposób:

```
List<Tweet> tweets = searches.stream()
    .map(params -> twitter.searchOperations().search(params))
    .map(searchResults -> searchResults.getTweets())
    .collect(Collectors.toList());
```

Jeżeli jednak się zastanowisz, odkryjesz, że powyższy kod zwraca listę tweetów. Tobie jednak potrzebna jest prosta, „płaska” lista. Operację przetwarzania mapy i „spłaszczenia” wyniku realizuje metoda flatMap. Można więc utworzyć następujący kod:

```
List<Tweet> tweets = searches.stream()
    .map(params -> twitter.searchOperations().search(params))
    .flatMap(searchResults -> searchResults.getTweets().stream())
    .collect(Collectors.toList());
```

Parametrem metody flatMap jest strumień, co na początku może wydawać się niezrozumiałe. Przedstawię najpierw pełny kod klasy SearchService, aby uzyskać całościowy obraz:

```
package masterSpringMvc.search;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.SearchParameters;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.social.twitter.api.Twitter;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.stream.Collectors;

@Service
public class SearchService {
```

```

private Twitter twitter;

@Autowired
public SearchService(Twitter twitter) {
    this.twitter = twitter;
}

public List<Tweet> search(String searchType, List<String> keywords) {
    List<SearchParameters> searches = keywords.stream()
        .map(taste -> createSearchParam(searchType, taste))
        .collect(Collectors.toList());
    List<Tweet> results = searches.stream()
        .map(params -> twitter.searchOperations()
            .search(params))
        .flatMap(searchResults -> searchResults.getTweets())
        .stream()
        .collect(Collectors.toList());
    return results;
}

private SearchParameters.ResultType getResultType(String searchType) {
    for (SearchParameters.ResultType knownType :
        SearchParameters.ResultType.values()) {
        if (knownType.name().equalsIgnoreCase(searchType)) {
            return knownType;
        }
    }
    return SearchParameters.ResultType.RECENT;
}

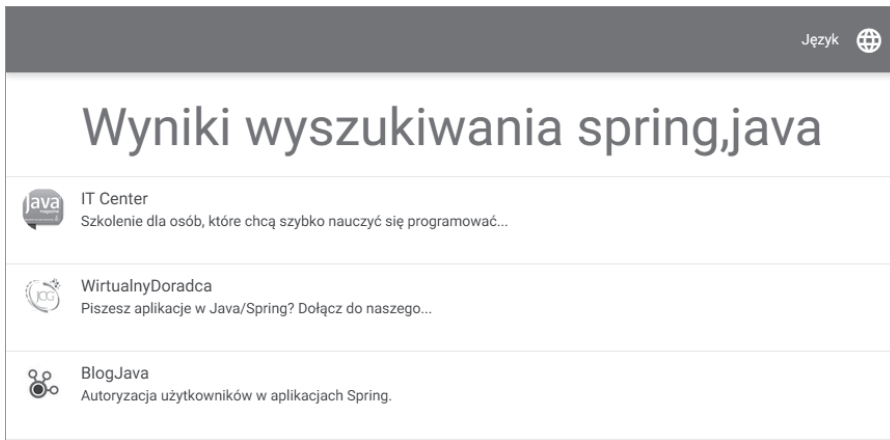
private SearchParameters createSearchParam(String searchType, String taste) {
    SearchParameters.ResultType resultType = getResultType(searchType);
    SearchParameters searchParameters = new SearchParameters(taste);
    searchParameters.resultType(resultType);
    searchParameters.count(3);
    return searchParameters;
}
}

```

Teraz, po otwarciu adresu <http://localhost:8080/search/mixed;keywords=spring,java>, pojawi się spodziewany efekt, tj. najpierw wyniki wyszukania słowa *Spring*, a następnie *Java* (patrz rysunek na następnej stronie).

Wszystko razem

Do tej pory poszczególne funkcjonalności działały osobno, więc teraz pora je zebrać wszystkie razem. Wykonaj następujące kroki:



1. Do strony profilu przenieś formularz ze strony do ładowania obrazów, a następnie usuń ją.
2. Zmień przycisk *Wyślij* na stronie profilu tak, aby od razu rozpoczynał wyszukiwanie tweetów według preferencji użytkownika.
3. Zmień stronę startową aplikacji tak, aby od razu pojawiały się na niej wyniki wyszukiwania spełniające preferencje użytkownika. Jeżeli preferencje nie będą zdefiniowane, powinna pojawić się strona profilu.

