Tutorial Spring Framework – moduł MVC

Autor: Andrzej Klusiewicz
klusiewicz@jsystems.pl

Spis treści

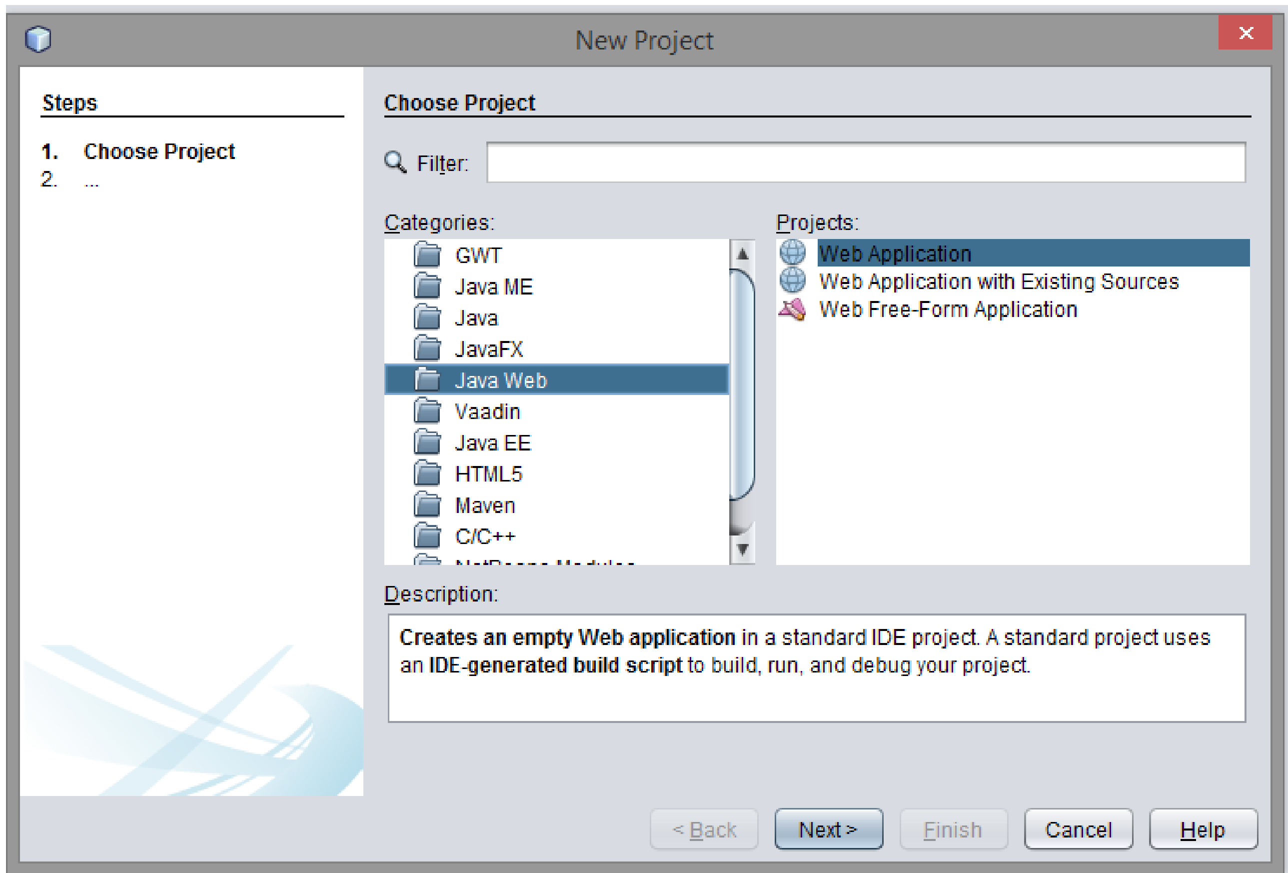
Tutorial Spring Framework – moduł MVC....................1
Spring MVC – pierwsza aplikacja – jak to wszystko działa....................3
Położenie pliku konfiguracji Springa....................25
Przekazywanie obiektów i list do warstwy widoku....................26
Mapowanie na poziomie klasy....................32
Zmienne ścieżki....................34
Zmienne tablicowe....................37
Parametry żądania....................39
Przechwytywacze....................42
Najprostszy formularz....................45
Walidacja formularzy....................50
Upload plików....................52

Spring MVC – pierwsza aplikacja – jak to wszystko działa

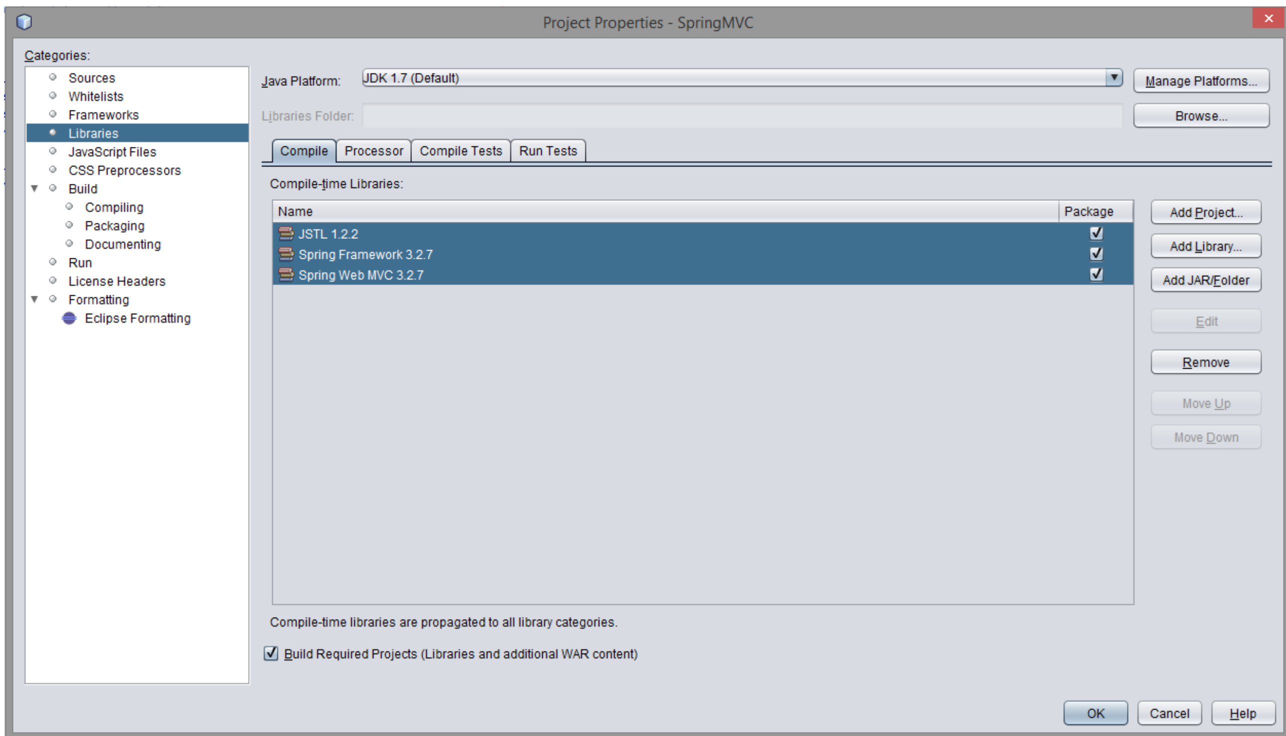
Kod źródłowy aplikacji którą tworzę w niniejszym kursie jest do pobrania z adresu:
http://www.jsystems.pl/storage/spring/springmvc1.zip

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

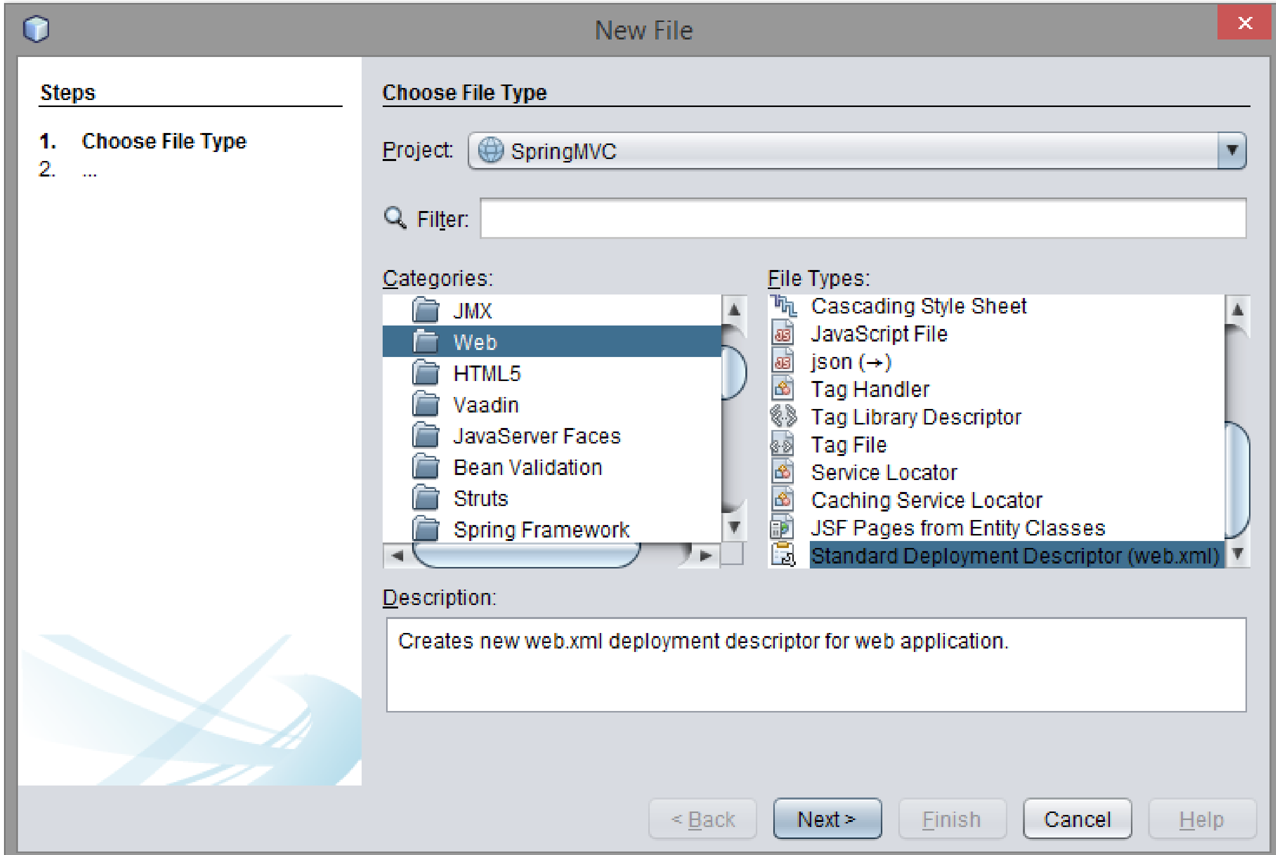
Zaczniemy od stworzenia zwykłej aplikacji WEBowej:



