

Laboratorium 4. Wprowadzenie do *Spring MVC*

Cel zajęć

Realizacja zadań proponowanych w niniejszym laboratorium pozwoli studentom poznać podstawowe zasady tworzenia aplikacji internetowych z wykorzystaniem szkieletu programistycznego *Spring MVC* [6, 19, 25], na przykładzie aplikacji typu *CRUD* do zarządzania pracownikami, przechowywanymi w bazie *MySQL*. Zadania opracowano na podstawie [17]:

<https://www.javatpoint.com/spring-mvc-crud-example>

Zakres tematyczny

- Przygotowanie aplikacji internetowej typu *CRUD* z wykorzystaniem szkieletu programistycznego *Spring MVC*.
- Przygotowanie bazy danych *MySQL* do obsługi pracowników.
- Konfiguracja projektu *Spring MVC*.
- Implementacja kontrolera i komponentów danych z wykorzystaniem modułów *Spring MVC*.
- Przygotowanie stron widoków w *JSP*.
- Wykonywanie operacji na danych (*CREATE*, *READ*, *UPDATE* i *DELETE*) z wykorzystaniem biblioteki *JdbcTemplate*.
- Implementacja obsługi wyjątków w klasie kontrolera.

Wprowadzenie

Spring jest wielowarstwowym, modułowym, najpopularniejszym obecnie szkieletem programistycznym dla języka *Java* [6, 24]. Projekt w *Spring* tworzony jest z perspektywy aplikacji, a nie serwera. *Spring* promuje dobre praktyki programowania obiektowego i jest w pełni modułowy (można wykorzystać dowolną część *Spring*, niezależnie od pozostałych). Podstawowe moduły *Spring* to *Core* i *Beans*. Moduły te zapewniają podstawowe części struktury projektu.


Zasadniczym elementem projektu *Spring* są zwykle klasy *POJO* (ang. *Plain Old Java Object*), czyli obiekty aplikacji, zarządzane poprzez kontener *Spring Core*, nazywane ziarnami (ang. *Beans*). Ziarno *Spring Bean* jest obiektem, który jest instancjonowany, montowany i zarządzany przez kontener. *Spring* nie nakłada na klasy *POJO* żadnych dodatkowych wymagań dotyczących dziedziczenia lub implementowania interfejsów. Kontener podstawowy *Spring Core Container* czyta metadane konfiguracyjne, które mogą być reprezentowane przez pliki

XML, adnotacje w klasach *Java* lub bezpośrednio w kodzie. Kontener *Spring* korzysta z klas *POJO* i metadanych konfiguracyjnych, do utworzenia gotowej aplikacji. Metadane konfiguracyjne określają obiekty, które zawierają aplikację oraz współzależności między tymi obiektami. Kontener *Spring Core* czyta metadane i na tej podstawie określa, które obiekty mają być instancjonowane, konfigurowane oraz montowane. Następnie kontener wstrzykuje zależności (ang. *dependency injection* – *DI*) podczas tworzenia ziarna.

Spring Web MVC to moduł *Spring* ukierunkowana na tworzenie aplikacji sieciowych, oparty na architekturze wzorca *MVC*. Dostarcza komponenty do tworzenia elastycznych i luźno połączonych aplikacji internetowych. Model łączy dane aplikacji, które definiowane są za pomocą klas *POJO*. Widok jest odpowiedzialny za renderowanie danych modelu i generowanie wyjściowego *HTML*. Kontroler jest odpowiedzialny za przetwarzanie żądań użytkownika, budowę odpowiedniego modelu i przekazywanie go do renderowania przez widok. Moduł *Web* zapewnia podstawowe funkcje zorientowane na sieć, inicjalizuje kontener *Spring* z wykorzystaniem *servletów* i kontekstu aplikacji webowej.

Zadanie 4.1. Przygotowanie bazy danych *MySQL*

Na serwerze *MySQL* w bazie danych o nazwie *test* utwórz tabelę *pracownik* (Rys. 4.1), korzystając z kodu *SQL* z przykładu 4.1.