Zachęcam Cię do samodzielnego przygotowania kodu. Być może po drodze napotkasz bardzo poważne problemy, ale posiadasz już wystarczającą wiedzę, aby je samodzielnie rozwiązać. Wierzę w Ciebie.

OK. Jeżeli wykonałeś zadanie (na pewno tak), rzuć okiem na moje rozwiązanie.

Pierwszym krokiem było usunięcie starej strony *uploadPage.html*. Śmiało, zrób to.

Następnie tuż po elemencie tytułu na stronie *profilePage.html* wpisałem poniższe wiersze:

```
<div class="row">
  <div class="col m8 s12 offset-m2">
    
  </div>
  <div class="col s12 center red-text" th:text="{error}" th:if="{error}">
    Błąd ładowania pliku
  </div>
  <form th:action="@{/profile}" method="post" enctype="multipart/form-data"
    class="col m8 s12 offset-m2">
    <div class="input-field col s6">
      <input type="file" id="file" name="file"/>
    </div>
    <div class="col s6 center">
```



```

<button class="btn indigo waves-effect waves-light" type="submit"
↳name="upload"
  th:text="#{upload}">Załaduj
  <i class="mdi-content-send right"></i>
</button>
</div>
</form>
</div>

```

Powyższy kod jest bardzo podobny do kodu poprzedniej strony *uploadPage.html*. Usunąłem tylko tytuł i zmieniłem etykietę na przycisku do wysyłania formularza. W plikach ustawień regionalnych umieściłem też odpowiednie tłumaczenia napisów:

Dla języka polskiego:

```
upload=Załaduj
```

Dla języka angielskiego:

```
upload=Upload
```

Zmieniłem również nazwę przycisku na `upload`. Dzięki temu łatwiej będzie można zidentyfikować operację ładowania pliku po stronie kontrolera.

Teraz przy próbie załadowania obrazu użytkownik zostanie przekierowany na starą stronę do ładowania plików. Musiałem poprawić kod metody `onUpload` w klasie `PictureUploadController`:

```

@RequestMapping(value = "/profile", params = {"upload"}, method =
↳RequestMethod.POST)
public String onUpload(@RequestParam MultipartFile file,
  RedirectAttributes redirectAttrs) throws IOException {
  if (file.isEmpty() || !isImage(file)) {
    redirectAttrs.addFlashAttribute("error",
      "Niewłaściwy plik. Załaduj plik z obrazem.");
    return "redirect:/profile";
  }
  Resource picturePath = copyFileToPictures(file);
  userProfileSession.setPicturePath(picturePath);
  return "redirect:/profile";
}

```

Zwróć uwagę, że adres URL obsługujący zapytanie POST został zmieniony z */upload* na */profile*. Obsługa formularza jest o wiele prostsza, jeżeli zapytania GET i POST dotyczą tego samego adresu URL. W szczególności oszczędzają mnóstwa kłopotów przy obsłudze wyjątków, ponieważ po wystąpieniu błędu nie trzeba użytkownika przekierowywać na inną stronę.

Usunąłem również atrybut `picturePath` modelu danych. Ponieważ jest teraz klasa `UserProfile` ↳`Session` reprezentująca sesję użytkownika, więc została tutaj wykorzystana. Atrybut `picturePath` dodałem do klasy `UserProfileSession` wraz z odpowiednimi metodami przypisującymi i odczytującymi wartości właściwości.

Nie mogłem zapomnieć o zastosowaniu klasy `UserProfileSession` i udostępnieniu jej jako pola klasy `PictureUploadController`.

Pamiętaj, że wszystkie właściwości klasy reprezentującej sesję muszą — w odróżnieniu od zasobów — nadawać się do serializacji. Dlatego ich wartości należy przechowywać w inny sposób. Klasa `URL` wydaje się dobrze nadawać do tego celu. Można ją serializować, a za pomocą klasy `UrlResource` można utworzyć zasób na podstawie adresu `URL`:

```
@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class UserProfileSession implements Serializable {
    private URL picturePath;
    public void setPicturePath(Resource picturePath) throws IOException {
        this.picturePath = picturePath.getURL();
    }
    public Resource getPicturePath() {
        return picturePath == null ? null : new
            UrlResource(picturePath);
    }
}
```

Ostatnią rzeczą, jaką zrobiłem, było udostępnienie obiektu `profileForm` jako atrybutu modelu po wystąpieniu błędu, ponieważ wymaga tego strona `profilePage.html` w momencie wyświetlenia.

