Spring Framework

moduły: MVC, WebFlow, Security, REST

autor: Andrzej Klusiewicz

www.jsystems.pl

Spis treści

Spring MVC....3
Spring MVC – pierwsza aplikacja – jak to wszystko działa....3
Położenie pliku konfiguracji Springa....26
Przekazywanie obiektów i list do warstwy widoku....27
Mapowanie na poziomie klasy....34
Zmienne ścieżki....36
Zmienne tablicowe....39
Parametry żądania....41
Przechwytywacze....44
Najprostszy formularz....47
Walidacja formularzy....52
Upload plików....54
Spring WebFlow....57
Proste formularze z przejściami....57
Zaawansowane elementy Spring WebFlow....71
Stany decyzyjne....72
Wyświetlanie danych....76
Wieloetapowe uzupełnianie obiektu podczas przepływu....78
Przejście globalne....88
Podprzepływy....90
Spring Security....97
Konfiguracja oparta o XML....97
Bazowa aplikacja....97
Zabezpieczamy aplikację....102
Użytkownicy i role przechowywane w bazie danych....105
Logowanie i wylogowywanie....109
Spring Rest....113
Obsługa żądań GET....115
Obsługa żądań POST....118

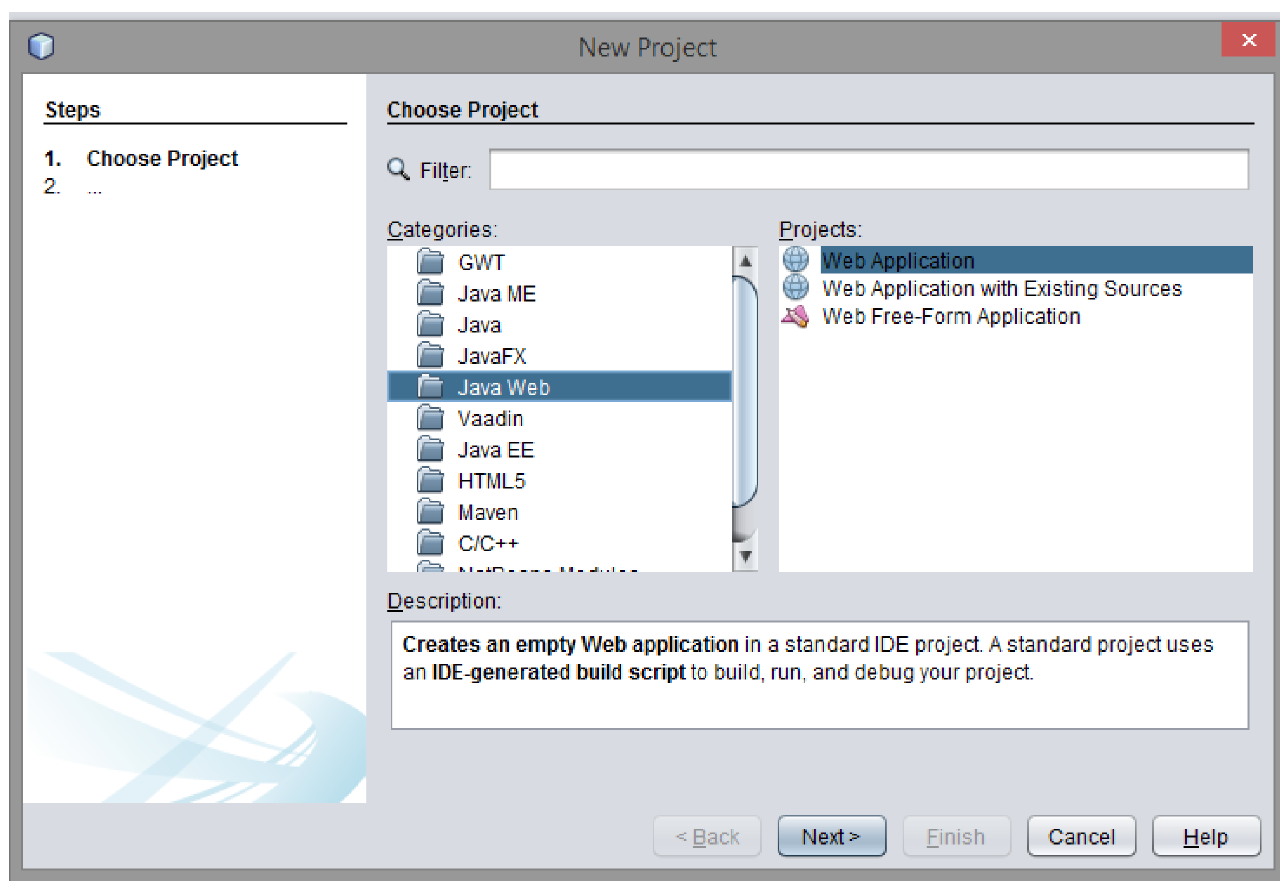
Spring MVC

Spring MVC – pierwsza aplikacja – jak to wszystko działa

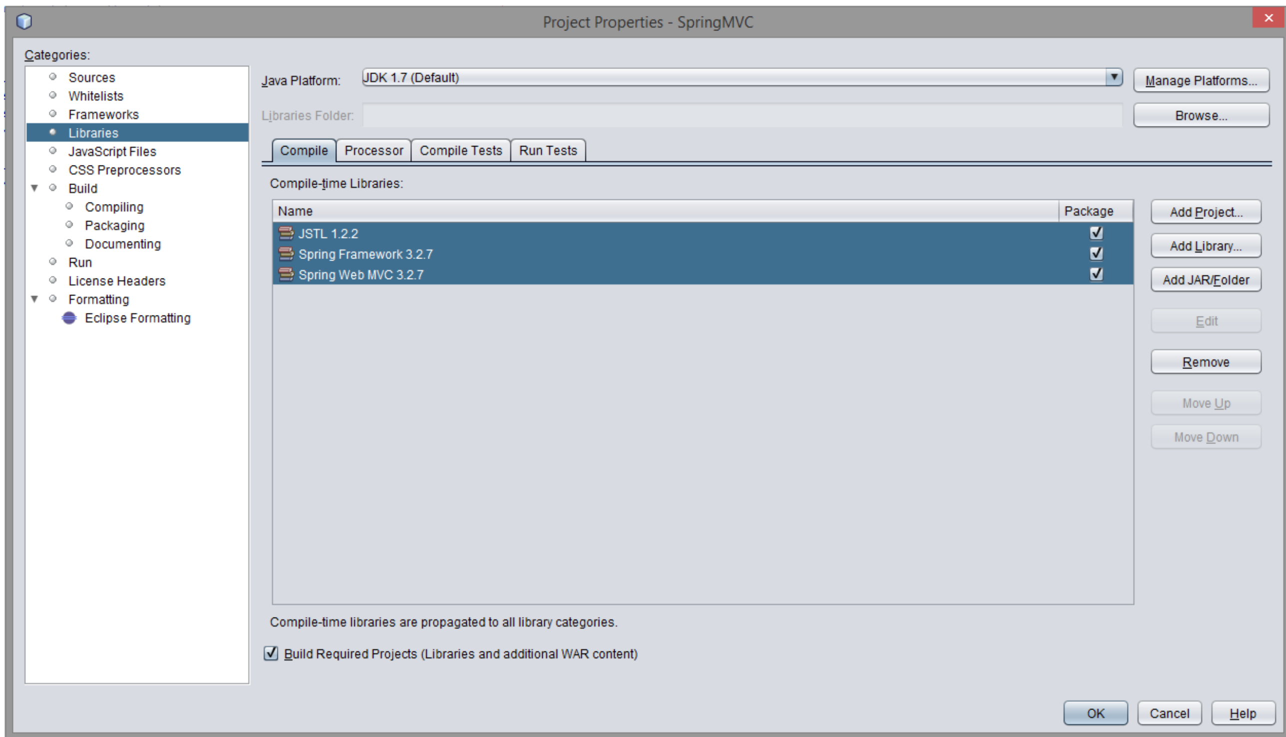
Kod źródłowy aplikacji którą tworzę w niniejszym kursie jest do pobrania z adresu:
http://www.jsystems.pl/storage/spring/springmvc1.zip

Aplikacja jest tworzona w NetBeans, a uruchamiana na serwerze Glassfish który to jest dołączany do w.w. IDE.

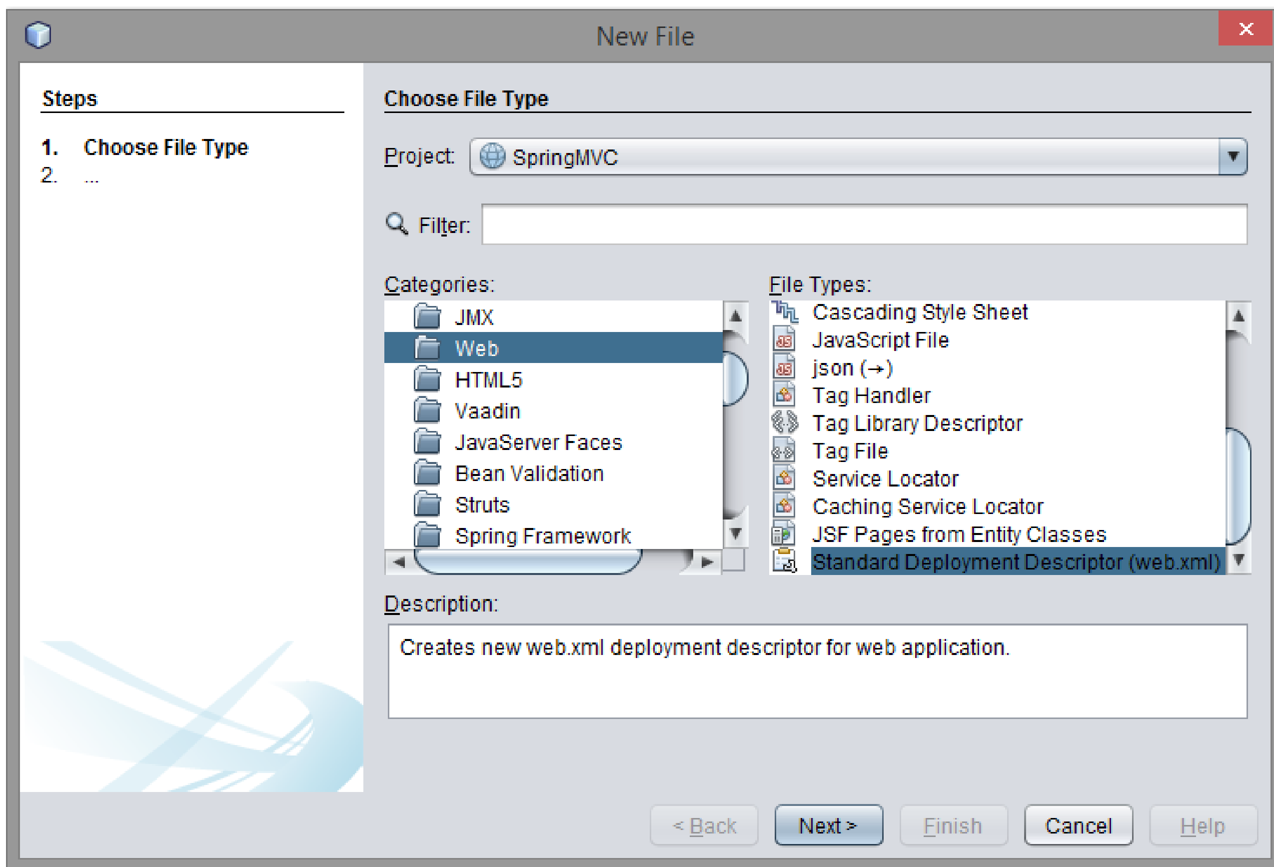
Zaczniemy od stworzenia zwykłej aplikacji WEBowej:



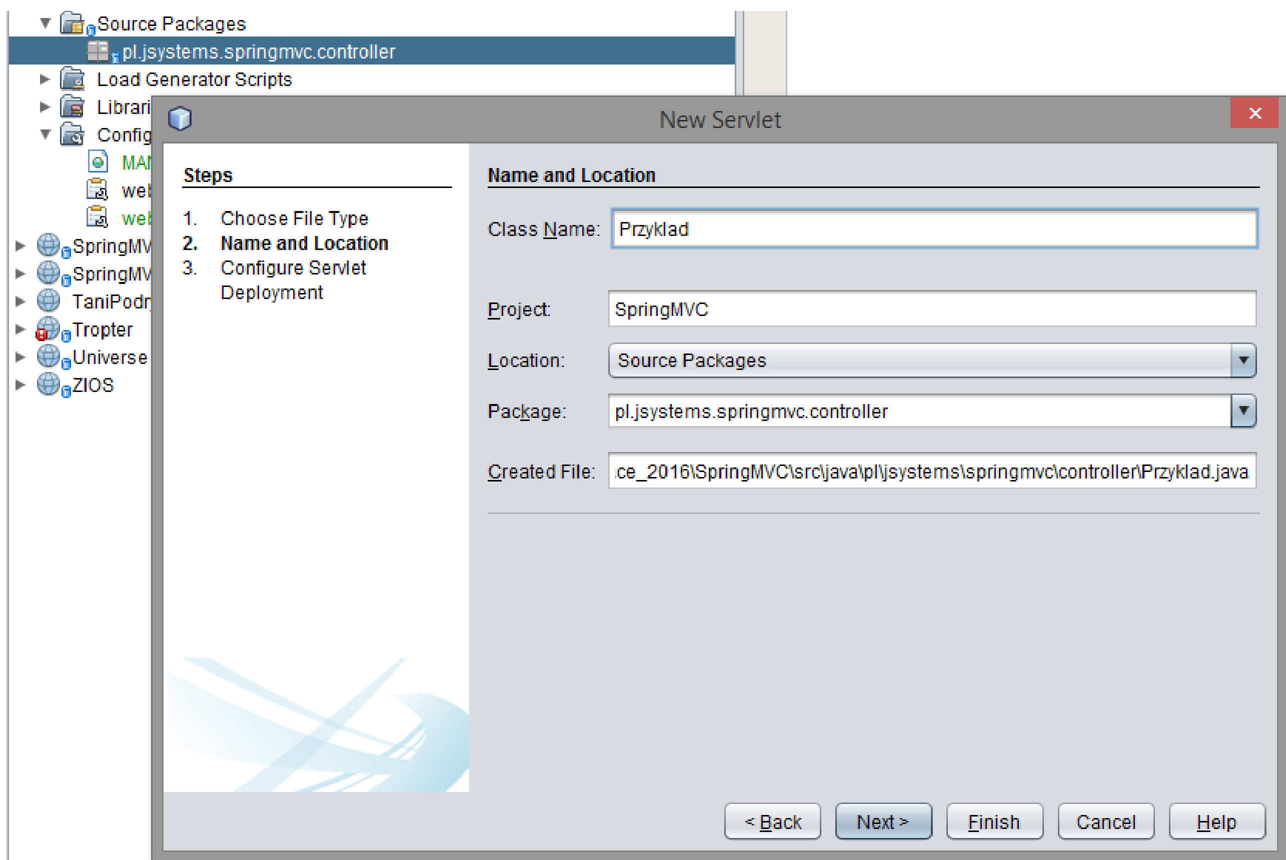
Po jej utworzeniu musimy dodać niezbędne biblioteki. W Netbeans należy wybrać właściwości projektu, przejść do sekcji „Libraries” a następnie kliknąć AddLibrary i wybrać potrzebne:



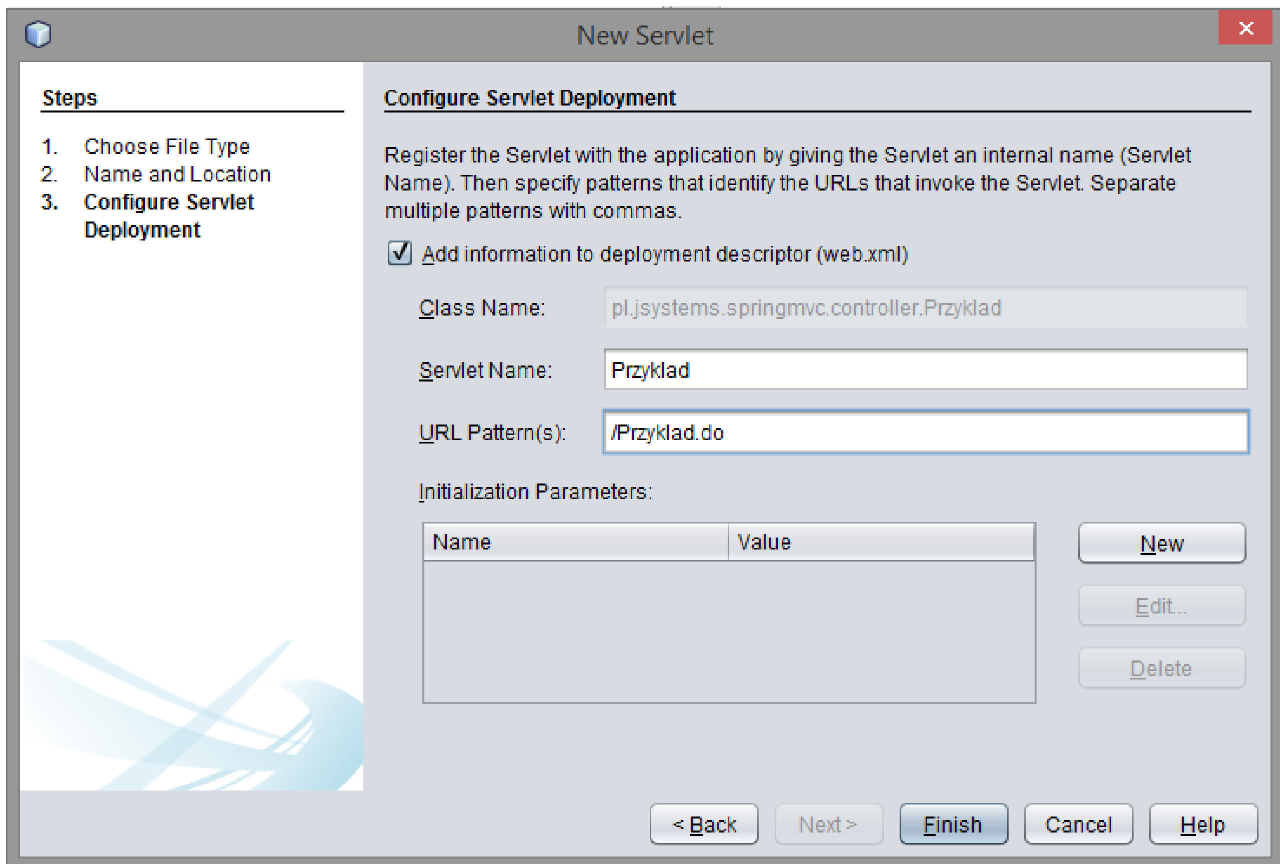
Będzie nam też potrzebny plik konfiguracyjny web.xml, dlatego dodajemy do do katalogu WEB-INF:



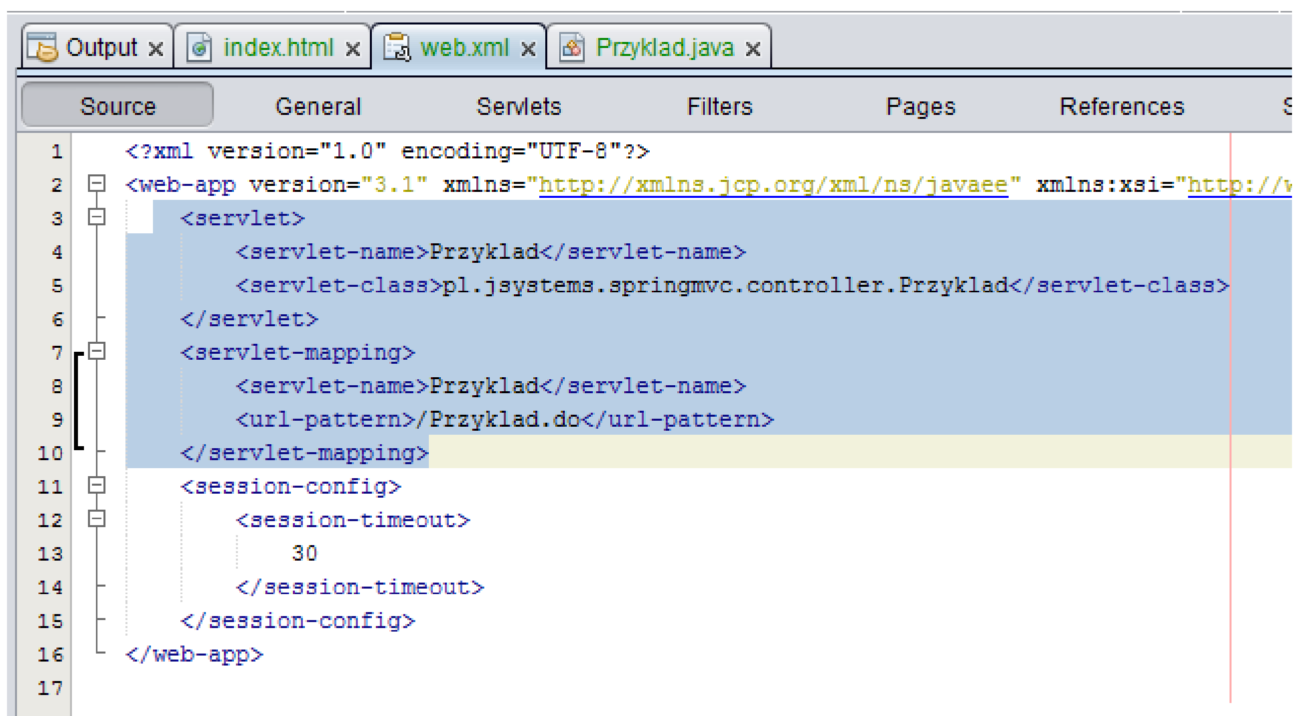
Będzie nam też potrzebny jakiś pakiet w którym umieścimy nasze klasy:



Na początek aby przyjrzeć się sposobowi przekazywania kontroli nad wywołaniami Springowi, stworzymy zwyczajny serwlet. Pamiętaj by zaznaczyć dodanie informacji o nim do web.xml!

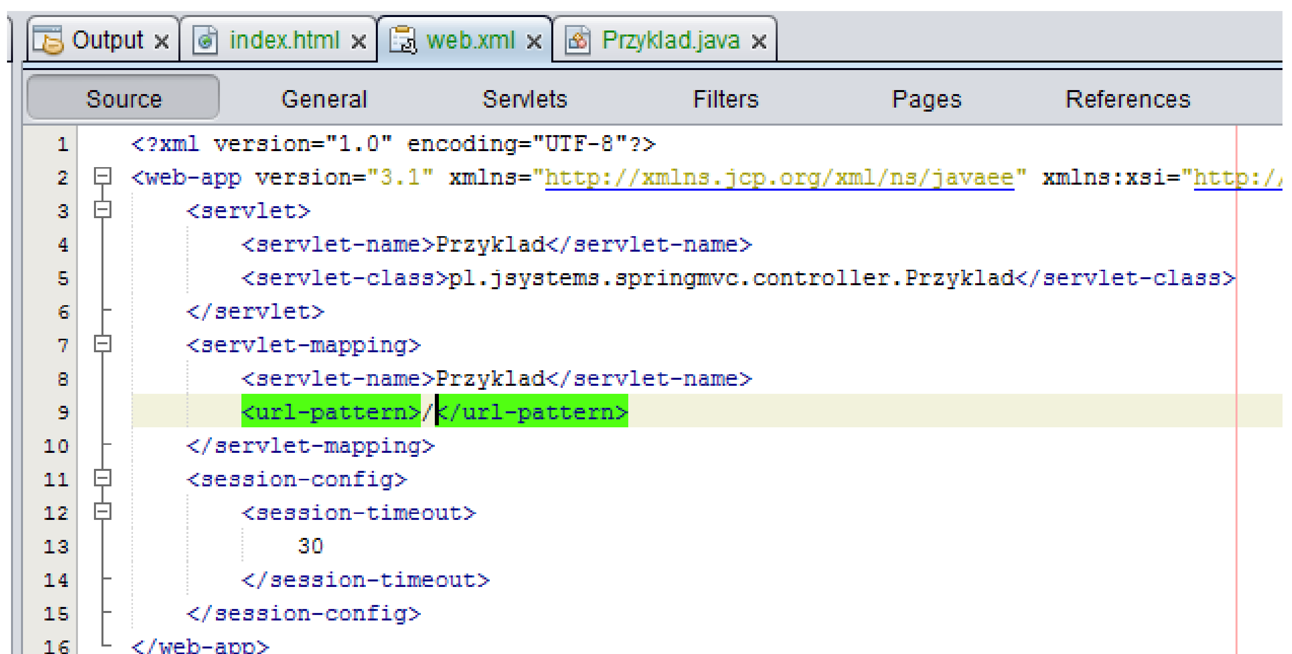


Gdy zajrzymy do web.xml po dodaniu serwletu, zobaczymy że pojawił się w nim taki oto wpis:



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://
    <servlet>
        <servlet-name>Przyklad</servlet-name>
        <servlet-class>pl.jsystems.springmvc.controller.Przyklad</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Przyklad</servlet-name>
        <url-pattern>/Przyklad.do</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

W liniach 7-9 mamy zapisane, że wywołanie podstrony „Przyklad.do” będzie obsługiwane przez nasz nowy serwlet. Nieco ten wpis przerobimy. Przyjrzyj się linii 9. Wpis „/” oznacza, że strona początkowa naszej aplikacji będzie obsługiwana przez nasz serwlet. Gdybyśmy wprowadzili tam wpis „/*” oznaczałoby to, że każde wywołanie adresu w naszej aplikacji będzie przez ten serwlet obsługiwane- tj. każdy podadres np. http://localhost:8080/SpringMVC/niematakiejstrony.do również byłoby obsłużone. Różnica w gwiazdce :)



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http:/
    <servlet>
        <servlet-name>Przyklad</servlet-name>
        <servlet-class>pl.jsystems.springmvc.controller.Przyklad</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Przyklad</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

Ważna informacja!!