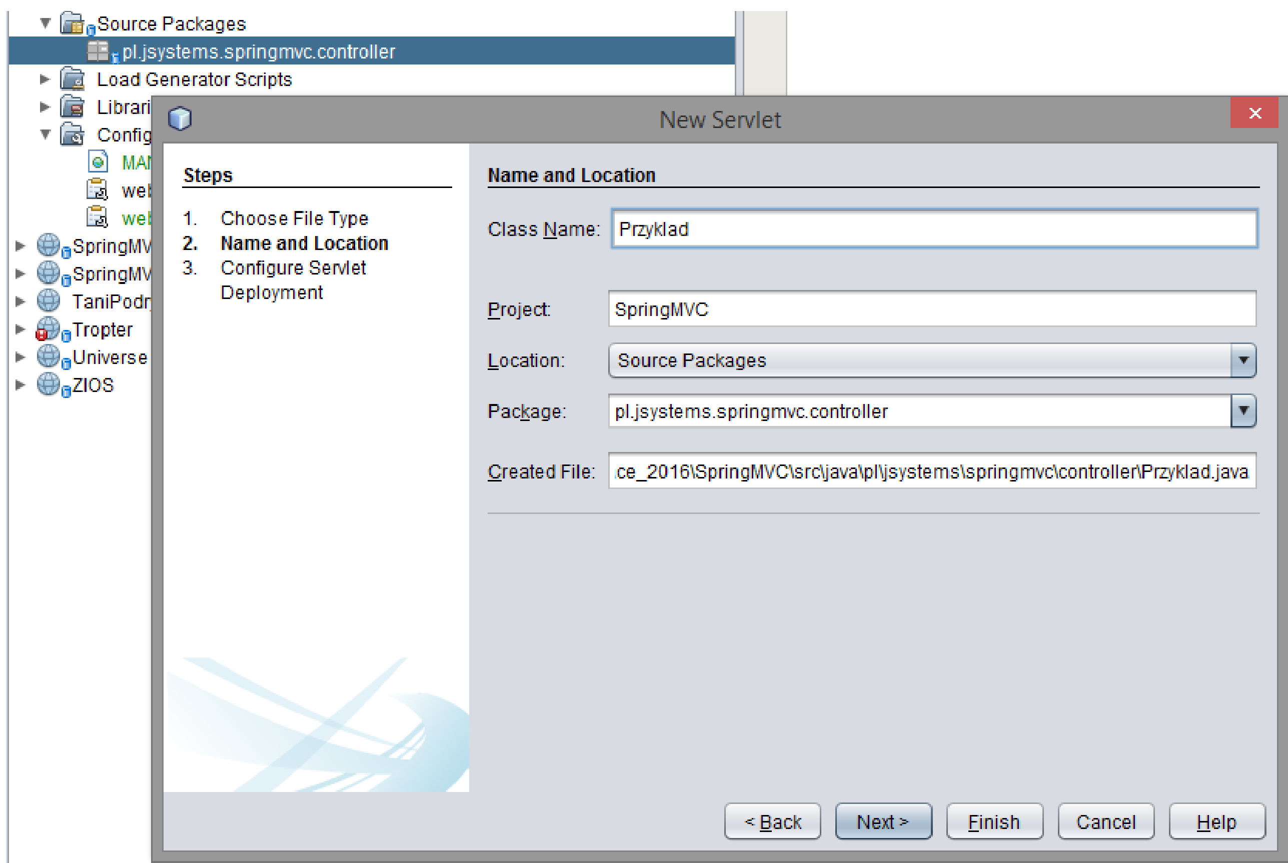
Po jej utworzeniu musimy dodać niezbędne biblioteki. W Netbeans należy wybrać właściwości projektu, przejść do sekcji „Libraries” a następnie kliknąć AddLibrary i wybrać potrzebne:



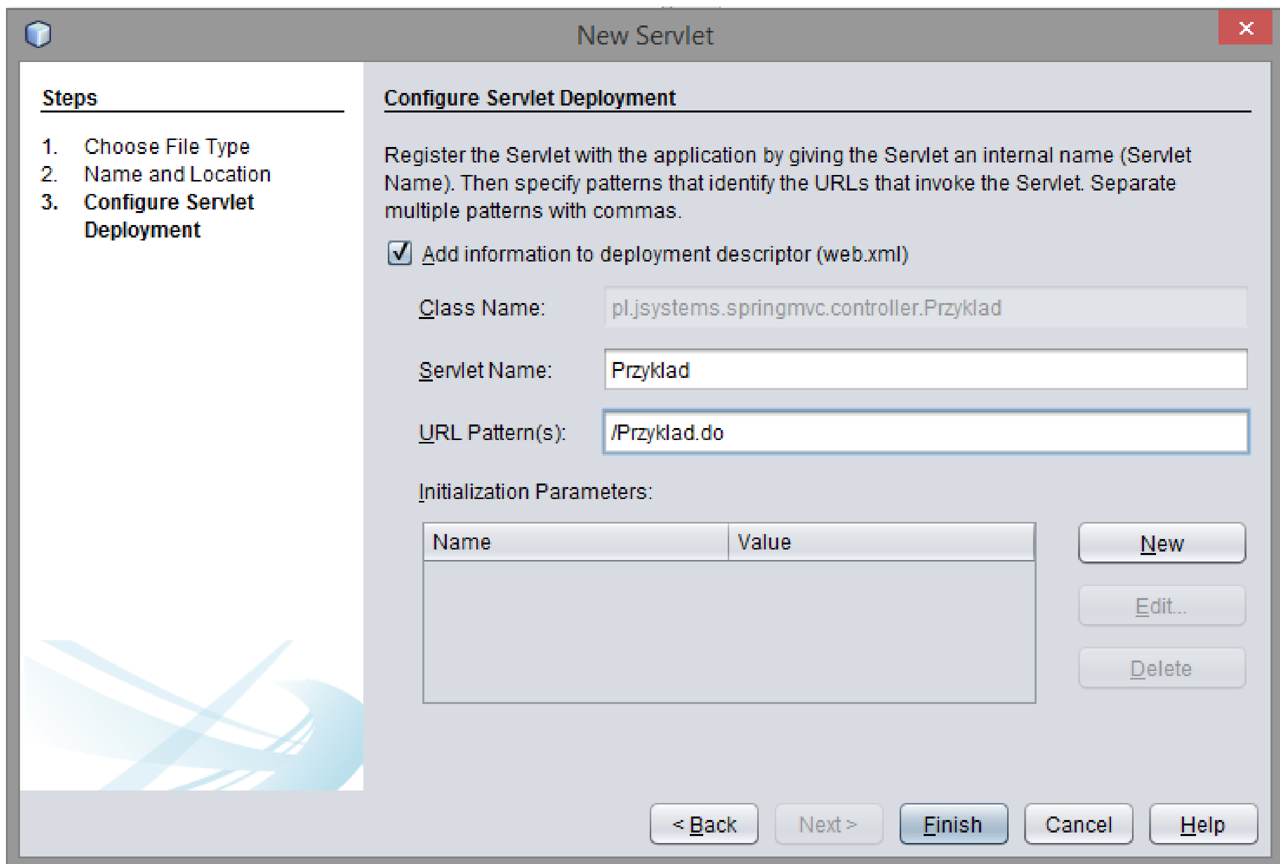
Będzie nam też potrzebny plik konfiguracyjny web.xml, dlatego dodajemy do do katalogu WEB-INF:



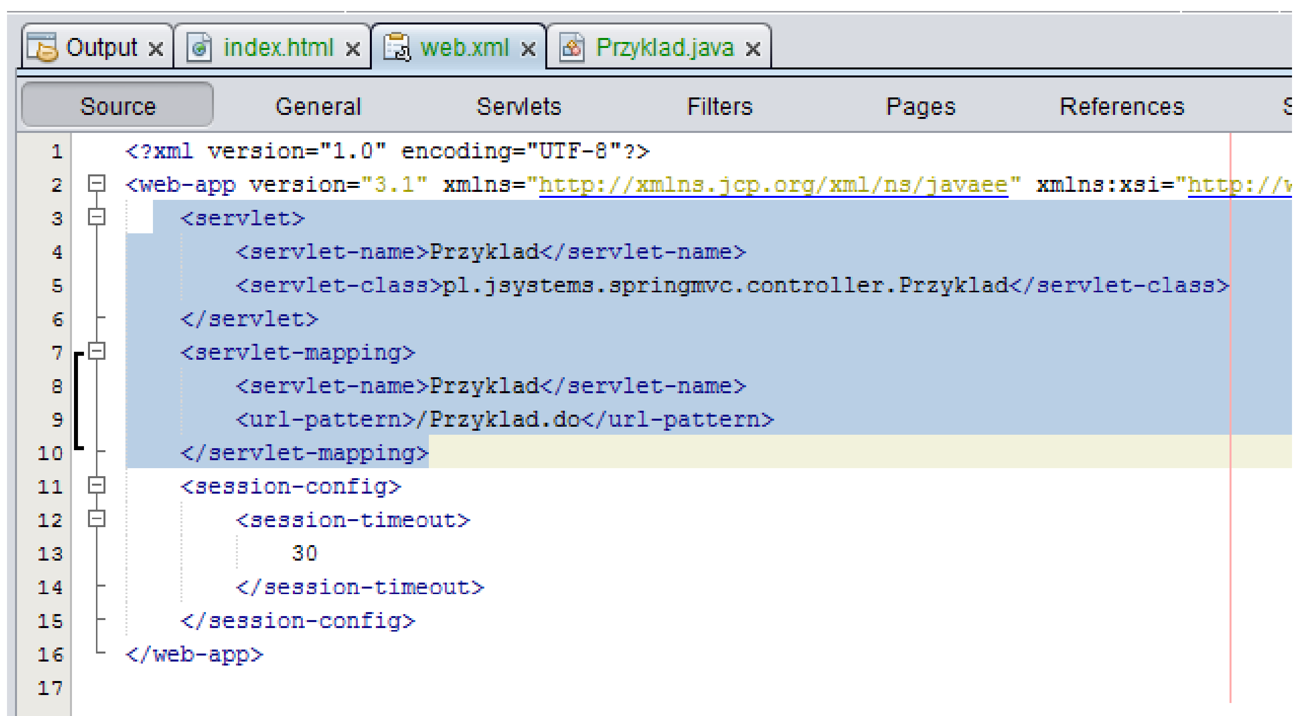
Będzie nam też potrzebny jakiś pakiet w którym umieścimy nasze klasy:



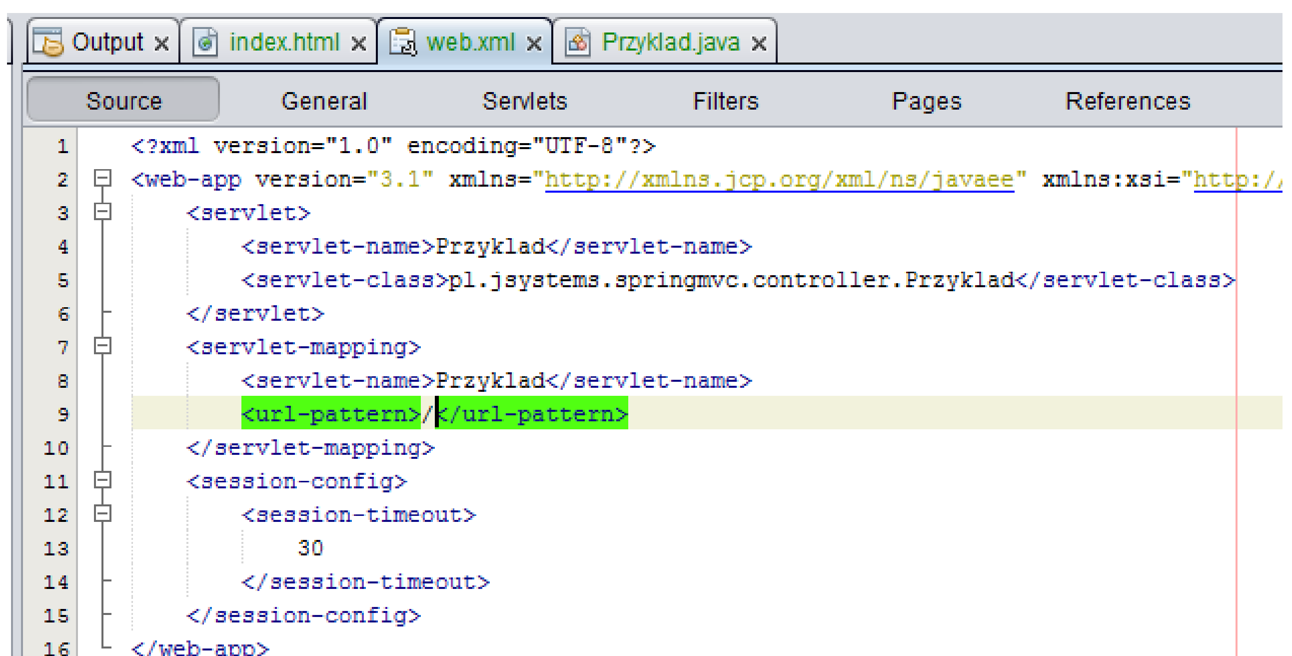
Na początek aby przyjrzeć się sposobowi przekazywania kontroli nad wywołaniami Springowi, stworzymy zwyczajny serwlet. Pamiętaj by zaznaczyć dodanie informacji o nim do web.xml!



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



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 :)



Ważna informacja!!

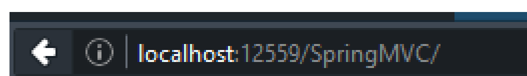
Taki sposób deklaracji wzorca URL obsługiwanego przez Spring MVC sprawi, że również wszystkie statyczne zasoby będą obsługiwane przez Spring. To może uniemożliwić np. osadzenie plików obrazków czy PNG w aplikacji – te przecież nie będą obsługiwane przez żadne kontrolery. Bezpieczniej więc będzie użyć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