Poniżej przedstawiona jest ostateczna wersja klasy `PictureUploadController`:

```
package masterSpringMvc.profile;

import masterSpringMvc.config.PictureUploadProperties;
import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import javax.servlet.http.HttpServletResponse;
import java.io.*;
import java.net.URLConnection;
import java.util.Locale;

@Controller
public class PictureUploadController {
    private final Resource picturesDir;
```

```

private final Resource anonymousPicture;
private final MessageSource messageSource;
private final UserProfileSession userProfileSession;

@Autowired
public PictureUploadController(PictureUploadProperties uploadProperties,
    MessageSource messageSource, UserProfileSession userProfileSession) {
    picturesDir = uploadProperties.getUploadPath();
    anonymousPicture = uploadProperties.getAnonymousPicture();
    this.messageSource = messageSource;
    this.userProfileSession = userProfileSession;
}

@RequestMapping(value = "/uploadedPicture")
public void getUploadedPicture(HttpServletRequest response) throws IOException {
    Resource picturePath = userProfileSession.getPicturePath();
    if (picturePath == null) {
        picturePath = anonymousPicture;
    }
    response.setHeader("Content-Type",
        URLConnection.guessContentTypeFromName(picturePath.getFilename()));
    IOUtils.copy(picturePath.getInputStream(), response.getOutputStream());
}

@RequestMapping(value = "/profile", params = {"upload"}, method =
↳RequestMethod.POST)
public String onUpload(@RequestParam MultipartFile file,
    RedirectAttributes redirectAttrs) throws IOException {
    if (file.isEmpty() || !isImage(file)) {
        redirectAttrs.addFlashAttribute("error",
            "Niewłaściwy plik. Załaduj plik z obrazem.");
        return "redirect:/profile";
    }
    Resource picturePath = copyFileToPictures(file);
    userProfileSession.setPicturePath(picturePath);
    return "redirect:profile";
}

private Resource copyFileToPictures(MultipartFile file) throws IOException {
    String fileExtension = getFileExtension(file.getOriginalFilename());
    File tempFile = File.createTempFile("pic", fileExtension,
↳picturesDir.getFile());
    try (InputStream in = file.getInputStream();
        OutputStream out = new FileOutputStream(tempFile)) {
        IOUtils.copy(in, out);
    }
    return new FileSystemResource(tempFile);
}

```

```

    @ExceptionHandler(IOException.class)
    public ModelAndView handleIOException(Locale locale) {
        ModelAndView modelAndView = new ModelAndView("profile/profilePage");
        modelAndView.addObject("error", messageSource.getMessage("upload.
        ↳io.exception",
            null, locale));
        modelAndView.addObject("profileForm", userProfileSession.toForm());
        return modelAndView;
    }

    @RequestMapping("uploadError")
    public ModelAndView onUploadError(Locale locale) {
        ModelAndView modelAndView = new ModelAndView("profile/profilePage");
        modelAndView.addObject("error", messageSource.getMessage("upload.file.
        ↳too.big",
            null, locale));
        modelAndView.addObject("profileForm", userProfileSession.toForm());
        return modelAndView;
    }

    private boolean isImage(MultipartFile file) {
        return file.getContentType().startsWith("image");
    }

    private static String getFileExtension(String name) {
        return name.substring(name.lastIndexOf("."));
    }
}

```

Teraz więc użytkownik może otworzyć stronę profilu i załadować obraz, jak również wprowadzić informacje osobiste (patrz rysunek na następnej stronie).

Użytkownik po wpisaniu danych na stronie profilu powinien zostać przekierowany na stronę z wynikami wyszukiwania. W tym celu zmieniłem metodę `saveProfile` w klasie `ProfileController`:

```

    @RequestMapping(value = "/profile", params = {"save"}, method =
    ↳RequestMethod.POST)
    public String saveProfile(@Valid ProfileForm profileForm, BindingResult
    ↳bindingResult) {
        if (bindingResult.hasErrors()) {
            return "profile/profilePage";
        }
        userProfileSession.saveForm(profileForm);
        return "redirect:/search/mixed;keywords=" + String.join(",",
        profileForm.getTastes());
    }
}

```

Ponieważ teraz można wyszukiwać tweety na stronie profilu, nie są już potrzebne pliki *searchPage.html* i *TweetController.java*. Można je po prostu usunąć.