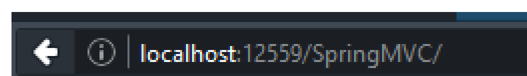
Taki sposób deklaracji wzorca URL obsługiwanego przez Spring MVC sprawi, że również wszystkie statyczne zasoby będą obsługiwane przez Spring. To może uniemożliwić np. osadzenie plików obrazków czy PNG w aplikacji – te przecież nie będą obsługiwane przez żadne kontrolery. Bezpieczniej więc będzie użyć takiej konstrukcji:

```
<servlet-mapping>
   <servlet-name>springmvc</servlet-name>
   <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Dokonamy teraz małej zmiany w naszym serwlecie. W momencie wywołania naszej aplikacji na ekranie w przeglądarce powinna się wyświetlić treść „Halo, tutaj servlet!”.

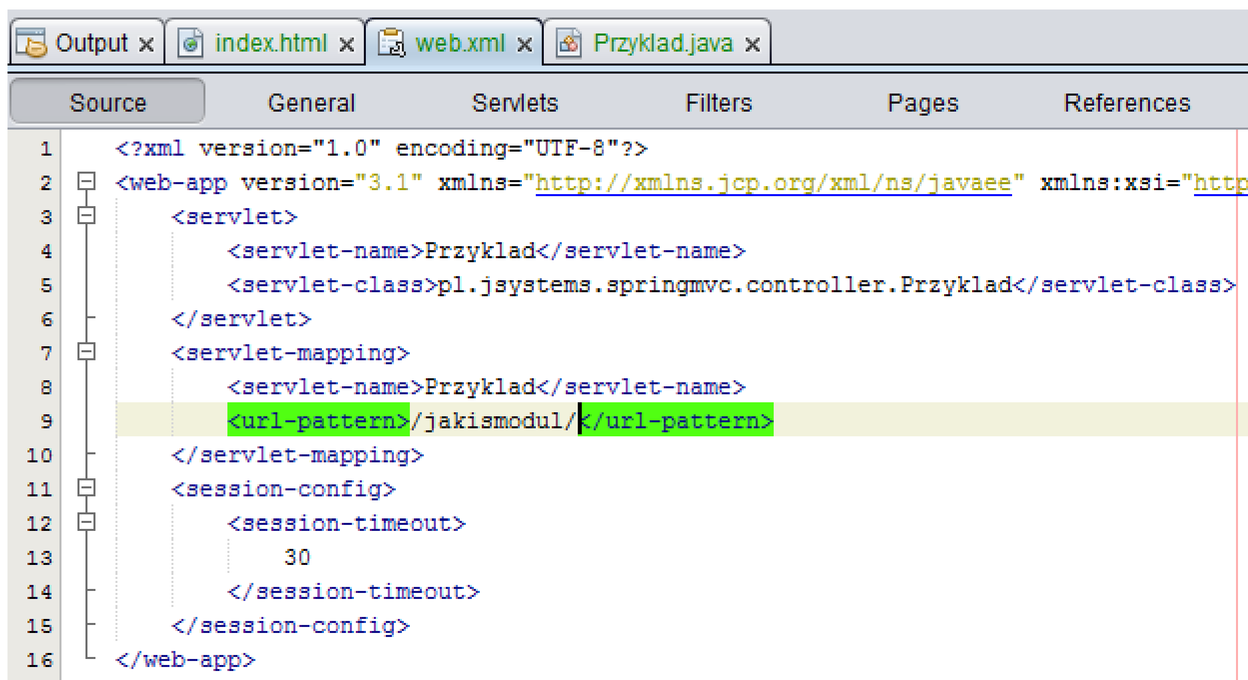
```java
6   package pl.jsystems.springmvc.controller;
7
8   import java.io.IOException;
9   import java.io.PrintWriter;
10  import javax.servlet.ServletException;
11  import javax.servlet.http.HttpServlet;
12  import javax.servlet.http.HttpServletRequest;
13  import javax.servlet.http.HttpServletResponse;
14
15  /**
16   *
17   * @author andrzej
18   */
    public class Przyklad extends HttpServlet {
20
21      @Override
        protected void doGet(HttpServletRequest request, HttpServletResponse response)
23              throws ServletException, IOException {
24          response.setContentType("text/html;charset=UTF-8");
25          PrintWriter out = response.getWriter();
26          out.println("Halo, tutaj servlet!");
27      }
28
29  }
30
```

Uruchommy więc naszą aplikację:



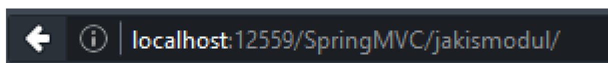
Halo, tutaj servlet!

Moglibyśmy podzielić naszą aplikację na moduły i obsługiwać je przez różne serwlety... albo np. tylko jeden moduł obsługiwać z użyciem Spring MVC. Nieco przerabiam mój plik web.xml:



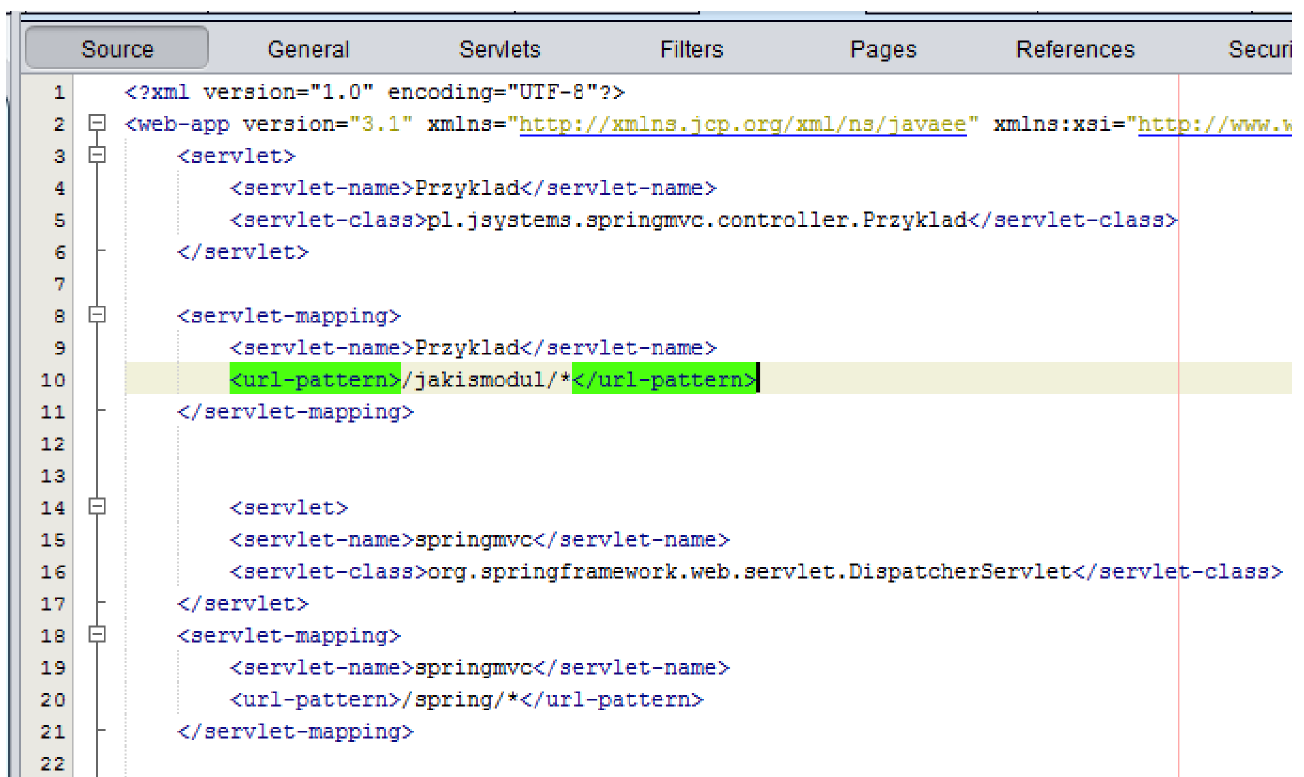

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http
    <servlet>
        <servlet-name>Przyklad</servlet-name>
        <servlet-class>pl.jsystems.springmvc.controller.Przyklad</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Przyklad</servlet-name>
        <url-pattern>/jakismodul/</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

Porównajmy adres wywołania naszego serwletu:

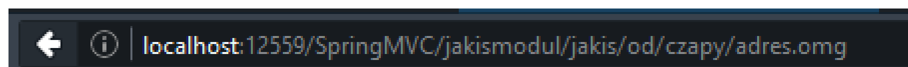


Halo, tutaj servlet!

Teraz będzie drobna zmiana. W linii 10 do adresu /jakismodul/ dodałem *. To oznacza że każde wywołanie z początkiem /jakismodul/ będzie obsługiwane przez nasz przykładowy serwlet:



I sprawdzamy :



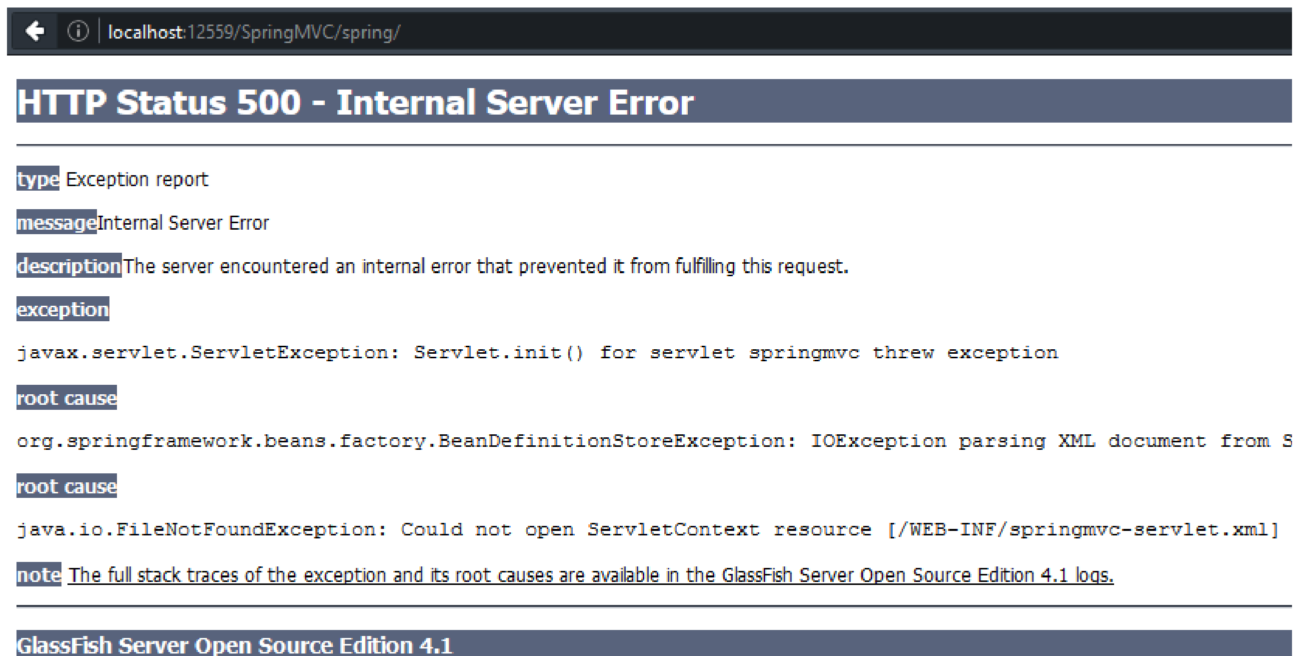
Halo, tutaj servlet!

Mam nadzieję, że po tych przykładach to co się dzieje w nowych liniach 14-21 w pliku web.xml będzie dla Ciebie oczywiste :) Wszystkie wywołania których adres będzie się zaczynał od /SpringMVC/spring/ będą obsługiwane przez serwlet o nazwie „springmvc”. A ten serwlet to klasa DispatcherServlet której.... nie definiowaliśmy :) To właśnie tutaj następuje przekazanie kontroli do Springa. To jest klasa dostarczana z bibioteką Springa, która przejmie kontrolę nad wywołaniami w naszej aplikacji (przynajmiej we wskazanym zakresie adresowym). Co by się stało gdybyśmy w linii 20 zdeklarowali „/*” ? Wszystkie wywołania w naszej aplikacji byłyby obsługiwane przez Springa..

Source | General | Servlets | Filters | Pages | References | Security

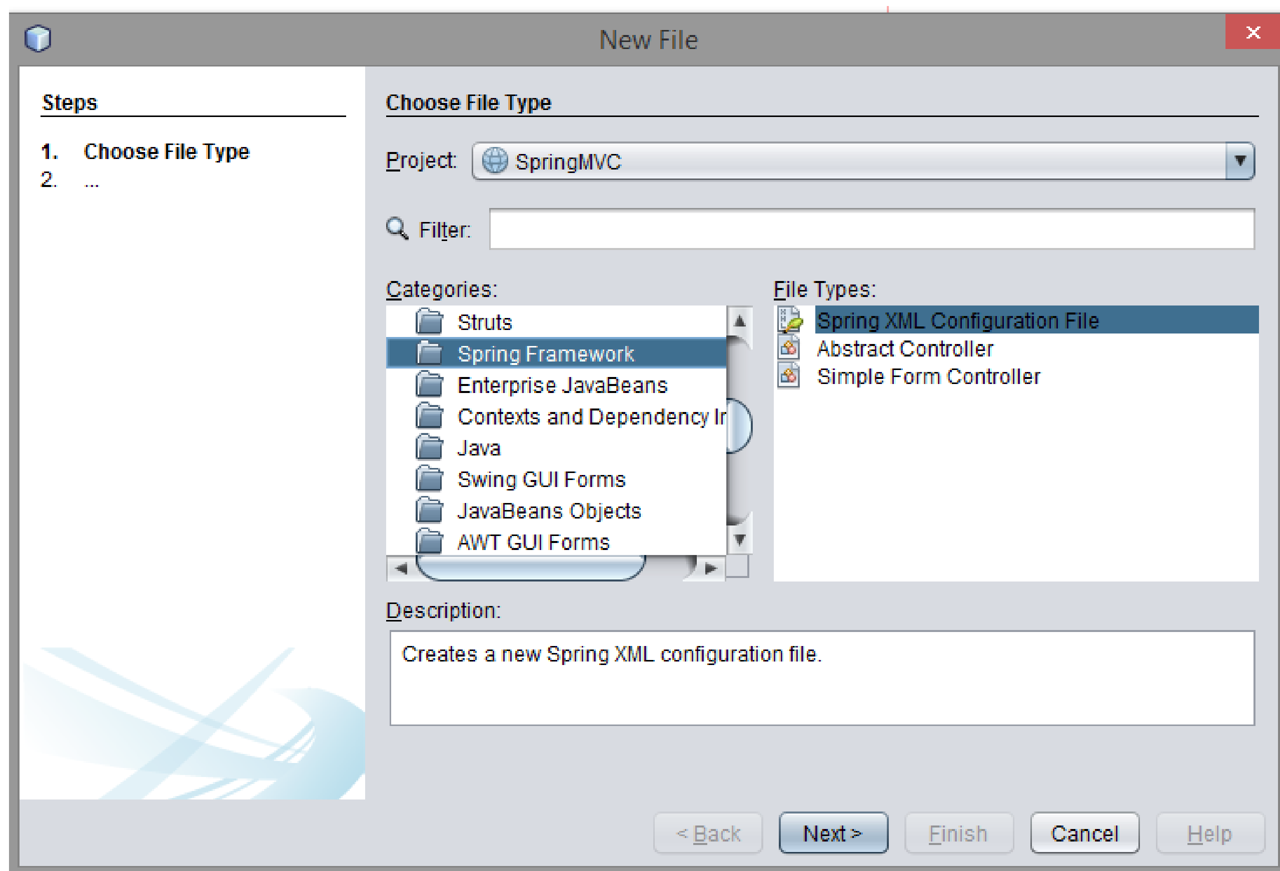
```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.or
3       <servlet>
4           <servlet-name>Przyklad</servlet-name>
5           <servlet-class>pl.jsystems.springmvc.controller.Przyklad</servlet-class>
6       </servlet>
7
8       <servlet-mapping>
9           <servlet-name>Przyklad</servlet-name>
10          <url-pattern>/jakismodul/*</url-pattern>
11      </servlet-mapping>
12
13
14          <servlet>
15          <servlet-name>springmvc</servlet-name>
16          <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
17      </servlet>
18      <servlet-mapping>
19          <servlet-name>springmvc</servlet-name>
20          <url-pattern>/spring/*</url-pattern>
21      </servlet-mapping>
22
23
24
25      <session-config>
26          <session-timeout>
27              30
28          </session-timeout>
29      </session-config>
30  </web-app>
31
```

OK, sprawdźmy teraz co się stanie kiedy wywołamy w przeglądarce adres nowego modułu:

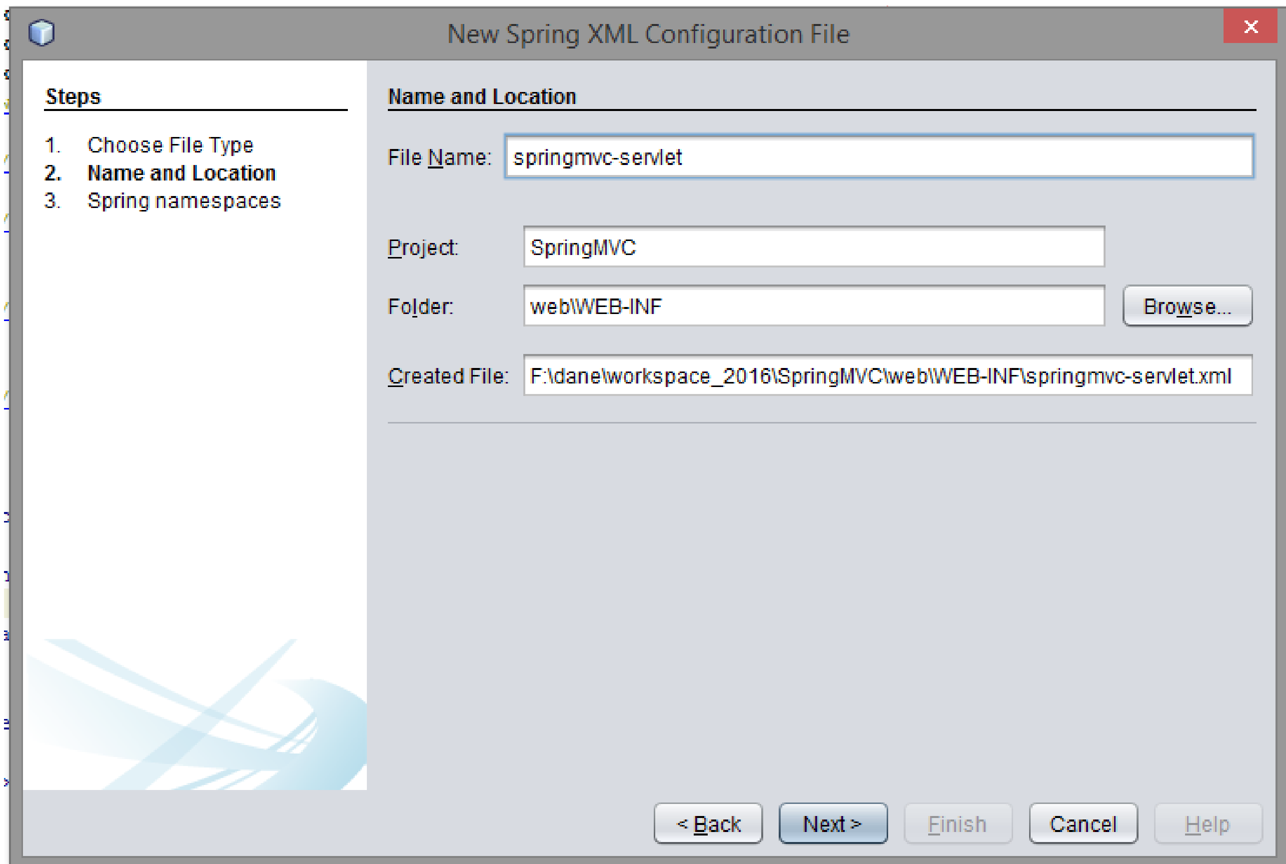


Pojawił się błąd, a gdy mu się przyjrzymy zobaczymy że problem polega na braku pliku springmvc-servlet.xml To jest plik konfiguracyjny Spring MVC w którym będziemy deklarowali m.in. kontrolery naszej aplikacji. Nazwa pliku springmvc-servlet.xml nie jest przypadkowa. Wynika ona z tego jak nazwaliśmy servlet dla klasy DispatcherServlet w linii 15 pliku web.xml. Gdybyś w tym miejscu wpisał zamiast springmvc np. gdzieJestNemo, Spring szukałby pliku gdzieJestNemo-servlet.xml. Wielkość liter ma znaczenie.

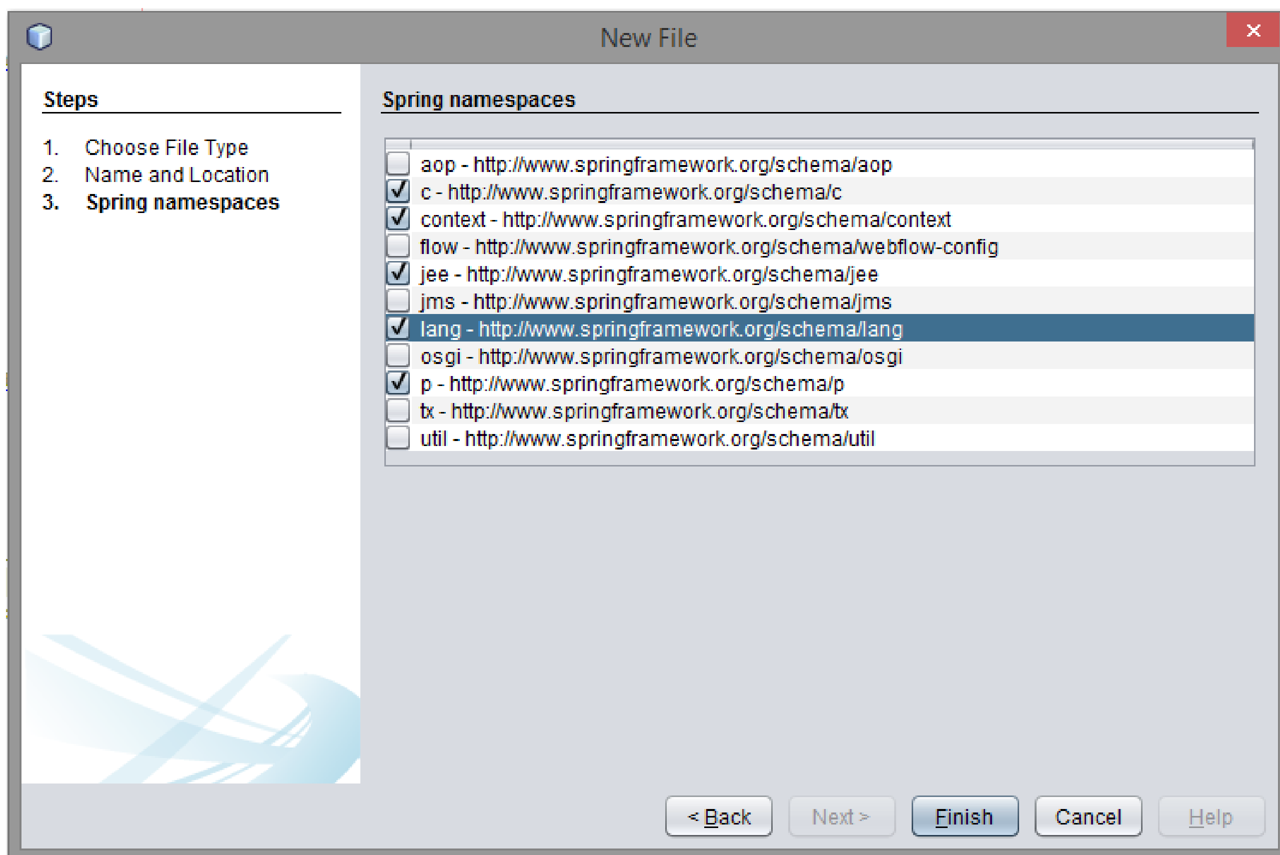
Skoro Spring tak bardzo potrzebuje tego pliku, to mu go dajmy :



Nadamy mu nazwę springmvc-servlet.xml:



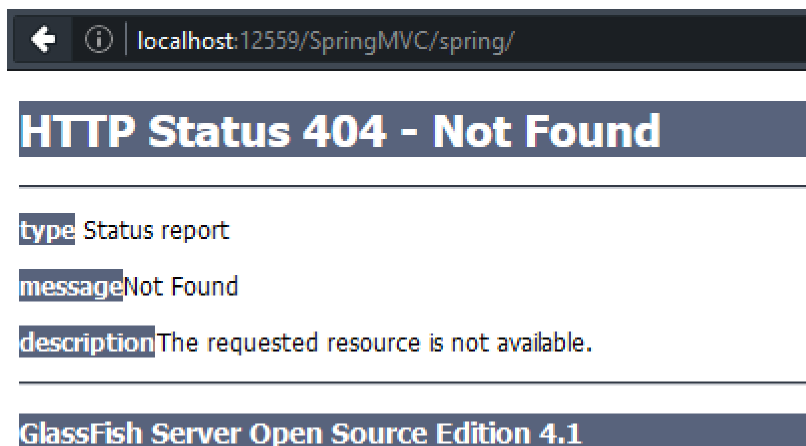
Będą nam też potrzebne pewne przestrzenie nazw XML które będziemy wykorzystywać, dlatego je zaznaczamy:



Nie ma możliwości wyboru przestrzeni MVC dlatego musimy dodać ją ręcznie do nowego pliku. Możesz tez gotowy plik pobrać z przykładowego kodu którego adres znajdziesz na początku tego rozdziału i umieścić go w katalogu WEB-INF.

```
Source   History
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:c="http://www.springframework.org/schema/c"
5         xmlns:context="http://www.springframework.org/schema/context"
6         xmlns:jee="http://www.springframework.org/schema/jee"
7         xmlns:lang="http://www.springframework.org/schema/lang"
8         xmlns:p="http://www.springframework.org/schema/p"
9
10        xmlns:mvc="http://www.springframework.org/schema/mvc"
11
12        xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
13           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd
14           http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.2.xsd
15           http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-3.2.xsd
16
17           http://www.springframework.org/schema/mvc
18  http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd
19 ">
20
```

Ponówmy próbę dostępu do modułu:



Zaglądamy do konsoli serwera:

```
Info:    SpringMVC was successfully deployed in 970 milliseconds.
Info:    WebModule[null] ServletContext.log():Initializing Spring FrameworkServlet 'springmvc'
Info:    FrameworkServlet 'springmvc': initialization started
Info:    Refreshing WebApplicationContext for namespace 'springmvc-servlet': startup date [Tue Jan 26 11:33:26 CET 2016];
Info:    Loading XML bean definitions from ServletContext resource [/WEB-INF/springmvc-servlet.xml]
Info:    Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@51cf6e71: d
Info:    FrameworkServlet 'springmvc': initialization completed in 471 ms
Warning:   No mapping found for HTTP request with URI [/SpringMVC/spring/] in DispatcherServlet with name 'springmvc'
```

Widzimy że Spring odnalazł nasz nowy plik (linia z Loading XML bean definitions....). Przyjrzyjmy się teraz ostatniej linii z ostrzeżeniem. Problem polega na tym, że nie określiliśmy w pliku springmvc-servlet.xml przez jaką klasę ma być obsługiwany adres /SpringMVC/spring/.

Dodajemy więc nowy wpis do springmvc-servlet.xml:

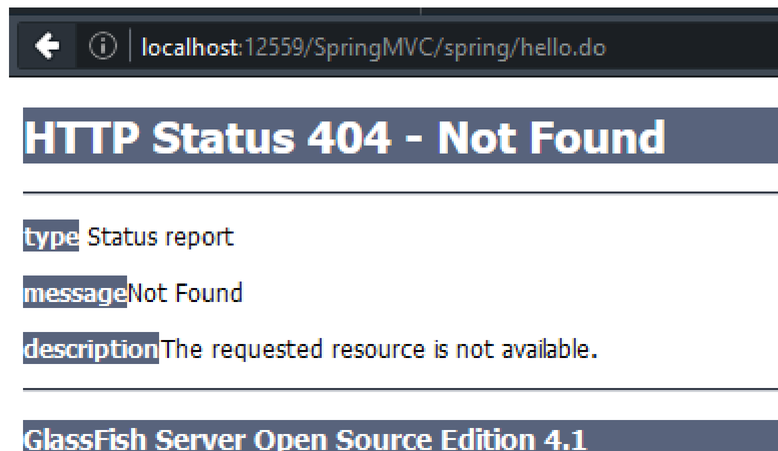
```
17              http://www.springframework.org/schema/mvc
18   http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd
19  ">
20
21      <mvc:annotation-driven/>
22
23      <context:component-scan base-package="pl.jsystems.springmvc.controller"/>
24
25
26  </beans>
27
```

Linia 21 oznacza, że Spring ma poszukać deklaracji mapowań obsługiwanych adresów w adnotacjach znajdujących się w klasach pakietu (i jego podpakietów) który wskazaliśmy w linii 23 :)
Do pakietu pl.jsystems.springmvc.controller dodajemy teraz zwyczajną klasę Hello. Wprowadzamy metodę „sayHello” która wypisuje na konsoli serwera tekst „HELLO MUPPET”. Koniecznie dodaj wpis @Controller nad deklarają klasy, oraz @RequestMapping nad metodą SayHello.

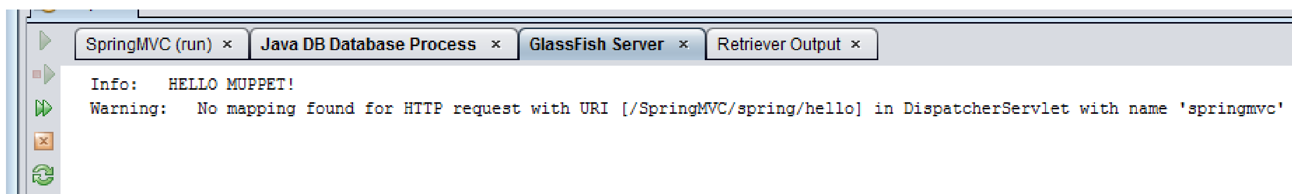
```
6   package pl.jsystems.springmvc.controller;
7
8   import org.springframework.stereotype.Controller;
9   import org.springframework.ui.Model;
10  import org.springframework.web.bind.annotation.RequestMapping;
11
12  /**
13   *
14   * @author andrzej
15   */
16  @Controller
17  public class Hello {
18
19      @RequestMapping("/hello.do")
20      public String sayHello(Model model) {
21          System.out.println("HELLO MUPPET!");
22          return "hello";
23      }
24  }
25
```

Wpis @Controller z linii 16 deklaruje, że klasa ta obsługuje żądania HTTP, a @RequestMapping z parametrem wskazują która klasa do robi i dla jakiego konkretnie żądania. Adres /hello.do jest względny i oznacza wywołanie /SpringMVC/spring/hello.do.

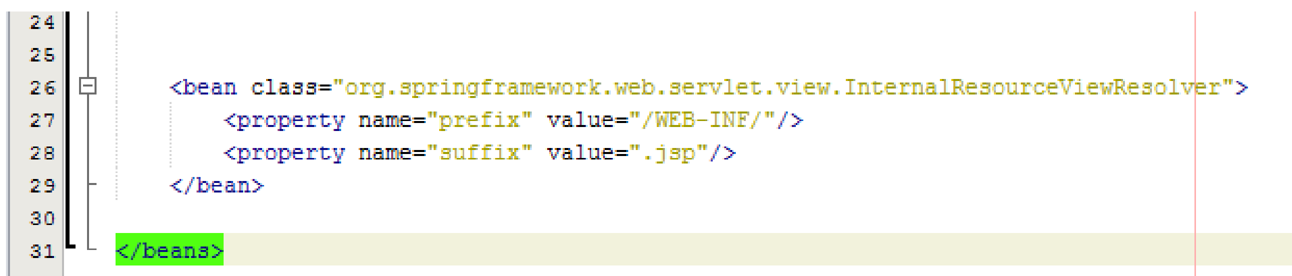
Wywołajmy więc ten adres:



Nasze ulubione 404. Czy to znaczy że coś nie zadziałało? To zależy :) „This is not a bug, this is a feature” :) A tak poważnie – wszystko zgodnie z planem. Zajrzyjmy do konsoli serwera:



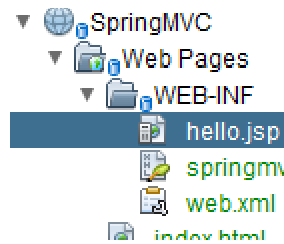
Wyświetlił nam się komunikat „Hello Muppet”, a to oznacza że nasza metoda sayHello została zgodnie z założeniem wywołana. Błąd 404 pojawia się dlatego, że nie mam strony JSP którą powinienem w odpowiedzi wyświetlić. Konkretnie to plik powinien się nazywać hello.jsp – ponieważ metoda sayHello zwraca ciąg tekstowy „hello” i powinien znajdować się bezpośrednio w katalogu WEB-INF co zdeklarujemy sobie w pliku springmvc-servlet:



```
24
25
26     <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
27         <property name="prefix" value="/WEB-INF/"/>
28         <property name="suffix" value=".jsp"/>
29     </bean>
30
31 </beans>
```

W linii 27 deklaruję gdzie Spring ma szukać plików JSP, a w linii 28 jakie mają mieć rozszerzenie. Jeśli chcesz, w linii 27 możesz dodać jakiś podkatalog np. /WEB-INF/jsp/, albo wydzielić w ogóle osobny katalog na pliki jsp związane z tym modułem np. /WEB-INF/jsp/spring/

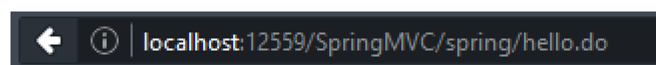
Stwórzmy jeszcze w zadeklarowanym katalogu plik hello.jsp:



i umieśćmy w nim taki oto kod:

```
7   <%@page contentType="text/html" pageEncoding="UTF-8"%>
8   <!DOCTYPE html>
9   <html>
10      <head>
11          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12          <title>JSP Page</title>
13      </head>
14      <body>
15          <h1>Hello Muppet!</h1>
16      </body>
17  </html>
18
```

Teraz przy wywołaniu adresu /SpringMVC/spring/hello.do powinniśmy zobaczyć taki komunikat:



Hello Muppet!

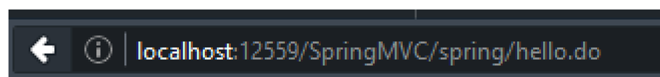
Przydałoby się jednak zrobić coś więcej niż wyświetlanie statycznej wartości. Przekażmy więc jakieś dane z kontrolera do widoku (patrz linia 23):

```
12  /**
13   *
14   * @author andrzej
15   */
16  @Controller
17  public class Hello {
18
19      @RequestMapping("/hello.do")
20      public String sayHello(Model model) {
21          System.out.println("HELLO MUPPET!");
22          String info="witaj w świecie Spring MVC!";
            model.addAttribute("wiadomosc", info);
24          return "hello";
25      }
26  }
27
```

a następnie wyświetlmy ją na stronie JSP (patrz linia 16):

```
6   <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
7   <%@page contentType="text/html" pageEncoding="UTF-8"%>
8   <!DOCTYPE html>
9   <html>
10      <head>
11          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12          <title>JSP Page</title>
13      </head>
14      <body>
15          <h1>Hello Muppet!</h1>
            <h3>${wiadomosc}</h3>
17      </body>
18  </html>
19
```

Zauważ że w JSP odwołuję się do przekazanego elementu po nazwie która ustaliłem w pierwszym parametrze metody addAttribute tj. „wiadomosc”. Efekt:



Hello Muppet!

witaj w świecie Spring MVC!

Przekazywać oczywiście możemy też obiekty, listy, a także możemy obsługiwać formularze, ale tym zajmiemy się w kolejnych częściach tego kursu. W tej chwili zrobiliśmy chyba najprostszą możliwą implementację Spring MVC. Mamy oczywiście wiele możliwych wariantów – jak choćby w miejsce adnotacji użycie deklaracji beanów w pliku XML.

Jeszcze pozwolę sobie na małe rozwinięcie tematu zarządzania wywołaniami. Przypuśćmy że tworzymy dużą aplikację złożoną z kilku modułów i chcielibyśmy mieć osobne pliki konfiguracyjne. Kod źródłowy do następnych przykładów znajdziesz pod adresem :

http://www.jsystems.pl/storage/spring/springmvc2.zip

Dodamy sobie kolejną paczkę do web.xml:

```
23
24      <servlet>
25        <servlet-name>drugimodul</servlet-name>
26        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
27    </servlet>
28    <servlet-mapping>
29        <servlet-name>drugimodul</servlet-name>
30        <url-pattern>/drugimodul/*</url-pattern>
31    </servlet-mapping>
32
33
```

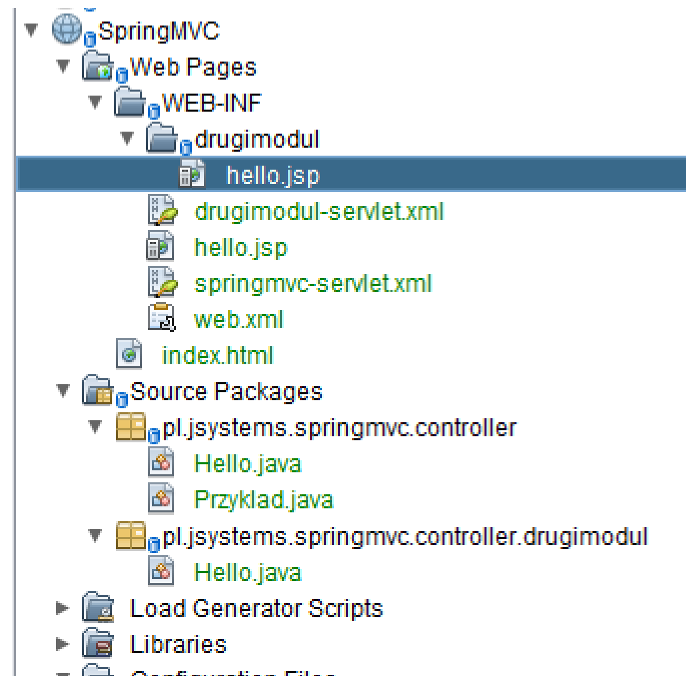
W związku z tym że nasz serwlet nazywa się drugimodul dodajemy też plik „drugimodul-servlet.xml” . W nim dodajemy takie wpisy jak poprzednio, z tą różnicą że pliki JSP znajdą się w osobnym podkatalogu w WEB-INFie (patrz linia 27), a dodatkowo wydzielimy sobie osobny pakiet na kontrolery tego modułu. Dzięki temu będziemy mogli mieć klasy kontrolerów o takiej samej nazwie.

```
20
21      <mvc:annotation-driven/>
22
23      <context:component-scan base-package="pl.jsystems.springmvc.controller.drugimodul"/>
24
25
26      <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
27          <property name="prefix" value="/WEB-INF/drugimodul/"/>
28          <property name="suffix" value=".jsp"/>
29      </bean>
30
31  </beans>
32
```

Oczywiście taki pakiet tworzymy, a w nim umieszczamy odrobinę różniącą się klasę kontrolera:

```
5    */
6   package pl.jsystems.springmvc.controller.drugimodul;
7
8   import pl.jsystems.springmvc.controller.*;
9   import org.springframework.stereotype.Controller;
10  import org.springframework.ui.Model;
11  import org.springframework.web.bind.annotation.RequestMapping;
12
13  /**
14   *
15   * @author andrzej
16   */
17  @Controller
18  public class Hello {
19
20      @RequestMapping("/hello.do")
21      public String sayHello(Model model) {
22          System.out.println("HELLO MUPPET w drugim module!");
23          String info="witaj w świecie Spring MVC!";
24          model.addAttribute("wiadomosc", info);
25          return "hello";
26      }
27  }
28
```

Zauważ że tutaj również określone jest mapowanie dla „/hello.do”, ale jak pamiętamy jest to adres względny i w tym przypadku oznacza wywołanie „/SpringMVC/drugimodul/hello.do”, a nie „/SpringMVC/spring/hello.do”. Metoda nadal zwraca tekst hello, co oznacza że Spring poszuka pliku hello.jsp by go wyświetlić, tym razem jednak będzie go szukał w katalogu „WEB-INF/drugimodul”.
W katalogu WEB-INF tworzymy podkatalog (taki jaki wskazaliśmy we wpisie w pliku drugimodul-servlet.xml) i umieszczamy w nim plik jsp analogiczny do poprzedniego, z tym że dla odróżnienia zmienimy troszkę jego zawartość.