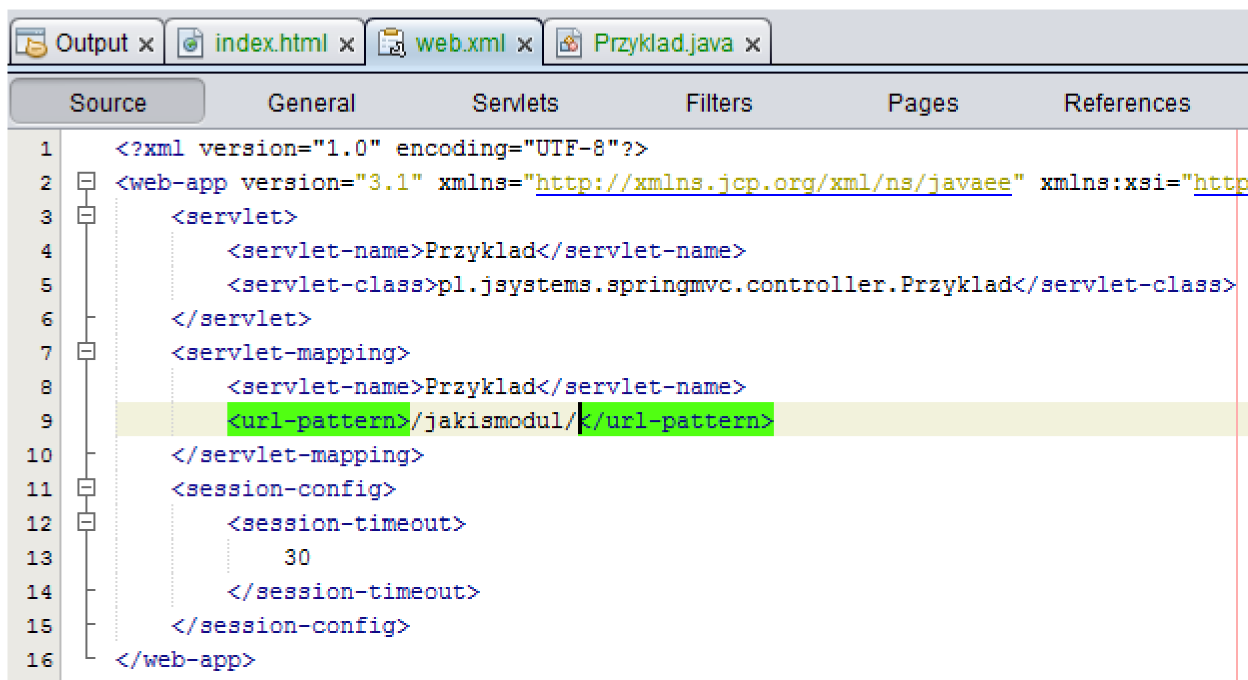
```
6   package pl.jsystems.springmvc.controller;
7
8   import java.io.IOException;
9   import java.io.PrintWriter;
10  import javax.servlet.ServletException;
11  import javax.servlet.http.HttpServlet;
12  import javax.servlet.http.HttpServletRequest;
13  import javax.servlet.http.HttpServletResponse;
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:

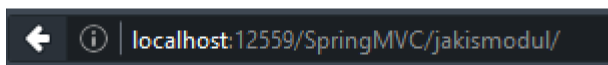


Halo, tutaj servlet!

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:

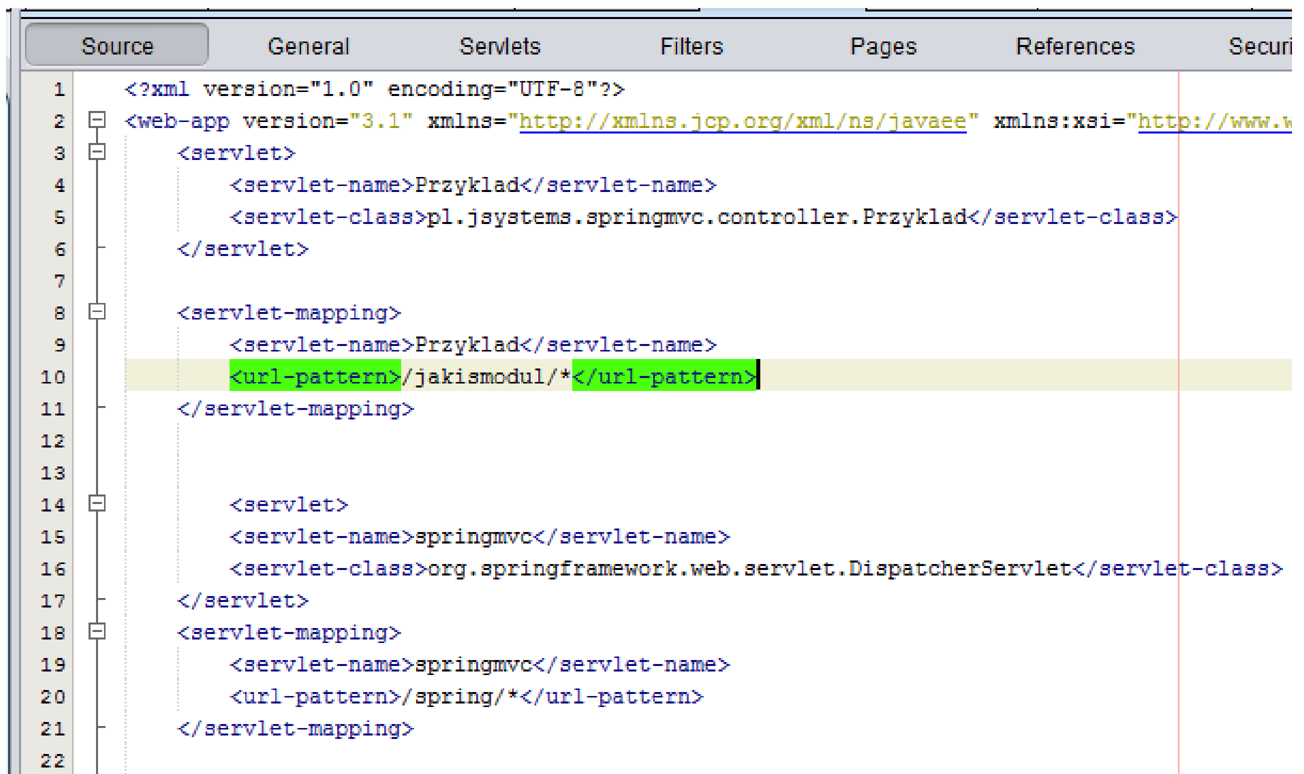


Porównajmy adres wywołania naszego serwletu:

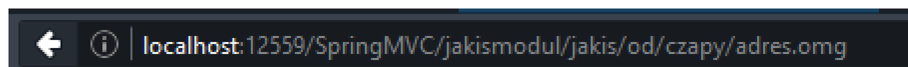


Halo, tutaj servlet!

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:

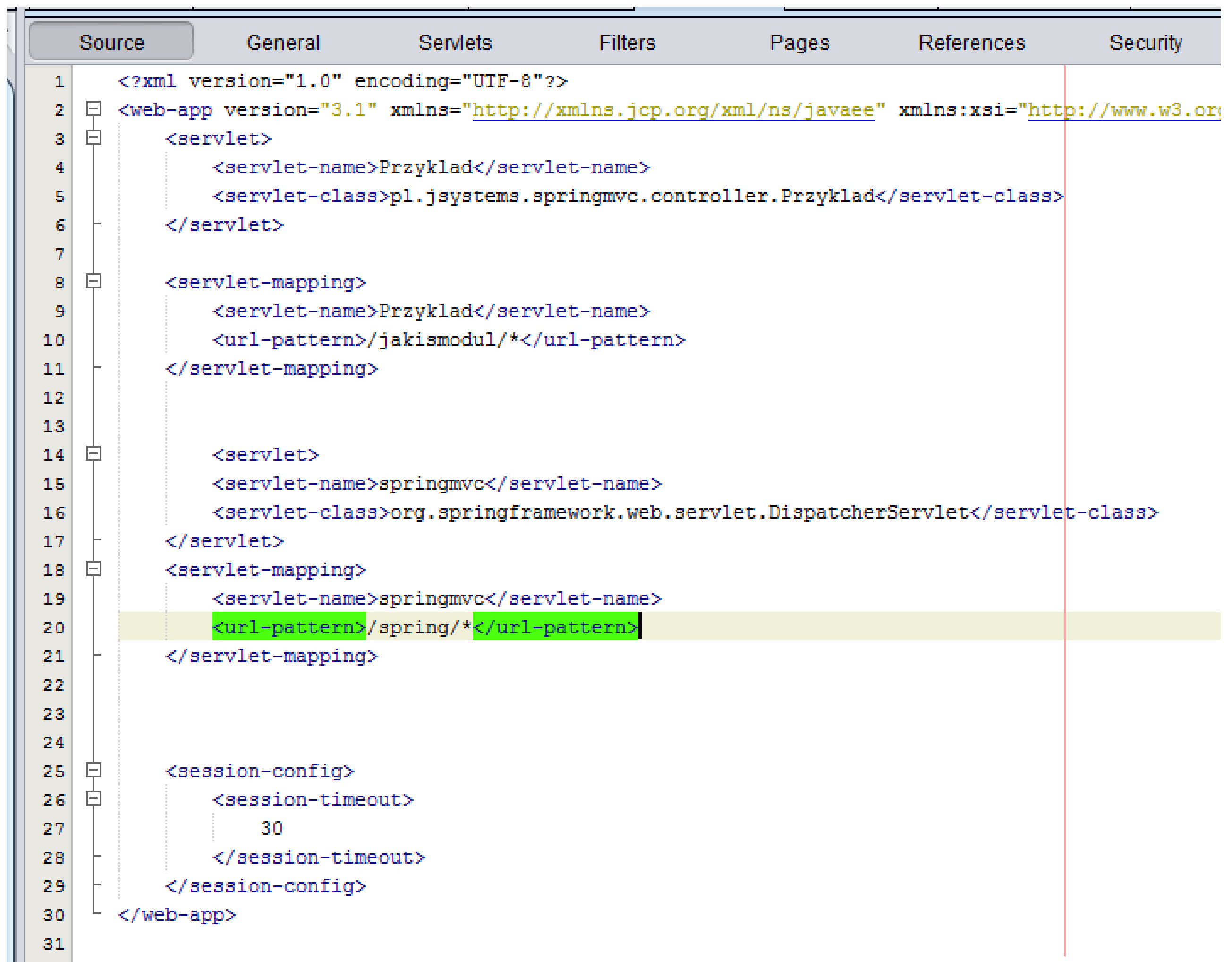


I sprawdzamy :

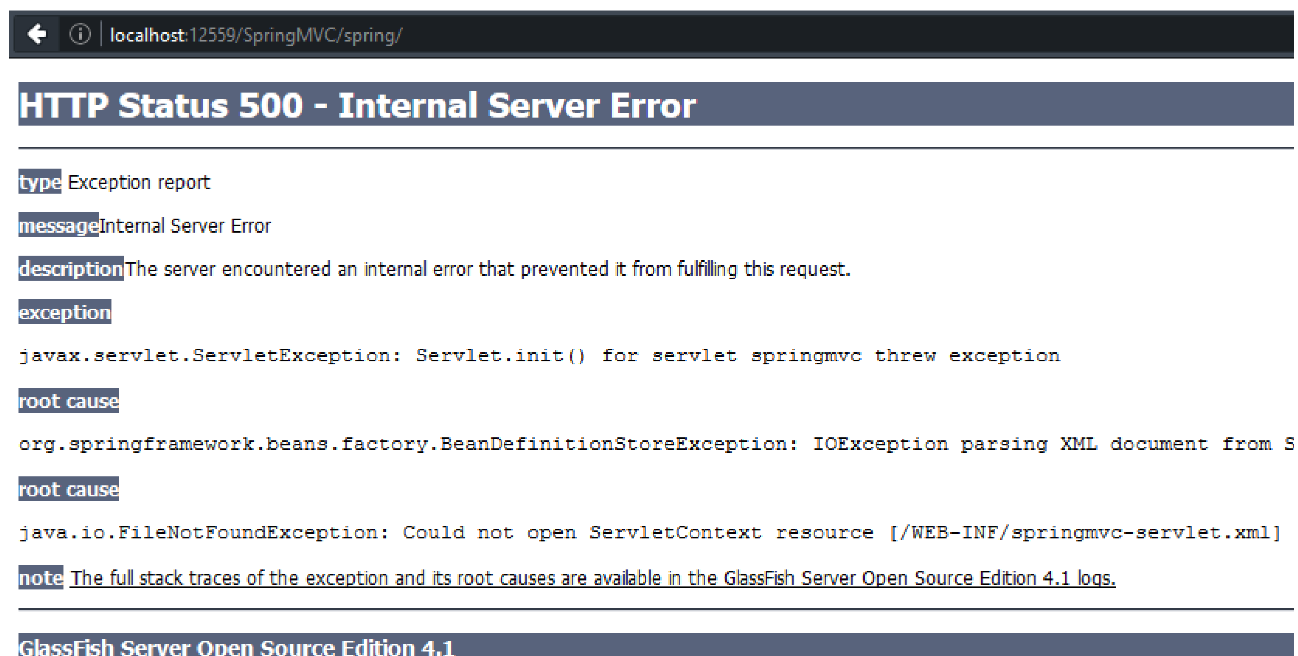


Halo, tutaj servlet!

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..

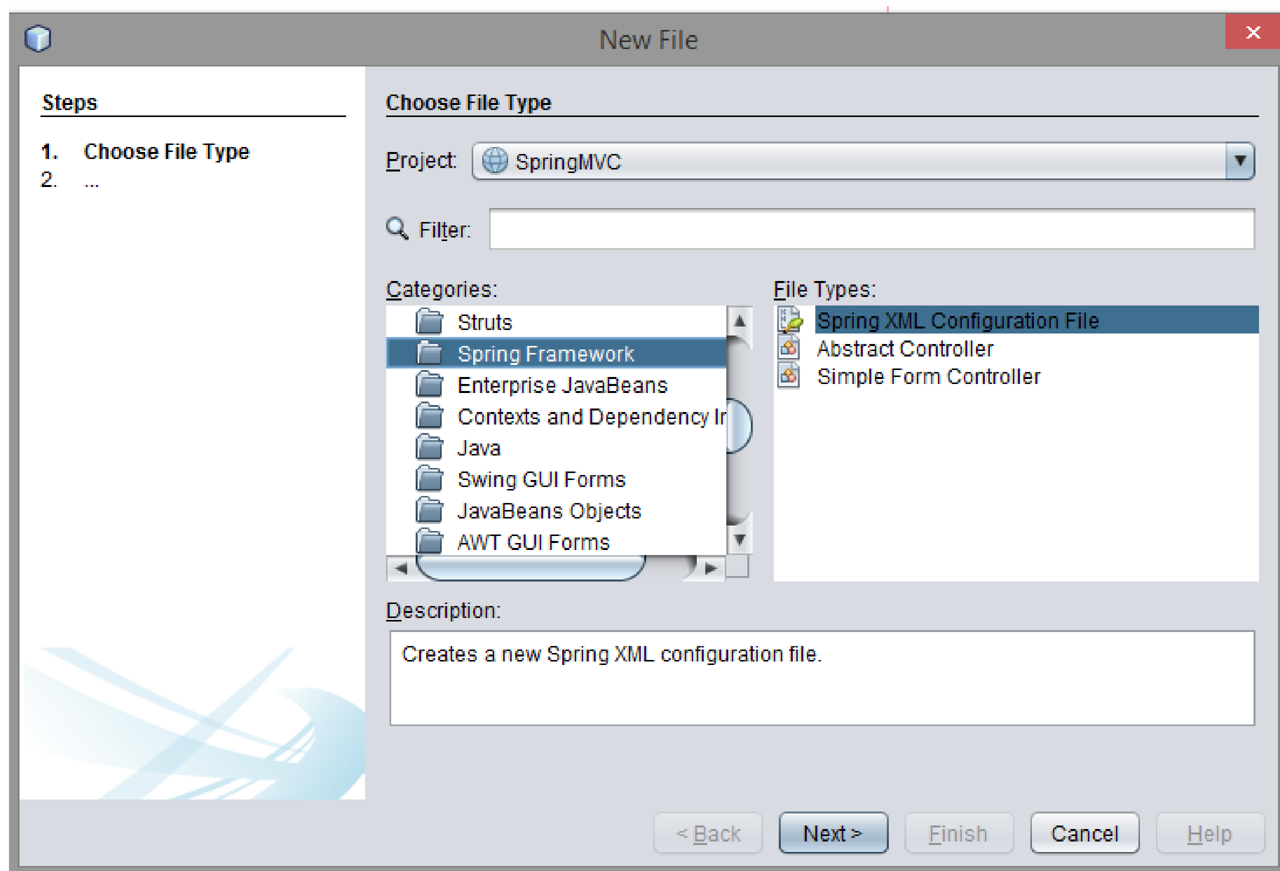


OK, sprawdźmy teraz co się stanie kiedy wywołamy w przeglądarce adres nowego modułu:

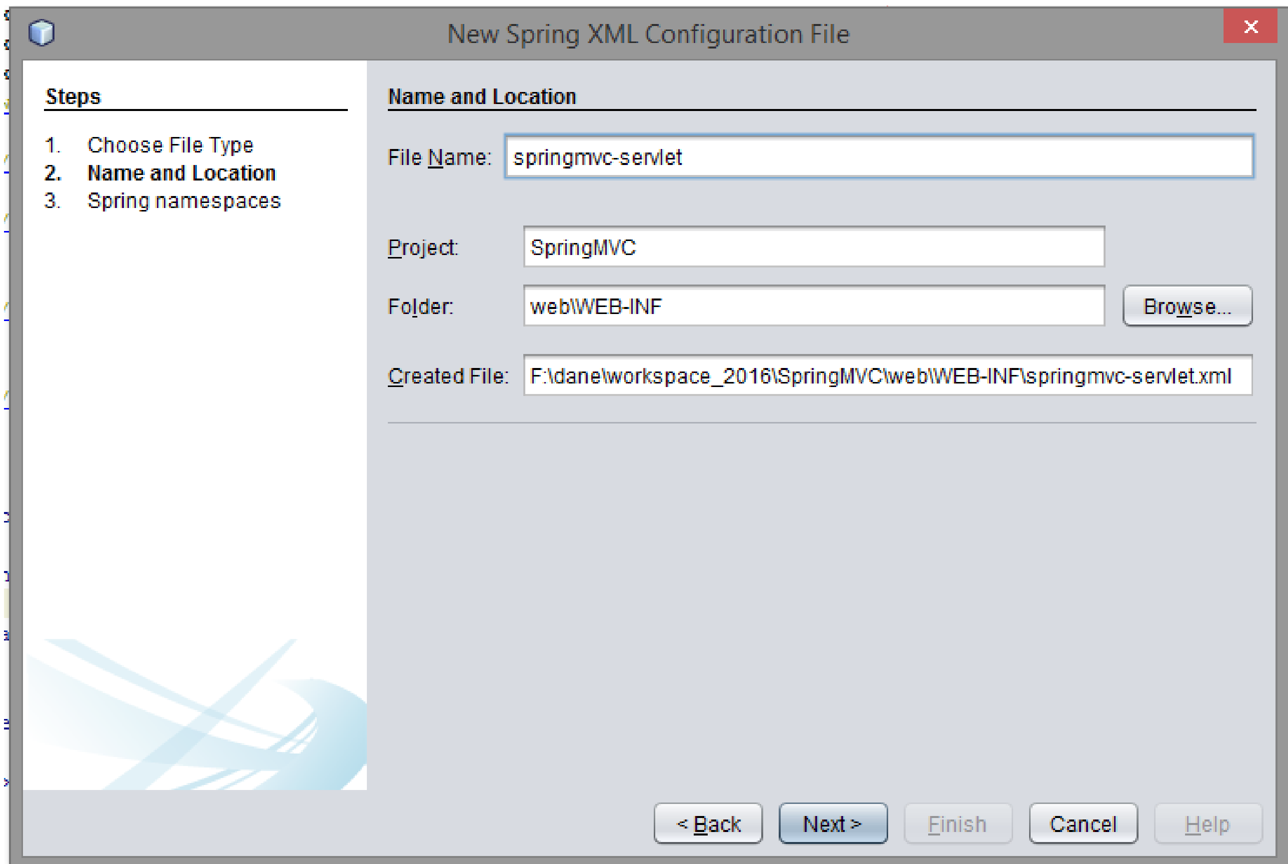


Pojawił się błąd, a gdy mu się przyjrzymy zobaczymy że problem polega na braku pliku springmvc-servlet.xml To jest plik konfiguracyjny Spring MVC w którym będziemy deklarowali m.in. kontrolery naszej aplikacji. Nazwa pliku springmvc-servlet.xml nie jest przypadkowa. Wynika ona z tego jak nazwaliśmy servlet 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.

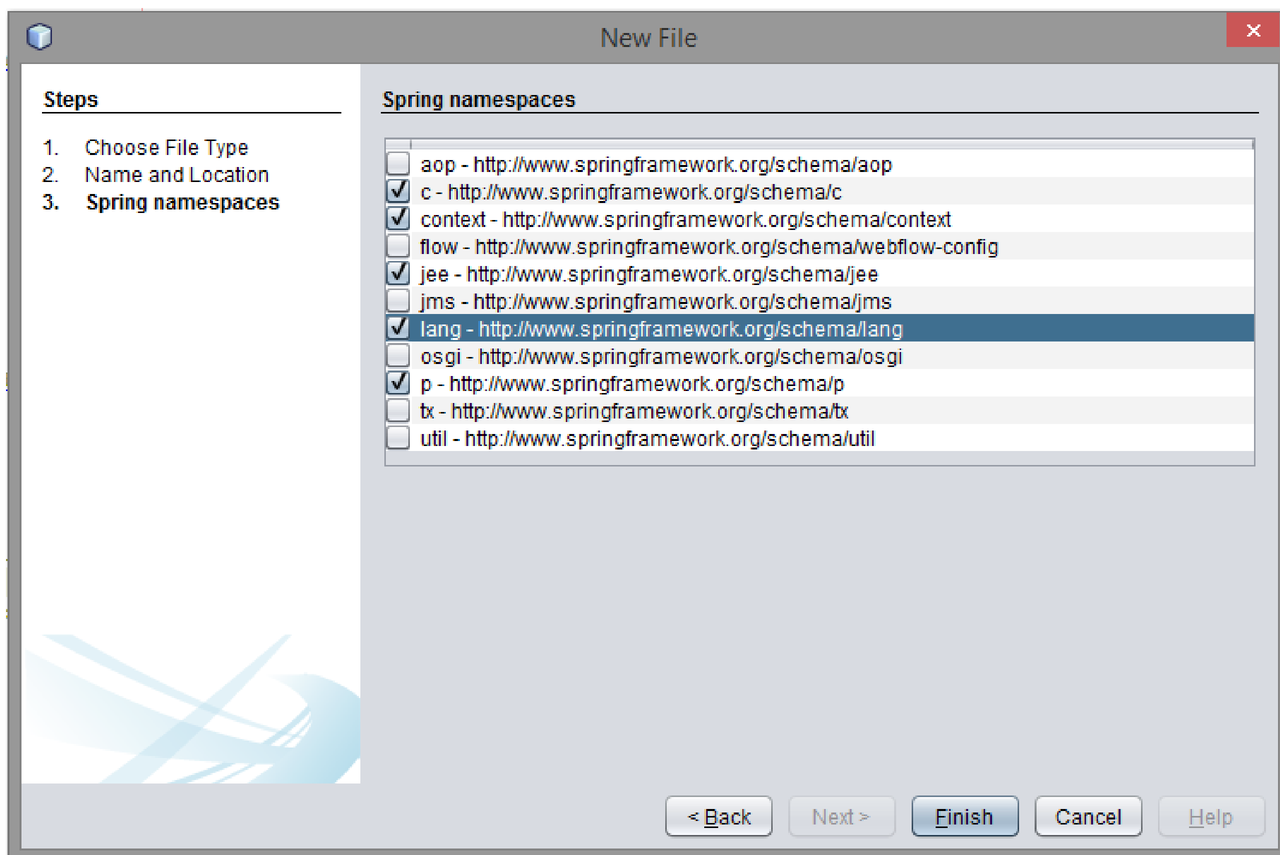
Skoro Spring tak bardzo potrzebuje tego pliku, to mu go dajmy :



Nadamy mu nazwę springmvc-servlet.xml:



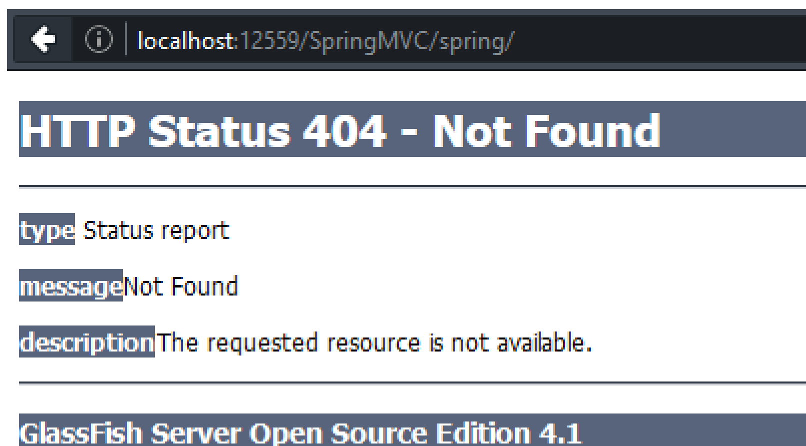
Będą nam też potrzebne pewne przestrzenie nazw XML które będziemy wykorzystywać, dlatego je zaznaczamy:



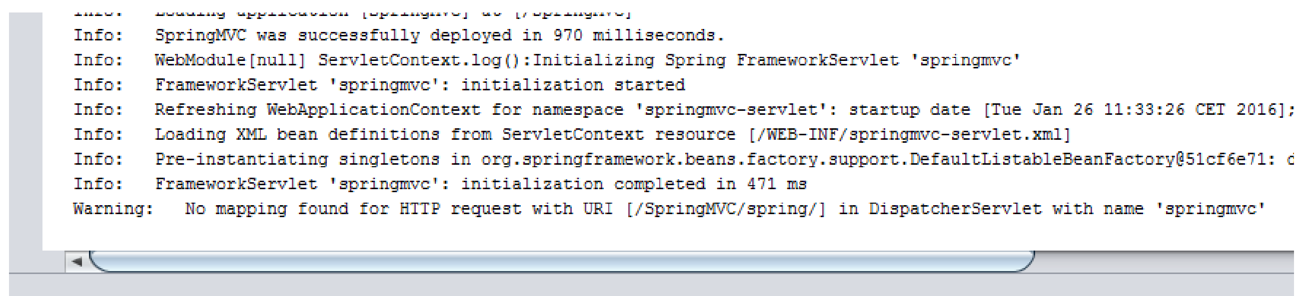
Nie ma możliwości wyboru przestrzeni MVC dlatego musimy dodać ją ręcznie do nowego pliku. Możesz tez gotowy plik pobrać z przykładowego kodu którego adres znajdziesz na początku tego rozdziału i umieścić go w katalogu WEB-INF.