Na koniec trzeba zmienić stronę startową tak, aby użytkownik był kierowany na stronę wyszukiwania tweetów spełniających preferencje, o ile zostały podane na stronie profilu.

W pakiecie *controller* utworzyłem klasę nowego kontrolera, który jest odpowiedzialny za przekierowanie użytkownika do głównej strony aplikacji lub do strony profilu w celu uzupełnienia danych, lub do strony *resultPage.html*, jeżeli wpisane zostały preferencje:

```
package masterSpringMvc.controller;

import masterSpringMvc.profile.UserProfileSession;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import java.util.List;

@Controller
public class HomeController {
    private UserProfileSession userProfileSession;

    @Autowired
    public HomeController(UserProfileSession userProfileSession) {
        this.userProfileSession = userProfileSession;
    }
}
```

```

    }

    @RequestMapping("/")
    public String home() {
        List<String> tastes = userProfileSession.getTastes();
        if (tastes.isEmpty()) {
            return "redirect:/profile";
        }
        return "redirect:/search/mixed;keywords=" + String.join(",", tastes);
    }
}
}

```

Punkt kontrolny

W tym rozdziale utworzyłeś dwa kontrolery: kontroler `PictureUploadController`, odpowiedzialny za zapisywanie załadowanych plików obrazów na dysk i obsługę błędów, oraz kontroler `SearchController`, wyszukujący tweety za pomocą parametrów tablicowych zawierających listę słów kluczowych.

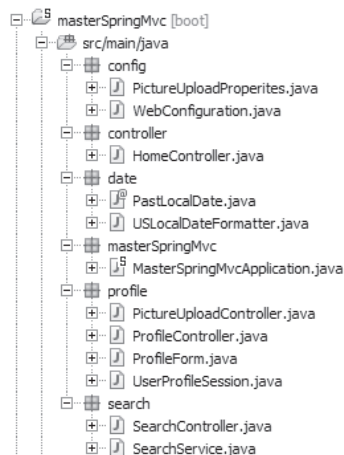
Kontroler ten przekazuje wyszukiwanie do nowej klasy `SearchService`.

Usunąłeś niepotrzebny kontroler `TweetController`.

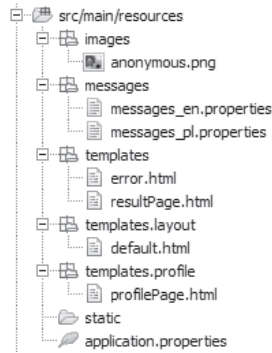
Utworzyłeś klasę `UserProfileSession` przechowującą informację o użytkowniku aplikacji.

Na koniec dodałeś do klasy `WebConfiguration` dwie metody: jedną, dla kontenera serwetów do wyświetlania strony z komunikatem o błędzie, i drugą, obsługującą zmienne tablicowe.

Struktura projektu wygląda teraz jak na poniższym rysunku:



Do zasobów zostały dodane obraz przedstawiający anonimowego użytkownika oraz statyczna strona wyświetlająca komunikat o błędzie. Strona *profilePage.html* została rozbudowana o możliwość ładowania plików, natomiast strona *searchPage.html* została usunięta. Strukturę zasobów przedstawia poniższy rysunek:



Podsumowanie

W tym rozdziale opisane zostały metody ładowania plików i obsługi błędów. Ładowanie plików w rzeczywistości nie jest skomplikowane. Ważna jest jednak decyzja, co należy zrobić z załadowanymi plikami. Można je przechowywać w bazie danych jako obrazy, ale w tym przypadku zostały zapisane na dysku, a informacje o ich położeniu były przechowywane w sesji użytkownika.

Opisane zostały typowe metody obsługi wyjątków na poziomie kontrolera i kontenera serwletów. Dodatkowe informacje na temat obsługi błędów w środowisku Spring MVC są dostępne pod adresem <https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc>.