```
5    --%>
6    <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
7    <%@page contentType="text/html" pageEncoding="UTF-8"%>
8    <!DOCTYPE html>
9    <html>
10       <head>
11           <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12           <title>JSP Page</title>
13       </head>
14       <body>
15           <h1>Hello Muppet w drugim module!</h1>
16           <h3>${wiadomosc}</h3>
17       </body>
18   </html>
```

Sprawdźmy:

localhost:12559/SpringMVC/drugimodul/hello.do

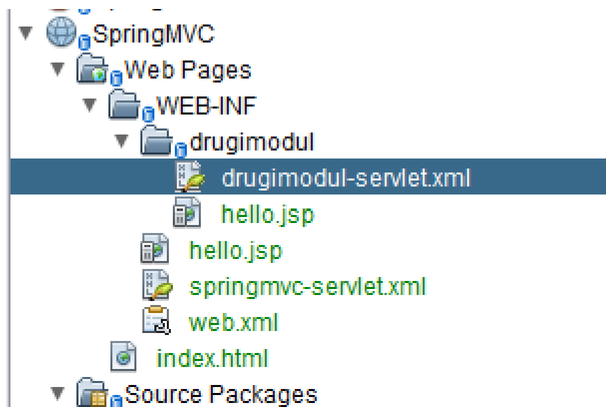
Hello Muppet w drugim module!

witaj w świecie Spring MVC!

Położenie pliku konfiguracji Springa

Domyślnie pliki konfiguracji Spring MVC muszą znajdować się bezpośrednio w katalogu WEB-INF i posiadać nazwę typu ******-servlet.xml. Gdybyś zechciał zmienić nazwę lub położenie tego pliku musisz wykonać następujące kroki:

Przenieść plik:



Wcześniej mój plik drugimodul-servlet.xml znajdował się bezpośrednio w katalogu WEB-INF. Dokonać zmiany w pliku web.xml dodając sekcję "init-param" dla Dispatcher Serwletu dla danego modułu/aplikacji:

```
22
23
24      <servlet>
25        <servlet-name>drugimodul</servlet-name>
26        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
27        <init-param>
28            <param-name>contextConfigLocation</param-name>
29            <param-value>/WEB-INF/drugimodul/drugimodul-servlet.xml</param-value>
30        </init-param>
31    </servlet>
32
```

I zweryfikować czy działa :)

```
Info:   FrameworkServlet 'drugimodul': initialization started
Info:   Refreshing WebApplicationContext for namespace 'drugimodul-servlet': startup date [Sat Jan 30 20:55:33 CET 2016];
Info:   Loading XML bean definitions from ServletContext resource [/WEB-INF/drugimodul/drugimodul-servlet.xml]
Info:   JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
Info:   Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@3b169aff: de
```

Przekazywanie obiektów i list do warstwy widoku

Kod źródłowy do poniższych przykładów znajdziesz pod adresem:

http://jsystems.pl/storage/spring/springmvc3.zip

Sama idea MVC zakłada rozdzielenie warstw biznesowej, danych i widoku od siebie. Tak więc i w naszym przypadku danych nie będziemy pobierali bezpośrednio na poziomie widoku. Kontroler przekaże do widoku dane pochodzące z DAO. Dao będzie jednak zaślepką, nie będziemy pobierali danych z bazy danych, gdyż nie tym się tutaj zajmujemy :)

Konstrukcja aplikacji jest bardzo zbliżona do przykładów z poprzednich rozdziałów, dochodzi nam za to kilka nowych elementów. Zakładam że masz już działającą aplikację, którą teraz będziemy rozbudowywać. Zaczniemy od stworzenia osobnych pakietów na DAO i klasy domenowe. Następnie tworzymy zwykłą klasę POJO jaką widac poniżej. Obiekty tej klasy będą reprezentowały samochody których dane będziemy wyświetlać. Konstruktor sparametryzowany w tym przykładzie nie jest niezbędny, stosuje go dla własnej wygody. Musimy za to mieć w tej przynajmniej gettery do pól, gdyż w warstwie widoku będziemy używali tagów JSTL które tego wymagają.

```
7
8   /**
9    *
10   * @author andrzej
11   */
12  public class Samochod {
13      private String marka;
14      private String model;
15      private String numerRejestracyjny;
16      private String kolor;
17
18      public Samochod(String marka, String model, String numerRejestracyjny, String kolor) {
19          this.marka = marka;
20          this.model = model;
21          this.numerRejestracyjny = numerRejestracyjny;
22          this.kolor = kolor;
23      }
24
25
26
27      /**
28       * @return the marka
29       */
30      public String getMarka() {
31          return marka;
32      }
33
34      /**
35       * @param marka the marka to set
36       */
37      public void setMarka(String marka) {
38          this.marka = marka;
39      }
40
41      /**
```

Tworzę teraz takie fake'owe DAO które będzie nam zwracało przykładowe dane. Potrzebujemy dwóch metod – jednej zwracającej pojedynczy samochód, drugiej zwracającej ich całą listę:

```java
package pl.jsystems.springmvc.dao;

import java.util.ArrayList;
import java.util.List;
import pl.jsystems.springmvc.domain.Samochod;

/**
 *
 * @author andrzej
 */
public class SamochodDao {

    public Samochod getOne() {
        Samochod s = new Samochod("Polonez", "Borewicz", "OMG 12345", "czerwony");
        return s;
    }

    public List<Samochod> getAll(){
        List<Samochod> fury = new ArrayList<Samochod>();
            fury.add(new Samochod("Audi", "A4", "WTF 98765", "czarny"));
            fury.add(new Samochod("BMW", "e61", "LLU 112112", "grafitowy"));
            fury.add(new Samochod("Mercedes", "SL", "WOW 00000", "biały"));
            fury.add(new Samochod("Solaris", "przegubowiec", "WR 12345", "żółto-czerwony"));
        return fury;
    }

}
```

Będziemy mieli dwa widoki – jeden wyświetlający listę samochodów, drugi wyświetlający tylko Poloneza ;) Tworzę więc dwa kontrolery:

```
7
8  import org.springframework.stereotype.Controller;
9  import org.springframework.ui.Model;
10 import org.springframework.web.bind.annotation.RequestMapping;
11 import pl.jsystems.springmvc.dao.SamochodDao;
12
13 /**
14  *
15  * @author andrzej
16  */
17
18 @Controller
19 public class PokazSamochody {
20
21     @RequestMapping("pokazSamochody.do")
22     public String wyswietlSamochody(Model model){
23
24         SamochodDao dao = new SamochodDao();
25         model.addAttribute("samochody", dao.getAll());
26         return "pokazSamochody";
27     }
28
29 }
30
```

```
6  package pl.jsystems.springmvc.controller;
7
8  import org.springframework.stereotype.Controller;
9  import org.springframework.ui.Model;
10 import org.springframework.web.bind.annotation.RequestMapping;
11 import pl.jsystems.springmvc.dao.SamochodDao;
12
13 /**
14  *
15  * @author andrzej
16  */
17
18 @Controller
19 public class PokazJeden {
20
21     @RequestMapping("pokazJeden.do")
22     public String pokazSamochod(Model model){
23         SamochodDao dao = new SamochodDao();
24         model.addAttribute("samochod", dao.getOne());
25         return "pokazJeden";
26     }
27
28 }
```

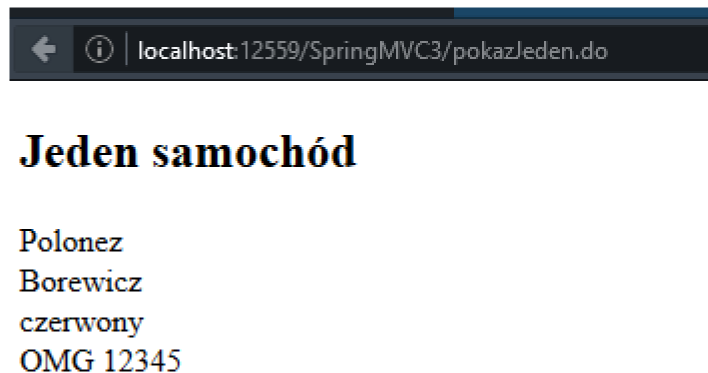
Mała uwaga dla osób które już troszeczkę Spring MVC znają : "tak, wiem że powinienem tutaj użyć AOP i @Autowired zamiast po prostu deklarować obiekt klasy dao (i to jeszcze na poziomie metody !!!) , ale popełniam takie bluźnierstwo aby nie komplikować życia osobom początkującym :)

Nowy element jaki się tutaj pojawia to model.addAttribute z przekazaniem obiektu, lub w drugim kontrolerze listy obiektów. Wcześniej przekazywaliśmy wyłącznie komunikat tekstowy. Właściwie możemy tutaj przekazać cokolwiek, różnica będzie jedynie w dostępie do tych danych na poziomie widoku. Pamiętać musimy że z racji używania tagów JSTL w widoku, jeśli przekazujemy jakiś obiekt a zamierzamy odwoływać się na poziomie widoku do jego pól, to pola te muszą posiadać gettery.

Dodałem też dwa pliki JSP na potrzeby dwóch osobnych widoków. Poniżej widok prezentujący dane jednego samochodu:

```
5   --%>
6   <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
7   <%@page contentType="text/html" pageEncoding="UTF-8"%>
8   <!DOCTYPE html>
9   <html>
10      <head>
11          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12          <title>JSP Page</title>
13      </head>
14      <body>
15
16          <h2>Jeden samochód</h2>
17
18          ${samochod.marka}<br>
19          ${samochod.model}<br>
20          ${samochod.kolor}<br>
21          ${samochod.numerRejestracyjny}<br>
22
23      </body>
24  </html>
25
```

Zauważ że przed polem obiektu podaję taką nazwę, pod jaką przekazałem obiekt w klasie PokazJeden (linia 24). Efekt działania:

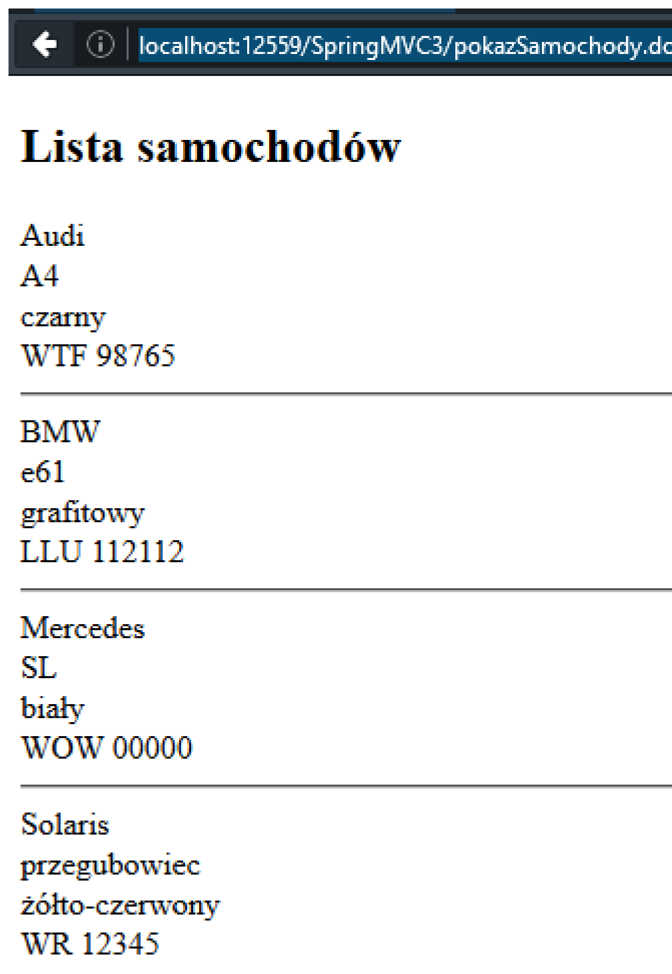


Poniżej kod pliku JSP odpowiedzialnego za wyświetlenie listy samochodów:

```
6   <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
7   <%@page contentType="text/html" pageEncoding="UTF-8"%>
8   <!DOCTYPE html>
9   <html>
10      <head>
11          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12          <title>JSP Page</title>
13      </head>
14      <body>
15
16          <h2>Lista samochodów</h2>
17
18          <c:forEach items="${samochody}" var="s">
19
20              ${s.marka}<br>
21              ${s.model}<br>
22              ${s.kolor}<br>
23              ${s.numerRejestracyjny}<br>
24
25              <hr>
26
27          </c:forEach>
28
29      </body>
30  </html>
```

Konstrukcja c:forEach nie jest elementem Springa. To są standardowe taki JSTL jakimi mógłbys się posługiwać w aplikacji złożonej np. Z serwletów i plików JSP. Pamiętaj jednak o konieczności wskazania co oznacza prefix c (u mnie w linii 6). Bez tego nic się nie wyswietli.

Efekt działania:



Mapowanie na poziomie klasy

Kod źródłowy do poniższych przykładów można znaleźć pod adresem :
http://jsystems.pl/storage/spring/springmvc4.zip

Nie musisz tworzyć osobnych kontrolerów dla każdego wywołania osobno. Możesz wiele adresów wywołań obsługiwać na poziomie jednej klasy. Staje się to przydatne szczególnie wówczas, gdy widok dla wszystkich żądań jest taki sam, ale np prezentowane są odfiltrowane dane w zależności od adresu wywołania.

W tym przykładzie zrobimy szkielet katalogu samochodów u typowego Mirka-handlarza-przedsiębiorcy :) Konfiguracja na poziomie plików XML jest analogiczna jak w poprzednich przykładach. Wszystkie żądania zaczynające się od /miro będą obsługiwane przez Springa:

```
5
6    <servlet>
7        <servlet-name>miro</servlet-name>
8        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
9    </servlet>
10   <servlet-mapping>
11       <servlet-name>miro</servlet-name>
12       <url-pattern>/miro/*</url-pattern>
13   </servlet-mapping>
14
```

Cała magia w zasadzie zawiera się w kontrolerze:

```
12  /**
13   *
14   * @author andrzej
15   */
16
17  @Controller
18  @RequestMapping("/niemiecplakaljaksprzedawal")
19  public class MapowanieNaPoziomieKlasy {
20
21
22      @RequestMapping
23      public String getAll(Model model){
24          return "wszystkie";
25      }
26
27      @RequestMapping("/bite")
28      public String getBite(Model model){
29          return "bite";
30      }
31
32      @RequestMapping("/lewyprzebieg")
33      public String getPrzebieg(Model model){
34          return "przebieg";
35      }
36  }
37
```

Zauważ że pojawiła się tutaj nowa rzecz, mianowicie "RequestMapping" na poziomie klasy. Ponadto nad poszczególnymi metodami mamy osobne "RequestMapping". O co chodzi? Chodzi o to, że wszystkie wywołania zaczynające się od "/niemiecplakaljaksprzedawal" (a tak naprawdę to w zasadzie "/miro/niemiecplakaljaksprzedawal") będą obsługiwane przez tę klasę. Teraz wywołując po prostu "/miro/niemiecplakaljaksprzedawal" mapowanie zostanie przypisane do domyślnego tj w tym przypadku do metody getAll, wywołanie "/miro/niemiecplakaljaksprzedawal/bite" do metody getBite, a ""/miro/niemiecplakaljaksprzedawal/lewyprzebieg" do metody getPrzebieg.

Zmienne ścieżki

Kod źródłowy z przykładami do tego rozdziału możesz pobrać pod adresem:

http://jsystems.pl/storage/spring/springmvc5.zip

W poprzednim przykładzie stworzyliśmy mapowania zależne od końcówki adresu. Przypomnę screena:

```
12  /**
13   *
14   * @author andrzej
15   */
16
17  @Controller
18  @RequestMapping("/niemiecplakaljaksprzedawal")
19  public class MapowanieNaPoziomieKlasy {
20
21
22      @RequestMapping
23      public String getAll(Model model){
24          return "wszystkie";
25      }
26
27      @RequestMapping("/bite")
28      public String getBite(Model model){
29          return "bite";
30      }
31
32      @RequestMapping("/lewyprzebieg")
33      public String getPrzebieg(Model model){
34          return "przebieg";
35      }
36  }
37
```

Co pozwoliłoby nam np zaimplementować sklep internetowy z kategoriami produktów. Co jednak jeśli mielibyśmy setki różnych kategorii? Tworzyć osobne metody dla każdej kategorii? Dużo wygodniej byłoby wychwycić nazwę kategorii z paska adresu i przekazać do zapytania SQL w celu odfiltrowania produktu. Zobaczmy:

```
12
13  /**
14   *
15   * @author andrzej
16   */
17  @Controller
18  public class ZmiennaSciezki {
19
20      @RequestMapping("/{zmienna}")
21      public String pobierzZmienna(Model model,@PathVariable("zmienna") String x) {
22          System.out.println("zmienna ścieżki="+x);
23          return "somejsp";
24      }
25
```

Porównaj @RequestMapping z tego i poprzedniego przykładu. Tym razem część adresu objęta jest w nawiasach klamrowych. Oznacza to, że tutaj może się pojawić dowolny tekst a zostaje on przechwycony dzięki @PathVariable. Parametrem @PathVariable jest nazwa zmiennej którą podaliśmy w @RequestMapping. Wartość zostaje przypisana do zmiennej x.

Do przeglądarki wprowadziłem adres: http://localhost:12559/SpringMVC5/zs/bulbulatory
Efekt:

```
Info:   ### --> pl
Info:   ### --> pl
Info:   zmienna ścieżki=bulbulatory
```

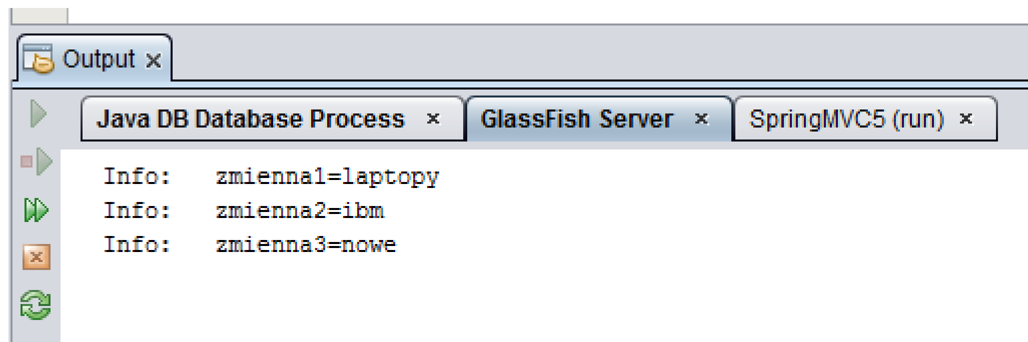
Inny przykład:

```
26
27      @RequestMapping("/{zmienna1}/{zmienna2}/{zmienna3}")
28      public String pobierzZmienne(Model model,
29              @PathVariable("zmienna1") String x,@PathVariable("zmienna2") String y,@PathVariable("zmienna3") String z) {
30          System.out.println("zmienna1="+x);
31          System.out.println("zmienna2="+y);
32          System.out.println("zmienna3="+z);
33          return "somejsp";
34      }
35  }
```

Tym razem zdefiniowałem trzy zmienne ścieżki. Spring będzie oczekiwał **dokładnie** trzech zmiennych ścieżkowych. Jeśli podamy jedną, wywołanie zostanie obsłużone przez pierwszą metodę. Przy 2 dostaniemy błąd, ponieważ nie mamy żadnej metody obsługującej dokładnie 2 parametry. Tym razem wprowadziłem taki adres:

http://localhost:12559/SpringMVC5/zs/laptopy/ibm/nowe

i efekt:



Zmienne tablicowe

Kod źródłowy z przykładami do tego rozdziału możesz pobrać pod adresem:

http://jsystems.pl/storage/spring/springmvc6.zip

Bywają sytuacje kiedy chcesz przekazać przez pasek wiele parametrów, ale ich liczba może być zmienna, a także wartości w tych parametrach mogą występować pojedynczo, lub jako zbiór. Weźmy za przykład wyszukiwarkę na popularnym portalu Allegro. Dajmy na to że szukamy dla siebie samochodu. Korzystamy z wyszukiwarki i wybieramy np markę, model, cenę, pojemność silnika. My się możemy skupić na konkretnym modelu konkretnej marki za określone pieniądze, ale ktoś inny może szukać np samochodów kilku marek, nie określając ceny ani pojemności silnika za to określając przybliżone położenie.... Teraz chcemy zaimplementować taką wyszukiwarkę. Wyobraźcie sobie to drzewo ifów które trzeba byłoby naklepać w zwykłych serwletach. Na szczęście Spring znów przychodzi nam z pomocą. Nasze wywołanie będzie wyglądało mniej więcej tak:
http://localhost:6060/SpringMVC6/zs/filtruj/wojewodztwo=mazowieckie,wielkopolskie;marka=BMW

chociaż oczywiście parametrów może być więcej lub mniej, możemy też mieć inną liczbę wartości w poszczególnych parametrach. Zaczniemy od małej modyfikacji w naszym pliku konfiguracyjnym ****-servlet.xml. Linię :

```
<mvc:annotation-driven/>
```

przerabiamy na :

```
<mvc:annotation-driven enableMatrixVariables="true"/>
```

Dorabiamy kontoler i tworzymy metodę która będzie nam mapowała adres /filtruj/ z parametrami:

```
18     * @author andrzej
19     */
20    @Controller
21    public class ZmienneTablicowe {
22
23        @RequestMapping("/filtruj/{filtry}")
24        public String filtruj(@MatrixVariable(pathVar="filtry") Map<String, List<String>> filtry,  Model model) {
25            System.out.println("Pobieram parametry....");
26            Set<String> warunki = filtry.keySet();
27
28            if(warunki.contains("wojewodztwo")){
29                System.out.println(" ##### Jest warunek dotyczący województw #####");
30                for (String p : filtry.get("wojewodztwo")) {
31                    System.out.println(p);
32                }
33            }
34            if(warunki.contains("marka")){
35                   System.out.println(" #####Jest warunek dotyczący marki #####");
36                for (String p : filtry.get("marka")) {
37                    System.out.println(p);
38                }
39            }
40
41            return "somejsp";
42        }
43
44
45    }
46
```

Zwróć uwagę, że @RequestMapping uwzględnia parametr ścieżkowy objęty nawiasami klamrowymi {filtry} tak jak w poprzednim rozdziale. Pojawia nam się tutaj też nowy parametr metody @MatrixVariable. Służy on właśnie do odbierania zmiennych tablicowych. Poruszamy się po nich tak jak po każdej mapie, z tą różnicą że dodatkowo każdy element w mapie jest listą (może być jednoelementową listą). Elementy Mapy to parametry, a elementy list przyporządkowanych do tych parametrów to ich wartości. Po uruchomieniu wskazanego wcześniej adresu dostajemy:

```
Output - Apache Tomcat or TomEE
06-Feb-2016 23:03:06.069 INFO [http-nio-6060-exec-23]
06-Feb-2016 23:03:06.092 INFO [http-nio-6060-exec-23]
06-Feb-2016 23:03:06.128 INFO [http-nio-6060-exec-23]
06-Feb-2016 23:03:06.433 INFO [http-nio-6060-exec-23]
06-Feb-2016 23:03:06.560 INFO [http-nio-6060-exec-23]
06-Feb-2016 23:03:06.813 INFO [http-nio-6060-exec-23]
Pobieram parametry....
 ##### Jest warunek dotyczący województw #####
mazowieckie
wielkopolskie
 #####Jest warunek dotyczący marki #####
BMW
```

Parametry żądania

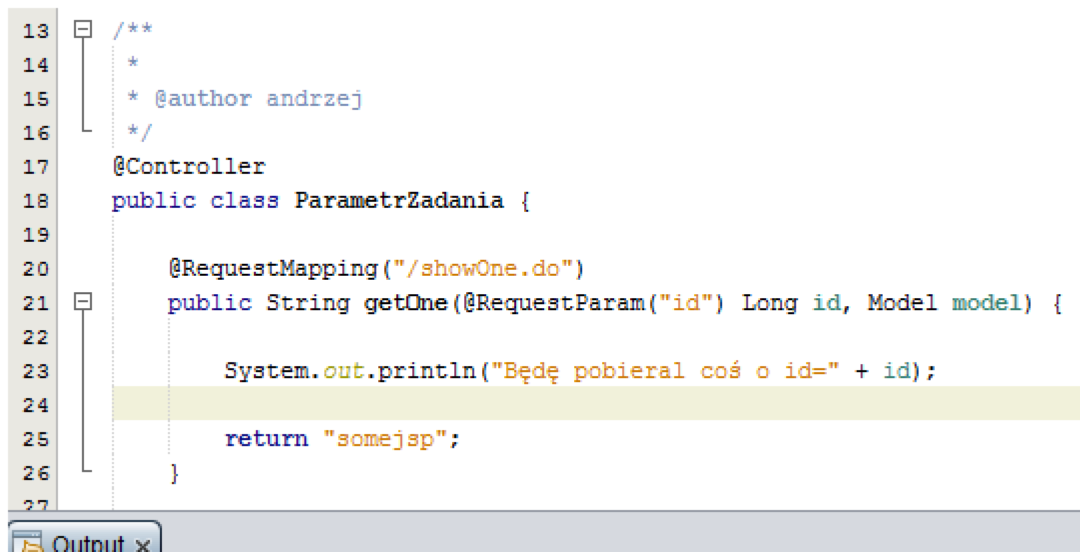
Kod źródłowy z przykładami do tego rozdziału możesz pobrać pod adresem:

http://jsystems.pl/storage/spring/springmvc7.zip

Parametry żądania to parametry przekazywane np. W ten sposób:

www.jakissklep.pl/pokazProdukt.do?idProduktu=78

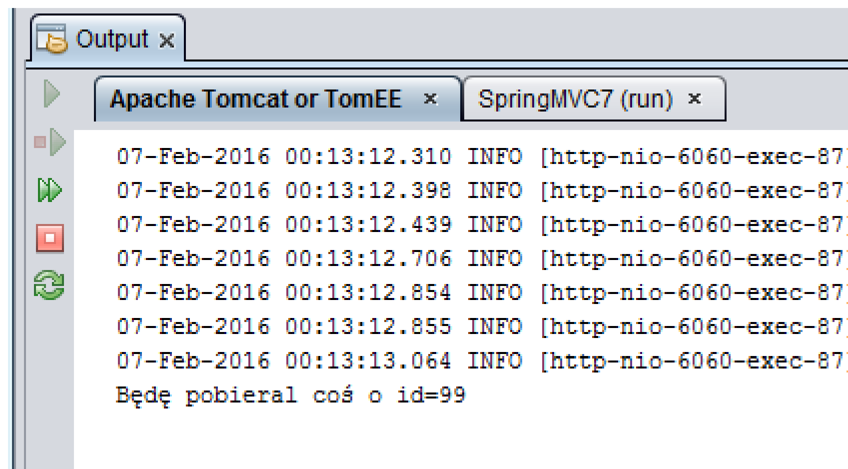
Można je wykorzystać np do filtrowania danych, wyświetlenia szczegółów produktu. Z jednej strony przedstawiona poniżej metoda jest lepsza od zmiennych tablicowych – ponieważ jest znacznie mniej roboty, z drugiej jest gorsza o tyle że tutaj podanie wartości parametru jest obligatoryjne. Nie możemy go pominąć, albo podać tylko częsci parametrów przy wywołaniu.



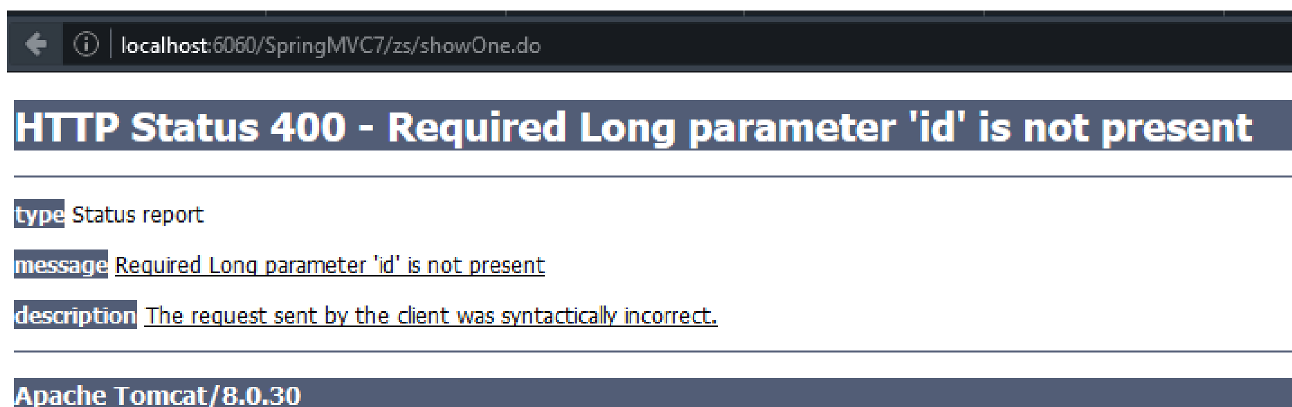
Pokawia się tutaj jedna nowa rzecz: @RequestParam. W parametrze adnotacji podaję nazwę parametru w pasku , którego wartość następnie trafia do mojej zmiennej id typu Long. Wywołanie ma postać:

http://localhost:6060/SpringMVC7/zs/showOne.do?id=99

A wynik działania prezentuje się tak:



Jeśli nie podam tego parametru, Spring się o niego upomni:



W razie gdybym w adresie podał:
http://localhost:6060/SpringMVC7/zs/showOne.do?id=

czyli nie podał wartości dla parametru, nie skończy się błędem, do mojej zmiennej zostanie po prostu przypisany null.

W ramach takiego małego rozszerzenia dodam jeszcze, że parametrów może być oczywiście więcej. Chcę przykładowo obsłużyć takie wywołanie:

http://localhost:6060/SpringMVC7/zs/showMustGoOn.do?kategoria=3&podkategoria=7

Mam dwa parametry: kategoria i podkategoria. Dodaję więc jeszcze jedną metodę, tym razem wymieniając @RequestParam dwa razy. Poniżej metoda wraz z wynikiem działania po wywołaniu.

```
27
28      @RequestMapping("/showMustGoOn.do")
29      public String getSome(@RequestParam("kategoria") Long kategoria, @RequestParam("podkategoria") Long podkategoria, Model model) {
30
31          System.out.println("Będę pobieral dane z kategorii=" + kategoria + ", podkategorii=" + podkategoria);
32
33          return "somejsp";
34      }
35
36  }
37
```

```
Output ×
Apache Tomcat or TomEE ×    SpringMVC7 (run) ×
Będę pobieral dane z kategorii=3, podkategorii=7
```

Przechwytywacze

Kod źródłowy z przykładami do tego rozdziału możesz pobrać pod adresem:

http://jsystems.pl/storage/spring/springmvc8.zip

Przechwytywacze to bardzo przydatne narzędzie pozwalające na kontrolę przepływu. Dzięki nim możemy wykonywać operacje przed każdym wywołaniem, po nim ale przed renderowaniem widoku lub na sam koniec. Gdzie to się może przydać? Od monitoringu wydajności (liczymy czas od wywołania do zakończenia całej operacji) , do czegoś w rodzaju filtrów – np. Ograniczających dostęp do wybranych podstron ze wskazanych adresów IP.

Zaczynamy od dodania klasy implementującej interfejs HandlerInterceptor (org.springframework.web.servlet.HandlerInterceptor).