```
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:



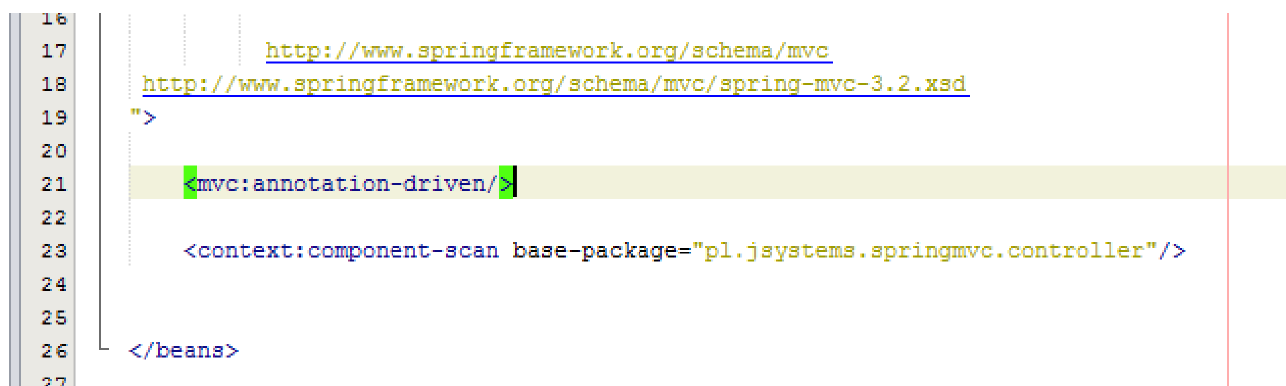
Zaglądamy do konsoli serwera:



```
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:



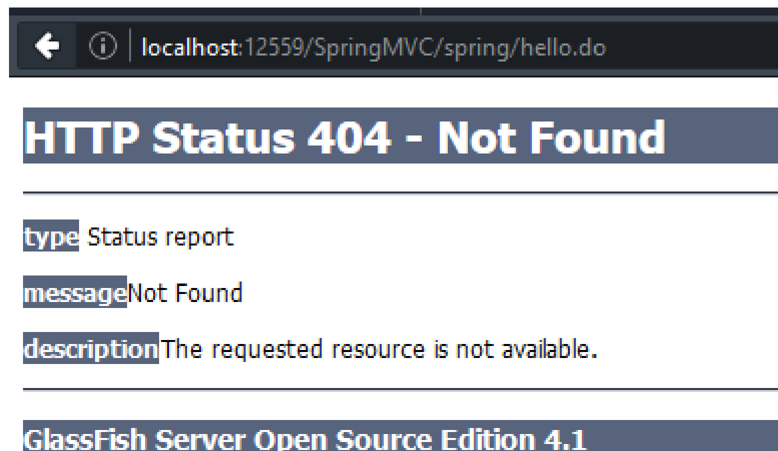
```
16
17              http://www.springframework.org/schema/mvc
18   http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd
19  ">
20
21      <mvc:annotation-driven/>
22
23      <context:component-scan base-package="pl.jsystems.springmvc.controller"/>
24
25
26  </beans>
27
```

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.

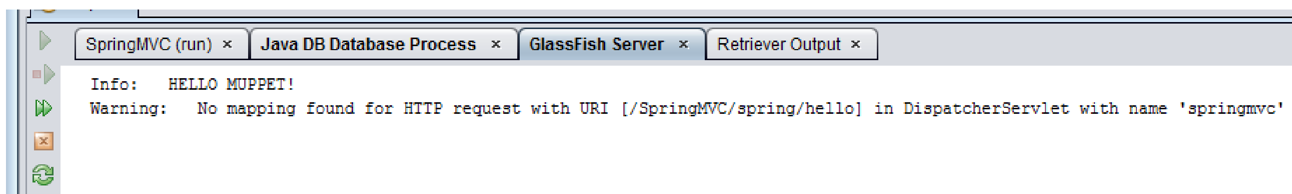
```
6    package pl.jsystems.springmvc.controller;
7
8    import org.springframework.stereotype.Controller;
9    import org.springframework.ui.Model;
10   import org.springframework.web.bind.annotation.RequestMapping;
11
12   /**
13    *
14    * @author andrzej
15    */
16   @Controller
17   public class 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.

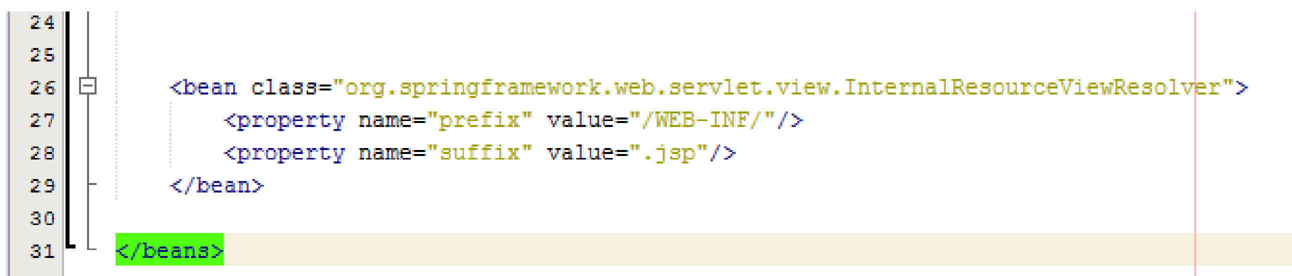
Wywołajmy więc ten adres:



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:

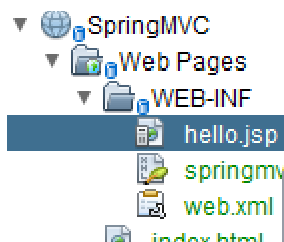


Wyświetlił nam się komunikat „Hello Muppet”, a to oznacza że nasza metoda sayHello została zgodnie z założeniem wywołana. Błąd 404 pojawia się dlatego, że nie mam strony JSP którą powinienem w odpowiedzi wyświetlić. Konkretnie to plik powinien się nazywać hello.jsp – ponieważ metoda sayHello zwraca ciąg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



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/

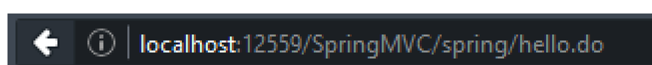
Stwórzmy jeszcze w zadeklarowanym katalogu plik hello.jsp:



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!

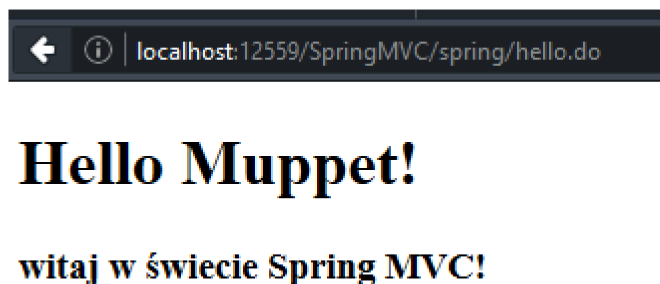
Przydałoby się jednak zrobić coś więcej niż wyświetlanie statycznej wartości. Przekażmy więc jakieś dane z kontrolera do widoku (patrz linia 23):

```
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!";
23          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>
16          <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:



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
```

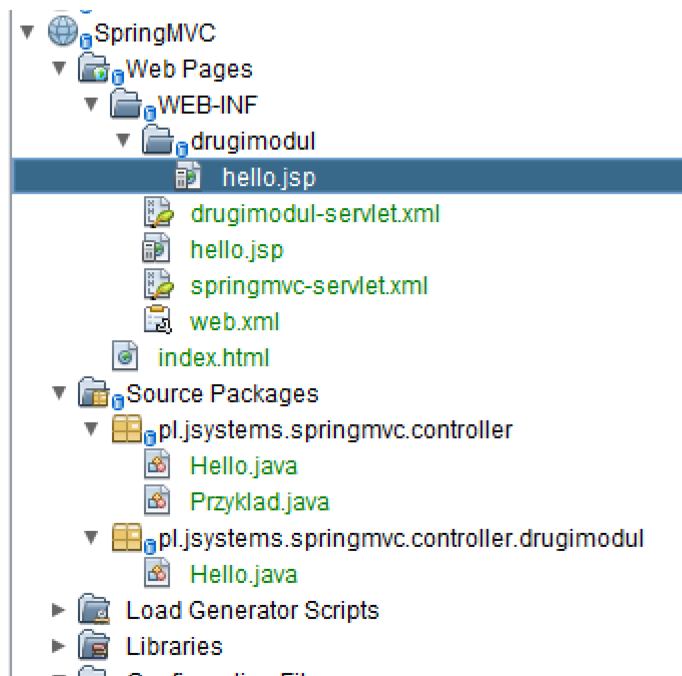
W związku z tym że nasz serwlet nazywa się drugimodul dodajemy też plik „drugimodul-servlet.xml” . W nim dodajemy takie wpisy jak poprzednio, z tą różnicą że pliki JSP znajdą się w osobnym podkatalogu w WEB-INFie (patrz linia 27), a dodatkowo wydzielimy sobie osobny pakiet na kontrolery tego modułu.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

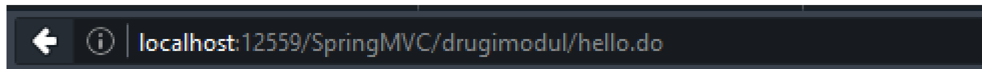
```
5    */
6   package pl.jsystems.springmvc.controller.drugimodul;
7
8   import pl.jsystems.springmvc.controller.*;
9   import org.springframework.stereotype.Controller;
10  import org.springframework.ui.Model;
11  import org.springframework.web.bind.annotation.RequestMapping;
12
13  /**
14   *
15   * @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:



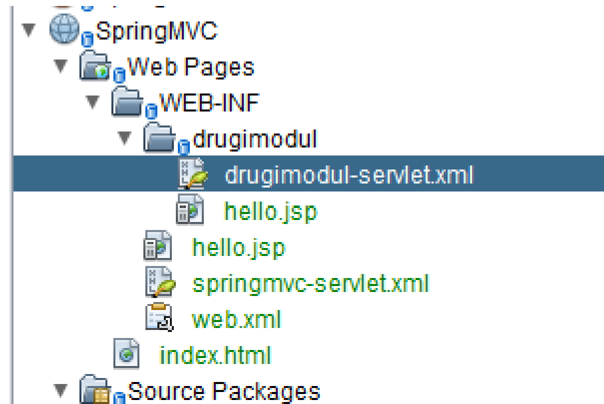
Hello Muppet w drugim module!

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
```

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:

```
6    package pl.jsystems.springmvc.dao;
7
8    import java.util.ArrayList;
9    import java.util.List;
10   import pl.jsystems.springmvc.domain.Samochod;
11
12   /**
13    *
14    * @author andrzej
15    */
16   public class SamochodDao {
17
18       public Samochod getOne() {
19           Samochod s = new Samochod("Polonez", "Borewicz", "OMG 12345", "czerwony");
20           return s;
21       }
22
23       public List<Samochod> getAll(){
             List<Samochod> fury = new ArrayList<Samochod>();
25               fury.add(new Samochod("Audi", "A4", "WIF 98765", "czarny"));
26               fury.add(new Samochod("BMW", "e61", "LLU 112112", "grafitowy"));
27               fury.add(new Samochod("Mercedes", "SL", "WOW 00000", "biały"));
28               fury.add(new Samochod("Solaris", "przegubowiec", "WR 12345", "żółto-czerwony"));
29           return fury;
30       }
31
32   }
```