Twoja aplikacja wygląda już całkiem niezłe, a ilość napisanego kodu jest dość duża.

Zapoznaj się z następnym rozdziałem, w którym dowiesz się, że platforma Spring MVC doskonale nadaje się również do tworzenia aplikacji w stylu REST.

Skorowidz

A

adnotacja

- @Async, 244
- @Cacheable, 227
- @CacheConfig, 230
- @CachePut, 230
- @CacheEvict, 230
- @Caching, 230
- @ControllerAdvice, 152
- @EnableCache, 226
- @EnableGlobalMethodSecurity, 161
- @JsonIgnore, 140
- @JsonIgnoreProperties, 140
- @ModelAttribute, 107
- @Primary, 194
- @RequestMapping, 69, 138
- @RestController, 138
- @SessionAttributes, 113
- @SpringApplicationConfiguration, 194
- @SpringBootApplication, 33
- @WebIntegrationTest, 206
- javax.validation.Valid, 81

adnotacje do weryfikacji danych, 85

adres URL, 73

- uprawniony, 163
- zmienne tablicowe, 117

aktywacja usługi Redis, 252, 258

anemiczny model domeny, 48

API, 135

aplikacja Gradle, 23

aplikacje

- jednostronicowe, 240, 264
- Spring, 19
- w stylu REST, 131

architektura

- MVC, 47
- platformy Spring MVC, 54
- REST, 131

atrapy, 191

atrybut

- dateFormat, 80
- request, 69

atrybuty redirectAttributes, 69

B

BDD, Behaviour Driven Development, 187

bezstanowość, 266

biblioteka

- AngularJS, 265
- AssertJ, 206
- Backbone.js, 240
- FluentLenium, 206, 209
- Geb, 215, 217
- hamcrest, 188
- Jolokia, 46
- mockito, 188
- NIO, 101
- Spock, 212, 214
- spring-test, 188
- WebJar, 241
- WebJars, 63, 72

błąd 400, 77

błędy, 40

- formularza, 81
- ładowania plików, 108

C

certyfikat z własnym podpisem, 179
 chmura, 245
 ciasteczko, 113
 ciągła integracja, 183
 Cloud Foundry, 246
 CRUD, Create Read Update Delete, 133
 CSRF, Cross Site Request Forgery, 164

D

debugowanie, 32
 deklaracja lokalizacji, 39
 diagnostyka interfejsu REST API, 139
 diagnozowanie błędów, 111
 dokumentowanie interfejsu, 153
 dostęp do serwisu Twitter, 60
 dostosowanie

- komunikatów o błędach, 82
- odpowiedzi JSON, 139

 DSL, Domain Specific Language, 214
 DTO, Data Transfer Object, 75
 dyspozytor, 35

F

format

- JSON, 139
- MIME, 103

 formularz, 73

- logowania, 165

 funkcja displayTweets, 243
 funkcje lambda, 62

G

GDK, Groovy Development Kit, 212
 generowanie certyfikatu, 179

H

HATEOAS, 134
 Heroku, 247, 253
 HTTP, 132

I

IaaS, Infrastructure as a Service, 245
 imitowanie klas, 191
 instalacja

- narzędzi konsolowych, 247
- narzędzia Gradle, 29
- systemu Git, 28

 IntelliJ IDEA, 19, 25
 interceptorzy, 88
 interfejs

- API, 135, 144
- CacheManager, 244
- java.util.Date, 76
- MultipartFile, 101
- REST API, 139
- SessionStrategy, 175
- UsersConnectionRepository, 175

 internacjonalizacja, 85
 interpreter widoków, 38

J

Java 8, 62
 JavaScript, 237
 Język

- DSL, 214
- Groovy, 212
- SpEL, 56

K

kanał

- HTTP, 180
- HTTPS, 179

 klasa

- ApiSecurityConfiguration, 166, 225
- AuthenticatingSignInAdapter, 171
- Bean, 154
- CacheConfiguration, 226, 244
- CompletableFuture, 237
- ContentNegotiatingViewResolver, 154
- DefaultMessageCodesResolver, 84
- DefaultResourceLoader, 106
- ErrorController, 117
- FixedLocaleResolver, 86
- HomeController, 188

LocalDate, 76
 LocalDateTimeSerializer, 143
 LocaleChangeInterceptor, 88
 MasterSpringMvcApplication, 24, 71
 MasterSpringMvcApplicationTests, 24
 MBean, 38
 MessageSource, 112
 MockHttpSession, 190
 MultipartProperties, 101
 PictureUploadController, 106, 124
 ProfileController, 93, 100, 126
 ProviderSignInController, 174
 RedirectAttributes, 69
 RedisConfig, 177
 SearchApiController, 137, 156, 195
 SearchCache, 244
 SearchControllerMockTest, 193
 SearchParamsBuilder, 227
 SearchService, 141, 193
 SessionLocaleResolver, 87
 SignupController, 175
 SimpMessagingTemplate, 242
 SocialWebAutoConfiguration, 174
 StubConfiguration, 194
 Tweet, 140
 TwitterSearch, 230, 234
 UrlResource, 124
 UserApiController, 156, 200
 UserProfileSession, 113, 115, 128
 UserRepository, 144, 198
 WebConfiguration, 79, 110, 128, 143
 WebMvcConfigurerAdapter, 79
 WebSecurityConfiguration, 166, 168
 klasy konfiguracyjne, 181, 244
 klient, 137
 REST, 139
 kodowanie znaków, 40
 kody
 błędów, 148
 HTTP, 136
 stanu, 147
 kompilowanie kodu, 28, 184
 kompresja gzip, 224
 komunikat o błędzie, 78, 82, 112
 400, 77
 konfiguracja
 aplikacji, 254
 elementów aplikacji, 35
 interpretera widoków, 39

ładowania plików, 104
 obsługi błędów, 40
 protokołu SSL, 44
 systemu Gradle, 202
 uwierzytelnienia społecznościowego, 170
 wbudowanego serwletu, 42
 kontrola pamięci podręcznej, 224
 kontroler, 48
 PictureUploadController, 128
 ProviderSignInController, 174
 SearchApiController, 193
 SearchController, 128, 193
 TweetController, 128
 kontrolki hipermediów, 134

L

lista w formularzu, 91
 logowanie, 165
 lokalizacja, 39

Ł

ładowanie
 obrazu, 102
 plików, 99, 104, 108

M

magazyn kluczy, keystore, 179
 metadane, 76
 metoda
 asyncFetch, 235
 await, 235
 CountDownLatch, 235
 DELETE, 133, 199
 GET, 133
 getInputStream, 101
 HEAD, 133
 HttpServletRequest.sendError, 149
 onUpload, 107
 OPTIONS, 133
 PATCH, 133
 POST, 133
 PUT, 133
 search, 138
 toString, 76
 TwitterTemplate, 229

metody
 asynchroniczne, 233
 HTTP, 133
 MIME, 103
 model, 48
 dojrzałości Richardsona, 132
 MVC, model-view-controller, 47

N

narzędzia konsolowe Cloud Foundry, 247
 narzędzie
 AssertJ, 187
 BackboneJS, 265
 DbUnit, 187
 Ember, 265
 geb, 20
 Gradle, 19, 28–30, 254
 httpie, 139
 JUnit, 187
 Maven, 19
 Mockito, 187, 191
 React, 265
 Spock, 187

O

obiekt
 DTO, 75
 Map, 144
 ObjectMapper, 143
 POJO, 76
 ResponseEntity, 148
 obiekty
 MBean, 33
 obiekty stron, 209, 217
 obsługa
 adresów URL, 73
 błędów, 40, 103, 149
 błędów ładowania plików, 108
 formularzy, 73
 sesji rozproszonych, 176
 wyjątków, 147, 152
 odczytywanie danych, 56
 odgałęzienie, branch, 254
 opcje konfiguracyjne, 45
 OpenShift, 246
 operacja
 forward, 69
 operacja redirect, 69

operacje
 CRUD, 146
 POST, 94
 optymalizacja zapytań, 223

P

PaaS, Platform as a Service, 245
 pakiet
 authentication, 168, 182
 config, 177
 ConfigurationProperties, 105
 GDK, 212
 hibernate-validator, 81
 java.util.concurrent.Executor, 233
 JDK, 179, 250
 org.springframework.web, 78
 utils, 199
 pamięć podręczna, 224
 aplikacji, 226
 rozproszona, 232
 unieważnianie danych, 231
 parametr
 proxyMode, 115
 search, 67
 platforma
 Docker, 263
 MVC 1-0-1, 50
 Spring, 261
 Spring Boot, 19, 263
 Swagger, 153
 plik
 application.properties, 24, 71, 78, 84, 104, 181
 application-cloud.properties, 252
 build.gradle, 30, 60, 142, 203
 default.html, 164
 JAR, 249
 JSESSIONID, 113
 messages_pl.properties, 86
 Procfile, 255
 profilePage.html, 76
 resultPage.html, 142, 239
 searchPage.html, 127
 TweetController.java, 127
 pliki
 JAR, 23
 WAR, 23
 POJO, Plain Old Java Object, 76