```
13  /**
14   *
15   * @author andrzej
16   */
17  public class Przechwytywacz implements HandlerInterceptor {
18
19      @Override
20      public boolean preHandle(HttpServletRequest hsr, HttpServletResponse hsr1, Object o) throws Exception {
21          System.out.println("preHandle!");
22          return true;
23      }
24
25      @Override
26      public void postHandle(HttpServletRequest hsr, HttpServletResponse hsr1, Object o, ModelAndView mav) throws Exception {
27          System.out.println("postHandle!");
28
29      }
30
31      @Override
32      public void afterCompletion(HttpServletRequest hsr, HttpServletResponse hsr1, Object o, Exception excptn) throws Exception {
33          System.out.println("afterCompletion!");
34
35      }
36
37  }
38
```

Będziemy musieli zaimplementować trzy metody wymagane przez ten interfejs.

PreHandle – jest wywoływana przed kontrolerem obsługującym dane żądanie. Metoda ta zwraca true lub false, a w zależności od tego co zwróci – request jest przekazywany do kontrolera lub nie. Jeśli zwróci true, kontroler przejmie dalszą obsługę żądania. I to jest miejsce gdzie moglibyśmy np ograniczyć dostęp do wybranych adresów z jakiejś wybranej puli adresowej.

PostHandle – jest wywoływana po zadziałaniu kontrolera, ale jeszcze przed renderowaniem widoku. Pozwala też manipulować na danych modelu przed wyświetleniem widoku. To może być

miejsce gdzie byśmy ograniczyli wyświetlane dane w zależności od tego jaki zalogowany użytkownik ich żąda.

AfterCompletion – wywoływana na koniec, już po zrenderowaniu i wyswietleniu widoku. Tę metodę moglibyśmy wykorzystać do wyliczania czasu przetworzenia całego żądania.

Zamiast implementować interfejs HandlerInterceptor, możesz dziedziczyć po klasie HandlerInterceptorAdapter i przesłonić tylko wybrane metody. Jak kto woli, jak komu wygodnie :)

W tym pierwszym przykładzie ograniczam się tylko do wyswietlenia stosownych komunikatów w poszczególnych metodach – to nam pozwoli zaobserwować kolejność wykonywanych operacji.

Dodaję też zwyczajny kontroler z metodą obsługującą żądanie http:

```
12  /**
13   *
14   * @author andrzej
15   */
16
17  @Controller
18  public class JakisKontroler {
19
20      @RequestMapping(value="jakieswywolanie.do")
21      public String jakasMetoda(Model model){
22          System.out.println("jakiś napis w kontrolerze...");
23          return "jakiesjsp";
24      }
25  }
26
```

Zawartość pliku *****-servlet została wzbogacona o fragment z tagami <mvc:interceptors>:

```
20
21      <mvc:annotation-driven />
22
23
24      <context:component-scan base-package="pl.jsystems.spring.controller"/>
25
26      <mvc:interceptors>
27          <bean class="pl.jsystems.spring.utils.Przechwytywacz"/>
28      </mvc:interceptors>
29
30      <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
31          <property name="prefix" value="/WEB-INF/jsp/"/>
32          <property name="suffix" value=".jsp"/>
33      </bean>
34
```

Działa to w ten sposób, że przy każdym żądaniu Dispatcher przetwarza listę interceptorów i wywołuje dla każdego z nich stosowne interceptory. Możesz pomiędzy tagami <mvc:interceptors> i

</ mvc:interceptors> wstawić kolejne beany z odwołaniami do klas implementujących interfejst HandlerInterceptor lub dziedziczących po klasie HandlerInterceptorAdapter i wszystkie one będą wywoływane.

Wywołuję adres obsługiwany przez nasz kontroler:

```
Info:   Mapped "{[/jakieswywolanie.do],methods=[],params=[],headers=[]
Info:   FrameworkServlet 'mapet': initialization completed in 964 ms
Info:   preHandle!
Info:   jakiś napis w kontrolerze...
Info:   postHandle!
Info:   afterCompletion!
```

Drobna uwaga – jedna z pierwszych myśli jaka mi przyszła do głowy po zapoznaniu się z tym mechanizmem – to czy można ograniczyć działanie interceptorów tylko do wybranych adresów? Przykładowo jakiejś pilnie strzeżonej części aplikacji? W samej konfiguracji interceptorów tego nie ustawimy, możemy za to stosunkowo łatwo sobie to samemu zaimplementować, wykorzystując fakt że w interceptorze mamy cały czas dostęp do obiektu klasy HttpServlerRequest naszego żądania.

Przykładowy kod mógłby wyglądać tak:

```
if(request.getRequestURI.endsWith("tajnapodstrona")) {
        response.sendRedirect("zalogujSieNajpierw.do");
}
```

Najprostszy formularz

Kod źródłowy aplikacji którą tworzę w niniejszym kursie jest do pobrania z adresu:
http://www.jsystems.pl/storage/spring/springform1.zip

Zrobimy sobie najprostszy możliwy formularz do dodawania samochodów. Struktura plików konfiguracyjnych jest identyczna jak w poprzednich rozdziałach, tak więc od razu przejdę do części właściwej tj. Tego co potrzebne do stworzenia i obsługi formularza.

Zaczynamy od stworzenia klasy domenowej – czyli klasy której obiekty będą reprezentować dodawane samochody.

```
7
8   /**
9    *
10   * @author andrzej
11   */
12  public class Samochod {
13
14      private String marka;
15      private String model;
16      private String numerRejestracyjny;
17
18      @Override
19      public String toString() {
20          return "Samochod{" + "marka=" + marka + ", model=" +
21                  model + ", numerRejestracyjny=" + numerRejestracyjny + '}';
22      }
23
24      public Samochod(){}
25      public Samochod(String marka,String model, String numerRejestracyjny){
26          this.marka=marka;
27          this.model=model;
28          this.numerRejestracyjny=numerRejestracyjny;
29      }
```

Klasa posiada trzy pola które będziemy uzupełniać poprzez formularz. Pola są prywatne, ponieważ w klasie dostępne są również settery i gettery do nich. Te musimy posiadać ze względu na wymagania samego Springa. Przeciążona metoda toString ani konstruktory widoczne poniżej nie są wymagane. Stworzyłem je wyłącznie dla własnej wygody przy późniejszej pracy.

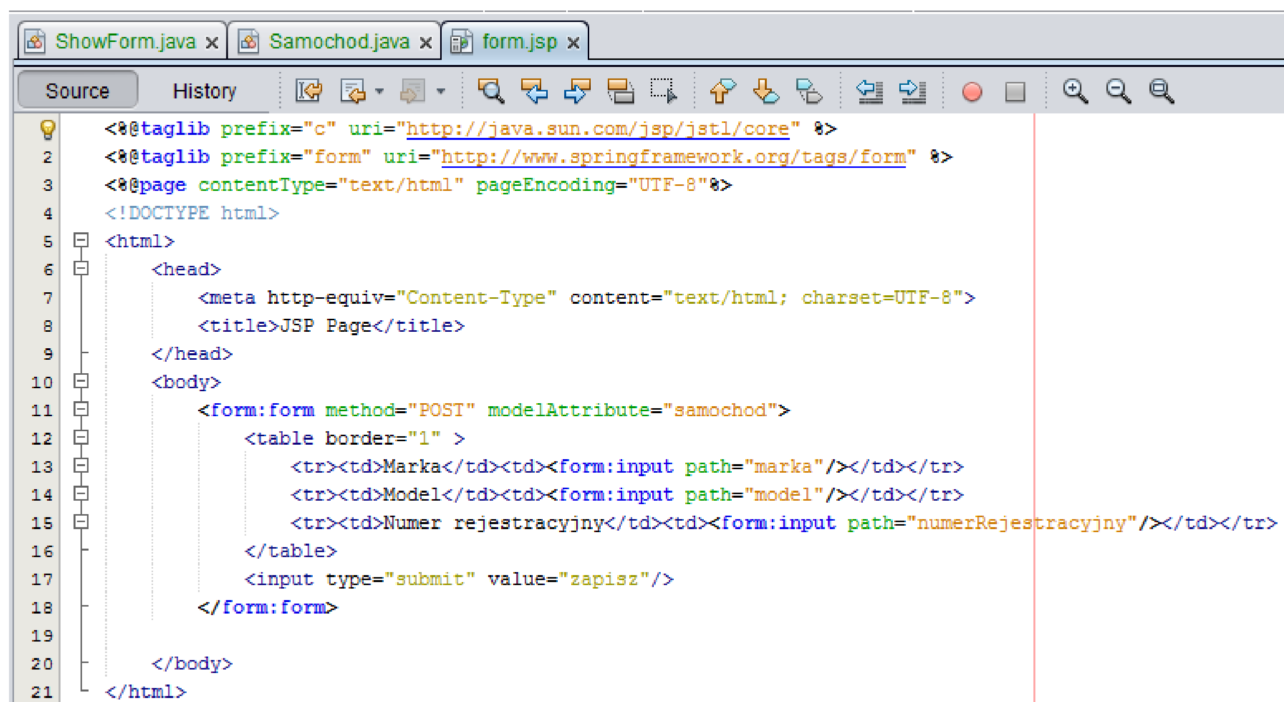
Przejdźmy teraz do klasy kontrolera:

```
13
14  /**
15   *
16   * @author andrzej
17   */
18  @Controller
19  @RequestMapping(value = "formularz")
20  public class ShowForm {
21
22      /*
23       W związku z istnieniej parametru params = "nowy" metoda zostanie wywołana
24       wyłącznie wtedy, kiedy w wywołaniu zostanie podany ten parametr
25       */
26      @RequestMapping(method = RequestMethod.GET, params = "nowy")
27      public String viewNewForm(Model model) {
28          System.out.println("nowy samochód!");
29          model.addAttribute("samochod", new Samochod());
30          return "form";
31      }
32
33      @RequestMapping(method = RequestMethod.GET)
34      public String viewForm(Model model) {
35          System.out.println("nowy samochód!");
36          model.addAttribute("samochod", new Samochod("Skoda","Rapid","WWL 12345"));
37          return "form";
38      }
39
40      @RequestMapping(method = RequestMethod.POST)
41      public String postThis(Samochod samochod) {
42          System.out.println("przesłano dane nowego samochodu");
43          System.out.println(samochod.toString());
44          return "form";
45      }
46
```

Spring obsługuje adresy zaczynające się od "/formularze" – kwestia konfiguracji w web.xml. Sama klasa kontrolera obsługuje więc adres : /formularze/formularz
Jest to mapowanie na poziomie klasy omawiane we wcześniejszych rozdziałach. Kontroler zawiera metody które będą obsługiwały ten właśnie adres wywołania, z tym że jedna z nich jeśli będzie to żądanie POST, dwie pozostałe GET – ale jedna tylko jeśli wystąpi dodatkowy parametr. Nie ma oczywiście problemu by każda metoda mapowała zupełnie osobny adres. Należałoby wtedy usunąć mapowanie na poziomie klasy a dodać parametr value do adnotacji @RequestMapping właściwych metod.
Zacznijmy od przyjrzenia się metodzie viewForm w liniach 33-38. Ponieważ mamy mapowanie na poziomie klasy , w @RequestMapping dla tej metody dodaję tylko metodę wywołania – GET. Tak więc gdy ktoś wywoła adres /formularze/formularz bez żadnego dodatkowego parametru, właśnie ta metoda zostanie wywołana. Do modelu dodaję samochód marki "Skoda". Obiekt ten pod nazwą "samochod" zostaje przekazany do warstwy widoku i tam wyświetlony. Z tego wynika, że jeśli ktoś wywoła adres /formularze/formularz bez parametru, to w jego formularzu pola będą uzupełnione informacjami o naszej przykładowej Skodzie. Jako strona JSP której chcę użyć do wyświetlenia formularza wybrałem plik form.jsp którego nazwę zwracam w linii 37.

Porównajmy tę metodę z metodą viewNewForm z linii 27-31. W związku z mapowaniem na poziomie klasy, metoda ta obsługuje ten sam adres (/formularze/formularz), jednak ponieważ mamy dopisek params="nowy" zostanie ona wywołana tylko wtedy, gdy zostanie przez pasek adresu przekazany parametr o nazwie "nowy". Tak więc w przypadku wywołania adresu "/formularze/formularz" zostanie wywołana poprzednia metoda viewForm, a w przypadku wywołania "/formularze/formularz?nowy" metoda viewNewForm. W pierwszym przypadku do modelu zostaje przekazany obiekt samochodu z uzupełnionymi danymi, w drugim nowy pusty obiekt. Obie metody są obsługiwane przez tę samą stronę JSP.

Do metody postThis wrócimy po omówieniu formularza, ponieważ jest ona wywoływana dopiero po jego zatwierdzeniu. Przyjrzyjmy się teraz zawartości pliku form.jsp obsługującego nasz formularz:

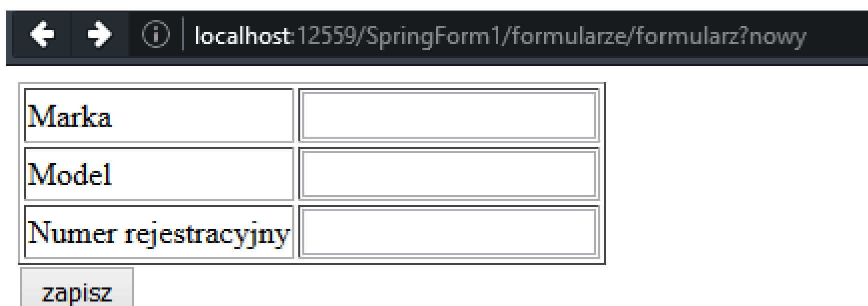


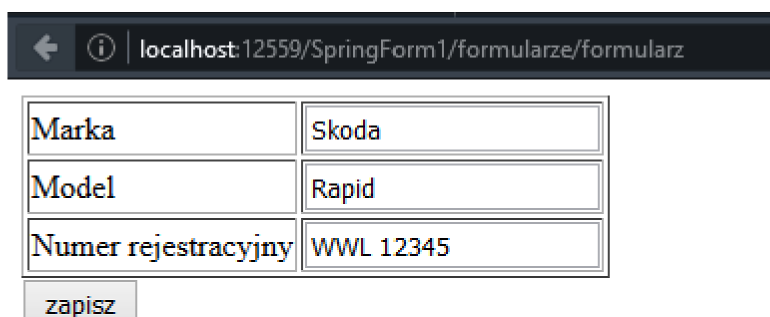
```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <form:form method="POST" modelAttribute="samochod">
            <table border="1" >
                <tr><td>Marka</td><td><form:input path="marka"/></td></tr>
                <tr><td>Model</td><td><form:input path="model"/></td></tr>
                <tr><td>Numer rejestracyjny</td><td><form:input path="numerRejestracyjny"/></td></tr>
            </table>
            <input type="submit" value="zapisz"/>
        </form:form>

    </body>
</html>
```

Koniecznie musisz posiadać odwołanie do biblioteki tagów takie jak u mnie w linii 2. Często zdarzało mi się o tym zapominać a potem się dziwić czemu formularz nie działa. Właściwy formularz to linie 11-18. Cały formularz musi zostać objęty tagami <form:form> i </form:form>, co chyba nie jest zaskakujące dla osób mających styczność z podstawami HTML :) Jak widać w linii 11 formularz obsługuje przesłanie danych metodą POST. Nie ma jednak znacznika "action" ... Oznacza to że dane z formularza zostaną przesłane pod ten sam adres pod którym się teraz znajdujemy, z tym że wywołanie będzie typu POST. Parametr "modelAttribute" określa nazwę obiektu którego pola będziemy uzupełniać w formularzu. Przyjrzyj się liniom 29 i 36 w kontrolerze. Nazwa obiektu musi się tutaj pokrywać. Wróć teraz na moment i przyjrzyj się polom w klasie Samochód które deklarowałem kilka kroków wcześniej. Teraz spójrz na linie 13-15 niniejszego formularza. Jak widzisz, kolejne pola do wprowadzania danych mają parametr "path" którego wartość pokrywać się musi z nazwami pól obiektu modelowego przekazywanego do i z formularza. Ponadto pola te muszą posiadać gettery i settery aby można się było do nich odwoływać z formularza.

W zależności od tego czy przekazywany obiekt będzie uzupełniony danymi, nasze pola do wprowadzania danych będą uzupełnione lub nie. Wszystko zależy od zawartości obiektu przekazywanego do widoku pod nazwą wskazaną przez "modelAttribute" w linii 11. W linii 17 jest zwyczajny guzik zatwierdzenia formularza.

Przypomnijmy sobie teraz sposób wywołania metod viewForm i viewNewForm, a następnie przyjrzyjmy się obu poniższym ilustracjom:





Porównaj adresy wywołań i zobacz co pojawia się w polach edycyjnych. Teraz już wszystko powinno być jasne ;)

Pozostaje nam obsługa zatwierdzenia formularza. Wróćmy na chwilę do naszego kontrolera:

```
39
40    @RequestMapping(method = RequestMethod.POST)
41    public String postThis(Samochod samochod) {
42        System.out.println("przesłano dane nowego samochodu");
43        System.out.println(samochod.toString());
44        return "form";
45    }
```

Do obsługi wywołania POST służy metoda postThis. Obiekt uprzednio przekazany do modelu, a następnie uzupełniony w formularzu a dalej wraca do metody postThis przez parametr. Nazwa tego parametru metody nie musi być zgodna z nazwą w modelAttribute. Zawartość konsoli po zatwierdzeniu formularza:

```
Info:   Pre-instantiating singletons in org.springframework.beans.factory.support.Def
Info:   Mapped "{[/formularz],methods=[GET],params=[],headers=[],consumes=[],produces
Info:   Mapped "{[/formularz],methods=[POST],params=[],headers=[],consumes=[],produce
Info:   Mapped "{[/formularz],methods=[GET],params=[nowy],headers=[],consumes=[],prod
Info:   FrameworkServlet 'formularze': initialization completed in 937 ms
Info:   przesłano dane nowego samochodu
Info:   Samochod{marka=Skoda, model=Rapid, numerRejestracyjny=WWL 12345}
```

Walidacja formularzy

Kod źródłowy aplikacji którą tworzę w niniejszym kursie jest do pobrania z adresu:
http://www.jsystems.pl/storage/spring/springform2.zip

W tym rozdziale przerobimy nieco poprzedni przykład. Klasę modelową "Samochód" wzbogacamy o kilka dodatkowych adnotacji. @Min – określa minimalną długość wprowadzanego do tego pola ciągu tekstowego. @Size pozwala określić jego długość od-do. W obu przypadkach parametr "message" pozwala określić wiadomość która zostanie wyświetlona na formularzu w przypadku nie przejścia walidacji dla danego pola. Jeśli nie określisz własnego komunikatu, zostanie wyświetlony domyślny szablonowy od Springa.

```
12  /**
13   *
14   * @author andrzej
15   */
16  public class Samochod {
17
18      @Min(value = 3, message = "marka musi się składać z minimum 3 znaków")
19      private String marka;
20
21      //@NotNull(message = "musisz uzupelnić model")//nie bangla.. :(
22      @Min(value = 1,message = "musisz uzupełnić model")
23      private String model;
24
25      @Size(min = 7, max = 9, message = "numer rejestracyjny może się składać z 7-8 znaków")
26      private String numerRejestracyjny;
27
```

Metodę obsługującą żądanie POST w kontrolerze uzupełniam o adnotację @Valid, oraz dodatkowy parametr. @Valid oznacza, że obiekt "samochod" zostanie przesłany z formularza dopiero po pomyślnej walidacji danych. Obiekt klasy BindingResult pozwala nam wyciągnąć konkretne błędy które się pojawiają. Nie wykorzystuję tutaj jego możliwości.

```
39       */
40      @RequestMapping(method = RequestMethod.POST)
41      public String postThis(@Valid Samochod samochod,BindingResult br) {
42          System.out.println(samochod.toString());
43          return "form";
44      }
45
46  }
```

Przejdźmy do pliku JSP formularza:

```
11    <form:form method="POST" modelAttribute="samochod">
12        <table border="1" >
13            <tr><td>Marka</td><td><form:input path="marka"/>
14                    <font color="red"><form:errors path="marka"/></font>
15            </td></tr>
16            <tr><td>Model</td><td><form:input path="model"/>
17                    <font color="red"><form:errors path="model"/></font>
18            </td></tr>
19            <tr><td>Numer rejestracyjny</td><td><form:input path="numerRejestracyjny"/>
20                    <font color="red"><form:errors path="numerRejestracyjny"/></font>
21            </td></tr>
22        </table>
23        <input type="submit" value="zapisz"/>
24    </form:form>
```

Nowa rzecz która się tutaj pojawia to znacznik:

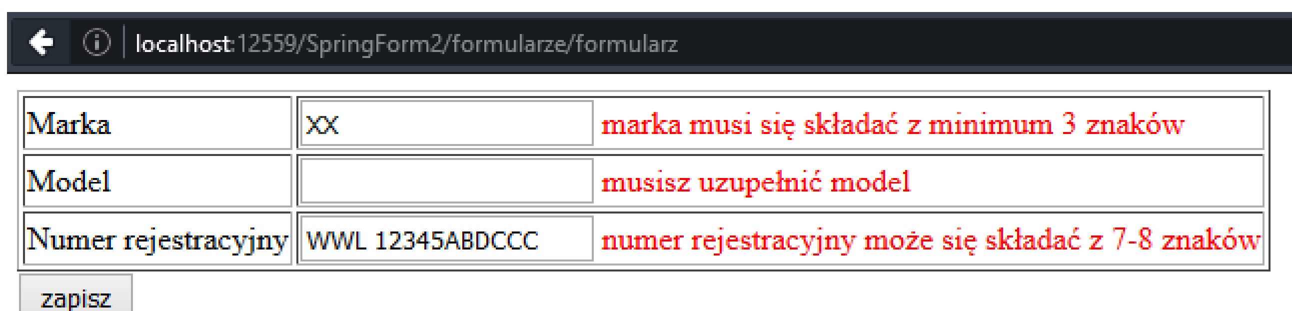
<form:errors path="marka"/>

Określa on, że w tym miejscu mają zostać wyświetlone ewentualne błędy związane z danym polem – i tak dla każdego pola :)

Inne przykładowe adnotacje do walidacji:

@Max – określa maksymalną długość ciągu
@NotNull – określa że pole nie może pozostawać puste
@Pattern – pozwala walidować z użyciem wyrażeń regularnych
@NotEmpty – nie pusty ciąg (z hibernate)

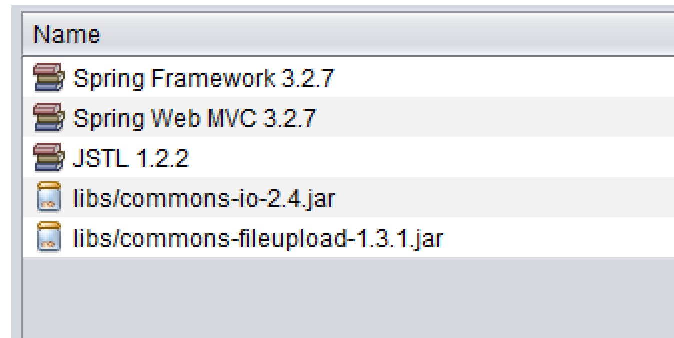
Sposób działania walidacji po próbie zatwierdzenia nie poprawnie wypełnionego formularza:



Upload plików

Kod źródłowy aplikacji którą tworzę w niniejszym kursie jest do pobrania z adresu:
http://www.jsystems.pl/storage/spring/springform3.zip

Dodanie możliwości uploadu plików jest nieco bardziej złożona. Zacznijmy od dodania dwóch niezbędnych bibliotek:



Commons-IO i commons-fileupload. Obie znajdują się w katalog libs projektu przykładowego do tego rozdziału. Skoro już jesteśmy przy rzeczach przyziemnych to od razu dodajmy do naszego pliku ***-servlet.xml beana o ID "multipartResolver" w którego parametrze p:maxUploadSize zdeklarujemymaksymalną wielkość wrzucanego pliku w bajtach. Spring będzie wymagał konfiguracji tego beana, więc nie możemy tego etapu pominąć.

```
23
24      <context:component-scan base-package="pl.jsystems.spring.controller"/>
25
26
27      <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver"
28      p:maxUploadSize="1000000"/>
29
30
31      <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
32          <property name="prefix" value="/WEB-INF/jsp/"/>
33          <property name="suffix" value=".jsp"/>
34      </bean>
35
36  </beans>
37
```

Przejdźmy teraz do przeróbki samego formularza:

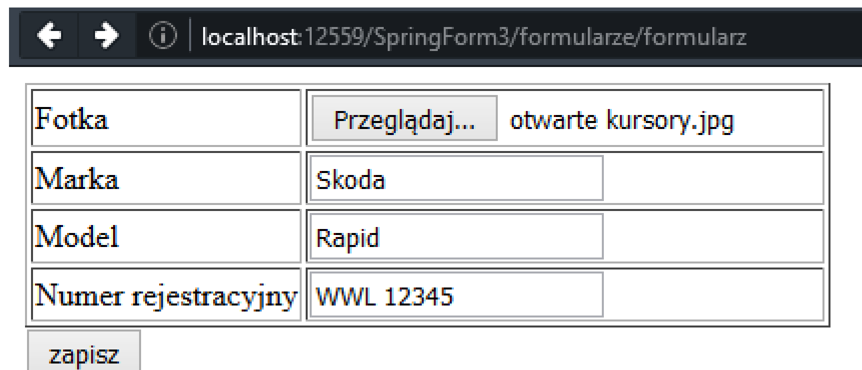
```
10      <body>
11          <form:form method="POST" modelAttribute="samochod"  enctype="multipart/form-data">
12              <table border="1" >
13
14                  <tr><td>Fotka</td><td><input name="fotka"  type="file" />
15                  </td></tr>
16
17                  <tr><td>Marka</td><td><form:input path="marka"/>
18                          <font color="red"><form:errors path="marka"/></font>
19                  </td></tr>
20                  <tr><td>Model</td><td><form:input path="model"/>
21                          <font color="red"><form:errors path="model"/></font>
22                  </td></tr>
23                  <tr><td>Numer rejestracyjny</td><td><form:input path="numerRejestracyjny"/>
24                          <font color="red"><form:errors path="numerRejestracyjny"/></font>
25                  </td></tr>
26              </table>
27              <input type="submit" value="zapisz"/>
28          </form:form>
29
```

Zaczniemy od dodania parametru enctype="multipart/form-data" do forma. Umożliwia on przesyłanie plików (patrz linia 11). Spójrz teraz na linię 14. Tutaj bardzo istotny drobiazg: mamy tutaj input a nie form:input. To nie jest pole klasy Samochod! Będziemy wrzucany obrazek przetwarzać jako osobny parametr wywołania. Przejdźmy teraz do kontrolera obsługującego formularz.

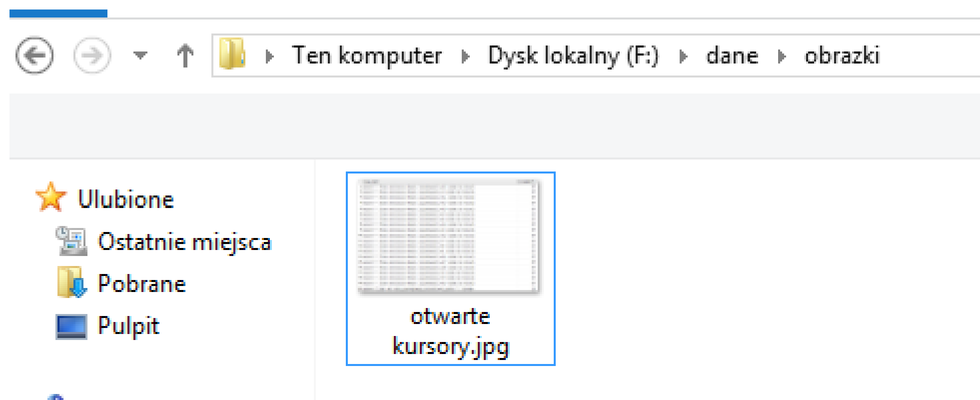
```
39
40      @RequestMapping(method = RequestMethod.POST)
41      public String postThis(@Valid Samochod samochod, BindingResult br,
42              @RequestParam(value = "fotka") MultipartFile fotka) {
43
44          if (fotka.getContentType().equals("image/jpeg")) {
45              System.out.println("format pliku jest OK!");
46          }
47          File f = new File("f:\\dane\\obrazki\\"+fotka.getOriginalFilename());
48          try {
49              FileUtils.writeByteArrayToFile(f, fotka.getBytes());
50          } catch (Exception e) {e.printStackTrace(); }
51
52          System.out.println(samochod.toString());
53          return "form";
54      }
55
```

Doszedł nam nowy parametr do metody postThis. Chodzi o parametr "fotka". Zauważ że nie jest on przekazywany przez model, tylko jako osobny parametr.
Element z linii 44-46 to tylko bajer, weryfikuje czy obrazek jest obrazkiem. Możesz tutaj rzucić jakimś wyjątkiem gdyby okazał się nim nie być. Linie 47-50 to zapisanie pliku na dysku.

Przetestujmy :)



Po zatwierdzeniu formularza plik pojawił się we wskazanym miejscu:



Spring WebFlow

Proste formularze z przejściami

Kod źródłowy z przykładami do tego rozdziału możesz pobrać pod adresem:

http://jsystems.pl/storage/spring/springflow1.zip

Do czego może nam się przydać Spring WebFlow? Ten moduł umożliwia robienie wieloetapowych przepływów – coś na kształt wizardów instalacyjnych. Możemy dzięki niemu w stosunkowo prosty sposób stworzyć np. kilkustronnicowy formularz rejestracyjny.

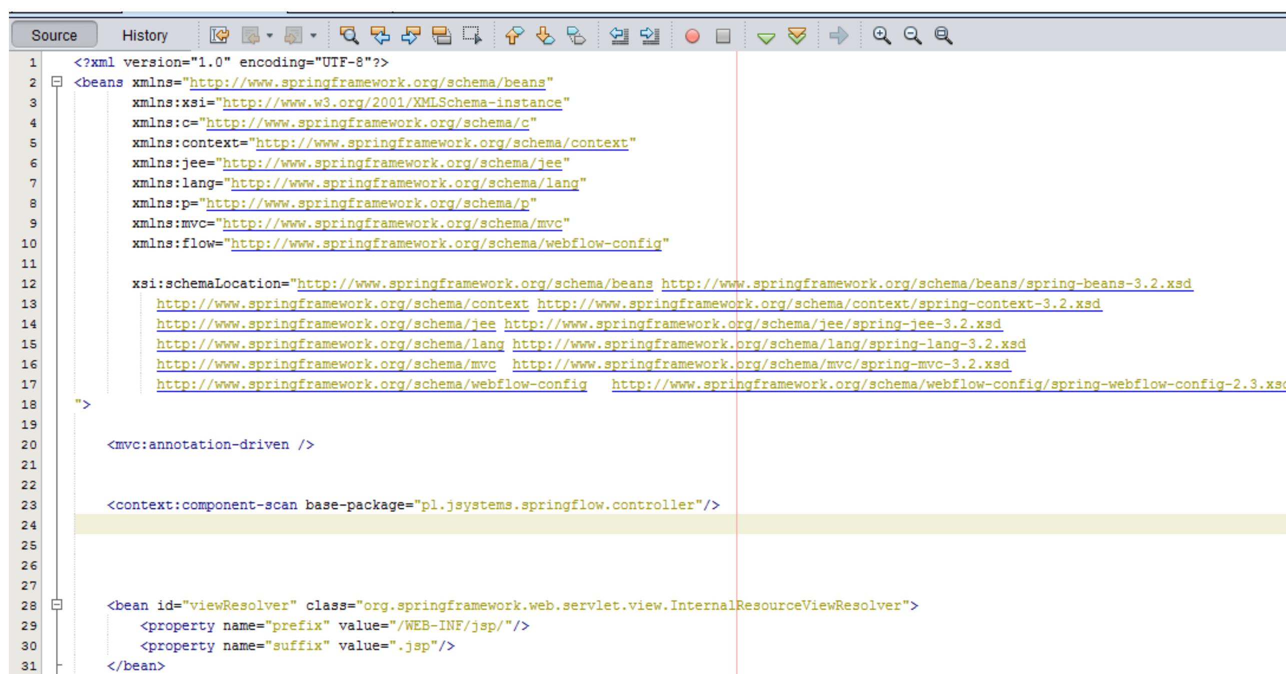
W tym rozdziale rozpoczniemy od dodania dwóch bardzo prostych funkcjonalności na potrzeby sklepu internetowego. Jedna będzie pozwalała zapisać się do newslettera, druga założyć konto w sklepie. Zaczynamy od stworzenia zwyczajnej aplikacji opartej o Spring MVC. Nie pojawia się tutaj nic nowego ani nadzwyczajnego, niemniej zaznaczę kilka kluczowych elementów:

Wpis w web.xml:

```
5
6
7   <servlet>
8       <servlet-name>sklep</servlet-name>
9       <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
10  </servlet>
11  <servlet-mapping>
12      <servlet-name>sklep</servlet-name>
13      <url-pattern>*.do</url-pattern>
14  </servlet-mapping>
15
```

Wszystkie podstrony tej aplikacji z rozszerzeniem .do będą obsługiwane przez Springa.

Do pliku *****-servlet.xml trzeba będzie dodać przestrzenie nazw związane ze Spring WebFlow, więc zrobimy to od razu:




```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:c="http://www.springframework.org/schema/c"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:jee="http://www.springframework.org/schema/jee"
        xmlns:lang="http://www.springframework.org/schema/lang"
        xmlns:p="http://www.springframework.org/schema/p"
        xmlns:mvc="http://www.springframework.org/schema/mvc"
        xmlns:flow="http://www.springframework.org/schema/webflow-config"

        xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
          http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd
          http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.2.xsd
          http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-3.2.xsd
          http://www.springframework.org/schema/mvc  http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd
          http://www.springframework.org/schema/webflow-config   http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.3.xsd
">