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 :)

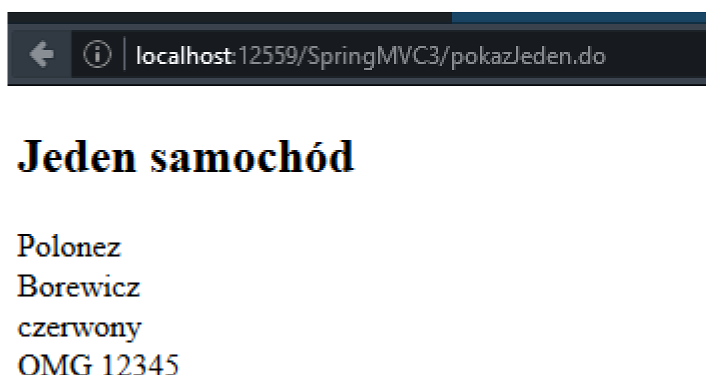
Nowy element jaki się tutaj pojawia to model.addAttribute z przekazaniem obiektu, lub w drugim kontrolerze listy obiektów. Wcześniej przekazywaliśmy wyłącznie komunikat tekstowy. Właściwie możemy tutaj przekazać cokolwiek, różnica będzie jedynie w dostępie do tych danych na poziomie widoku. Pamiętać 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:

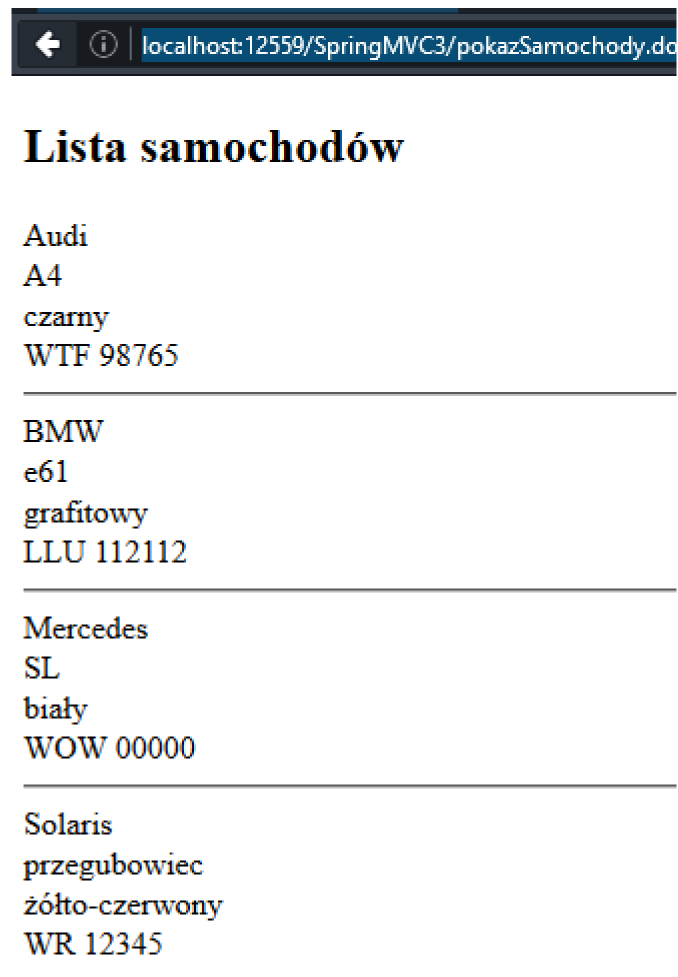


Poniżej kod pliku JSP odpowiedzialnego za wyświetlenie listy samochodów:

```
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
```

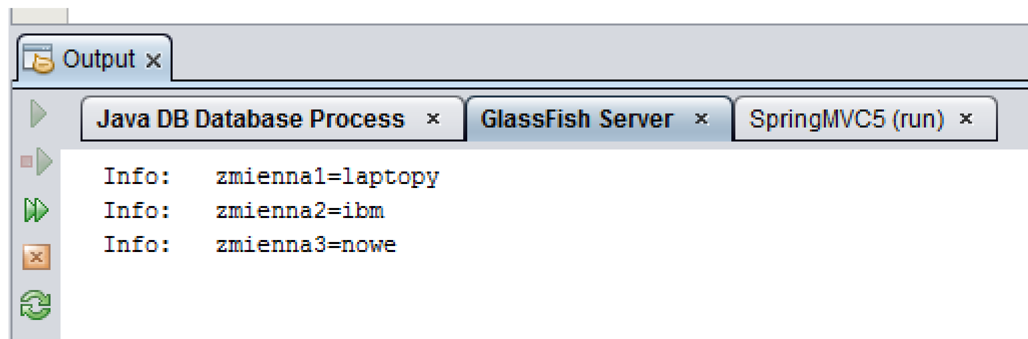
Inny przykład:

```
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
```

Proste, wygodne, skuteczne :)

Parametry żądania

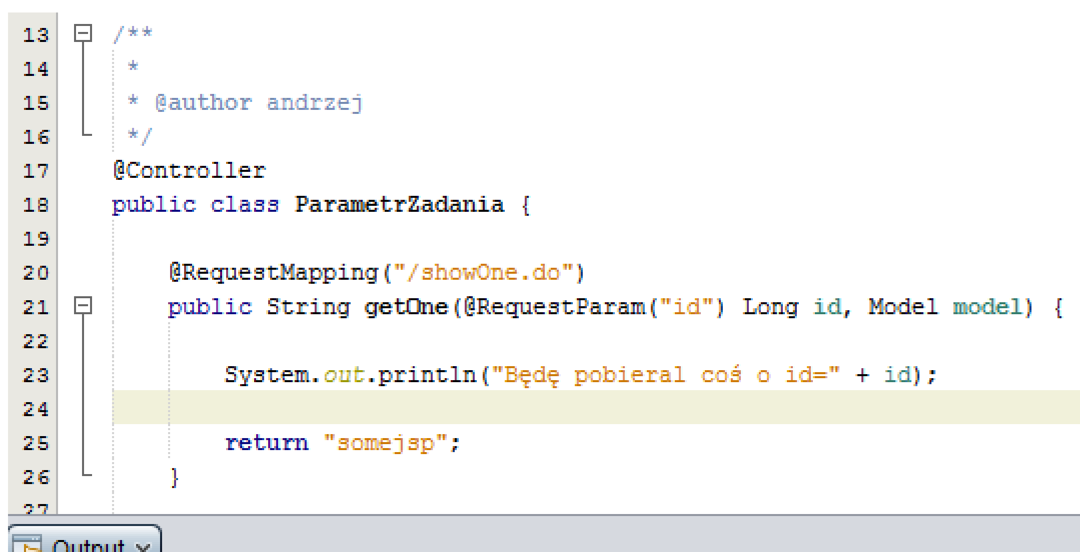
Kod źródłowy z przykładami do tego rozdziału możesz pobrać pod adresem:

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

Parametry żądania to parametry przekazywane np. W ten sposób:

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

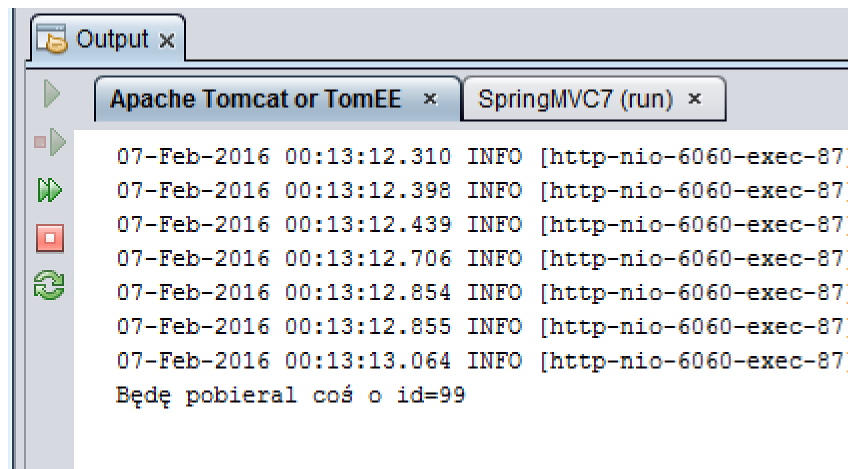
Można je wykorzystać np do filtrowania danych, wyświetlenia szczegółów produktu. Z jednej strony przedstawiona poniżej metoda jest lepsza od zmiennych tablicowych – ponieważ jest znacznie mniej roboty, z drugiej jest gorsza o tyle że tutaj podanie wartości parametru jest obligatoryjne. Nie możemy go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



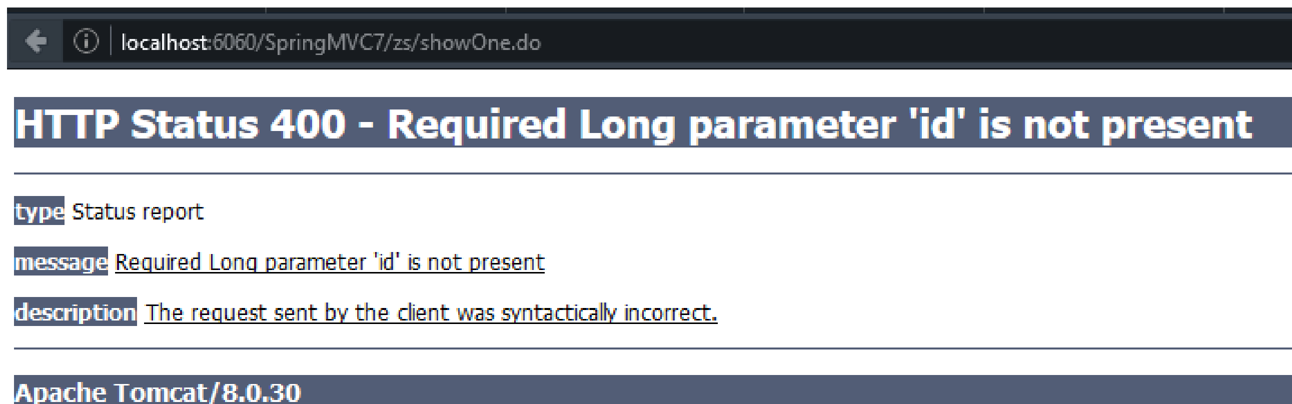
Pokawia się tutaj jedna nowa rzecz: @RequestParam. W parametrze adnotacji podaję nazwę parametru w pasku , którego wartość następnie trafia do mojej zmiennej id typu Long. Wywołanie ma postać:

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
```

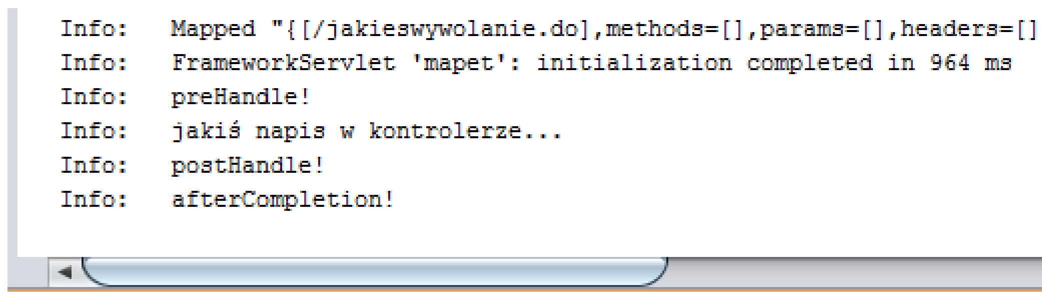
Zawartość pliku *****-servlet została wzbogacona o fragment z tagami <mvc:interceptors>:

```
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:



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.