#	Nazwa	Typ	Metoda porównywania napisów	Atrybuty	Null	Ustawienia domyślne
1	id 	int(11)		UNSIGNED	Nie	Brak
2	nazwisko	varchar(50)	utf8_general_ci		Nie	Brak
3	pensja	double		UNSIGNED	Nie	0
4	firma	varchar(400)	utf8_general_ci		Nie	Brak

Rys. 4.1. Tabela *pracownik* w bazie *test*

Przykład 4.1. Kod *SQL* do utworzenia tabeli *pracownik*

```
--
-- Struktura tabeli `pracownik`
--
```

```
CREATE TABLE `pracownik` (  
  `id` int(11) UNSIGNED NOT NULL AUTO_INCREMENT,  
  `nazwisko` varchar(50) NOT NULL,  
  `pensja` double UNSIGNED NOT NULL DEFAULT '0',  
  `firma` varchar(400) NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Zadanie 4.2. Utworzenie projektu *Spring MVC*

Utwórz nowy projekt aplikacji webowej o nazwie *pai_spring* (*File* → *New Project* → *Java with Maven/Web Application*). W wygenerowanym pliku konfiguracyjnym *pom.xml* zmień wersję *JEE* 6 na *JEE* 7 (jeśli trzeba):

```
<dependency>  
  <groupId>javax</groupId>  
  <artifactId>javaee-web-api</artifactId>  
  <version>7.0</version>  
</dependency>
```

W elemencie `<build>` dla elementu `<plugin>` zmodyfikuj element `<configuration>`:

```
<plugin>  
  <groupId>org.apache.maven.plugins</groupId>  
  <artifactId>maven-compiler-plugin</artifactId>  
  <version>2.3.2</version>  
  <configuration>  
    <source>1.7</source>  
    <target>1.7</target>  
    <compilerArguments>  
      <endorseddirs>${endorsed.dir}</endorseddirs>  
    </compilerArguments>  
  </configuration>  
</plugin>
```

W kolejnym elemencie `<plugin>` (jeśli trzeba) zmień wersję *javaee-endorsed-api* z 6.0 na 7.0:

```
<artifactItem>  
  <groupId>javax</groupId>  
  <artifactId>javaee-endorsed-api</artifactId>  
  <version>7.0</version>  
  <type>jar</type>  
</artifactItem>
```

Następnie dodaj do pliku **pom.xml** kolejne zależności (bibliotekę *spring-webmvc*, sterownik *mysql-connector-java* i *spring-jdbc* do pracy z *MySQL* oraz bibliotekę znaczników *jstl*) (Przykład 4.2).

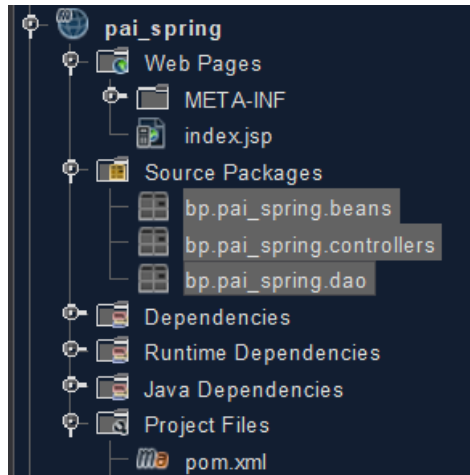
Przykład 4.2. Dodatkowe zależności w pliku pom.xml

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-  
webmvc -->  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-webmvc</artifactId>  
    <version>5.1.1.RELEASE</version>  
</dependency>  
  
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->  
<dependency>  
    <groupId>javax.servlet</groupId>  
    <artifactId>jstl</artifactId>  
    <version>1.2</version>  
</dependency>  
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->  
<dependency>  
    <groupId>mysql</groupId>  
    <artifactId>mysql-connector-java</artifactId>  
    <version>8.0.11</version>  
</dependency>  
<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc  
-->  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-jdbc</artifactId>  
    <version>5.1.1.RELEASE</version>  
</dependency>
```

Po tych modyfikacjach zbuduj projekt: *Run*→*Clean and Build Main Project*.

Następnie w pakiecie (tutaj o nazwie **bp.pai_spring**) utwórz 3 kolejne pakiety (*New* → *Java package*) o nazwach: **beans**, **controllers** i **dao** tak, jak przedstawiono na rysunku 4.2. Przy tworzeniu zwróć uwagę, żeby pakiety te były **bezpośrednimi podpakietami** pakietu projektu (np. **bp.pai_spring**).

Do utworzenia funkcjonalnej aplikacji webowej typu *CRUD* należy wykonać kilka podstawowych kroków, pokazanych w punktach 4.2.1–4.2.5.



Rys. 4.2. Struktura projektu z dodanymi pakietami

4.2.1. Klasa *Pracownik* jako *POJO*

Do pakietu *beans* dodaj definicję klasy o nazwie *Pracownik*, której obiekt będzie odwzorowany na rekord w tabeli *pracownik* (Przykład 4.3). Do klasy dodaj odpowiednie metody publiczne *get* i *set*.