    <mvc:annotation-driven />


    <context:component-scan base-package="pl.jsystems.springflow.controller"/>




    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
```

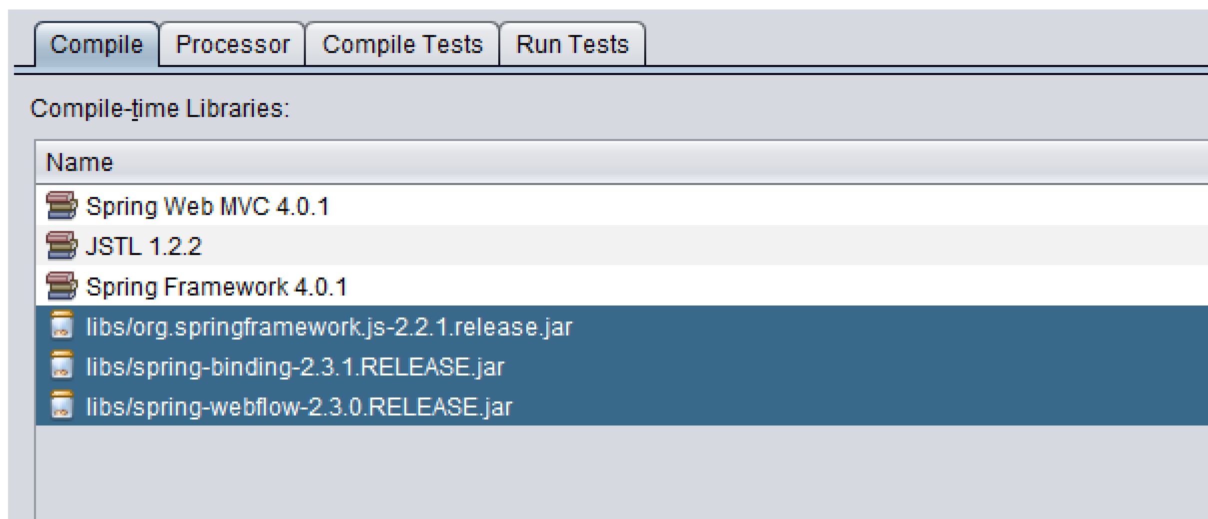
a konkretniej chodzi mi o dodanie przestrzeni nazw:

xmlns:flow="http://www.springframework.org/schema/webflow-config"

oraz jej położenia:

http://www.springframework.org/schema/webflow-config
http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.3.xsd

WebFlow wymaga dodatkowych bibliotek, więc poza tym co zwykle dodać trzeba jeszcze będzie biblioteki spring-webflow,spring-binding i springframework.js:



Wszystkie trzy pliki jar znajdują się z katalogu libs projektu dołączonego do niniejszego artykułu.

Dodaję też najzwyklejszy w świecie kontroler z jedną metodą mapującą adres „/start.do” :

```
12
13  /**
14   *
15   * @author andrzej
16   */
17  @Controller
18  public class Startowa {
19
20      @RequestMapping(value="/start.do")
21      public String startowa(Model model){
22          System.out.println("Ktoś wszedł na stronę startową...");
23          return "startowa";
24      }
25  }
26
```

Do pliku JSP obsługującego to żądanie dodałem dwa linki:

```
15
16          <h3>Witamy w naszym sklepie </h3>
17          <br>
18          <a href="newsletter.do">Zapisz się do newslettera!</a>
19          <br>
20          <a href="rejestracja.do">Rejestracja konta</a>
21
```

I teraz się zacznie... W pliku ****-servlet.xml musimy dodać kilka niezbędnych elementów. Na potrzeby choćby jednego przepływu musimy mieć całą taką konfigurację jaką wprowadziłem w swoim pliku w liniach 29-63... Omówimy elementy zmienne, które będą się różnić w różnych aplikacjach.

Dodamy na początek przepływ który będzie się sprowadzał do dodania swojego adresu email, do newslettera. Nieco później rozbudujemy nasz projekt.

Każdy przepływ będzie miał osobny plik konfiguracyjny w którym opiszemy co po czym ma się uruchamiać i przy spełnieniu jakich warunków. Rejestrujemy te pliki konfiguracyjne w sekcji flow-registry (linie 51-53). Bardzo ważny jest tutaj parametr ID, ponieważ jest on związany z tak zwanym adresem wyzwalającym przepływ. Jeśli adres naszego formularza jest /newsletter.do, to WebFlow będzie poszukiwał w rejestrze wpisu o id newsletter, jeśli adres rejestracji konta w sklepie będzie miał postać /rejestracja.do to będzie szukał wpisu o id rejestracja. Liczy się tylko „końcówka", same adresy wyzwalające mogą mieć postać np. /modul1/newsletter.do.

Ten adres wyzwalający oznacza rozpoczęcie przepływu – w tym przypadku rejestracji do newslettera.

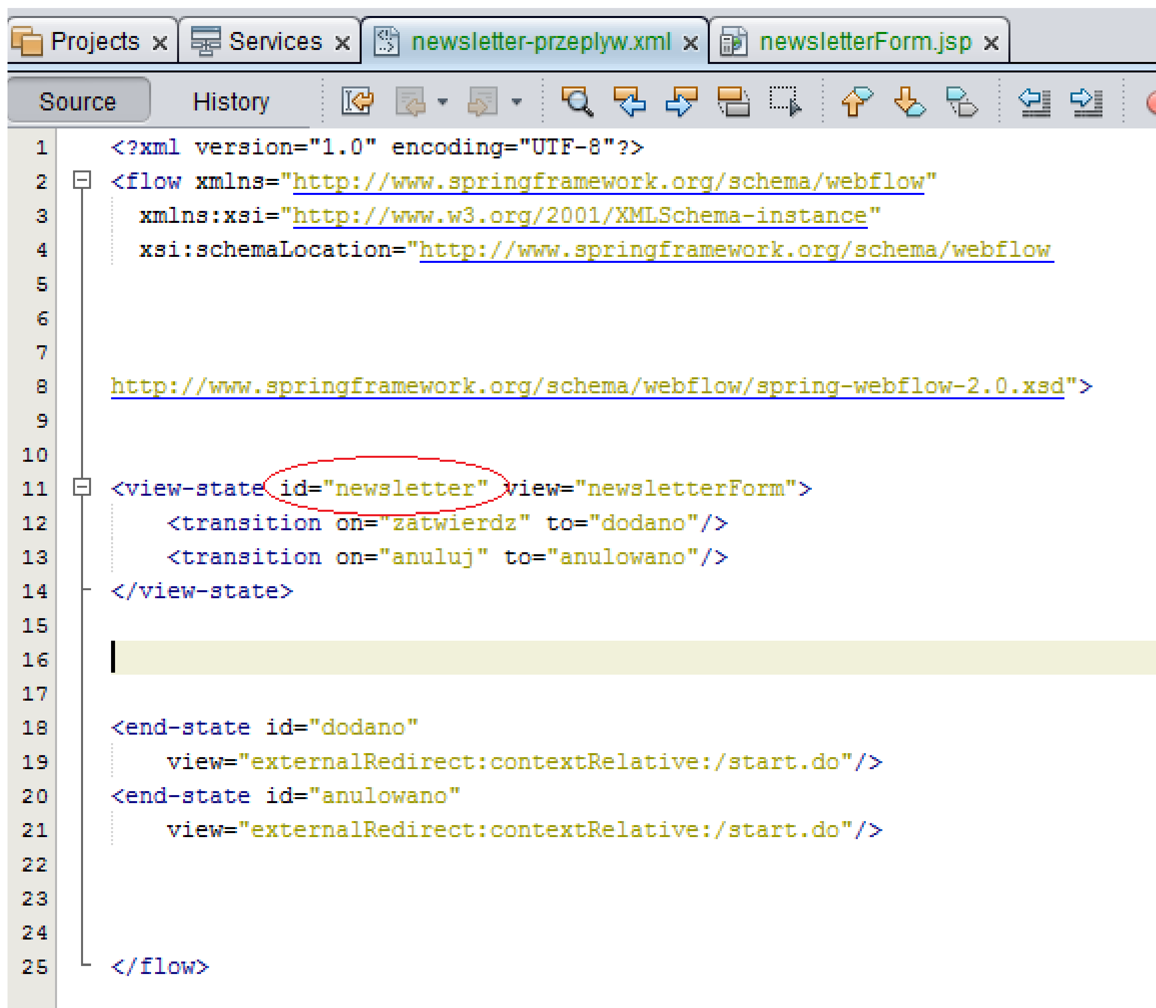
Startowa.java | startowa.jsp | sklep-servlet.xml

Source | History

```xml
    <context:component-scan base-package="pl.jsystems.springflow.controller"/>



    <!-- SPRING SHIT FLOW -->

    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>


 <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /newsletter.do=flowController
            </value>
        </property>
    </bean>


    <bean id="flowController" class="org.springframework.webflow.mvc.servlet.FlowController">
        <property name="flowExecutor" ref="flowExecutor"/>
    </bean>

    <!--Tworzy przeplywy na podstawie flowRegistry -->
    <flow:flow-executor id="flowExecutor" flow-registry="flowRegistry"/>
     <!--Rejestr plików z opisami poszczególnych przeplywów-->
    <flow:flow-registry id="flowRegistry" flow-builder-services="flowBuilderServices">
        <flow:flow-location path="/WEB-INF/flows/newsletter-przeplyw.xml" id="newsletter"/>
    </flow:flow-registry>

    <flow:flow-builder-services id="flowBuilderServices" view-factory-creator="viewFactoryCreator"/>

    <bean id="viewFactoryCreator" class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator">
        <property name="viewResolvers">
            <list>
                <ref bean="viewResolver"/>
            </list>
        </property>
    </bean>


</beans>
```

Jak widać w rejestrze przepływów dla przepływu o id „newsletter” wskazałem plik newsletter-przeplyw.xml znajdujący się w katalogu /WEB-INF/flows. Tworzymy więc podkatalog flows w WEB-INF i umieszczamy w nim na razie pusty plik o nazwie newsletter-przeplyw.xml

Wewnątrz tego pliku umieszczamy taką konfigurację:



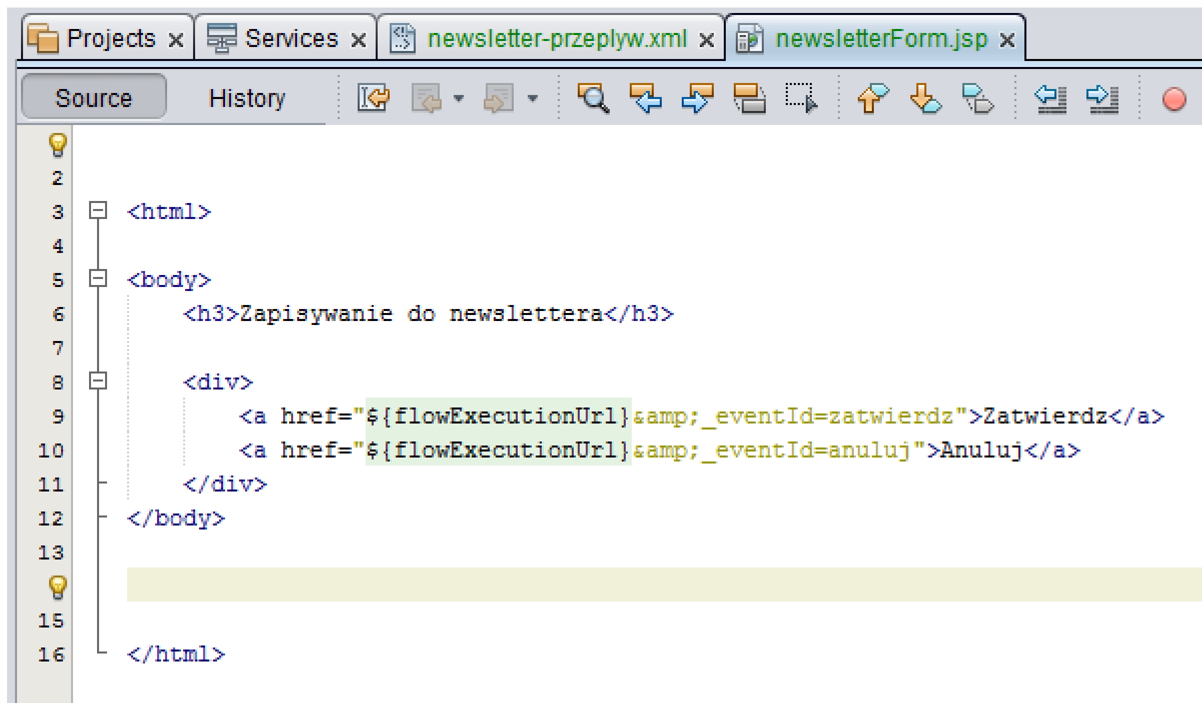
Każdy element zamieszczony tutaj to jeden „krok” w naszym przepływie. Takim krokiem może być wyświetlenie strony, wykonanie jakiejś akcji, podjęcie decyzji warunkowej. Każdy przepływ musi mieć dokładnie jeden stan początkowy i przynajmniej jeden końcowy. Oznaczają one pierwszy i ostatni krok w przepływie. Pomiędzy tymi krokami możemy dodać kolejne, nic nie stoi na przeszkodzie. W naszym mini projekcie w tym przepływie będą tylko dwa kroki. Jeden to wyświetlenie formularza do którego wprowadzimy nasz adres email, drugi to stan końcowy który po prostu przekieruje nas do strony początkowej. Nieco później zadbamy o to, by w zależności od tego czy zatwierdzimy czy anulujemy formularz wyświetlił się stosowny komunikat.

Przyjrzyjmy sięteraz elementowi w liniach 11-14. Oznacza on wyświetlenie strony jsp o nazwie newsletterForm (która powinna znaleźć się w WEB-INF/jsp – jak mamy skonfigurowane w beanie o ID viewResolver w pliku ****-servlet.xml). Zauważ, że element ten ma również id „newsletter”. To jest stan początkowy, co oznacza że jako pierwszy krok w tym przepływie zostanie wyświetlona strona newsletterForm. Pojawiają nam się tutaj też znaczniki <transition on =”x” to=”y”/> Co to oznacza? Jeśli pojawi się (na razie bliżej nie określone) „COŚ” o nazwie „zatwierdz”, sterowanie

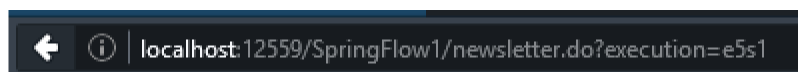
ma zostać przekierowane do elementu „dodano”. Jeśli pojawi się takie samo „COŚ” o nazwie „anuluj”, sterowanie ma zostać przekierowane do elementu o nazwie „anulowano”.

To „COŚ” to są komunikaty wysyłane z poziomu widoku i za chwilę się tym zajmiemy. W tej chwili przyjrzyjmy się jeszcze elementom z linii 18-21. Są to dwa możliwe stany końcowe. Oba w tej chwili robią to samo – przekierowują do strony początkowej. Który ze stanów końcowych nastąpi zależy od tego, czy z widoku zostanie przesłany komunikat „zatwiedz” czy „anuluj”.

Przejdźmy teraz do naszego pliku widoku „newsletterForm” który jest wyświetlany w pierwszym kroku naszego przepływu. Niedługo pojawią się tutaj też pola do których wprowadzać będziemy dane, teraz interesuje nas tylko przepływ jako taki.



Może zastanawiać dziwny adres linka który podałem. Zacznijmy od lewej. Pojawia się tutaj **${flowExecutionUrl}** - dzięki temu fragmentowi powracamy do przepływu w którego trakcie jesteśmy. Ten znacznik widoczny jest zresztą w pasku adresu:



Zapisywanie do newslettera

Zatwierdz Anuluj

Dalej mamy tajemniczy ciąg zawierający **_eventId=zatwierdz ,** oraz drugi podobny **_eventId=anuluj** . Myślę że kiedy przyjrzysz się zaznaczonemu poniżej fragmentowi pliku konfiguracyjnego tego przepływu to wszystko stanie się jasne:

```
10
11  <view-state id="newsletter" view="newsletterForm">
12      <transition on="zatwierdz" to="dodano"/>
13      <transition on="anuluj" to="anulowano"/>
14  </view-state>
15
```

Gdybyś jednak nie wypił porannej mocnej kawy to wyjaśniam – to jest to nasze „COŚ" które wywołane powoduje przejście do kolejnego stanu. Ale jakiego stanu? W pierwszym przypadku do stanu o nazwie „dodano", w drugim do stanu o nazwie „anulowano". A teraz przyjrzyj się znajdującemu się nieco niżej fragmentowi pliku newsletter-przeplyw.xml:

```
16
17
18  <end-state id="dodano"
19      view="externalRedirect:contextRelative:/start.do"/>
20  <end-state id="anulowano"
21      view="externalRedirect:contextRelative:/start.do"/>
22
23
```

Kliknięcie tych linków spowoduje przejście do stanów końcowych przepływu. Oczywiście można by tutaj było dać jakiś inny stan – choćby kolejny widok. Wszystko fajno, ale trochę słabe takie zapisywanie się do newslettera, jeśli nigdzie nie podajemy swojego adresu email ;)

Wzbogaćmy teraz naszą aplikację o obiekt w którym będziemy gromadzili dane na temat użytkownika. Uzyszkownik będzie obiektem klasy którą musimy stworzyć:

```
9
10  /**
11   *
12   * @author andrzej
13   */
14  public class Uzyszkodnik implements Serializable{
15
16      private Long id;
17      private String imie;
18      private String nazwisko;
19      private String email;
20
21      @Override
22      public String toString() {
23          return "Uzyszkodnik{" + "id=" + getId() + ", imie=" + getImie() + ", nazwisko=" + getNazwisko() + ", email=" + getEmail() + '}';
24      }
25
```

Moja klasa zawiera cztery proste pola . Nie są ujęte na screenie, ale klasa posiada też gettery i settery do tych pól. Klasa musi być serializowalna.

Dodaję też jakieś fake'owe dao do obiektów klasy Uzyszkodnik. Metoda save ma przyjmować obiekt użytkownika po wypełnieniu formularza i zwracać ID pod jakim obiekt ten wylądował w bazie. Tutaj jest to jakaś wartość „na sztywno", abyśmy teraz nie komplikowali sobie życia sprawami związanymi z bazami danych. Jest też metoda print która przyjmuje przez parametr obiekt użytkownika i wypisuje jego zawartość na konsolę. Oczywiście w żadnym normalnym DAO taka metoda by się nie pojawiała, tutaj jest jedynie do celów prezentacyjnych działania przepływów.

```
10  /**
11   *
12   * @author andrzej
13   */
14  public class UzyszkodnikDao {
15
16      public Long save(Uzyszkodnik u){
17          System.out.println("zapisuję użytkownika "+u.toString());
18          return 123L;
19      }
20
21      public void print(Uzyszkodnik u){
22          System.out.println("Skompletowany użyszkodnik : "+u.toString());
23      }
24  }
25
```

Ponieważ do obiektu klasy UzyszkodnikDao zamierzam się odwoływać w ramach mojego przepływu na zasadzie „uzyszkodnikDao.jakasMetoda()” w pliku konfiguracji przepływu, Spring musi wiedzieć co to takiego jest ten uzyszkodnikDao. W związku z tym deklarujemy tę klasę jako bean w pliku ***-servlet.xml:

```
<bean id="uzytkownikDao" class="pl.jsystems.springflow.dao.UzyszkodnikDao"/>
```

Przejdźmy teraz do pliku konfiguracji przepływu. Pojawiło się tutaj kilka nowych rzeczy:

```
sklep-servlet.xml × | newsletter-przeplyw.xml × | newsletterForm.jsp × | Uzyszkodnik.java × | Uzyszkodni
Source   History
1  <?xml version="1.0" encoding="UTF-8"?>
2  <flow xmlns="http://www.springframework.org/schema/webflow"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://www.springframework.org/schema/webflow
5
6
7
8  http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">
9                                                                       1.
10 <var name="uzyszkodnik" class="pl.jsystems.springflow.model.Uzyszkodnik"/>
11
12 <view-state id="newsletter" view="newsletterForm" model="uzyszkodnik">  2.
13     <transition on="zatwierdz" to="zapiszUzytkownika"/>
14     <transition on="anuluj" to="anulowano"/>
15 </view-state>                                                           3.
16
17 <action-state id="zapiszUzytkownika">
18     <evaluate expression="uzytkownikDao.save(uzyszkodnik)" result="uzyszkodnik.id"   />
19     <transition to="wydrukujUzytkownika" />
20 </action-state>
21                                                                         4.
22 <action-state id="wydrukujUzytkownika">
23     <evaluate expression="uzytkownikDao.print(uzyszkodnik)" />
24     <transition to="dodano" />
25 </action-state>
26
27 <end-state id="dodano"
28     view="externalRedirect:contextRelative:/start.do"/>
29 <end-state id="anulowano"
30     view="externalRedirect:contextRelative:/start.do"/>
31
32
33
34 </flow>
```

Element oznaczony jako nr 1 to zmienna która widoczna będzie w całym przepływie. Spring stworzy obiekt o nazwie podanej w parametrze „name” klasy podanej w parametrze „class” i umieści ją w przepływie. Nasz widoczek newsletterForm zyskał nowego kolegę pod postacią parametru model. Zauważ że jest tutaj podana nazwa naszej nowej zmiennej – uzyszkodnik. W formularzu będziemy uzupełniać pola obiektu – więc wypadałoby wskazać jakiego :). Bez tego dodatku w prawdzie przepływ będzie działał (w sensie będą działały przejścia i nie będzie sypało żadnymi błędami), ale wypełniane pola pójdą „w kosmos” a obiekt na koniec przepływu pozostanie pusty (sprawdzone organoleptycznie). W linii 13 też jest mała zmiana – przejście następuje nie do końcowego stanu a do stanu „zapiszUzytkownika” który to się właśnie pojawił. Element action-state oznacza wykonanie jakiejś akcji – wywołania metody , a nie jak view-state wyświetlenia widoku. Przyjrzyjmy się więc teraz temu elementowi. „Evaluate expression” wskazuje metodę która ma zostać wykonana. Jest to metoda „save” z obiektu klasy uzytkownikDao któgo bean przed momentem dodawaliśmy do pliku ****-servlet.xml . Do tej metody jako parametr przekazuje obiekt uzyszkodnik który.... też przed momentem definiowaliśmy ;). Chodzi więc o obiekt który sobie w ramach naszego przepływu uzupełniamy. W naszym formularzu który jest uzupełniany stan temu uzupełnialiśmy pola właśnie tego obiektu. W tym kroku przekazujemy go do metody by zapisać go w bazie. Ten obiekt jest już uzupełniony wartościami wprowadzonymi przez formularz. Pojawia się tutaj tez parametr result. Obiekt który zapisujemy wyląduje w bazie pod jakimś identyfikatorem, a identyfikator ten zwracany jest przez metodę save. Przyda się on do późniejszego wyświetlenia, lub np. podpinania kolejnych obiektów pod ten. Result spowoduje podstawienie pod pole id naszego przepływowego obiektu wartości zwracanej przez metodę wywoływaną w evaluate expression. Action-state po wykonaniu oczekiwanych operacji przekazuje sterowanie do stanu „wydrukujUzytkownika”. Stan wydrukujUzytkownika to stan akcji który wywołuje stworzoną w dao metodę print i przekazuje jej uzupełniony obiekt do wyswietlenia. Dalej przepływ przekazywany jest do stanu końcowego dodano.

Podsumowując – pierwszy stan to stan widoku wyświetlający formularz z pliku newsletterForm.jsp, przy użyciu którego uzupełniamy obiekt uzyszkodnik który jest widoczny w całym przepływie. Po zatwierdzeniu formularza następuje stan akcji „zapiszUzytkownika” który zapisuje obiekt „uzyszkodnik” do bazy i uzupełnia w nim pole id identyfikatorem pod jakim wylądował w bazie. Sterowanie jest przekazywane do stanu „wydrukujUzytkownika” który na konsoli drukuje zawartość obiektu i przekazuje sterowanie do stanu końcowego, który przekierowuje do strony startowej.

Troszkę zmian nastąpiło też w samym formularzu do wypełniania danych.

```
2   <%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
3   <%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
4   <%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
5   <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
6   <html>
7
8       <body>
9           <h3>Zapisywanie do newslettera</h3>
10
11          <div>
12              <form:form commandName="uzyszkodnik">
13                  Imię: <form:input path="imie"/><br>
14                  Nazwisko: <form:input path="nazwisko"/><br>
15                  Email: <form:input path="email"/><br>
16
17                  <input type="submit" name="_eventId_zatwierdz" value="Zatwierdź"/>
18                  <input type="submit" name="_eventId_anuluj" value="Anuluj"/>
19
20
21                  <a href="${flowExecutionUrl}&amp;_eventId=zatwierdz">Zatwierdz</a>
22                  <a href="${flowExecutionUrl}&amp;_eventId=anuluj">Anuluj</a>
23
24
25              </form:form>
26          </div>
27      </body>
28
29
30
31  </html>
```

W liniach 2-5 pojawiły się importy do bibliotek tagów których używamy. Musisz je mieć, inaczej formularz nie zadziała. Elementy z linii 21 i 22 już znasz, nie będą nam więcej potrzebne ale zostawiłem je do porównania z elementami z linii 17 i 18 które je zastąpią. Zacznijmy jednak od początku – linia 12. <form:form.... z zasady oznacza początek formularza, </form:form> jego koniec. Wszystkie pola służące do wprowadzania danych muszą się znajdować pomiędzy nimi.

W tagu otwierającym formularz pojawia się parametr commandName=”uzyszkodnik”. Pewne podejrzenie odnośnie tego co to takiego, powinno już Ci się nasunąć samo. To jest wskazanie obiektu który będziemy uzupełniać, a ściślej obiektu klasy Uzytkownik który zdeklarowaliśmy w pliku konfiguracyjnym przepływu. Możemy tam przecież mieć kilka deklaracji <var..../>

<form:input path=”nazwaPola”/> odnosi się do pól tego obiektu. Krótko mówiąc są to pola tekstowe do uzupełnienia pól imię, nazwisko, email w obiekcie uzyszkodnik.

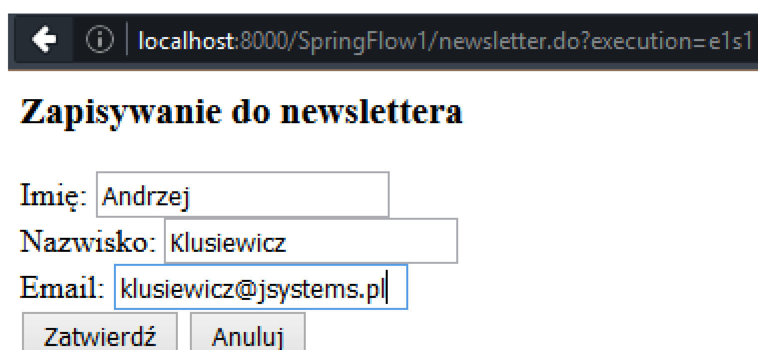
Linie 17 i 18 to przyciski. Jeden będzie zatwierdzał formularz, drugi anulował. Ściślej – oba będą wywoływały akcję którą zdeklarowaliśmy w pliku konfiguracji przepływu. Zwróć uwagę na ich

nazwy. Aby spełniały założoną rolę, muszą składać się z elementów _eventId_ oraz naszego uruchamiacza przejścia do następnego stanu.

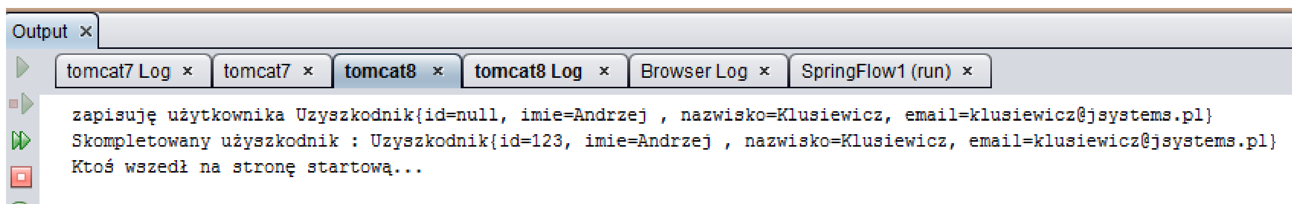
Jeśli przyjrzysz się teraz elementom „transition on …. to …” to myślę że nie trzeba będzie nic więcej wyjaśniać. Linki zrobione wcześniej (linia 21 i 22) można już usunąć.

Skoro już wszystkie elementy mamy, zobaczmy jak to działa. Rozpoczynamy przepływ wchodząc na stronę newsletter.do:

Jesteśmy na pierwszym stanie. Jest to stan widoku „newsletter”. Uzupełniam wszystkie pola:



Po zatwierdzeniu zostaje wywołane zdarzenie „zatwierdź” które powoduje przejście do stanu akcji „zapiszUzytkownika”. Zajrzyjmy teraz do konsoli:



Pierwsza linia pochodzi z dao wywołanego w stanie akcji „zapiszUzytkownika”. Jak widać wyświetlają się dane wprowadzone przez formularz, ale pole id pozostaje puste. Jest ono uzupełniane w stanie akcji „zapiszUzytkownika” . Następnie do akcji wkracza „wydrukujUzytkownika” który prezentuje nam na konsoli już uzupełnione razem z ID dane. Ostatnia linia pochodzi z naszego kontrollera od strony głównej gdzie przepływ został przekierowany przez stan końcowy.

Zaawansowane elementy Spring WebFlow

Kod źródłowy z przykładami do tego rozdziału możesz pobrać pod adresem:

http://jsystems.pl/storage/spring/springflow2.zip

W poprzedniej części stworzyliśmy banalnie prosty przepływ z przejściami. W niniejszym będziemy rozwijać kod z poprzedniej części wzbogacając go o kilka elementów które pomogą nam ulepszyć program. Co chcemy osiągnąć:

- Dodać element który będzie sprawdzał czy użytkownik o takim emailu już jest zapisany do newslettera. Jeśli okaże się że owszem, to przekierowanie widoku informującego o tym.
- Dodać stronę podsumowującą z wyświetleniem wszystkich danych i podziękowaniem za zapisanie się do newslettera.
- Dodanie drugiego przepływu który będzie pozwalał rejestrować konta w sklepie, ale tym razem pojawi się tutaj więcej widoków.
- Dorobić jakiś jeden globalny stan końcowy oznaczający poprawne zakończenie przepływu bez potrzeby definiowania przejść do niego w każdym po kolei stanie na wypadek naciśnięcia przycisku „anuluj” na którymś stanie.
- Dodanie podprzepływu który będzie uruchamiany kiedy użytkownik wprowadzi kod promocyjny.

Stany decyzyjne

Zaczynamy od zadania: „**Dodać element który będzie sprawdzał czy użytkownik o takim emailu już jest zapisany do newslettera. Jeśli okaże się że owszem, to przekierowanie widoku informującego o tym.”**

Do naszego wcześniej stworzonego DAO dodaję nową metodę, która z założenia ma sprawdzać czy użytkownik o podanym adresie email już istnieje czy jeszcze nie. Nie bawimy się tutaj w kurs baz danych, zrobiłem po prostu sprawdzanie czy podano mój email czy jakiś inny. Jeśli ktoś poda adres email klusiewicz@Jsystems.pl, metoda zwróci „true” informując o istnieniu takiego użytkownika.

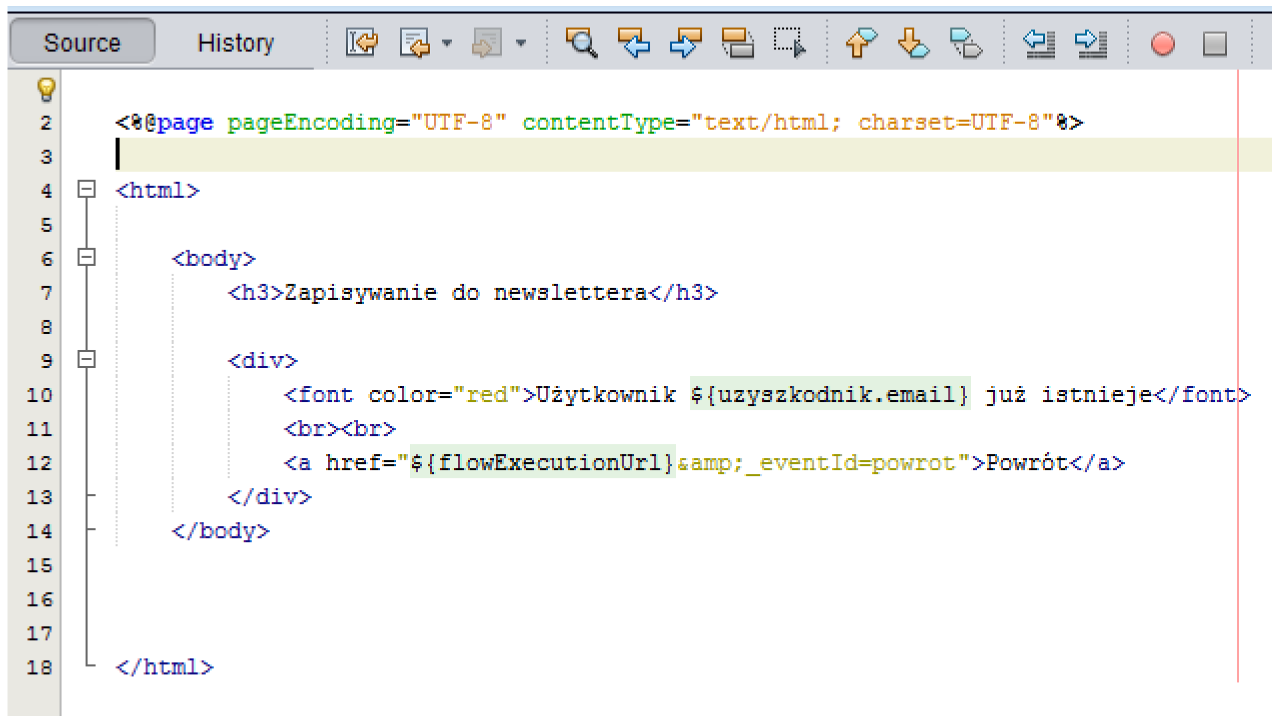
```
10  /**
11   *
12   * @author andrzej
13   */
14  public class UzyszkodnikDao {
15
16      public boolean userExists(String email) {
17          System.out.println("Sprawdzam czy użytkownik "+email+" już istnieje");
            if (email.equalsIgnoreCase("klusiewicz@jsystems.pl")) {
19              return true;
20          } else {
21              return false;
22          }
23      }
```

Przejdźmy teraz do pliku konfiguracji przepływu. Przyjrzyjmy się linii 13. Wcześniej w przypadku zatwierdzenia formularza było tutaj przekierowanie do stanu „zapiszUzytkownika”, teraz jest przekierowanie do nowego stanu – „czyUserIstnienie”. Jest to stan decyzyjny. W zależności od sprawdzanych warunków, przepływ może pójść w całkiem różnych kierunkach. Nasz stan decyzyjny (linie 17-22) sprawdza czy użytkownik o podanym adresie email już istnieje czy jeszcze nie. Robi to z użyciem metody userExists naszego DAO (dao to zostało zdeklarowane jako bean w poprzednim rozdziale). Do metody userExists przekazujemy zawartość pola email z uzupełnianego w trakcie tego przepływu obiektu uzyszkodnik. W zależności od tego czy metoda zwróci true czy false, zostaniemy przekierowani albo do stanu „userIstnieje”, albo do stanu „zapiszUzytkownika”. W przypadku gdyby okazało się że takiego użytkownika jeszcze nie było, to sprawa jest o tyle prosta że wystarczy przejść do zapisu obiektu. Jeśli jednak okazałoby się że mamy już użytkownika o takim adresie email, należy wyświetlić jakąś stronę informacyjną. Przyjrzymy się więc teraz liniom 24-26. Mamy tutaj kolejny stan widoku.