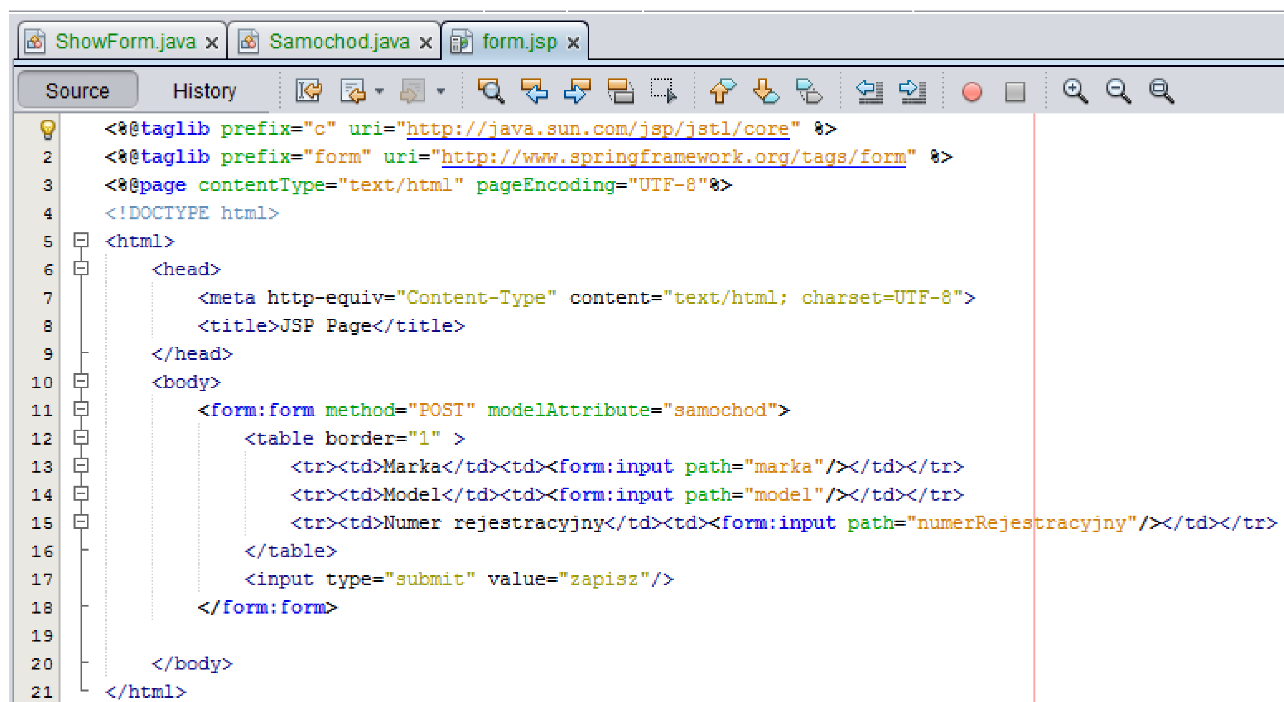
Przejdźmy teraz do klasy kontrolera:

```java
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:

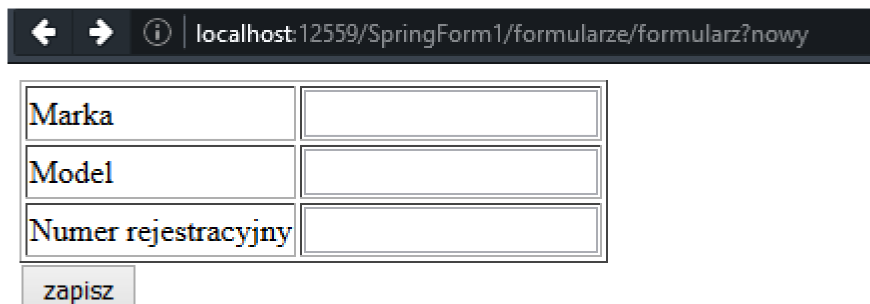


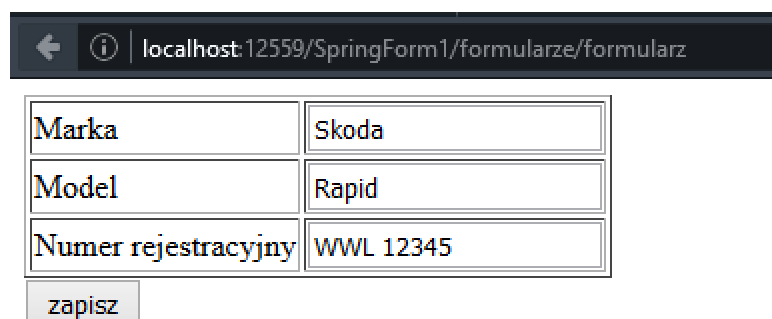
```
ShowForm.java  Samochod.java  form.jsp
Source  History
1  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2  <%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
3  <%@page contentType="text/html" pageEncoding="UTF-8"%>
4  <!DOCTYPE html>
5  <html>
6      <head>
7          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8          <title>JSP Page</title>
9      </head>
10     <body>
11         <form:form method="POST" modelAttribute="samochod">
12             <table border="1" >
13                 <tr><td>Marka</td><td><form:input path="marka"/></td></tr>
14                 <tr><td>Model</td><td><form:input path="model"/></td></tr>
15                 <tr><td>Numer rejestracyjny</td><td><form:input path="numerRejestracyjny"/></td></tr>
16             </table>
17             <input type="submit" value="zapisz"/>
18         </form:form>
19
20     </body>
21 </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)
        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:

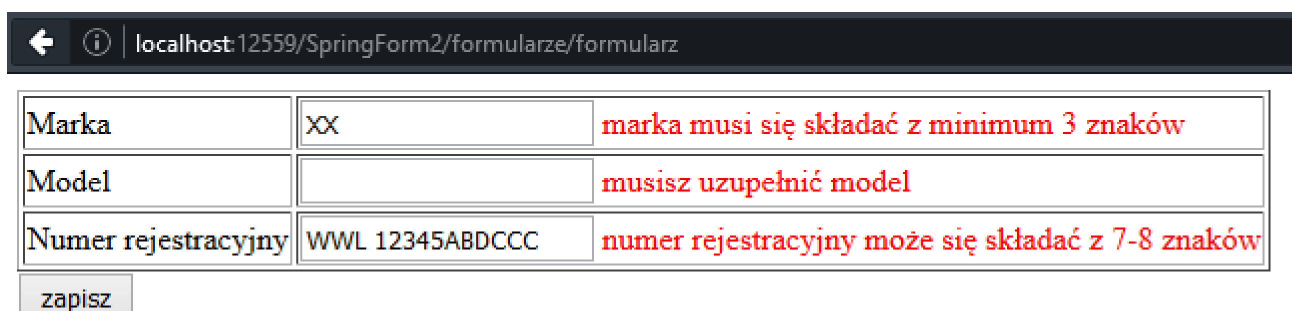
<form:errors path="marka"/>

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)

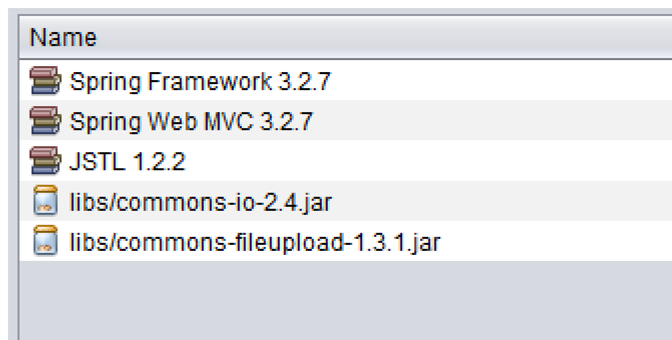
Sposób działania walidacji po próbie zatwierdzenia nie poprawnie wypełnionego formularza:



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:

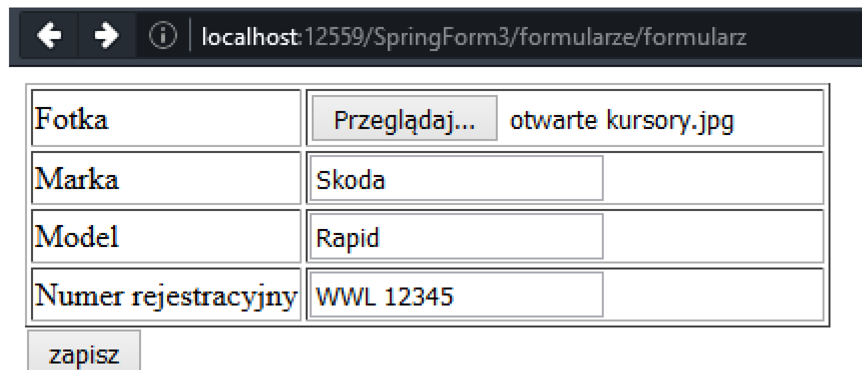
```
10      <body>
11          <form:form method="POST" modelAttribute="samochod"  enctype="multipart/form-data">
12              <table border="1" >
13
14                  <tr><td>Fotka</td><td><input name="fotka"  type="file" />
15                  </td></tr>
16
17                  <tr><td>Marka</td><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());
            } 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: