ЛАБОРАТОРНАЯ РАБОТА №8

РАЗРАБОТКА ЭФФЕКТИВНЫХ ВЕБ-ПРИЛОЖЕНИЙ (SPRING MVC)

Цель работы: разработка веб-приложений с реализацией шаблона MVC.

Рекомендуемое программное обеспечение: IntelliJ IDEA, JDK 1.8+, Spring.

Необходимая теоретическая подготовка:

- объектно-ориентированное программирование;
- Maven;
- шаблоны проектирования.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

https://www.youtube.com/playlist?list=PLAma_mKffTOR5o0WNHnY0mTjKxnCgSXrZ

https://habr.com/ru/post/336816/

https://spring-projects.ru/guides/serving-web-content/

Задание

- создать проект Spring MVC и подключить к нему все необходимые зависимости (2 часа).
- создать html формы добавления параметров объектов в проект. Поля, которые впоследствии будут получены из базы данных реализовать в виде заглушек.
- все поля для ввода текста должны иметь валидацию.

Порядок выполнения работы

Для работы с Spring-MVC добавляем зависимость в pom.xml.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>3.0.6.RELEASE</version>
</dependency>
```

Spring-MVC – фреймворк, который строится вокруг DespatcherServlet, сервлета, перенаправляющего все запросы специальным обработчикам.

Конфигурация

в файле web.xml нужно описать слушателя, который будет загружать основной контекст приложения:

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

По умолчанию он ищет контекст в файле WEB-INF/applicationContext.xml. Если он находится в другом месте, то нужно указать путь к файлу в параметре web-контекста:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
```

```
        classpath:context.xml
    </param-value>
</context-param>
```

В данном случае мы указываем, что наш файл называется context.xml и лежит в корне.

Теперь нужно настроить DespatcherServlet:

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

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

Тут мы указали имя сервлета и что он будет обрабатывать все запросы оканчивающиеся на .html

DispatcherServlet создает свой собственный контекст, конфигурация которого ищется в файле /WEB-INF/<servlet-name>-servlet.xml. В нашем случае имя сервлета springmvc и следовательно файл будет находиться по пути /WEB-INF/springmvc-servlet.xml. Этот контекст будет являться дочерним и если при поиске в этом контексте не обнаружится компонент он будет искаться в родительском.

Так как один контекст может перекрыть другой контекст, то нужно быть внимательным при конфигурировании. Например, если указать для сканирования аннотаций все пакеты, то может поломаться управление транзакциями. Поэтому лучше вынести компоненты связанные в web в отдельные пакеты и указывать их при сканировании.

Определяем контроллер
Стереотипная аннотация @org.springframework.stereotype.Controller показывает, что данный компонент выполняет роль контроллера:

```
package com.nixsolutions.usermanagement.controller;

    @Controller
    public class UserController {

     …

    }
```

Контроллер содержит набор обработчиков, которые можно настроить на обработку запросов по разным критериям, например по определенному URL, типу запроса (GET или POST), передаваемым параметрам и т.п. Настройка происходит при помощи аннотации @org.springframework.web.bind.annotation.RequestMapping. Например, ниже показано, что обработчик будет вызываться если пользователь запросит ресурс /users/browse.html через GET запрос:

```
@Controller
public class UserController {

    @RequestMapping(value = "/users/browse.html", method = RequestMethod.GET)
    public ModelAndView browse() {

        ...

    }
}
```

Если несколько обработчиков мапятся на URL, который содержит общую часть (например /users/browse.html и /users/add.html), то общая часть может быть указана для всего контроллера:

```
    @Controller
    @RequestMapping(value = "/users")
    public class UserController {


        @RequestMapping(value = "/browse.html")
        public ModelAndView browse() {

          ...

        }
    }
```

Сигнатуры обработчиков (методов, помеченных аннотацией @RequestMapping) могут быть разными. Более подробно о параметрах и возвращаемых результатах смотри http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/mvc.html#mvc-ann-requestmapping-arguments.

В данном случае входящих параметров у обработчика нет, а возвращает он объект org.springframework.web.servlet.ModelAndView, который содержит данные отображаемые на view и логическое имя view.

```
@RequestMapping(value = "/browse.html")
public ModelAndView browse() {

    ModelAndView mav = new ModelAndView();
    mav.setViewName("browse");
    List<User> users = … //get users to browse
    mav.addObject(users);

    return mav;
}
```

Тут мы хотим отобразить view с именем browse и передаем туда список пользователей (мы можем так же указать имя, под которым список пользователей будет передаваться, но если не указано имя, то будет использоваться имя по-умолчанию, в данном случае userList)

В итоге получаем контроллер вида:

```
package com.nixsolutions.usermanagement.controller;


import java.util.ArrayList;
```

```java
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

import com.nixsolutions.usermanagement.User;
import com.nixsolutions.usermanagement.db.UserDao;


@Controller
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserDao userDao;

    @RequestMapping(value = "/browse.html")
    public ModelAndView browse() throws Exception {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("browse");
        List<User> users = new ArrayList<User>();
        users.addAll(userDao.findAll());
        mav.addObject(users);
        return mav;
    }
}
```

Определяем вид (view)

Для того чтобы отобразить результаты в браузере нам нужно сконфигурить как по логическому имени вида Spring найдет нужную страницу. Для этого в spring имеются ViewResolver'ы. Ресолверов несколько и это позволяет по-разному определять отношение вида и страницы. Мы будем использовать org.springframework.web.servlet.view. InternalResourceViewResolver, который использует имя вида как физическое имя страницы, при помощи суффиксов и префиксов уточнить путь к странице, а так же позволяет определять тип вида:

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

Данный пример показывает, что страницы находятся в папке /WEB-INF/jsp/ и имеют расширение .jsp. Т.е. подставляя к имени вида впереди значение префикса, а в конце значение суффикса получаем путь к странице.