```
sklep-servlet.xml  newsletter-przeplyw.xml  newsletterForm.jsp  Uzyszkodnik.java  Uzyszkod
Source  History
7
8    http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">
9
10   <var name="uzyszkodnik" class="pl.jsystems.springflow.model.Uzyszkodnik"/>
11
12   <view-state id="newsletter" view="newsletterForm" model="uzyszkodnik">
13       <transition on="zatwierdz" to="czyUserIstnieje"/>
14       <transition on="anuluj" to="anulowano"/>
15   </view-state>
16
17   <decision-state id="czyUserIstnieje">
18       <if test="uzytkownikDao.userExists(uzyszkodnik.email)"
19       then="userIstnieje"
20       else="zapiszUzytkownika"
21       />
22   </decision-state>
23
24   <view-state id="userIstnieje" view="userIstniejeForm" model="uzyszkodnik">
25        <transition on="powrot" to="newsletter"/>
26   </view-state>
27
28   <action-state id="zapiszUzytkownika">
29       <evaluate expression="uzytkownikDao.save(uzyszkodnik)" result="uzyszkodnik.id"   />
30       <transition to="wydrukujUzytkownika" />
31   </action-state>
```

Na ten moment jako widok podałem „userIstniejeForm”. Wcześniej go nie tworzyliśmy, tymczasowo dodałem sobie taki wpis by cokolwiek tutaj było. Zaraz stworzymy warstwę widoku. Zauważ że tutaj też przekazuję do modelu obiekt uzyszkodnik (ten który uzupełniamy podczas przeplywu). Po co? Ponieważ będę chciał sięgnąć do jego zawartości w tym widoku – konkretnie do adresu email. W linii 25 pojawia nam się nowe zdarzenie – powrot. Zostanie ono wywołane znaną już konstrukcją kiedy ktoś kliknie na nowej podstronie informacyjnej link „powrót”. W takim przypadku zostanie przekierowany do stanu „newsletter” czyli naszego formularza.

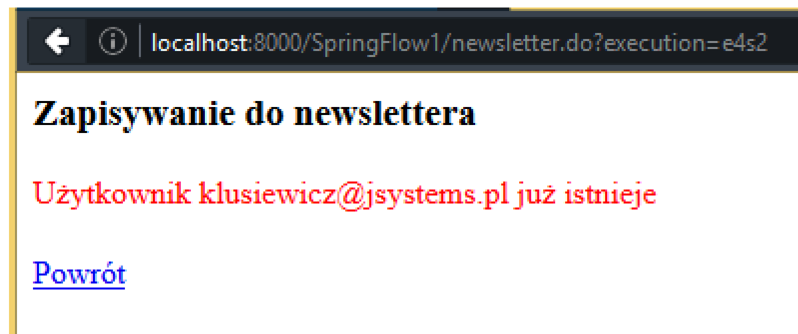
Przejdźmy teraz do warstwy widoku. Stworzyłem wspomniany wcześniej plik „userIstniejeForm.jsp” i umieściłem w nim taką oto treść:



W zasadzie interesujące są tylko linie 10 i 12, bo tylko tam coś się dzieje ;)

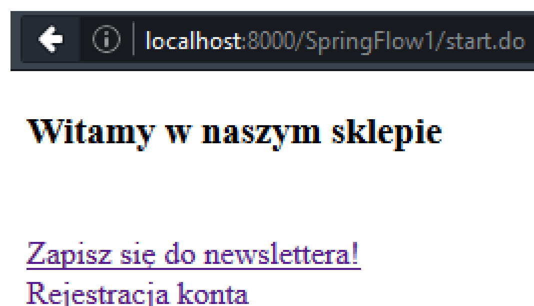
W linii 10 wyświetlam email uzupelnianego podczas przepływu użytkownika, czyli bezpośrednio ten email który wstukaliśmy w formularzu (wyjaśnia się podawanie parametru model w nowym view-state). Linia 12 to znana już konstrukcja linka wywołującego akcję. Tym razem jest to akcja o nazwie „powrot” która wg zapisów w naszym pliku konfiguracji przepływu spowoduje powrót do formularza. Sprawdźmy więc działanie. Uzupełniam formularz i zatwierdzam.

Wywoływana w naszym stanie decyzyjnym metoda weryfikuje że użytkownik o podanym emailu już istnieje i powoduje przejście do stanu widoku "userIstnieje”. Przy okazji zwróć uwagę na pasek adresu. Cały czas widnieje tam adres „newsletter.do”!



Kliknięcie linka „Powrót” powoduje wywolanie akcji „powrot” i co za tym idzie powrót do formularza wypelniania danych:

Teraz działa to tak, że po wypełnieniu formularza wracamy do strony startowej:



Wyświetlanie danych

Chcemy dodać jakąś stronę podsumowującą z podziękowaniem za rejestrację w newsletterze. Zacznijmy więc od dodania kolejnego stanu – stanu widoku i zadbania o właściwe przekierowanie. Przechodzę do edycji pliku newsletter-przeplyw.xml i wprowadzam nieco modyfikacji:

```
32
33  <action-state id="wydrukujUzytkownika">
34      <evaluate expression="uzytkownikDao.print(uzyszkodnik)" />
35      <transition to="podziekowanie" />
36  </action-state>
37
38
39  <view-state id="podziekowanie" view="podziekowanieView" model="uzyszkodnik">
40      <transition on="zakoncz" to="dodano" />
41  </view-state>
42
```

Stan akcji „wydrukujUzytkownika” nie przekazuje już przepływu do stanu dodano który po prostu przekierowywał do strony startowej, tylko do nowego stanu „podziekowanie” który jest stanem widoku mającym wyświetlać podziękowanie. Dopiero ten nowy stan przekaże przepływ do strony startowej kiedy nastąpi zdarzenie „zakoncz”. W linii 39 widzimy, że za wyświetlenie widoku odpowiedzialny jest plik podziekowanieView.jsp. Toteż go tworzymy i wprowadzam poniższy kod:

```
4       Author      : andrzej
5   --%>
6   <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
7   <%@page contentType="text/html" pageEncoding="UTF-8"%>
8   <!DOCTYPE html>
9   <html>
10      <head>
11          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12          <title>JSP Page</title>
13      </head>
14      <body>
15
16          <h3>Dziękujemy ${uzyszkodnik.imie} ${uzyszkodnik.nazwisko} za rejestrację w naszym newsletterze.</h3>
17          <h4>Będziesz otrzymywal na podany adres email (${uzyszkodnik.email}) nowości z naszej oferty!</h4>
18
19          <a href="${flowExecutionUrl}&amp;_eventId=zakoncz">Powrót do strony startowej</a>
20      </body>
21  </html>
22
```

Nie ma tutaj nic nadzwyczajnego, to jest zwykła strona podziękowania. Warte chwili jest za to przyjrzenie się liniom 16 i 17, oraz linii 19. W liniach 16 i 17 widzimy odwołanie do obiektu uzyszkodnik – a właściwie do jego pól. Zauważ, że jest to obiekt który uzupełniamy w ramach tego przepływu. Aby mieć dostęp do danych w nim zawartych, musimy dodać parametr „model”, który widzimy w linii 39 pliku newsletter-przeplyw.xml. Bez tej deklaracji nie będziemy mieli dostępu do tych danych! W linii 19 jest wywołanie przez link akcji „zakoncz” która powoduje przekazanie przepływu do stanu „dodano” (co widać w linii 40 pliku newsletter-przeplyw.xml).

Istnieje taka możliwość, że ktoś w formularzu wprowadzi polskie litery. Choć nie jest to akurat element Springa, warto dodać poniższy wpis do swojego pliku web.xml, aby wprowadzone polskie znaki nie zmieniły się w krzaczki.

```
5
6      <filter>
7      <filter-name>encoding-filter</filter-name>
8      <filter-class>
9          org.springframework.web.filter.CharacterEncodingFilter
10     </filter-class>
11     <init-param>
12         <param-name>encoding</param-name>
13         <param-value>UTF-8</param-value>
14     </init-param>
15     <init-param>
16         <param-name>forceEncoding</param-name>
17         <param-value>true</param-value>
18     </init-param>
19   </filter>
20
21   <filter-mapping>
22       <filter-name>encoding-filter</filter-name>
23       <url-pattern>/*</url-pattern>
24   </filter-mapping>
25
```

Po wszystkim uzupełniam formularz i sprawdzam działanie całości:

Dziękujemy Grześ Brzęczyszczykiewicz za rejestrację w naszym newsletterze.

Będziesz otrzymywal na podany adres email (grzes@lubiespringa.pl) nowości z naszej oferty!

Powrót do strony startowej

Wieloetapowe uzupełnianie obiektu podczas przepływu

Czas dodać drugi przepływ, tym razem umożliwiający założenie konta w naszym sklepie internetowym. Trzeba będzie podać imię i nazwisko, dane kontaktowe, adres, dane do faktur etc. Mało przejrzyste i wygodne to będzie jeśli zrobimy to na jednej stronie. Lepiej rozbić to na kilka etapów. Osobna podstrona do imienia i nazwiska, osobna do danych kontaktowych, do adresu i osobna do danych do faktur. Przepływ będzie jeden, ale podczas jego trwania będziemy uzupełniali jeden obiekt.
Zaczynamy od edycji naszego pliku ****-servlet.xml, aby zarejestrować nowy przepływ. Nowe wpisy zaznaczylem na czerwono. Dodałem adres strony początkowej przepływu – konto.do, oraz nazwę i położenie pliku konfiguracyjnego kolejnego przepływu – konto-przeplyw.xml.

```
37
38  <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
39      <property name="mappings">
40          <value>
41              /newsletter.do=flowController
42              /konto.do=flowController
43          </value>
44
45      </property>
46  </bean>
47
48
49  <bean id="flowController" class="org.springframework.webflow.mvc.servlet.FlowController">
50      <property name="flowExecutor" ref="flowExecutor"/>
51  </bean>
52
53  <!--Tworzy przeplywy na podstawie flowRegistry -->
54  <flow:flow-executor id="flowExecutor" flow-registry="flowRegistry"/>
55  <!--Rejestr plików z opisami poszczególnych przeplywów-->
56  <flow:flow-registry id="flowRegistry" flow-builder-services="flowBuilderServices">
57      <flow:flow-location path="/WEB-INF/flows/newsletter-przeplyw.xml" id="newsletter"/>
58      <flow:flow-location path="/WEB-INF/flows/konto-przeplyw.xml" id="konto"/>
59  </flow:flow-registry>
```

Przyjrzyjmy się naszemu plikowi konfiguracji przepływu.

Przepływ jest stosunkowo prosty, ponieważ ogranicza się do czterech następujących po sobie stanów widoku i na końcu jednego stanu akcji zapisującego nowe konto do bazy.

```
 9
10  <var name="konto" class="pl.jsystems.springflow.model.Konto"/>
11
12  <view-state id="step1" view="daneKontaktowe" model="konto">
13      <transition on="dalej" to="step2"/>
14      <transition on="wstecz" to="start"/>
15      <transition on="anuluj" to="start"/>
16  </view-state>
17
18  <view-state id="step2" view="adresDostawy" model="konto">
19      <transition on="dalej" to="step3"/>
20      <transition on="wstecz" to="step1"/>
21      <transition on="anuluj" to="start"/>
22  </view-state>
23
24  <view-state id="step3" view="daneDoFaktury" model="konto">
25      <transition on="dalej" to="podsumowanie"/>
26      <transition on="wstecz" to="step2"/>
27      <transition on="anuluj" to="start"/>
28  </view-state>
29
30  <view-state id="podsumowanie" view="podsumowanie" model="konto">
31      <transition on="dalej" to="zapisz"/>
32      <transition on="wstecz" to="step3"/>
33      <transition on="anuluj" to="start"/>
34  </view-state>
35
36  <action-state id="zapisz">
37      <evaluate expression="kontoDao.zapisz(konto)" />
38      <transition to="start" />
39  </action-state>
40
41  <end-state id="start"
42      view="externalRedirect:contextRelative:/start.do"/>
```

Podczas tego przepływu będziemy uzupełniać obiekt klasy Konto (co widać w deklaracji w linii 10). Za chwilę do niego wrócimy, teraz przyjrzyjmy się kolejnym przejściom tego przepływu. Na każdym z widoków będą trzy przyciski – dalej, wstecz, anuluj. Pierwszy powoduje przejście do następnego kroku, drugi do poprzedniego a trzeci anuluje cały przepływ i powraca do strony startowej. Wszystkie stany widoku mają zadeklarowany parametr model – będziemy uzupełniać obiekt, więc stany te muszą mieć do niego dostęp. Zauważ że w każdym kolejnym stanie mamy deklarację taką :

<transition on="anuluj" to="start"/>

Deklarowanie przejścia anulującego w każdym kroku jest co najmniej męczące. Co gdybyśmy np. chcieli zmienić stan do którego przechodzi przepływ po kliknięciu „anuluj”? W dalszych krokach zadeklarujemy przejście globalne i nie będziemy musieli deklarować tego przejścia w każdym stanie.

Przepływ składa się z trzech formularzy i jednego widoku podsumowującego. W pierwszym formularzu uzupełniamy dane kontaktowe i informacje o koncie, w drugim adres dostawy, w trzecim dane do faktury. W widoku podsumowującym widzimy wszystkie dane które uzupełnialiśmy i możemy ewentualnie wrócić i coś poprawić.

Przez wszystkie etapy uzupełniamy obiekt klasy konto, więc przyjrzyjmy się tejże klasie.

```
9
10  /**
11   *
12   * @author andrzej
13   */
    public class Konto implements Serializable{
15
16      private String login;
17      private String haslo;
18      private String imie;
19      private String nazwisko;
20      private String email;
21      private String telefon;
22      private Adres adresDostawy=new Adres();
23      private DaneDoFaktury daneDoFaktury=new DaneDoFaktury();
24
```

Oczywiście w klasie tej są również gettery i settery do tych pól. Widzimy tutaj obiekty klas Adres i DaneDoFaktury, więc zobaczmy również ich zawartość. Klasa Adres:

```
10  /**
11   *
12   * @author andrzej
13   */
    public class Adres  implements Serializable{
15      private Long Id;
16      private String miasto;
17      private String kodPocztowy;
18      private String ulica;
19      private String numerBudynku;
20      private String numerLokalu;
```

Klasa DaneDoFaktury:

```
9
10  /**
11   *
12   * @author andrzej
13   */
14  public class DaneDoFaktury  implements Serializable{
15
16      private Long id;
17      private String nazwaFirmy;
18      private String NIP;
19      private Adres adresFirmy=new Adres();
20
```

Zauważ że przy polach będących obiektami mamy użycie domyslnego konstruktora. Przy typach prostych tego nie robię. Musi tak być, ponieważ Spring tworzy nowy obiekt klasy którą zadeklarowaliśmy jako zmienną przepływu, ale już nie robi tego dla „podobiektów”. Jeśli sami nie zaincjalizujemy ich, dostaniemy błąd podczas próby dostępu do ich pól. Przy okazji zobaczmy też deklarację metody zapisz którą wywołuję w stanie „zapisz” naszego przepływu. Jak widać jest to zaślepka – nie zajmujemy się tutaj integracją Springa z bazami danych.

```
10  /**
11   *
12   * @author andrzej
13   */
14  public class KontoDao {
15      public void zapisz(Konto konto){
16          System.out.println("zapisuję do bazy nowe konto : "+konto.toString());
17      }
18  }
```

W pliku konfiguracji przepływu do tego DAO odwołuję się tak:

```
35
36  <action-state id="zapisz">
37      <evaluate expression="kontoDao.zapisz(konto)" />
38      <transition to="start" />
39  </action-state>
40
```

a to oznacza że musimy mieć jeszcze zadeklarowany bean do tej klasy w pliku *****-servlet.xml:

```
<bean id="uzytkownikDao" class="pl.jsystems.springflow.dao.UzyszkodnikDao"/>
<bean id="kontoDao" class="pl.jsystems.springflow.dao.KontoDao"/>
```

Zobaczmy teraz warstwę widoku, czyli nasze formularze i stronę podsumowującą. Formularz numer 1 (czyli stan step1) służy do uzupełniania danych o koncie i informacji kontaktowych. Tradycyjne pola do wprowadzania danych służące uzupełnianiu bezpośrednich pól obiektu klasy Konto i guziki o których wspominałem wcześniej:

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

    </head>
    <body>
        <h3>Rejestracja konta - krok 1 - dane kontaktowe i informacje o koncie</h3>

        <div>
            <form:form commandName="konto">
                <table>
                    <tr><td>Login: </td><td><form:input path="login"/></td></tr>
                    <tr><td>Haslo:</td><td><form:input path="haslo"/></td></tr>
                    <tr><td>Imię: </td><td><form:input path="imie"/></td></tr>
                    <tr><td>Nazwisko: </td><td><form:input path="nazwisko"/></td></tr>
                    <tr><td>Email: </td><td><form:input path="email"/></td></tr>
                    <tr><td>Telefon: </td><td><form:input path="telefon"/></td></tr>
                </table>
                <input type="submit" name="_eventId_dalej" value="Dalej"/>
                <input type="submit" name="_eventId_wstecz" value="Wstecz"/>
                <input type="submit" name="_eventId_anuluj" value="Anuluj"/>

            </form:form>
        </div>
    </body>



</html>
```

Kolejny krok i drugi formularz to uzupełnianie danych do dostawy zakupów:

```
11   <body>
12       <h3>Rejestracja konta - krok 2 - adres dostaw</h3>
13
14       <div>
15            <form:form commandName="konto">
16               <table>
17                   <tr><td>Miasto: </td><td><form:input path="adresDostawy.miasto"/></td></tr>
18                   <tr><td>Kod pocztowy: </td><td><form:input path="adresDostawy.kodPocztowy"/></td></tr>
19                   <tr><td>Ulica: </td><td><form:input path="adresDostawy.ulica"/></td></tr>
20                   <tr><td>Numer budynku: </td><td><form:input path="adresDostawy.numerBudynku"/></td></tr>
21                   <tr><td>Numer lokalu: </td><td><form:input path="adresDostawy.numerLokalu"/></td></tr>
22               </table>
23               <input type="submit" name="_eventId_dalej" value="Dalej"/>
24               <input type="submit" name="_eventId_wstecz" value="Wstecz"/>
25               <input type="submit" name="_eventId_anuluj" value="Anuluj"/>
26
27           </form:form>
28       </div>
29   </body>
```

Przyjrzyj się dobrze parametrom „path”... W taki właśnie sposób odwołujemy się do podobiektów i ich pól. Krok trzeci to uzupełnianie danych do faktury. Widzimy tutaj odwołania do jeszcze bardziej zagnieżdżonych podobiektów.

```
  <h3>Rejestracja konta - krok 3 - dane do faktury</h3>

<div>
     <form:form commandName="konto">
        <table>

              <tr><td>Nazwa firmy: </td><td><form:input path="daneDoFaktury.nazwaFirmy"/></td></tr>
            <tr><td>NIP: </td><td><form:input path="daneDoFaktury.NIP"/></td></tr>

            <tr><td>Miasto: </td><td><form:input path="daneDoFaktury.adresFirmy.miasto"/></td></tr>
            <tr><td>Kod pocztowy: </td><td><form:input path="daneDoFaktury.adresFirmy.kodPocztowy"/></td></tr>
            <tr><td>Ulica: </td><td><form:input path="daneDoFaktury.adresFirmy.ulica"/></td></tr>
            <tr><td>Numer budynku: </td><td><form:input path="daneDoFaktury.adresFirmy.numerBudynku"/></td></tr>
            <tr><td>Numer lokalu: </td><td><form:input path="daneDoFaktury.adresFirmy.numerLokalu"/></td></tr>
        </table>
        <input type="submit" name="_eventId_dalej" value="Dalej"/>
        <input type="submit" name="_eventId_wstecz" value="Wstecz"/>
        <input type="submit" name="_eventId_anuluj" value="Anuluj"/>

    </form:form>
```

No i na koniec nasza strona podsumowująca:

```
11  <body>
12      <h3>Podsumowanie rejestracji</h3>
13
14      <div>
15          <h4>Dane kontaktowe i informacje o koncie</h4>
16          <table>
17              <tr><td>Login: </td><td>${konto.login}</td></tr>
18              <tr><td>Haslo:</td><td>${konto.haslo}</td></tr>
19              <tr><td>Imię: </td><td>${konto.imie}</td></tr>
20              <tr><td>Nazwisko: </td><td>${konto.nazwisko}</td></tr>
21              <tr><td>Email: </td><td>${konto.email}</td></tr>
22              <tr><td>Telefon: </td><td>${konto.telefon}</td></tr>
23          </table>
24
25          <h4>Adres dostawy</h4>
26
27           <table>
28                  <tr><td>Miasto: </td><td>${konto.adresDostawy.miasto}</td></tr>
29                  <tr><td>Kod pocztowy: </td><td>${konto.adresDostawy.kodPocztowy}</td></tr>
30                  <tr><td>Ulica: </td><td>${konto.adresDostawy.ulica}</td></tr>
31                  <tr><td>Numer budynku: </td><td>${konto.adresDostawy.numerBudynku}</td></tr>
32                  <tr><td>Numer lokalu: </td><td>${konto.adresDostawy.numerLokalu}</td></tr>
33              </table>
34
35          <h4>Dane do faktury</h4>
36
37             <table>
38                  <tr><td>Nazwa firmy: </td><td>${konto.daneDoFaktury.nazwaFirmy}</td></tr>
39                  <tr><td>NIP: </td><td>${konto.daneDoFaktury.NIP}</td></tr>
40                  <tr><td>Miasto: </td><td>${konto.daneDoFaktury.adresFirmy.miasto}</td></tr>
41                  <tr><td>Kod pocztowy: </td><td>${konto.daneDoFaktury.adresFirmy.kodPocztowy}</td></tr>
42                  <tr><td>Ulica: </td><td>${konto.daneDoFaktury.adresFirmy.ulica}</td></tr>
43                  <tr><td>Numer budynku: </td><td>${konto.daneDoFaktury.adresFirmy.numerBudynku}</td></tr>
44                  <tr><td>Numer lokalu: </td><td>${konto.daneDoFaktury.adresFirmy.numerLokalu}</td></tr>
45              </table>
46
47
48          <form:form commandName="konto">
49              <input type="submit" name="_eventId_dalej" value="Dalej"/>
50              <input type="submit" name="_eventId_wstecz" value="Wstecz"/>
51              <input type="submit" name="_eventId_anuluj" value="Anuluj"/>
52          </form:form>
53      </div>
54  </body>
```

Tutaj standardowy sposób wyświetlania danych z obiektu na zasadzie „${obiekt.pole}”. Wyświetlamy dane uzupełniane we wszystkich etapach. Możemy tak zrobić ponieważ cały czas pracujemy na jednym obiekcie klasy Konto zainicjalizowanym na początku przepływu. Odwołujemy się do pól, podobiektów, pól podobiektów dokładnie tak jak przy np. servletach – w końcu robimy to z użyciem tagów JSTL. A teraz przyjrzyj się liniom 48-52. Guziki jak wczesniej, ale przecież wiemy że to jest strona wyświetlająca podsumowanie, nie ma tu formularza. Co więc robią tutaj tagi „<form:form>” ? Otóż bez nich nie działałyby nasze przyciski, ponieważ zatwierdzają one formularze! Inaczej musielibyśmy je zamienić na linki. Zobaczmy teraz jak to wszystko razem działa ;)

Krok 1:

localhost:8000/SpringFlow1/konto.do?execution=e5s1

Rejestracja konta - krok 1 - dane kontaktowe i informacje o koncie

Login:	mapet
Haslo:	mapettubyl
Imię:	Andrzej
Nazwisko:	Klusiewicz
Email:	klusiewicz@jsystems.pl
Telefon:	112

Dalej Wstecz Anuluj

Krok 2:

http://localh...ecution=e5s2

localhost:8000/SpringFlow1/konto.do?execution=e5s2

Rejestracja konta - krok 2 - adres dostaw

Miasto:	Warszawa
Kod pocztowy:	00-123
Ulica:	Programistyczna
Numer budynku:	6
Numer lokalu:	9

Dalej Wstecz Anuluj

Krok 3:

http://localh...ecution=e5s3

localhost:8000/SpringFlow1/konto.do?execution=e5s3

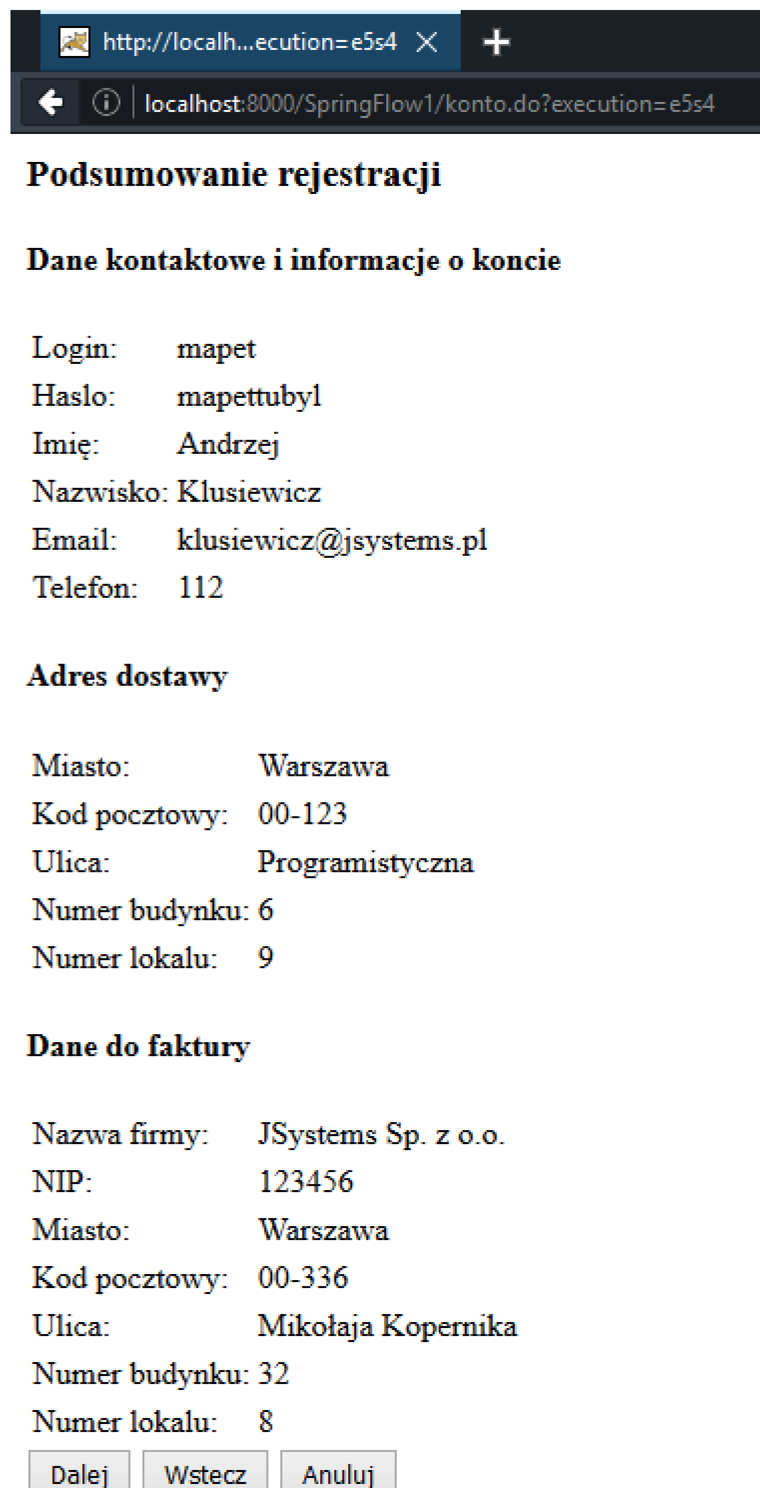
Rejestracja konta - krok 3 - dane do faktury

Nazwa firmy:	JSystems Sp. z o.o.
NIP:	123456
Miasto:	Warszawa
Kod pocztowy:	00-336
Ulica:	Mikołaja Kopernika
Numer budynku:	32
Numer lokalu:	8

Dalej Wstecz Anuluj

Taka mała ciekawostka przy okazji. Przyjrzyj się paskowi adresu, a właściwie to wartości w parametrze execution w poszczególnych krokach przepływu.

Czas na stronę podsumowującą:



Przejście globalne

Pojawił nam się wcześniej problem z ciągłym deklarowaniem przejścia do strony startowej w przypadku naciśnięcia anuluj na którymkolwiek z kroków przepływu:

```
12  <view-state id="step1" view="daneKontaktowe" model="konto">
13      <transition on="dalej" to="step2"/>
14      <transition on="wstecz" to="start"/>
15      <transition on="anuluj" to="start"/>
16  </view-state>
17
18  <view-state id="step2" view="adresDostawy" model="konto">
19      <transition on="dalej" to="step3"/>
20      <transition on="wstecz" to="step1"/>
21      <transition on="anuluj" to="start"/>
22  </view-state>
23
24  <view-state id="step3" view="daneDoFaktury" model="konto">
25      <transition on="dalej" to="podsumowanie"/>
26      <transition on="wstecz" to="step2"/>
27      <transition on="anuluj" to="start"/>
28  </view-state>
29
30  <view-state id="podsumowanie" view="podsumowanie" model="konto">
31      <transition on="dalej" to="zapisz"/>
32      <transition on="wstecz" to="step3"/>
33      <transition on="anuluj" to="start"/>
34  </view-state>
35
36  <action-state id="zapisz">
37      <evaluate expression="kontoDao.zapisz(konto)" />
38      <transition to="start" />
39  </action-state>
40
41  <end-state id="start"
42      view="externalRedirect:contextRelative:/start.do"/>
43
```

Można i tak, ale wygodniej będzie zdeklarować tak zwane przejście globalne. Ustalasz w jednym miejscu że np. skądkowiek wywołana akcja „powrot” przekierowuje do jakiegos poczatkowego stanu widoku i nie musisz tego deklarować w każdym stanie. W podobny sposób możesz np. zrobić przekierowanie na stronę główną czy do formularza logowania.

Dodajmy więc globalne przejście do naszego pliku przepływu:

```
10    <var name="konto" class="pl.jsystems.springflow.model.Konto"/>
11
12    <view-state id="step1" view="daneKontaktowe" model="konto">
13        <transition on="dalej" to="step2"/>
14        <transition on="wstecz" to="start"/>
15    </view-state>
16
17    <view-state id="step2" view="adresDostawy" model="konto">
18        <transition on="dalej" to="step3"/>
19        <transition on="wstecz" to="step1"/>
20    </view-state>
21
22    <view-state id="step3" view="daneDoFaktury" model="konto">
23        <transition on="dalej" to="podsumowanie"/>
24        <transition on="wstecz" to="step2"/>
25    </view-state>
26
27    <view-state id="podsumowanie" view="podsumowanie" model="konto">
28        <transition on="dalej" to="zapisz"/>
29        <transition on="wstecz" to="step3"/>
30    </view-state>
31
32    <action-state id="zapisz">
33        <evaluate expression="kontoDao.zapisz(konto)" />
34        <transition to="start" />
35    </action-state>
36
37
38    <end-state id="start"
39        view="externalRedirect:contextRelative:/start.do"/>
40
41
42    <global-transitions>
43        <transition on="anuluj" to="start"/>
44    </global-transitions>
45
46    </flow>
```

Pozbyłem się przejść kierujących do stanu start z poszczególnych stanów, a w ich miejsce wstawiłem element z linii 42-44. Przy okazji mała uwaga – z jakiegoś powodu sekcja globa-transitions musi być na końcu pliku. Ilekroć znajdowała się gdzieś indziej, dostawałem dziwne błędy i zwykle dotyczyły one sekcji następującej po niej.