Przykład 4.3. Definicja klasy Pracownik.java

```
package bp.pai_spring.beans;

public class Pracownik {
    private int id;
    private String nazwisko;
    private float pensja;
    private String firma;
    //dodaj metody get i set
}
```

4.2.2. Klasa *PracownikDao* do pracy z danymi

W pakiecie *dao* utwórz klasę *PracownikDao* (Przykład 4.4), której metody, korzystają z interfejsu *JdbcTemplate* i pomogą wykonać zapytania *SQL* do bazy danych *test* z tabelą *pracownik*.

Przykład 4.4. Definicja klasy PracownikDao.java

```
package bp.pai_spring.dao;

import bp.pai_spring.beans.Pracownik;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```

import java.util.List;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
public class PracownikDao {
    JdbcTemplate template;

    public void setTemplate(JdbcTemplate template) {
        this.template = template; //wstrzyknięcie przez metodę set
    }
    public int save(Pracownik p) {
        String sql = "insert into pracownik (nazwisko,pensja,firma) "
            + "values('" + p.getNazwisko() + "','" + p.get...()
            + "','" + p.get...() + "')";
        return template.update(sql);
    }
    public List<Pracownik> getAll() {
        return template.query("select * from pracownik",
            new RowMapper<Pracownik>() {
                @Override
                public Pracownik mapRow(ResultSet rs, int row)
                    throws SQLException{
                    Pracownik e = new Pracownik();
                    e.setId(rs.getInt(1));
                    e.setNazwisko(rs.getString(2));
                    // ustaw właściwość pensja
                    // ustaw właściwość firma
                    return e;
                }
            });
    }
}

```

4.2.3. Klasa kontrolera *PracownikController*

W pakiecie *controllers* utwórz klasę o nazwie *PracownikController* (Przykład 4.5), która będzie pełniła rolę kontrolera. Metody tej klasy (**akcje kontrolera**) będą obsługiwały odpowiednie żądania *HTTP*, wskazane w parametrach adnotacji *@RequestMapping*.

Przykład 4.5. Definicja klasy PracownikController.java

```

package bp.pai_spring.controllers;

import bp.pai_spring.beans.Pracownik;
import bp.pai_spring.dao.PracownikDao;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

```

```
import org.springframework.web.bind.annotation.RequestMethod;

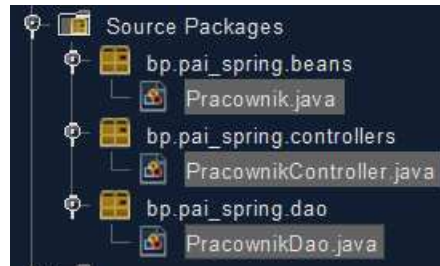
@Controller
public class PracownikController {
    @Autowired
    PracownikDao dao; //wstrzyknięcie dao z pliku XML

    /* Wynikiem działania metody jest przekazanie danych w modelu do
    * strony widoku addForm.jsp, która wyświetla formularz
    * wprowadzania danych, a „command” jest zastrzeżonym atrybutem
    * żądania, umożliwiającym wyświetlenie danych obiektu pracownika
    * w formularzu.
    */
    @RequestMapping("/addForm")
    public String showform(Model m){
        m.addAttribute("command", new Pracownik());
        return "addForm"; //przekierowanie do addForm.jsp
    }

    /* Metoda obsługuje zapis pracownika do BD. @ModelAttribute
    * umożliwia pobranie danych z żądania do obiektu pracownika.
    * Jawnie wskazano RequestMethod.POST (domyślnie jest to GET)
    */
    @RequestMapping(value="/save",method =
        RequestMethod.POST)
    public String save(@ModelAttribute("pr") Pracownik pr){
        dao.save(pr);
        return "redirect:/viewAll";
        //przekierowanie do endpointa o URL: /viewAll
    }

    /* Metoda pobiera listę pracowników z BD i umieszcza je w modelu */
    @RequestMapping("/viewAll")
    public String viewAll(Model m){
        List<Pracownik> list=dao.getAll();
        m.addAttribute("list",list);
        return "viewAll"; //przejdźcie do widoku viewAll.jsp
    }
}
```

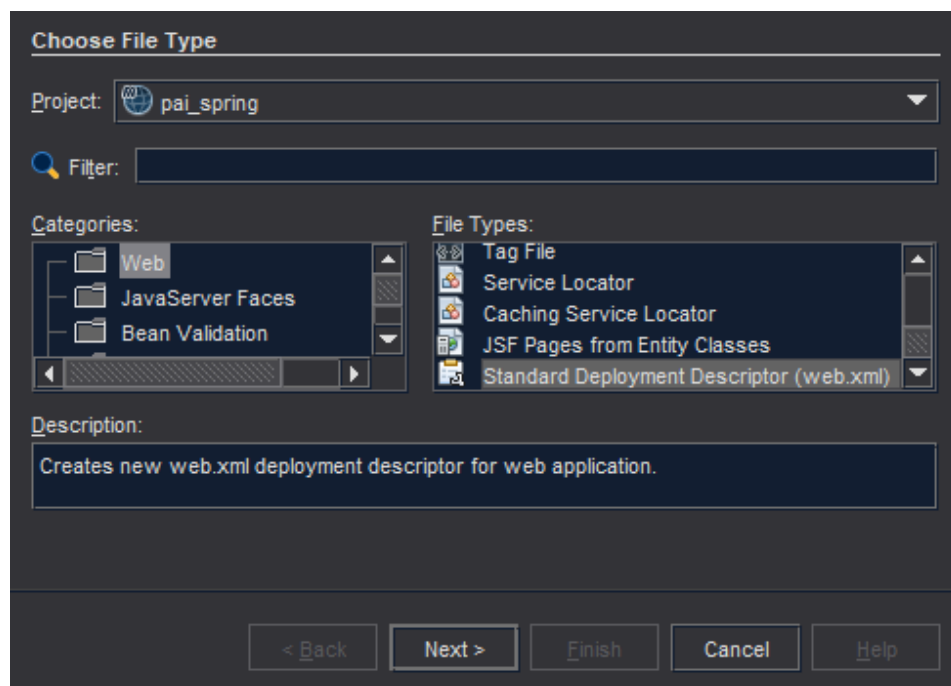
Struktura plików w projekcie po wykonaniu kroków z podrozdziałów 4.2.1–4.2.3 przedstawiona jest na rysunku 4.3.



Rys. 4.3. Pakiety z klasami *Pracownik*, *PracownikDao* i *PracownikController*

4.2.4. Konfiguracja w plikach *web.xml* i *spring-servlet.xml*

W folderze **WEB-INF** projektu utwórz (jeśli jeszcze nie istnieje) standardowy plik **web.xml** (deskryptor wdrożenia – Rys. 4.4) i dodaj do niego konfigurację punktu wejścia aplikacji – serwletu o nazwie **spring**, jak pokazuje przykład 4.6 (kolorem zielonym oznaczono dodany fragment kodu).



Rys. 4.4. Dodanie deskryptora wdrożenia *web.xml* w *NetBeans IDE*

Przykład 4.6. Uzupełniony plik *web.xml*

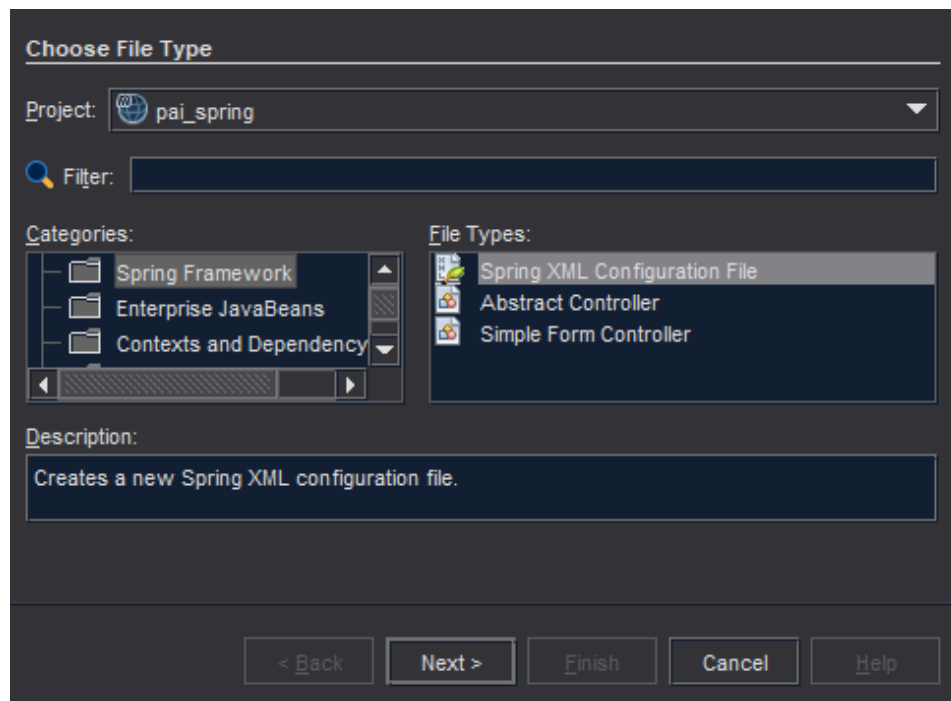
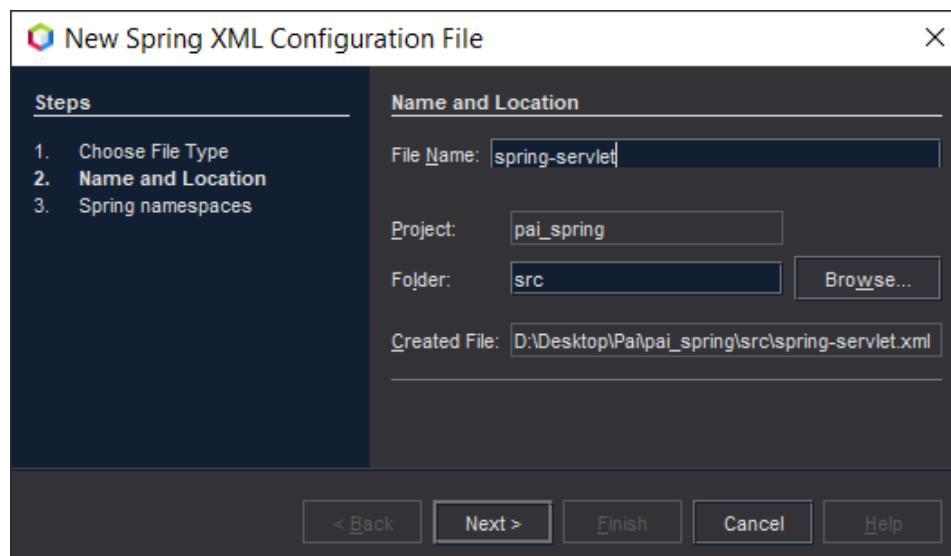
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
```



```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
version="3.1">
<session-config>
  <session-timeout>
    30
  </session-timeout>
</session-config>
<display-name>SpringMVC</display-name>
<servlet>
  <servlet-name>spring</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>spring</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

W tym samym folderze utwórz także plik konfiguracyjny *spring-servlet.xml* serwletu (Rys. 4.5 i 4.6), w którym należy dodać definicję ziaren (bean).

Przykład 4.7 przedstawia prawidłowy, gotowy plik konfiguracyjny. Zwróć uwagę na właściwe wskazanie nazw pakietów w projekcie oraz na konfigurację ziarna z połączeniem do bazy danych (Rys. 4.5 i 4.6).

Rys. 4.5. Dodanie pliku konfiguracyjnego *spring-servlet.xml* w NetBeansRys. 4.6. Ustawienia pliku konfiguracyjnego *spring-servlet.xml*

Przykład 4.7. Konfiguracja ziaren w pliku *spring-servlet.xml*

```

<?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:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
  <context:component-scan base-package="bp.pai.spring.controllers">
  </context:component-scan>
  <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
">
    <property name="prefix" value="/WEB-INF/jsp/"></property>
    <property name="suffix" value=".jsp"></property>
  </bean>

  <bean id="ds"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
value="com.mysql.cj.jdbc.Driver"></property>
    <property name="url"
value="jdbc:mysql://localhost:3306/test?serverTimezone=UTC"> </property>
    <property name="username" value="root"></property>
    <property name="password" value=""></property>
  </bean>
  <bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="ds"></property>
  </bean>

  <bean id="dao" class="bp.pai.spring.dao.PracownikDao">
    <property name="template" ref="jt"></property>
  </bean>
</beans>

```

4.2.5. Widoki JSP

Jako stronę startową aplikacji wykorzystaj istniejący (lub utwórz jeśli nie istnieje) plik *index.jsp* z zawartością jak podano w przykładzie 4.8. Zwróć uwagę, że adresy w hiperłączach są mapowane na odpowiednie metody kontrolera *PracownikController*. Lokalizacja pliku *index.jsp* pokazana jest na rysunku 4.7.

Przykład 4.8. Strona *index.jsp*

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>

```

```

<head>
  <meta charset="UTF-8">
  <title>Pracownicy</title>
</head>
<body>
  <div>
    <h3>Pracownicy</h3>
    <p>
      <a href="addForm">Dodaj pracownika</a><br />
      <a href="viewAll">Pokaż listę pracowników</a>
    </p>
  </div>
</body>
</html>

```

Do projektu dodaj 2 dodatkowe pliki widoków *JSP* (w lokalizacji jak na rysunku 4.7):

- ***viewAll.jsp*** – tabela z listą pracowników (Przykład 4.9),
- ***addForm.jsp*** – strona z formularzem dodawania nowego pracownika (Przykład 4.10).

Strony *JSP* korzystają z języka wyrażeń *JSTL* oraz z bibliotek znaczników *form* i *core*, które pozwoliły w prosty sposób wyświetlić dane przekazane z kontrolera do widoków w atrybutach modelu, dostępnych przez nazwę klucza. Klucze i wartości atrybutów zostały ustawione w odpowiednich akcjach kontrolera ***PracownikController***: ***showform()*** i ***viewAll()***. W kodzie strony ***viewAll.jsp***, przy każdym wierszu zostały już dodane hiperłącza do wykonania kolejnych akcji: edycji i usunięcia wskazanego przez *id* obiektu (**do uzupełnienia w kolejnych zadaniach**).

Przykład 4.9. Strona *viewAll.jsp*

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://www.springframework.org/tags/form"
prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Pracownicy</title>
  </head>
  <body>
    <div>
      <h1>Lista pracowników</h1>
      <table border>
        <tr> <th>Id</th> <th>Nazwisko</th> <th>Pensja</th> <th>Firma</th>
          <th>Edytuj</th> <th>Usuń</th>

```

```

</tr>
<c:forEach var="pr" items="${list}">
  <tr>
    <td> ${pr.id} </td>
    <td> ${pr.nazwisko} </td>
    <td> ${pr.pensja} </td>
    <td> ${pr.firma} </td>
    <td><a href="edit/${pr.id}"> Edytuj </a></td>
    <td><a href="..."> Usuń </a></td>
  </tr>
</c:forEach>
</table>
<br/>
<a href="addForm">Dodaj nowego pracownika</a>
</div>
</body>
</html>

```

Przykład 4.10. Strona addForm.jsp

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://www.springframework.org/tags/form"
  prefix="form"%>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Pracownicy</title>
  </head>
  <body>
    <div>
      <h1>Dodaj dane nowego pracownika</h1>
      <form:form method="post" action="save">
        <table >
          <tr>
            <td>Nazwisko : </td>
            <td> <form:input path="nazwisko" /> </td>
          </tr>
          <tr>
            <td>Pensja :</td>
            <td> <form:input path="pensja" /> </td>
          </tr>
          <tr>
            <td>Firma :</td>
            <td> <form:input path="firma" /> </td>
          </tr>
          <tr>
            <td> </td>
            <td> <input type="submit" value="Zapisz" /> </td>
          </tr>
        </table>
      </form>
    </div>
  </body>
</html>

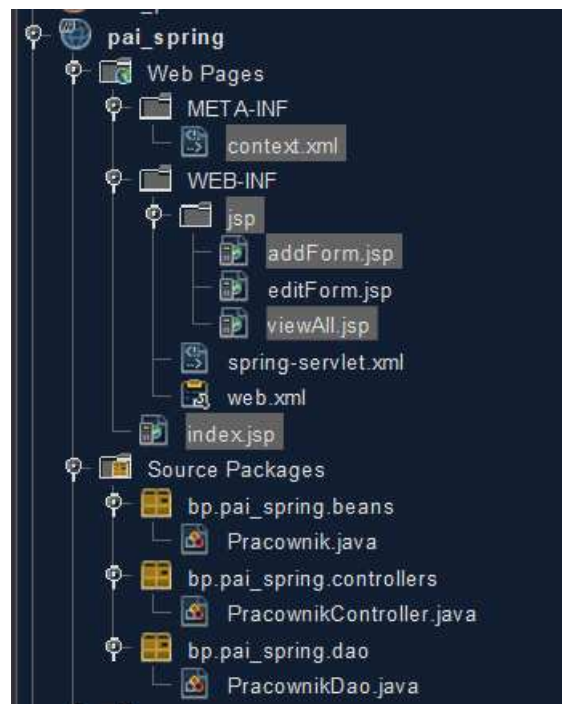
```

```
</form:form>
</div>
</body>
</html>
```

UWAGA! Jeśli w projekcie nie istnieje folder **META-INF**, to go utwórz (w tej samej lokalizacji co folder **WEB-INF**), a w nim umieść plik **context.xml** z zawartością:

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/pai_spring"/>
```

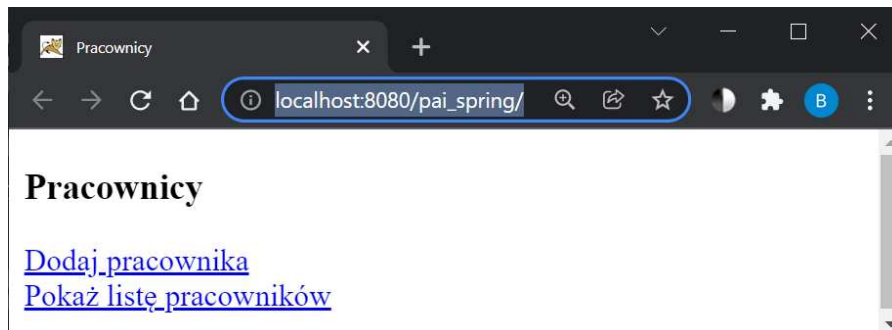
Struktura plików w gotowym projekcie pokazana jest na rysunku 4.7.



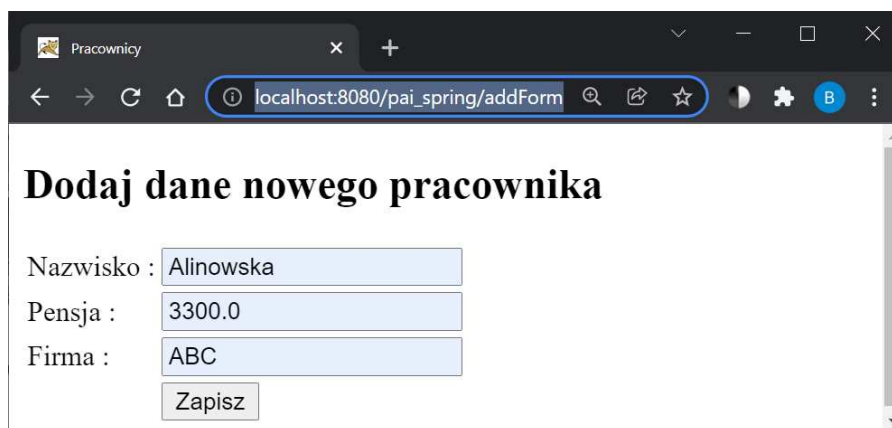
Rys. 4.7. Rozmieszczenie plików gotowego projektu

Zadanie 4.3. Zbudowanie i uruchomienie projektu

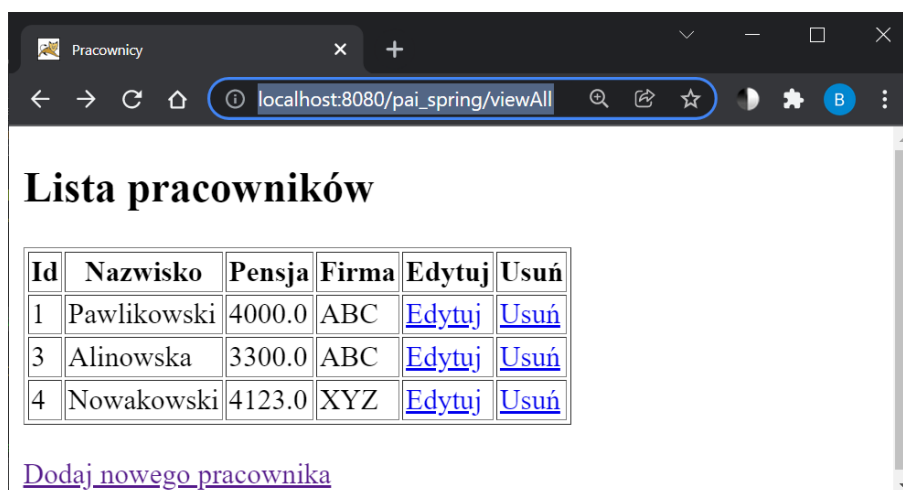
Zbuduj i uruchom projekt (*Run*→*Build Main Project*). Jeśli wszystko zostało poprawnie zrealizowane, to po uruchomieniu projektu w przeglądarce powinna pokazać się strona **index.jsp** (Rys. 4.8). Sprawdź efekt działania hiperłączy (Rys. 4.9) i poprawność pracy z bazą danych. Przeanalizuj kod klasy kontrolera i zwróć szczególną uwagę na jego metody obsługujące poszczególne żądania.



Rys. 4.8. Strona startowa projektu



Rys. 4.9. Formularz dodawania nowego pracownika



Rys. 4.10. Widok listy pracowników

Zadanie 4.4. Akcja kontrolera *DELETE*

Zaimplementuj możliwość usuwania wskazanego pracownika. W widoku listy pracowników istnieje już odpowiedni link do akcji kontrolera (sprawdź jaki).

W celu implementacji usunięcia pracownika:

- do klasy *PracownikDao* dodaj metodę *delete* z atrybutem *int id*,
- do klasy *PracownikController* dodaj metodę *delete*, która obsłuży żądanie */delete/{id}*. Nagłówek tej metody powinien mieć postać:
`public String delete(@PathVariable int id) {}`, gdzie *@PathVariable* umożliwia pobranie parametru przekazanego w ścieżce z adresem URL żądania.

Przetestuj działanie akcji usunięcia pracownika.

Zadanie 4.5. Akcja kontrolera *EDIT*

Zaimplementuj możliwość edycji danych wskazanego pracownika. W widoku listy pracowników istnieje już odpowiedni link do akcji kontrolera (sprawdź jaki).

Wykonaj podobne operacje jak w przypadku dodawania nowego pracownika do bazy:

- Do klasy *PracownikDao* dodaj metodę *update* (analogiczną do metody *save*, tylko z zapytaniem *update*).
- Do klasy *PracownikDao* dodaj metodę *getPracownikById()*:

```
public Pracownik getPracownikById(int id){  
    String sql="select * from pracownik where id=?";  
    return  
        template.queryForObject(sql, new Object[]{id},  
            new BeanPropertyRowMapper<>(Pracownik.class));  
}
```
- Do klasy *PracownikController* dodaj metodę *edit*, która obsłuży żądanie przesłane metodą *GET /edit/{id}* i przekaże do widoku *editForm* obiekt pracownika o podanym *id*.
- Do klasy *PracownikController* dodaj metodę *editsave*, która obsłuży przychodzące żądanie ze zmodyfikowanymi danymi pracownika, przesłane z formularza *editForm* metodą *POST* (sprawdź, jak działa metoda *save*).
- Utwórz stronę widoku *editForm.jsp* do edycji danych pracownika, analogicznie jak w *addForm.jsp*, tylko dodatkowo do formularza dodaj ukryte pole do przekazania *id*:
`<form:hidden path="id" />`

Przetestuj działanie akcji edycji danych pracownika.

Zadanie 4.6. Obsługa wyjątków w kontrolerze

W poprzednich zadaniach nie było jeszcze kontroli poprawności danych przesyłanych w żądaniu z formularza. Sprawdź, jaki będzie efekt po próbie zapisania danych pracownika z pensją wpisaną jako łańcuch, np. "abc". W takim przypadku nastąpi problem związany z pojawieniem się niekontrolowanego wyjątku.

Aby tego uniknąć, przygotuj strony widoków do obsługi błędów (umieść je w folderze *jsp* razem z istniejącymi już tam widokami) oraz w kontrolerze dodaj specjalną metodę, poprzedzoną adnotacją *@ExceptionHandler*.

W zależności od potrzeby, metoda taka może mieć postać prezentowaną w przykładzie 4.11 lub 4.12.

Przykład 4.11. Prosta obsługa wyjątku przez metodę kontrolera

```
//Metoda zwraca nazwę widoku, wykorzystanego do pokazania komunikatu
//o błędzie (sam obiekt Exception nie jest dostępny w widoku)
@ExceptionHandler({Exception.class}) //tu można wymienić wyjątki
public String error() {
    return "errorpage";
}
```

Przykład 4.12. Obsługa błędu z przekazaniem modelu do widoku

```
//Totalna kontrola - ustawienie danych o błędzie w modelu oraz
//zwrócenie nazwy widoku i modelu w obiekcie ModelAndView
@ExceptionHandler(Exception.class)
public ModelAndView handleError(HttpServletRequest req,
                                Exception ex) {
    ModelAndView mav = new ModelAndView();
    mav.addObject("exception", ex);
    mav.addObject("url", req.getRequestURL());
    mav.setViewName("error");
    return mav;
}
```

Przykład 4.13 przedstawia stronę widoku *error.jsp*, która pobiera dane przekazane w metodzie kontrolera z przykładu 4.12 w obiekcie *ModelAndView*.

Przykład 4.13. Strona error.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
```

```
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" >
  <title>JSP Page</title>
</head>
<body>
  <h1>Error Page</h1>
  <p>Failed URL: ${url}
    Exception: ${exception.message}
    <c:forEach items="${exception.stackTrace}" var="ste">
      ${ste}
    </c:forEach>
  </p>
</body>
</html>
```