- polecenie
 - brew, 253
 - git add, 29
 - heroku login, 253
- port HTTP, 44
- produkcyjny profil aplikacji, 223
- profil
 - aplikacji, 223
 - async, 237
 - Heroku, 255
 - prod, 252
 - redis, 252
 - użytkownika, 198
- program
 - keytool, 179
 - PhantomJS, 206
- programowanie zorientowane na testy, 185
- projekt
 - React, 265
 - Spring Batch, 263
 - Spring Integration, 263
 - Spring Reactor, 263
 - Spring Social, 60
- projektowanie domenowe, 49
- protokół
 - SSL, 44, 178, 181
 - WebSocket, 241, 242
- przekazywanie danych do widoku, 55
- przekierowanie, 69
- przeladowywanie widoków, 53
- PWS, Pivotal Web Services, 247

R

- rdzeń platformy Spring, 262
- Redis, 252, 258
- rejestracja aplikacji, 58
- rejestrowanie informacji, 78
- repozytorium Git, 254
- REST, Representational State Transfer, 131
- rozproszenie pamięci, 232

S

- serializacja, 140, 155
- serwer
 - Redis, 244
 - Tomcat, 29, 42

- serwis
 - Getting Started Content, 22
 - start.Spring.io, 26
 - Twitter, 60
- serwlet DispatcherServlet, 54
- sesje
 - przyklejane, 132
 - rozproszone, 176
- skrypt materialize, 239
- SpEL, Spring Expression Language, 56
- sprawdzanie uprawnień użytkownika, 165
- Spring
 - core, rdzeń, 262
 - data, dane, 262
 - execution, uruchamianie, 262
- Spring Boot, 19, 34
- Spring Boot Plugin, 31
- Spring Initializr, 25
- Spring Social, 60
- Spring Tool Suite, 20
- SSL, Secure Sockets Layer, 178
- strona
 - login.html, 167, 168
 - profilePage.html, 124, 129
 - resultPage.html, 243
- strumienie, 62
- STS, Spring Tool Suite, 20
- styl material design, 63
- Swagger, 153
- system
 - Git, 20, 27
 - Gradle, 202
 - Thymeleaf, 53
- szablon Thymeleaf, 51, 164, 241

Ś

- ścieżka
 - classpath, 197
 - twitterSearch, 242
- środowisko
 - IDE, 20
 - IntelliJ, 25

T

- tagi ETag, 237, 238
- test, 183, 184
 - FluentLenium, 204

testowanie

- metody DELETE, 199
- uwierzytelnienia, 201

testy

- integracyjne, 187, 215
- jednostkowe, 186, 212
- jednostkowe kontrolerów REST, 195

tłumaczenie tekstów aplikacji, 89

tweety, 235, 239

Twitter, 60, 170

tworzenie

- aplikacji Spring, 19
- atrap, 193
- odpowiedzi XML, 154
- profilu, 198
- strony, 52
- testów integracyjnych, 202
- testów jednostkowych, 188
- testów w języku Groovy, 212

typ

- ListenableFuture, 234
- ProfileForm, 77

U

udostępnianie aplikacji, 245, 253

układy stron, 66

ulepszanie aplikacji, 259

uprawnione adresy URL, 163

URI, unified resource identifier, 132

uruchomienie aplikacji, 256

usługa

- Cloud Foundry, 246
- Heroku, 247, 253
- OpenShift, 246
- PWS, 247, 248
- Redis, 252, 258
- REST, 131