Podprzepływy

Przypuśćmy teraz, że w zależności od jakichś warunków chcemy przeprowadzić jakąś dodatkową sekwencję kroków. Przykładowo użytkownik dysponuje kodem promocyjnym. Jeśli nim dysponuje, uruchomimy dodatkowy przepływ który sprawdzi poprawność kodu i powróci do pierwotnego przepływu. Chcemy by pytanie o posiadanie kodu promocyjnego pojawiało się od razu na początku. Będą do wyboru dwa przyciski - „Posiadam” i „Nie posiadam”. Jeśli użytkownik wybierze „Posiadam” to główny przepływ zostanie na chwilę wstrzymany, przejdzie przez dodatkowy podprzepływ i wróci do przepływu głównego. Jeśli ktoś wybierze „Nie posiadam”, to po prostu główny przepływ będzie kontynuowany.

Zaczynamy od poinformowania Springa o nowym przepływie w pliku *****-servlet.xml:

```
37
38    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
39        <property name="mappings">
40            <value>
41                /newsletter.do=flowController
42                /konto.do=flowController
43                /kodPromocyjny.do=flowController
44            </value>
45
46        </property>
47    </bean>
48
49
50    <bean id="flowController" class="org.springframework.webflow.mvc.servlet.FlowController">
51        <property name="flowExecutor" ref="flowExecutor"/>
52    </bean>
53
54    <!--Tworzy przeplywy na podstawie flowRegistry -->
55    <flow:flow-executor id="flowExecutor" flow-registry="flowRegistry"/>
56    <!--Rejestr plików z opisami poszczególnych przeplywów-->
57    <flow:flow-registry id="flowRegistry" flow-builder-services="flowBuilderServices">
58        <flow:flow-location path="/WEB-INF/flows/newsletter-przeplyw.xml" id="newsletter"/>
59        <flow:flow-location path="/WEB-INF/flows/konto-przeplyw.xml" id="konto"/>
60        <flow:flow-location path="/WEB-INF/flows/kodpromocyjny-przeplyw.xml" id="kodpromocyjny"/>
61    </flow:flow-registry>
```

Tworzymy plik konfiguracji podprzepływu który przed momentem obiecaliśmy: kodpromocyjny-przeplyw.xml. Poniżej tylko jego „mięsista zawartość.

```
 9
10    <var name="konto" class="pl.jsystems.springflow.model.Konto"/>
11
12    <view-state id="czymaszkod" view="czyMaszKod" model="konto">
13        <transition on="mam" to="kodpromocyjny"/>
14        <transition on="niemam" to="anulowano"/>
15    </view-state>
16
17
18    <view-state id="kodpromocyjny" view="kodPromocyjny" model="konto">
19        <transition on="zatwierdz" to="sprawdzKod"/>
20        <transition on="anuluj" to="anulowano"/>
21    </view-state>
22
23    <action-state id="sprawdzKod">
24        <evaluate expression="kontoDao.sprawdzKod(konto.kodPromocyjny)" />
25        <transition to="zakonczono" />
26    </action-state>
27
28    <end-state id="zakonczono">
29        <output name="konto"/>
30    </end-state>
31
32    <end-state id="anulowano"/>
```

Przepływ jest dosyć prosty, natomiast w związku z tym że jest to podprzepływ, muszę wyjaśnić kilka rzeczy. Posługuję się tutaj cały czas obiektem klasy Konto. Zostanie on przekazany do przepływu i z niego zwrócony. Sam kod promocyjny, jako że jest zwykłym tekstem umieścimy w nowym polu klasy Konto które też dopisałem:

```
10    /**
11     *
12     * @author andrzej
13     */
      public class Konto implements Serializable{
15
16        private String login;
17        private String haslo;
18        private String imie;
19        private String nazwisko;
20        private String email;
21        private String telefon;
22        private Adres adresDostawy=new Adres();
23        private DaneDoFaktury daneDoFaktury=new DaneDoFaktury();
24
25        private String kodPromocyjny;
26
```

Pierwszy stan to stan widoku który jedyne co robi to pyta czy posiadasz kod promocyjny czy nie. Zawartość pliku czyMaszKod.jsp:

```
11  <body>
12      <h3>Kod promocyjny</h3>
13
14      <div>
15          <form:form commandName="konto">
16              Czy posiadasz kod promocyjny?
17              <br><br>
18              <input type="submit" name="_eventId_mam" value="Posiadam"/>
19              <input type="submit" name="_eventId_niemam" value="Nie posiadam"/>
20
21          </form:form>
```

W zależności od wyboru albo wróci do głównego przepływu (w wyniku wywołania akcji „niemam” jest przejście do stanu „anulowano” który jest stanem końcowym), albo przejdzie do stanu „kodpromocyjny” w którym wprowadzamy owy kod. Zawartość pliku kodPromocyjny.jsp:

```
11  <body>
12      <h3>Kod promocyjny</h3>
13
14      <div>
15          <form:form commandName="konto">
16              Kod promocyjny: <form:input path="kodPromocyjny"/><br>
17
18              <input type="submit" name="_eventId_zatwierdz" value="Zatwierdź"/>
19              <input type="submit" name="_eventId_anuluj" value="Anuluj"/>

21          </form:form>
```

Przy wyborze „anuluj” jest przejście do stanu końcowego anulowano, ale jeśli ktoś wprowadzi kod i zatwierdzi wybierając „zatwierdź”, podprzepływ przejdzie do stanu akcji „sprawdzKod”. Pole do wprowadzania kodu promocyjnego jest podpięte do naszego nowego pola „kodPromocyjny” w klasie „Konto”. Stan sprawdxKod wywołuje metodę „sprawdzKod” z kontoDao która jest tylko zaślepką:

```
14  public class KontoDao {
15      public void zapisz(Konto konto){
16          System.out.println("zapisuję do bazy nowe konto : "+konto.toString());
17      }
18
19      public boolean sprawdzKod(String kod){
20          System.out.println("sprawdzam kod "+kod);
21          return true;
22      }
23  }
24
```

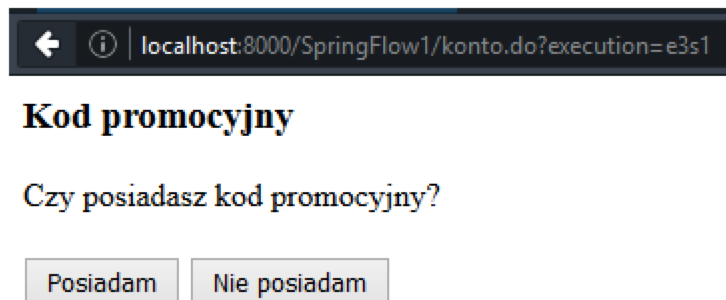
Oczywiście można byłoby tutaj podpiąć faktyczne sprawdzanie i jakiś stan decyzyjny / walidację. Taki stan jaki jest na razie całkiem nam wystarczy, ponieważ zajmujemy się w tej chwili podprzepływami jako takimi. Efekt jest taki, że jaki byś kod nie wprowadził to zawsze będzie ok ;)

Po zakończeniu tego stanu akcji następuje przejście do stanu „zakonczono" który jest stanem końcowym tego podprzepływu.Pojawia się tutaj za to nowa rzecz - <output name="konto"/>. Co to? To jest element który przekazuje do głównego przepływu uzupełniany obiekt. Zaraz się to wyjaśni, jak tylko przejdziemy do wpinania podprzepływu do głównego przepływu. Przejdźmy teraz do pliku konfiguracji głównego przepływu – konto-przeplyw.xml:

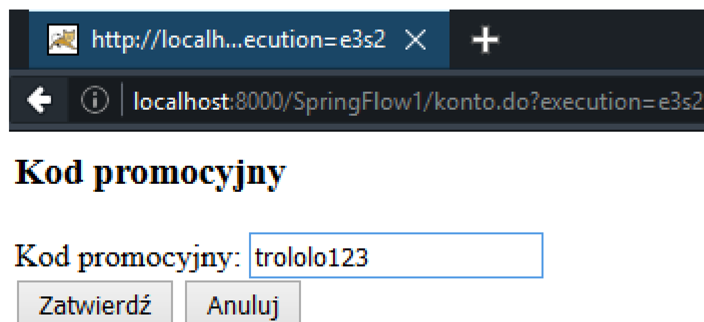
```
10      <var name="konto" class="pl.jsystems.springflow.model.Konto"/>
11
12      <subflow-state id="kodPromocyjny" subflow="kodpromocyjny">
13          <output name="konto" value="konto"/>
14          <transition to="step1"/>
15      </subflow-state>
16
17      <view-state id="step1" view="daneKontaktowe" model="konto">
18          <transition on="dalej" to="step2"/>
19          <transition on="wstecz" to="start"/>
20      </view-state>
21
```

Jako pierwszy stan na liście pojawił się teraz „subflow-state" - czyli stan podprzepływu. Zostanie on uruchomiony jako pierwszy, ponieważ jako pierwszy znajduje sięna liście. Parametr subflow określa nazwę podprzepływu który ma zostać uruchomiony. Parametr output dotyczy tego co nam „wypadnie" z podprzepływu. Będziemy uzupełniać cały czas obiekt klasy „Konto" - ten sam co w głównym przepływie. Transition to z linii 14 mówi o tym gdzie mamy przejść po zakończeniu podprzepływu – a wrócimy do kroku pierwszego – tj. formularza danych kontaktowych.

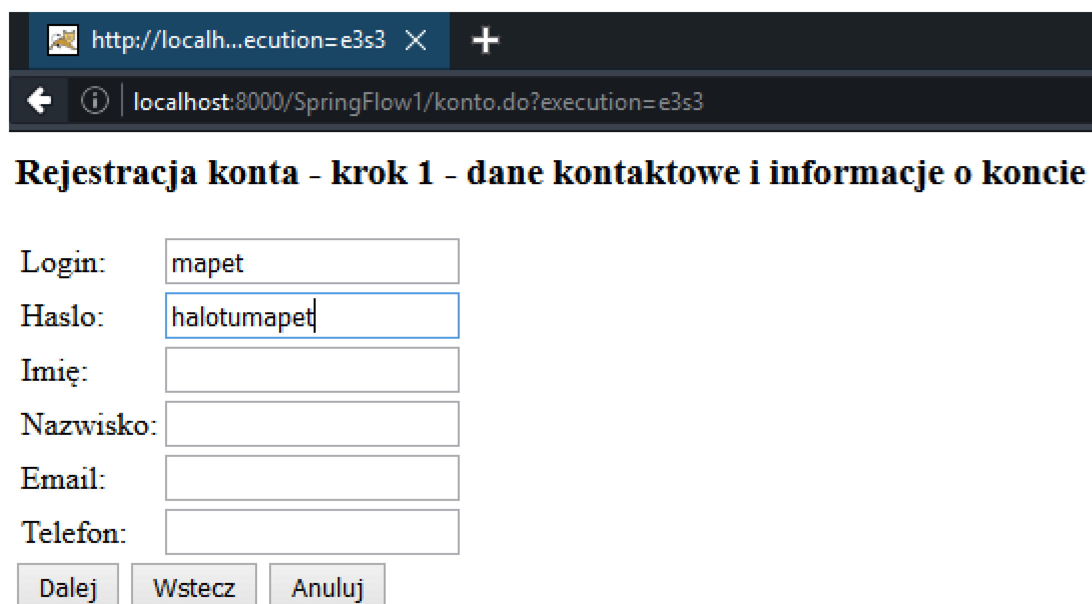
Uruchamiamy całość i sprawdzamy:



Wybrałem że posiadam kod promocyjny, który w następnym kroku wprowadzam:



Zatwierdzam i przepływ wraca do głównego przebiegu:

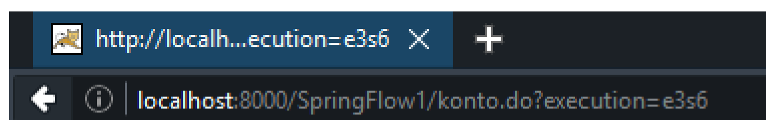


Uzupełniam tylko login i hasło, resztę we wszystkich krokach pozostawiam puste.

W pliku podsumowanie.jsp dodałem też wyświetlanie ewentualnie uzupełnionego kodu promocyjnego:

```
14        <div>
15            <h4>Dane kontaktowe i informacje o koncie</h4>
16            <table>
17                <tr><td>Login: </td><td>${konto.login}</td></tr>
18                <tr><td>Haslo:</td><td>${konto.haslo}</td></tr>
19                <tr><td>Imię: </td><td>${konto.imie}</td></tr>
20                <tr><td>Nazwisko: </td><td>${konto.nazwisko}</td></tr>
21                <tr><td>Email: </td><td>${konto.email}</td></tr>
22                <tr><td>Telefon: </td><td>${konto.telefon}</td></tr>
23                 <tr><td>Kod promocyjny: </td><td>${konto.kodPromocyjny}</td></tr>
            </table>
```

Strona z podsumowaniem po zakończeniu przebiegu:



Podsumowanie rejestracji

Dane kontaktowe i informacje o koncie

Login:	mapet
Haslo:	halotumapet
Imię:	
Nazwisko:	
Email:	
Telefon:	
Kod promocyjny:	trololo123

Spring Security

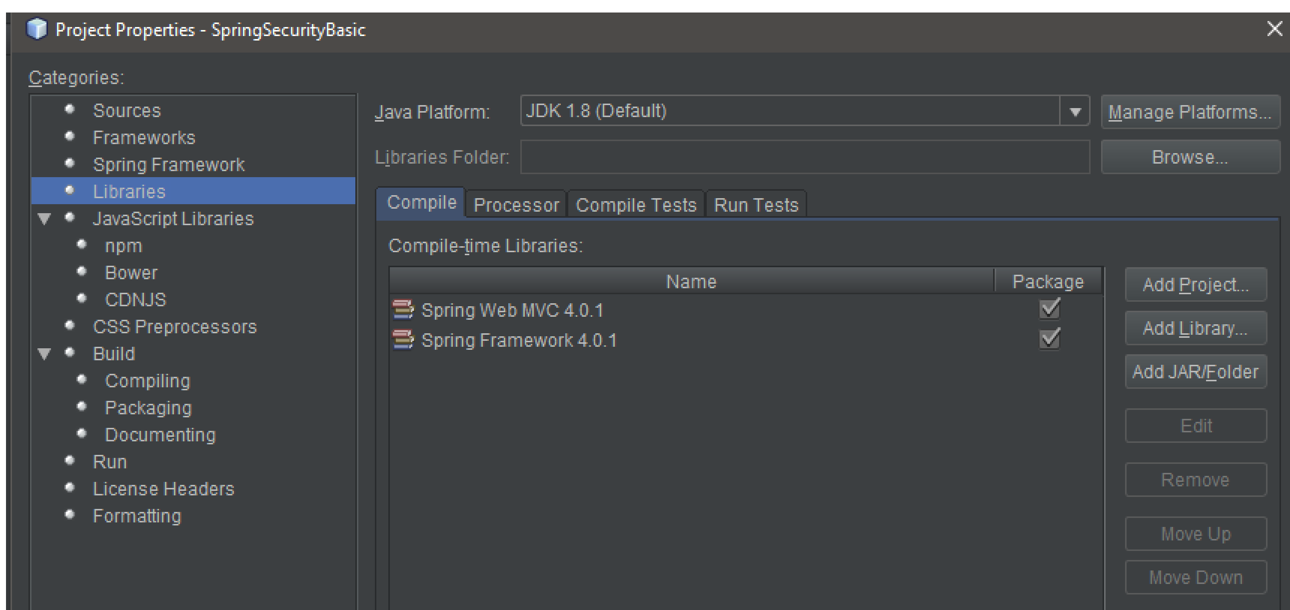
Spring Security to moduł frameworka Spring, służący do zabezpieczania aplikacji przed niepowołanym dostępem. W tym artykule przejdziemy proces tworzenia prostej aplikacji z użyciem Spring MVC, a następnie stworzę w niej 3 podstrony. Jedna z podstron będzie dostępna publicznie bez autoryzacji, druga będzie dostępna dla użytkowników oraz administratorów, a trzecia tylko dla administratorów.

Konfiguracja oparta o XML

Cały proces zaczynamy od stworzenia aplikacji. Ja użyłem akurat Spring MVC, ale możesz zrobić aplikację opartą o jakikolwiek inny framework albo zwyczajne serwlety. Spring Security jest oparte o zwykłe serwletowe filtry, więc może być podpięty do wszystkiego. Może też służyć do zabezpieczania np usług sieciowych zamiast stron html.

Bazowa aplikacja

Używając narzędzia Netbeans IDE stworzyłem aplikację o nazwie SpringSecurityBasic. Ponieważ działać bedzie w oparciu o Spring MVC, zaczynam od dodania niezbędnych bibliotek:



Utworzyłem plik web.xml w katalogu WEB-INF i dodałem do niego konfigurację która sprawi że wszystkie wywołania kończące się na ".do" będą przekierowywane do Springa:

```
<servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Spring będzie szukał pliku konfiguracyjnego nazywającego się **spring-servlet.xml** (ponieważ serwlet nazywa się spring, gdyby się nazywał jelonek to Spring szukałby pliku jelonek-servlet.xml) w katalogu WEB-INF. Taki więc plik tworzę i w jego wnętrzu umieszczam konfigurację jak niżej:

```
21    <mvc:annotation-driven/>
22
23    <context:component-scan base-package="pl.jsystems.ssb"/>
24
25    <mvc:resources mapping="/resources/**"
26                   location="/, classpath:/WEB-INF/resources/"
27                   cache-period="10000" />
28
29    <mvc:resources mapping="/jsp/*"
30                   location="/, classpath:/WEB-INF/jsp/"
31                   cache-period="10000" />
32
33
34
35    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
36        <property name="prefix" value="/WEB-INF/jsp/"/>
37        <property name="suffix" value=".jsp"/>
38    </bean>
```

W linii 23 wskazuję pakiet w którego wnętrzu znajdą się controllery-czyli klasy odpowiedzialne za obsługę wywołań http. W liniach 25-31 opisuję katalogi zawierające statyczne zasoby, które mają nie być mapowane przez Springa. Linie 35-38 określają gdzie będą znajdować się pliki JSP i jakie będą miały rozszerzenie. Dzięki temu w kontrolerach nie będę tego musiał każdorazowo wskazywać.

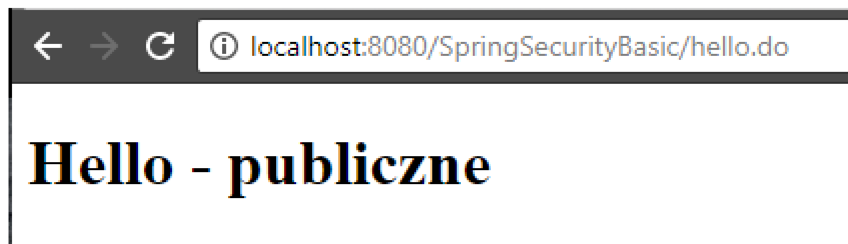
Utworzyłem też pakiet **"pl.jsystems.ssb"** i umieściłem w nim klasę która spełnia rolę kontrolera. Jak widać poniżej, kontroler ten będzie uruchamiany w reakcji na wywołanie **hello.do**.

```
 6  package pl.jsystems.ssb;
 7
 8  import org.springframework.stereotype.Controller;
 9  import org.springframework.ui.Model;
10  import org.springframework.web.bind.annotation.RequestMapping;
11
12  /**
13   *
14   * @author andrzej
15   */
16
17
18  @Controller
19  public class HelloController {
20      @RequestMapping("hello.do")
21      public String show(Model m){
22
23          return "hello";
24      }
25  }
26
```

Ponieważ metoda show zwraca ciąg "hello" a konfiguracja w pliku **spring-servlet.xml** określa że pliki jsp znajdują się w podkatalogu **WEB-INF/jsp**, to właśnie w tym miejscu tworzę plik **hello.jsp** którego zawartość to po prostu napis "hello - publiczne".

```
 7  <%@page contentType="text/html" pageEncoding="UTF-8"%>
 8  <!DOCTYPE html>
 9  <html>
10      <head>
11          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12          <title>JSP Page</title>
13      </head>
14      <body>
15          <h1>Hello - publiczne</h1>
16      </body>
17  </html>
18
```

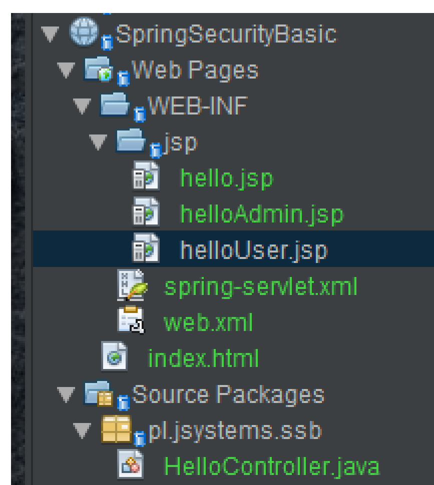
Uruchamiam aplikację i sprawdzam działanie nowo dodanego widoku:



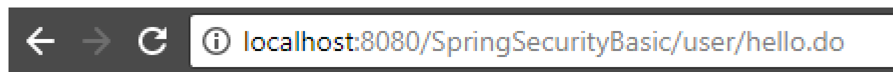
W podobny sposób tworzę jeszcze 2 widoki. Ja rozbudowałem swoją klasę HelloController o dwie dodatkowe metody, ale jeśli jest Ci tak wygodniej to możesz zrobić osobne klasy:

```
17
18  @Controller
19  public class HelloController {
20
21      @RequestMapping("hello.do")
22      public String show(Model m){
23          return "hello";
24      }
25
26       @RequestMapping("/user/hello.do")
27      public String showUser(Model m){
28          return "helloUser";
29      }
30
31      @RequestMapping("/admin/hello.do")
32      public String showAdmin(Model m){
33          return "helloAdmin";
34      }
35
```

Dodałem też brakujące pliki jsp. W tej chwili struktura aplikacji wygląda w ten sposób:

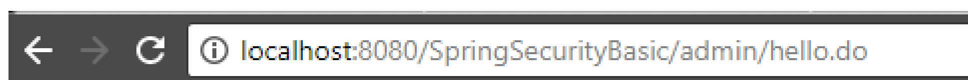


Sprawdzam teraz działanie nowych widoków. Widok który będzie widoczny dla zalogowanych użytkowników oraz administratorów:



Hello - user

Widok który będzie widoczny tylko dla administratorów:



Hello - admin

Kod aplikacji w aktualnym stanie można pobrać z :
http://jsystems.pl/static/download/blog/java/SpringSecurityBasic.zip

Zabezpieczamy aplikację

Do pliku **web.xml** dodaję wpisy jak niżej (możesz je skopiować z kodu źródłowego dołączonego do tej publikacji):

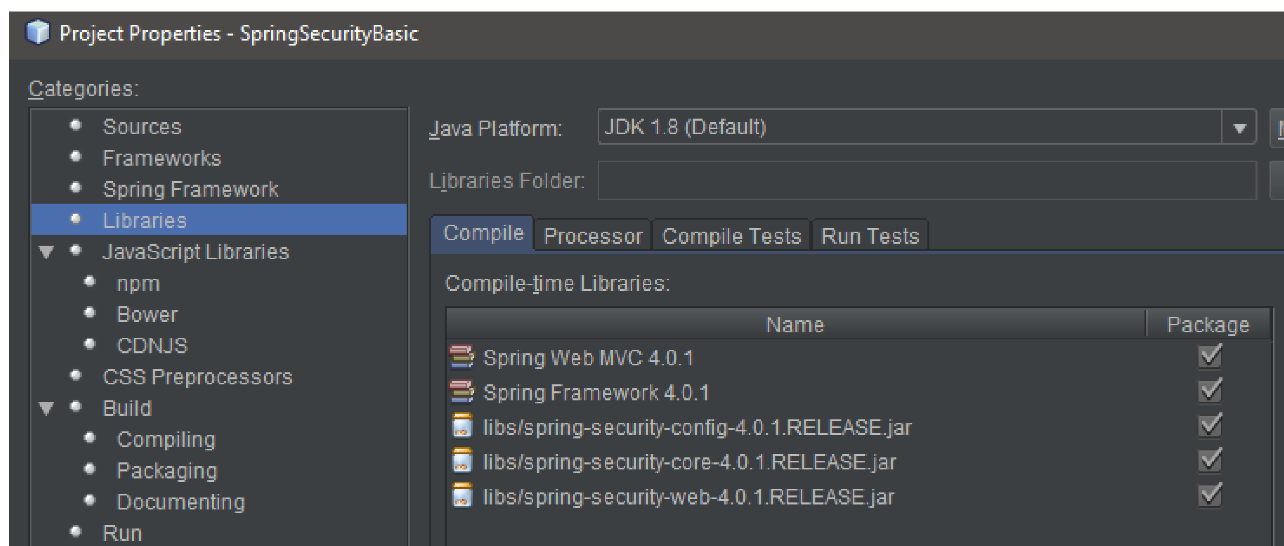
```
24    <filter>
25        <filter-name>springSecurityFilterChain</filter-name>
26        <filter-class>org.springframework.web.filter.DelegatingFilterProxy
27        </filter-class>
28    </filter>
29
30    <filter-mapping>
31        <filter-name>springSecurityFilterChain</filter-name>
32        <url-pattern>/*</url-pattern>
33    </filter-mapping>
34
35    <listener>
36        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
37    </listener>
38
39    <context-param>
40        <param-name>contextConfigLocation</param-name>
41        <param-value>
42            /WEB-INF/security.xml
43        </param-value>
44    </context-param>
```

Zasadniczo jest to kod szablonowy, to co będziemy zmieniać znajduje się w liniach 32 i 42. Jak wspominałem na samym początku, Spring Security jest oparte o zwykłe filtry. W linii 32 określiłem jakich wywołań ma Spring Security dotyczyć. Zapis **/*** oznacza wszystkie wywołania w ramach aplikacji. To kto będzie mógł gdzie zaglądać określę w pliku konfiguracyjnym Spring Security, którego położenie deklaruję w linii 42.

Zanim przejdziemy do właściwej konfiguracji, dodajemy niezbędne biblioteki. Spakowałem je i wrzuciłem osobno na serwer, więc możesz je pobrać z :

http://jsystems.pl/static/download/blog/java/libsSSB.zip

Po pobraniu trzeba je rozpakować i dodać do zależności projektu.



Przejdźmy teraz do zawartości pliku **security.xml**. Plik w całości możesz skopiować z dołączonego do materiałów kodu źródłowego. Poniżej znajduje się tylko najważniejsza część tego pliku. Przypomnijmy założenia: Do strony /hello.do może dostać się każdy, do strony /user/hello.do tylko zalogowany użytkownik lub administrator, a do /admin/hello.do tylko administrator. W liniach 19 i 20 określiłem użytkowników oraz ich role. Mamy więc użytkownika o nazwie "killer" i administratora o nazwie "siara". W liniach 12 i 13 określiłem kto ma mieć do czego dostęp.

```
10
11      <http auto-config="true">
12          <intercept-url pattern="/user/**" access="hasAnyAuthority('USER','ADMIN')" />
13          <intercept-url pattern="/admin/**" access="hasAnyAuthority('ADMIN')" />
14      </http>
15
16      <authentication-manager>
17          <authentication-provider>
18              <user-service>
19                  <user name="kiler" password="rozmach" authorities="USER" />
20                  <user name="siara" password="rysia" authorities="ADMIN" />
21              </user-service>
22          </authentication-provider>
23      </authentication-manager>
24
```

Przełącznik **"pattern"** określa wzorzec adresów których ma dotyczyć zabezpieczenie. Wszystkie adresy zaczynające się od /user/ będa wymagały zalogowania się jako użytkownik posiadający rolę "USER". Analogicznie adresy zaczynające się od "/admin/" będą wymagały logowania jako

ADMIN. Słowa USER i ADMIN nie są jakimiś specjalnymi słowami kluczowymi. Możesz sobie określić rolę o nazwie choćby "WĄSKI" i używać dokładnie tak samo. Pewnego wyjaśnienia wymaga zapis "**". Czym różni się "**" od "*"? Otóż zapis: /user/** dotyczy zarówno adresu: /user/strona.do jak i /user/costam/strona.do, a zapis "*" tylko tego pierwszego.

Ponieważ dla adresów nie poprzedzonych "/user/" albo "/admin/" nie określiłem wymaganej autoryzacji, są dostępne publicznie. Możesz teraz sprawdzić że strona hello.do jest publiczna, strona /user/hello.do jest widoczna po zalogowaniu zarówno jako kiler jak i jako siara, natomiast stronę /admin/hello.do może oglądać tylko siara.

Kod aktualnego stanu aplikacji możesz pobrać :

http://jsystems.pl/static/download/blog/java/SpringSecurityBasic2.zip

Użytkownicy i role przechowywane w bazie danych

Dotychczasowa konfiguracja spełnia swoją funkcję, jednak dodawanie kolejnych użytkowników wymagałoby modyfikowania pliku XML zawartego w aplikacji. Nie znajdzie to więc zastosowania w sytuacji w której np użytkownicy będą mogli się rejestrować online w aplikacji. Przerobimy nieco naszą aplikację w taki sposób, by informacje o użytkownikach i ich uprawnieniach były przechowywane w bazie danych. Na potrzeby przykładu przygotowałem klaster PostgreSQL i utworzyłem w nim bazę danych o nazwie **"spring_security"**. W tej bazie utworzyłem tabele z użyciem poniższych instrukcji:

```
create table uzytkownicy(
id_uzytkownika serial primary key,
login text not null,
haslo text not null,
aktywny boolean default true
);

create table role_uzytkownikow(
id_roli serial primary key,
nazwa_roli text not null
);

create table uzytkownik_rola_intersect(
id_uri serial primary key,
id_uzytkownika integer not null references uzytkownicy(id_uzytkownika),
id_roli integer not null references role_uzytkownikow(id_roli)
);
```

Konstrukcja tabel nie jest z góry narzucona. Twoje tabele mogą mieć taki kształt jaki chcesz, muszą jednak umożliwiać uruchomienie na nich zapytania które na podstawie nazwy użytkownika zwróci informację o haśle i aktywności konta, oraz zapytania które wylistuje użytkowników i przypisane im role. Poniżej zapytania które będą uruchamiane na tych tabelach:

```
select login,haslo, aktywny from uzytkownicy where login=?
```

```
select login,nazwa_roli from uzytkownicy join uzytkownik_rola_intersect
using(id_uzytkownika) join role_uzytkownikow using(id_roli);
```

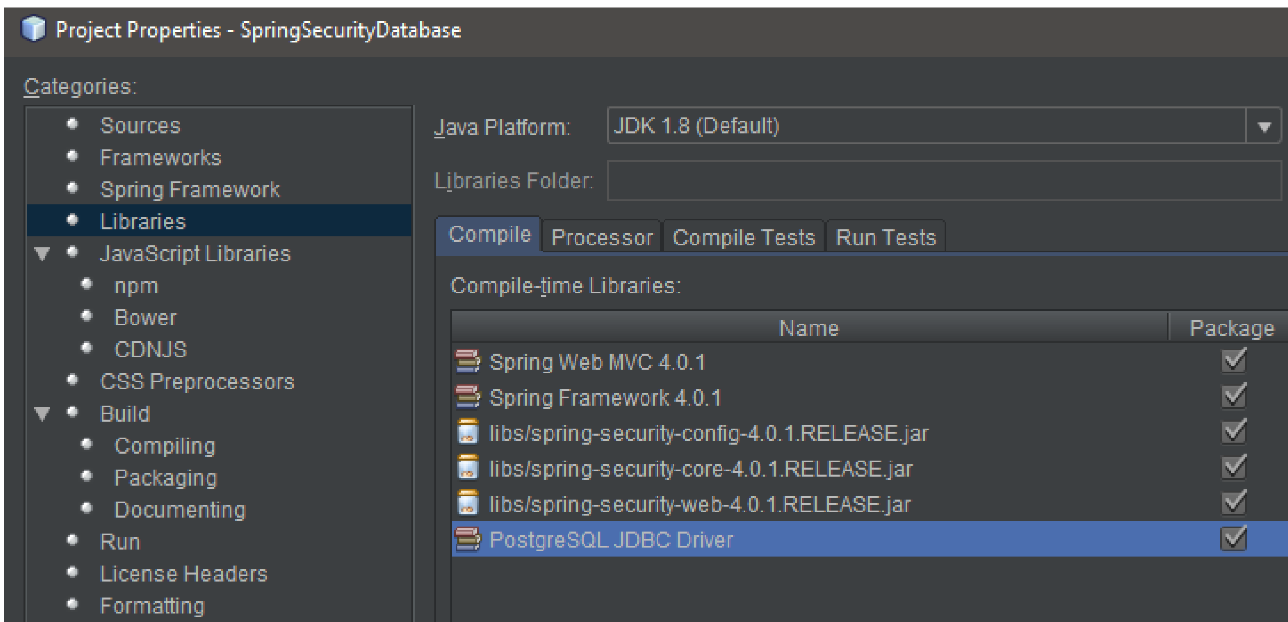
Do naszych tabel trzeba jeszcze dodać użytkowników, role i powiązania między nimi. Poniżej odpowiedni kod:

```
insert into uzytkownicy(login,haslo) values ('kiler','rozmach');
insert into uzytkownicy(login,haslo) values ('siara','rysia');
insert into role_uzytkownikow(nazwa_roli) values ('USER');
insert into role_uzytkownikow(nazwa_roli) values ('ADMIN');
insert into uzytkownik_rola_intersect(id_uzytkownika,id_roli) values (1,1);
insert into uzytkownik_rola_intersect(id_uzytkownika,id_roli) values (2,2);
```

Użytkownicy i ich hasła są identyczne jak i przy konfiguracji z użyciem XML. Stworzyłem też użytkownika bazodanowego i nadałem mu uprawnienia niezbędne do odpytywania tych tabel:

```
create user ssd with password 'jsystems';
grant select on uzytkownicy,role_uzytkownikow,uzytkownik_rola_intersect to ssd;
```

Ponieważ w celu autoryzacji Spring będzie sięgał do bazy, musimy dodać do projektu odpowiedni sterownik:



Przejdźmy teraz do konfiguracji XML. Edytuję plik **security.xml** - ten w którym dotychczas miałem bezpośrednio wprowadzonych użytkowników i hasła. Kod można skopiować z dołączonego to materiałów projektu.

```
10
11    <http auto-config="true">
12        <intercept-url pattern="/user/**" access="hasAnyAuthority('USER','ADMIN')" />
13        <intercept-url pattern="/admin/**" access="hasAnyAuthority('ADMIN')" />
14    </http>
15
16
17
18            <b:bean id="dataSource"
19                    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
20                <b:property name="driverClassName" value="org.postgresql.Driver"/>
21                <b:property name="url" value="jdbc:postgresql://localhost/spring_security"/>
22                <b:property name="username" value="ssd"/>
23                <b:property name="password" value="jsystems"/>
24            </b:bean>
25
26
27
28    <authentication-manager>
29
30        <authentication-provider>
31
32            <jdbc-user-service data-source-ref="dataSource"
33                        users-by-username-query=
34                    "select login,haslo, aktywny from uzytkownicy where login=?"
35                        authorities-by-username-query=
36                    "select login,nazwa_roli from uzytkownicy join uzytkownik_rola_intersect
37                    using(id_uzytkownika) join role_uzytkownikow using(id_roli) where login=?" />
38
39
40        </authentication-provider>
41
42    </authentication-manager>
43
```

Sekcja w liniach 18-24 to dane niezbędne do połączenia się z bazą danych w której znajdują się nasze tabele. Weź pod uwagę że nazwa sterownika i url będą różne w zależności od tego do jakiej bazy będziemy się łączyć.

Linie 28-42 zawierają informację o sposobie wyciągania z bazy informacji o użytkownikach. Wcześniej pomiędzy tagami authentication-provider była sekcja user-service w której użytkownicy i hasła byli po prostu wymienieni. Teraz w jej miejsce pojawia się sekcja jdbc-user-service która zawiera zapytania odpytujące odpowiednie tabele.

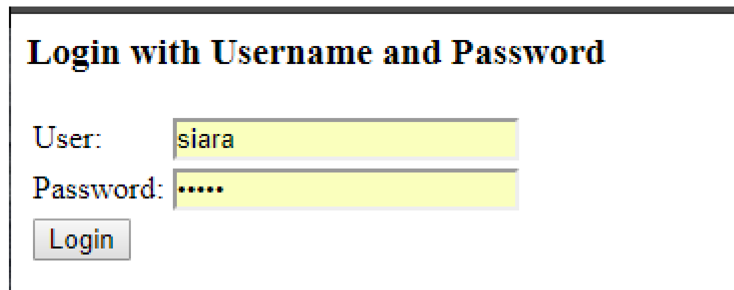
To już wszystko :) Ponieważ do bazy wprowadziliśmy tych samych użytkowników których używaliśmy wcześniej, aplikacja powinna zachowywać się identycznie jak wcześniej.

Kod aplikacji do tego tutoriala możesz pobrać :

http://jsystems.pl/static/download/blog/java/SpringSecurityDatabase.zip

Logowanie i wylogowywanie

Spring dostarcza gotowy formularz logowania i wygląda on domyślnie tak:



Zwykle będziemy go chcieli zmienić na jakąś własną implementację z własnym wyglądem, oraz dodać możliwość wylogowania się. Tym właśnie się teraz zajmiemy. Do pliku **security.xml** dodajemy wpis **"form-login"** jak niżej:

```
11      <http auto-config="true">
12          <intercept-url pattern="/user/**" access="hasAnyAuthority('USER','ADMIN')" />
13          <intercept-url pattern="/admin/**" access="hasAnyAuthority('ADMIN')" />
14
15          <form-login
16              login-page="/login.do"
17              login-processing-url="/j_spring_security_check"
18              default-target-url="/hello.do"
19              authentication-failure-url="/login.do?error"
20              username-parameter="username"
21              password-parameter="password" />
22
```

Co oznaczają poszczególne parametry? **"login-page" -** to adres pod jakim jest nasza strona logowania - czyli adres pod jaki użytkownik zostanie przekierowany w przypadku gdyby zażądał dostępu do strony wymagającej autoryzacji. **"login-processing-url"** to parametr określający adres kontrolera obsługującego autoryzację. Pod ten adres powinieneś wysyłać zawartość formularza logowania. Ja użyłem domyślnego springowego, więc tak naprawdę mogłoby tego parametru tu nie być. Dałem go w charakterze przykładu, gdybyś zechciał go zmienić. **"default-target-url"** to adres pod który domyślnie zostanie przekierowany użytkownik po zalogowaniu (o ile nie usiłował dostać się na stronę wymagającą wcześniejszej autoryzacji, w związku z czym znalazł się na stronie logowania).**"authentication-failure-url"** - to adres pod jaki użytkownik ma zostać przekierowany w przypadku nieprawidłowego logowania. W naszym przypadku jest to o tyle istotne, że chcemy taką sytuację obsługiwać poprzez wyświetlenie na stronie logowania odpowiedniego komunikatu. Będziemy ten komunikat przekazywać do widoku z kontrolera, o ile w pasku pojawi się parametr "error". Parametry **"username-parameter"** i **"password-parameter"** określają nazwy pól w

formularzu logowania w których znajdą się nazwa użytkownika i hasło. Możesz tych parametrów nie ustawiać, ale wtedy pola w formularzu będą musiały się nazywać konkretnie **"j_username"** i **"j_password"**.

Przejdźmy teraz do widoku logowania tj pliku **login.jsp**:

```
<h1>Moja implementacja logowania...</h1>

<form name='loginForm' action="<c:url value='j_spring_security_check' />" method='POST'>
    <table>
        <tr>
            <td>Użyszkodnik:</td>
            <td><input type='text' name='username' value=''></td>
        </tr>
        <tr>
            <td>Hasło:</td>
            <td><input type='password' name='password' /></td>
        </tr>
        <tr>
            <td ><input name="submit" type="submit"
                        value="submit" /></td>

            <td>
                <c:if test="${not empty error}">
                    <div class="error"><b>${error}</b></div>
                </c:if>
            </td>

        </tr>
    </table>

    <input type="hidden" name="${_csrf.parameterName}"
        value="${_csrf.token}" />
</form>
```

W linii 11 to co istotne to adres określony w parametrze **"action" -** określa on adres pod jaki wysyłane będą dane z formularza. W tym przypadku mamy domyślny kontroler dostarczany przez Springa - można to skonfigurować inaczej korzystając z parametru **"login-processing-url"**. Linie 12 - 32 to formularz z polami służącymi do wprowadzenia nazwy użytkownika i hasła. Zwróć uwagę na to, że nazwy te pokrywają się z nazwami określonymi przed momentem w pliku **security.xml**. Poza polami do wprowadzania danych mamy też przycisk do zatwierdzenia formularza, oraz (w liniach 26-28) miejsce zawierające ewentualny komunikat błędu (gdyby ktoś podał niepoprawne dane logowania). Element widoczny w liniach 34-35 to element zabezpieczenia przed atakami typu CSFR i domyślnie jest wymagany.

Przejdźmy teraz do kontrolera obsługującego logowanie:

```
20    @Controller
21    public class LoginController {
22
23        @RequestMapping(value = "login.do", method = RequestMethod.GET)
24        public String show(Model m,
25                HttpServletRequest request,
26                @RequestParam(value = "error", required = false) String error) {
27
28            if (error != null) {
29                m.addAttribute("error", "Błędne dane logowania");
30            }
31
32            return "login";
33        }
```

Kontroler ten zawiera metodę obsługującą wywołanie **login.do** (a więc adresu który w security.xml ustawiłem jako adres formularza logowania). Zasadniczo przy żądaniu GET kontroler ma tylko spowodować wyświetlenie omawianego przed chwilą widoku **login.jsp**. Gdyby w pasku został przekazany parametr error (a tak się stanie w przypadku wcześniejszego błędnego logowania) do widoku zostanie przekazany przez model dodatkowy komunikat.

Przy próbie dostępu do strony wymagającej uprzedniego zalogowania dostajemy widok jak niżej:

Przy podaniu niepoprawnych danych autoryzacyjnych:

Brakuje nam w tej chwili jedynie implementacji wylogowywania. Wylogowanie sprowadzać się będzie właściwie do wyrzucenia z sesji obiektu zalogowanego użytkownika, nie potrzebujemy nawet żadnego widoku. Dodałem metodę obsługującą adres **"logout.do"**. Zwróć uwagę że jako parametr tej metody dodałem obiekt klasy HttpServletRequest. Będzie mi ten obiekt potrzebny by spowodować wyrzucenie zalogowanego użytkownika z sesji. Obiekt ten zostanie mi podany automatycznie przez Springa. Wystarczy jak widać poniżej wywołać metodę **logout()** na rzecz obiektu klasy HttpServletRequest przekazanego przez parametr metody obsługującej żądanie adresu logout.do. Po zakończeniu przekierowuję użytkownika do strony logowania. Wystarczy teraz umieścić w jakimś dostępnym na wszystkich stronach menu linka do adresu "logout.do".

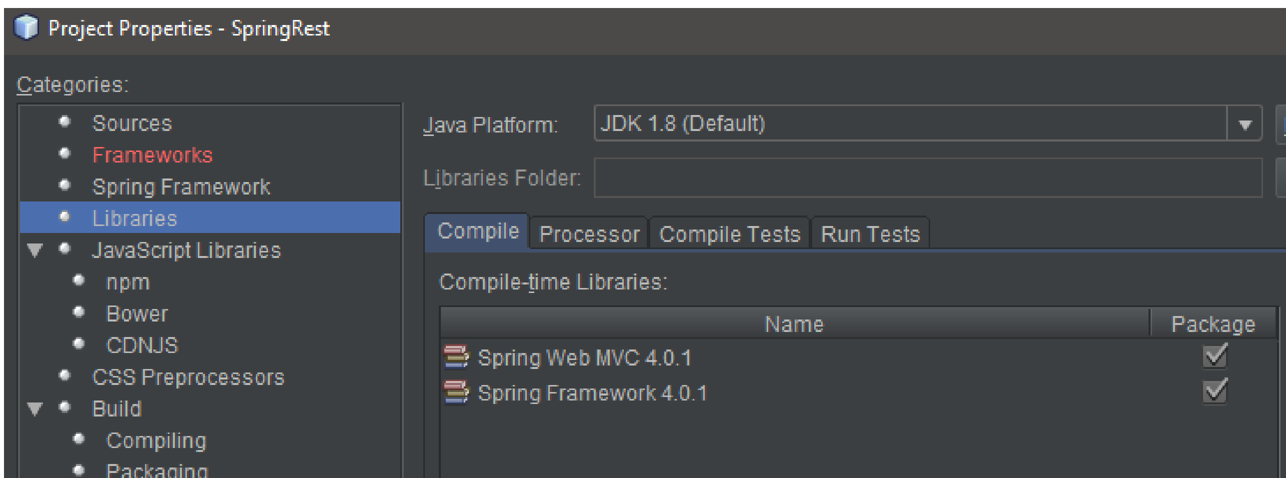
```
@RequestMapping(value = "logout.do", method = RequestMethod.GET)
public String show(HttpServletRequest request) {
    System.out.println("Wylogowanie...");
    try {
        request.logout();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return "redirect:login.do";
}
```

Kod źródłowy z niniejszego artykułu:

http://jsystems.pl/static/download/blog/java/SpringSecurityDatabase2.zip

Spring Rest

Spring Rest to moduł Spring Framework umożliwiające budowanie usług RestFull. Aby rozpocząć musimy stworzyć aplikację webową i wdrożyć do niej Springa. Istotne jest to by Spring był w **wersji <u>nie niższej niż 4.0.1!</u>** Wynika to z faktu, że będziemy korzystać z adnotacji które pojawiły się w tej właśnie wersji.



Do pliku **web.xml** dodaję konfigurację dzięki której wszystkie wywołania kończące się na **".do"** będą obsługiwane przez Springa:

```
<servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Tworzę też plik konfiguracyjny Springa tj w tym przypadku plik **spring-servlet.xml** w katalogu WEB-INF. Umieszczam w nim konfigurację dzięki której będę mógł się posługiwać adnotacjami, oraz wskazuję w którym pakiecie ma szukać kontrolerów.

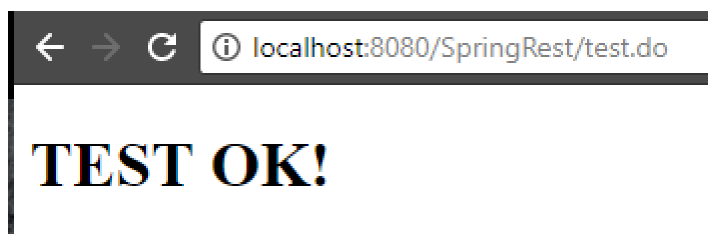
```
21    <mvc:annotation-driven/>
22
23    <context:component-scan base-package="pl.jsystems.springrest.controller"/>
24
```

Na wszelki wypadek by sprawdzić czy Spring faktycznie wyłapuje wywołania dodałem do projektu klasę Hello obsługującą żądanie http i zwracającą do wyświetlenia plik jsp:

```
16  @Controller
17  public class Hello {
18      @RequestMapping("/test.do")
19      public String show(Model m){
20          return "test";
21      }
22  }
```

```
7   <%@page contentType="text/html" pageEncoding="UTF-8"%>
8   <!DOCTYPE html>
9   <html>
10      <head>
11          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12          <title>JSP Page</title>
13      </head>
14      <body>
15          <h1>TEST OK!</h1>
16      </body>
17  </html>
```

Test się powiódł, więc możemy przejść do konfiguracji usług REST.



Obsługa żądań GET

Stworzyłem klasę Warzywo, jest to klasa domenowa która będzie wykorzystana w naszej usłudze do przekazywania danych. Zawiera tylko dwa pola, oraz ponadto gettery i settery i konstruktory:

```java
package pl.jsystems.springrest.model;

/**
 *
 * @author andrzej
 */
public class Warzywo {

    private Integer idWarzywa;
    private String nazwaWarzywa;

    public Warzywo(Integer idWarzywa, String nazwaWarzywa) {
        this.idWarzywa = idWarzywa;
        this.nazwaWarzywa = nazwaWarzywa;
    }


    public Warzywo() {
    }

    public Integer getIdWarzywa() {
        return idWarzywa;
    }

    public void setIdWarzywa(Integer idWarzywa) {
        this.idWarzywa = idWarzywa;
    }

    public String getNazwaWarzywa() {
        return nazwaWarzywa;
    }

    public void setNazwaWarzywa(String nazwaWarzywa) {
        this.nazwaWarzywa = nazwaWarzywa;
    }

}
```

Przejdźmy teraz do kontrolera naszej usługi. Zwykłe kontrolery obsługujące widoki mają adnotację **@Controller** - w tym przypadku pojawia się w jej miejsce adnotacja **@RestController**. Adnotacja @RestController pojawiła się w wersji 4 Springa. Spełnia taką samą funkcję jak połączenie @Controller i **@ResponseBody** we wcześniejszych wersjach Springa.

```
1   package pl.jsystems.springrest.controller;
2
3   import java.util.Arrays;
4   import java.util.List;
5   import org.springframework.web.bind.annotation.RequestMapping;
6   import org.springframework.web.bind.annotation.RestController;
7   import pl.jsystems.springrest.model.Warzywo;
8
9   /**
10   *
11   * @author andrzej
12   */
13
14  @RestController
15  public class Rest {
16
17      @RequestMapping("jednoWarzywo.do")
18      public Warzywo showOne() {
19          return new Warzywo(1, "Marchew");
20      }
21
22      @RequestMapping("wieleWarzyw.do")
23      public List<Warzywo> showList() {
24          return Arrays.asList(
25                  new Warzywo(1, "Marchew"),
26                  new Warzywo(2, "Ziemniak"),
27                  new Warzywo(3, "Cebula")
28          );
29      }
30  }
```

Zauważ że tym razem zamiast zwracać nazwę widoku JSP zwracam obiekt klasy Warzywo lub listę takich obiektów. W ramach jednej klasy stworzyłem dwie metody obsługujące dwa różne adresy.

Sprawdzam oba adresy:

localhost:8080/SpringRest/jednoWarzywo.do

{"idWarzywa":1,"nazwaWarzywa":"Marchew"}

localhost:8080/SpringRest/wieleWarzyw.do

[{"idWarzywa":1,"nazwaWarzywa":"Marchew"},{"idWarzywa":2,"nazwaWarzywa":"Ziemniak"},{"idWarzywa":3,"nazwaWarzywa":"Cebula"}]

Dane zostały automatycznie rzutowane na format JSON - obecnie najpopularniejszy i wypierający format XML charakteryzujący się dużym dodatkowym narzutem danych (tagi XML zajmują znacznie więcej miejsca). W przypadku usług REST możesz podobnie jak przy zwykłym MVC używać parametrów wywołania.

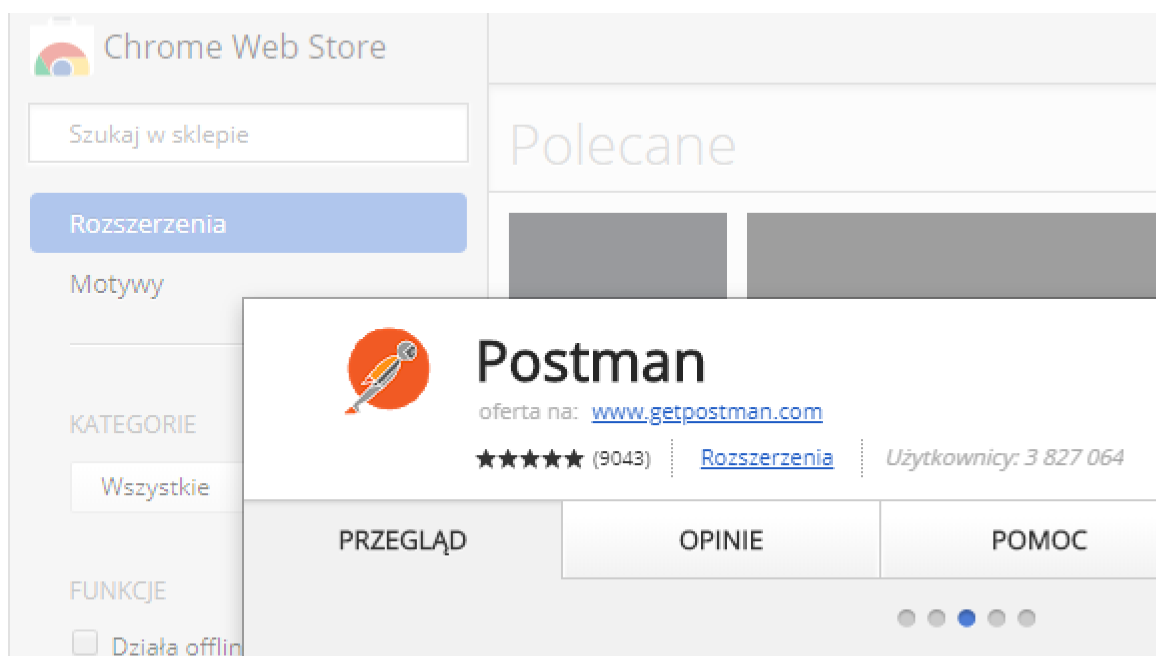
Obsługa żądań POST

Nasza usługa może też odebrać dane przesłane POSTem (czy innymi metodami HTTP - PUT, DELETE etc). Używam tej samej klasy "Warzywo" co w poprzednim przykładzie. Do naszego dotychczasowego kontrolera dodałem dwie dodatkowe metody - catchPost i catchPostStatus. Obie przyjmują przesłane do nich obiekty klasy Warzywo w formacie JSON. Pierwsza z nich nic nie zwraca, druga zwraca obiekt ResponseEntity który pozwala nam zwrócić jakiś status odpowiedzi.

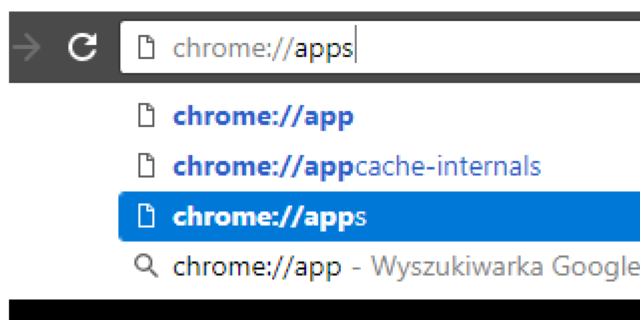
```
13  /**
14   *
15   * @author andrzej
16   */
17  @RestController
18  public class Rest {
19
20      @RequestMapping("jednoWarzywo.do")
21      public Warzywo showOne() {
22          return new Warzywo(1, "Marchew");
23      }
24
25      @RequestMapping("wieleWarzyw.do")
26      public List<Warzywo> showList() {
27          return Arrays.asList(
28                  new Warzywo(1, "Marchew"),
29                  new Warzywo(2, "Ziemniak"),
30                  new Warzywo(3, "Cebula")
31          );
32      }
33
34      @RequestMapping(value = "chwytajWarzywo.do", method = RequestMethod.POST)
35      public void catchPostVoid(@RequestBody Warzywo w) {
36          System.out.println("Otrzymałem: " + w);
37
38      }
39
40      @RequestMapping(value = "chwytajWarzywoStatus.do", method = RequestMethod.POST)
41      public ResponseEntity catchPostStatus(@RequestBody Warzywo w) {
42          System.out.println("Otrzymałem: " + w);
43          return new ResponseEntity(w, HttpStatus.OK);
44      }
45
46  }
47
```

W obu metodach pojawia się adnotacja **@RequestBody.** Obie metody przyjmują przez parametr obiekt klasy Warzywo. W takiej konfiguracji możesz wewnątrz metody zarządzać już gotowym obiektem który do metody został przesłany jako JSON i Spring automatycznie przepakował go do obiektu. Metody POST nie wywołamy już tak łatwo przez przeglądarkę jak metodę GET. Aby przetestować nasze nowe metody potrzebujemy jakiegoś klienta usług sieciowych. Może to być np **CURL** albo **Postman**.

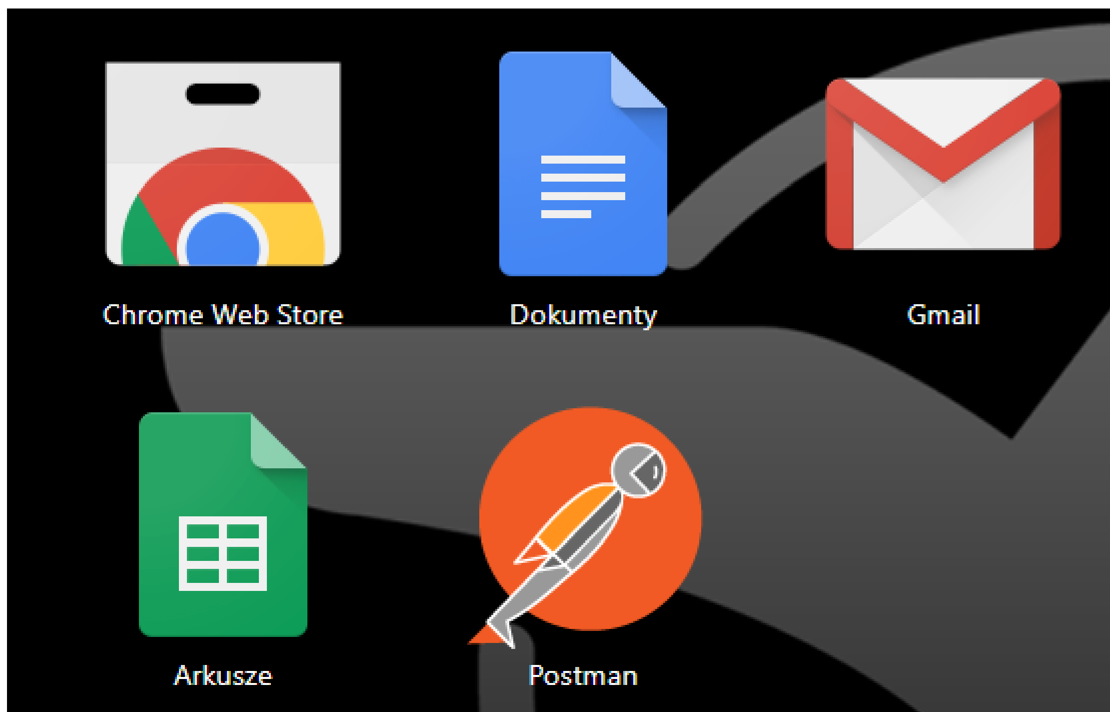
Wybieram tę drugą aplikację i instaluję w przeglądarce Chrome.



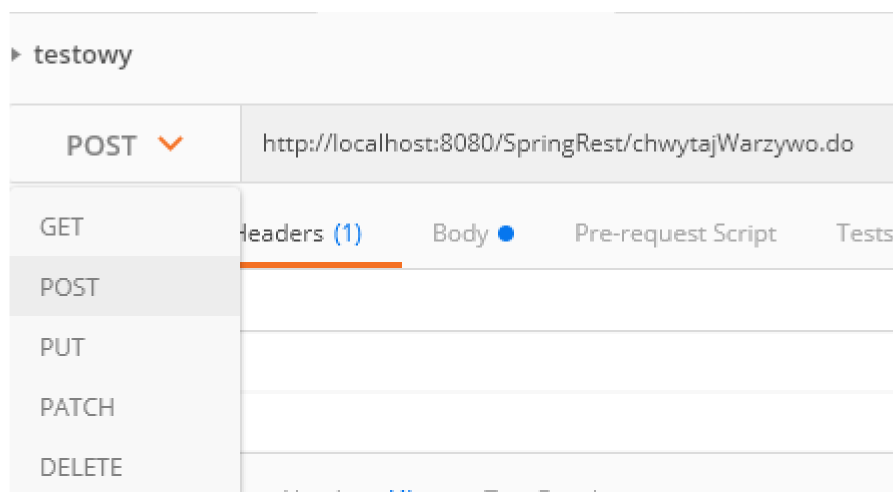
Po instalacji w pasku adresu Chroma wystarczy wprowadzić ciąg **chrome://apps** :



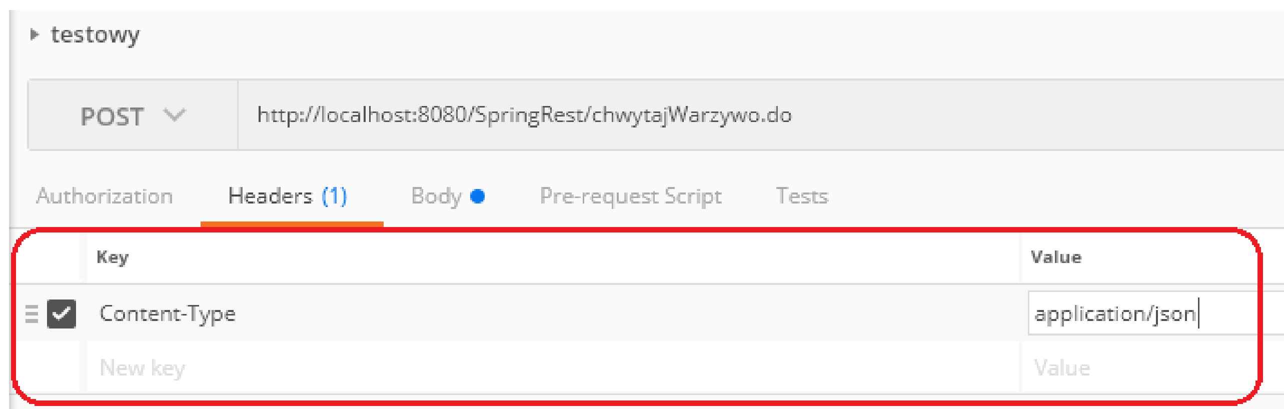
Pojawi nam się menu z którego możemy wybrać Postmana:



Po uruchomieniu narzędzia wprowadzamy adres usługi, oraz wybieramy metodę którą chcemy się z nią komunikować:



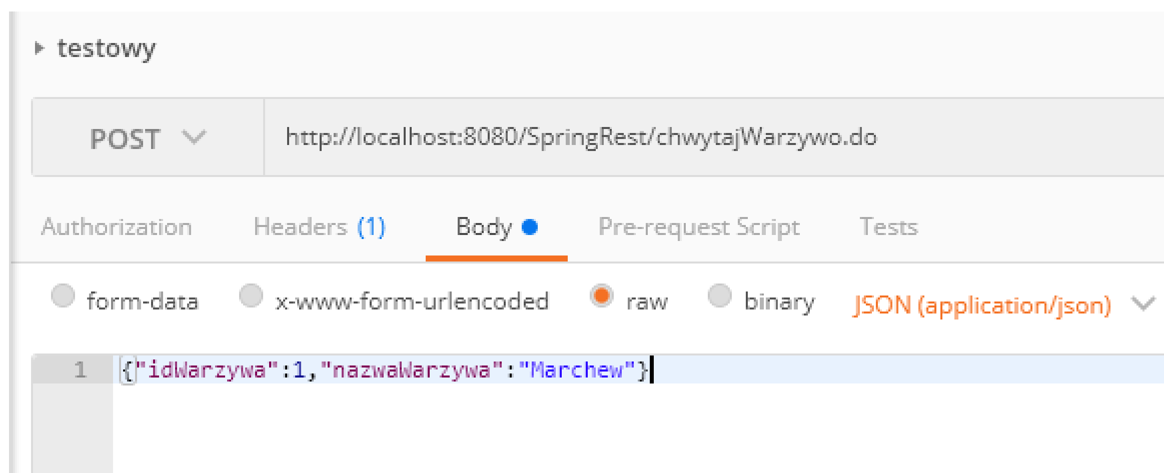
Ponieważ dane będziemy wysyłać w formacie JSON, w sekcji **"Headers"** deklarujemy format przesyłanych danych:



Jeśli tego nie ustawisz, dostaniesz w odpowiedzi błąd "415 Unsupported Media Type" (!!!!)

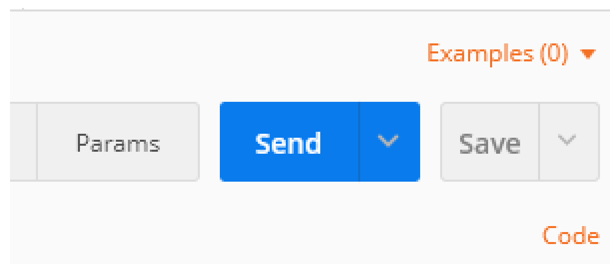
Przyczyn tego błędu może być na tyle dużo, że późniejsze docieranie do źródła problemu nie należy do rzeczy szybkich i przyjemnych, toteż polecam o tym ustawieniu pamiętać.

W sekcji **"Body"** wprowadzamy przesyłany do usługi obiekt w formacie JSON:

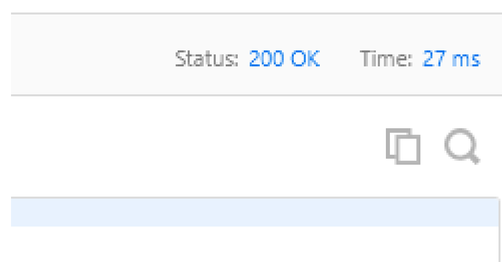


W tym miejscu po prawej stronie powinien być zaznaczony format JSON(application/json) - jeśli nie jest, zaznacz go. Powinno zostać to ustawione automatycznie w związku z wprowadzeniem odpowiedniego ustawienia w sekcji headers. To działa w dwie strony - jeśli tutaj wyklikasz ten format (a domyślnie jest text), to w headers powinno pojawić się stosowne ustawienie.

Wszystko mamy wprowadzone, więc wybieramy przycisk "Send" znajdujący się po prawej stronie.

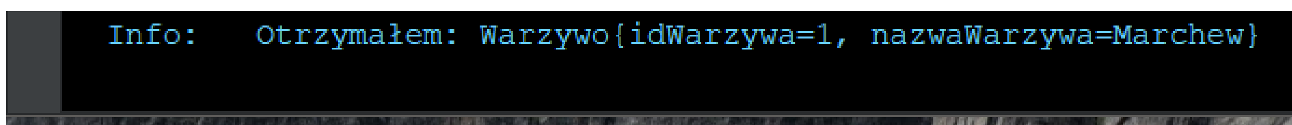


Po wysłaniu danych, w dolnej części formularza po prawej stronie powinniśmy dostać status odpowiedzi:



a w przypadku metody catchPostStatus zostanie nam dodatkowo zwrócony obiekt który przesłaliśmy (może się to przydać np. gdy wartość klucza głównego jest nadawana na poziomie bazy, a my zapisując obiekt przy użyciu usługi sieciowej chcielibyśmy dowiedzieć się pod jakim ID został owy obiekt zapisany w bazie.

Ja dostałem jeszcze oprogramowany wcześniej komunikat na konsoli:



Kod źródłowy z niniejszego artykułu:

http://jsystems.pl/static/download/blog/java/SpringRest.zip