ustawienia regionalne, 38, 87

uwierzytelnienie, 201

- podstawowe, 159
- przez Twitter, 170

użycie parametru, 56

użytkownik

- anonimowy, 169
- upoważniony, 160

W

wdrożenie aplikacji, 248, 263

weryfikacja danych, 75, 80, 85

po stronie klienta, 94

weryfikator, matcher, 189

widok, 48

wiersz poleceń, 26

własny podpis, 179

wstępne przetwarzanie wyrażenia, 92

wstrzykiwanie pól, 115

wtyczka Groovy-Eclipse, 20

wyjątek

- ArrayOutOfBoundsException, 61
- EntityNotFoundException, 149–152
- IOException, 109
- MultipartException, 102, 109, 110

wykonywalny plik JAR, 32

wyniki wyszukiwania, 208

wyszukiwanie tweetów, 71

wyświetlanie

- komunikatów, 116
- tweetów, 67
- załadowanego obrazu, 107

Z

zabezpieczanie aplikacji, 159

zapisywanie

- danych w sesji, 114
- hasła, 180
- profilu użytkownika, 112

zapytanie

- GET, 123
- POST, 123

zarządzanie

- certyfikatami, 179
- zasobami użytkowników, 144

zasoby, 132

- statyczne, 38

zgłaszanie wyjątków, 109

zmiana ustawień regionalnych, 87

zmiennie

- środowiskowe, 180
- tablicowe, 117

znaczniki bezpieczeństwa, 164

zwrot kodów stanu, 148, 149

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>

Spring MVC 4

Projektowanie zaawansowanych aplikacji WWW

Wszyscy jesteśmy świadkami dynamicznego rozwoju branży aplikacji internetowych. Projektanci i programiści muszą jeszcze szybciej tworzyć coraz doskonalsze i atrakcyjniejsze aplikacje, a następnie błyskawicznie udostępniać je użytkownikom, przy dość ograniczonym budżecie. Platforma Spring Boot i środowiska chmurowe pozwalają sprostać tym wymaganiom: niezwykle aplikacje można tworzyć i przekazywać w rekordowym tempie, w dodatku wyposażone w tak istotne funkcjonalności jak internacjonalizacja, sesje rozproszone, logowanie społecznościowe, wielowątkowość i wiele innych.

Jeśli programujesz w Javie, choć trochę znasz platformę Spring i chcesz tworzyć użyteczne oraz nowoczesne aplikacje WWW, masz w rękę właściwą książkę. Ten podręcznik w niezwykle praktyczny sposób podchodzi do zagadnienia budowy skomplikowanych aplikacji z wykorzystaniem nowoczesnych technologii. Podczas lektury poszczególnych rozdziałów będziesz mógł od podstaw przyjrzeć się konstruowaniu w pełni działającej aplikacji WWW, a potem spróbować własnych sił w tej dziedzinie, z wykorzystaniem internacjonalizacji, weryfikacji formularzy oraz obsługi rozproszonych sesji i pamięci podręcznej. Dowiesz się również, jak porządnie przetestować aplikację i opublikować ją w internecie.



W tej książce znajdziesz:

- praktyczne omówienie platformy Spring Boot i Spring Tool Suite
- wyjaśnienie implementacji architektury MVC
- zasady działania aplikacji REST i wykorzystywania zapytań HTTP
- wyczerpujące omówienie zagadnień bezpieczeństwa aplikacji
- opis dobrych praktyk, takich jak testy jednostkowe i testy akceptacji, optymalizacja zapytań, metody zarządzania pamięcią podręczną

Geoffroy Warin — programuje od dziecka. Jest gorącym orędownikiem tworzenia otwartego kodu. Wierzy w ideę Software Craftsmanship (osiągania mistrzostwa w programowaniu). Jest uznanym specjalistą w dziedzinie budowania biznesowych aplikacji WWW w języku Java i entuzjastą platform Groovy oraz Spring. Po godzinach prowadzi bloga, jest szkoleniowcem i autorem oraz współautorem książek.

PACKT open source
PUBLISHING community experience distilled

Helion

44488 numer katalogowy

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne

☎ 0 801 339900

☎ 0 601 339900

Informatyka w najlepszym wydaniu

Sprawdź najnowsze promocje:

● <http://helion.pl/promocje>

● <http://helion.pl/promocje>

● <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

● <http://helion.pl/novosci>

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

ISBN 978-83-283-2347-6



9 788328 323476

cena: 49,00 zł

ślednij po WIĘCEJ



KOD KORZYŚCI