У нас уже есть ранее написанные страницы, но их использовать в том виде, в котором они сейчас есть мы не можем, поэтому будем писать их заново и хранить их будем в папке WEB-INF/jsp (Это предотвратит доступ к этим страницам напрямую из браузера)

Опишем изменения на странице browse.jsp.
1. Форма больше не нужна так как все команды будем передавать серверу ссылками
2. К коллекции пользователей будем обращаться напрямую по имени

```
<c:forEach var="user" items="${userList}">
        ….
</c:forEach>
```

Это связано с тем, что сконфигурированный ранее ViewResolver сам будет следить, чтобы мы могли получать доступ к объектам по именам. Имена задаются, например, методом addObject(...) класса ModelAndView (явным указанием или по-умолчанию)

3. В таблицу пользователей добавим еще одну колонку, в которой будут ссылки на редактирование, удаление и детали, а радиокопку можно за ненадобностью убрать:

```
<tr>
<th></th>
<th>First name</th>
<th>Last name</th>
<th>Date of birth</th>
<th>Actions</th>
</tr>
        ….

<c:forEach var="user" items="${userList}">
<tr>
<td></td>
<td>${user.firstName}</td>
<td>${user.lastName}</td>
<td>${user.dateOfBirth}</td>
<td>
<a href="${editUrl}" >Edit</a>
<a href="${deleteUrl}" >Delete</a>
<a href="${detailsUrl}" >Details</a>
</td>
</tr>
</c:forEach>
```

Как видно ссылки являются переменными. Это связано с тем, что нам нужно динамически формировать URL для каждой строки таблицы:

```
<spring:url value="/users/edit.html" var="editUrl">
<spring:param name="id" value="${user.id}"/>
</spring:url>
<spring:url value="/users/browse.html" var="deleteUrl">
<spring:param name="id" value="${user.id}"/>
</spring:url>
<spring:url value="/users/details.html" var="detailsUrl">
<spring:param name="id" value="${user.id}"/>
</spring:url>
```

Тут мы используем теги spring, которые нужно описать в начале страницы:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
        4. Добавим ссылку на страницу добавления пользователя:
<spring:url value="/users/add.html" var="addUrl"/>
<a href="${addUrl}" >Add</a>
```

В итоге получаем страницу browse.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<html>
<head><title>User management</title></head>
<body>
<table id="userTable" border="1">
<tr>
<th></th>
<th>First name</th>
<th>Last name</th>
<th>Date of birth</th>
<th>Action</th>
</tr>
<c:forEach var="user" items="${userList}">
<spring:url value="/users/edit.html" var="editUrl">
<spring:param name="id" value="${user.id}"/>
</spring:url>
<spring:url value="/users/delete.html" var="deleteUrl">
<spring:param name="id" value="${user.id}"/>
</spring:url>
<spring:url value="/users/details.html" var="detailsUrl">
<spring:param name="id" value="${user.id}"/>
</spring:url>
<tr>
<td></td>
<td>${user.firstName}</td>
<td>${user.lastName}</td>
<td>${user.dateOfBirth}</td>
<td>
<a href="${editUrl}" >Edit</a>
<a href="${deleteUrl}" >Delete</a>
<a href="${detailsUrl}" >Details</a>
</td>
</tr>
</c:forEach>
</table>
<spring:url value="/users/add.html" var="addUrl"/>
<a href="${addUrl}" >Add</a>
<c:if test="${requestScope.error != null}">
<script>
            alert('${requestScope.error}');
</script>
</c:if>
</body>
</html>
```

Обработка форм
Теперь сделаем страницу добавления пользователей.
Для начала сделаем обработчик запроса страницы add.html:

```
@RequestMapping(value = "/add.html", method = RequestMethod.GET)
public String add(Model model) throws Exception {
        model.addAttribute(new User());
        return "add";
    }
```

Показан еще один вариант сигнатуры обработчика. Как можно понять метод add делает примерно тоже самое, что и метод browse, только иным способом. Объект, который будет использоваться на странице добавляется в экземпляр класса Model, который инжектится в

метод как параметр. Имя вида возвращаем как результат выполнения метода. Оба варианта (существуют и другие) могут использоваться по усмотрению разработчика.

Теперь напишем страницу с формой для добавления пользователя. На ней будем использовать теги спринга, поэтому добавим их описание в начало страницы:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
        Далее опишем форму.
<form:form method="post" modelAttribute="user">
        ….
</form:form>
```

В описании указываем метод post и определяем атрибут модели, с которой мы будем работать. В качестве значения используется имя, которое было указано при добавлении объекта в модель или, если имя не было указано, имя по-умолчанию.

Теперь напишем поля формы:

```
<form:form method="post" modelAttribute="user">
<form:hidden path="id"/><br>
            First name:
<form:input type="text" path="firstName" /><br>
            Last name:
<form:input type="text" path="lastName" /><br>
            Date of birth:
<form:input type="text" path="dateOfBirth"/><br>
<input type="submit" name="okButton" value="Ok">
        ….
</form:form>
```

В форме мы описали три поля ввода для полее объекта User и кнопку submit которая отправит форму на сервер. Нам еще понадобится кнопка Cancel. Ранее она также сабмитила форму, сейчас же мы просто сделаем по нажатию на кнопку переход на страницу browse:

```
<spring:url value="/users/browse.html" var="browseUrl"/>
<input type="button" name="cancelButton" value="Cancel"
        onclick="location.href='${browseUrl}'">
```

Сначала формируем URL на страницу browse и используем эту ссылку при обработки события onclick. Обратите внимание, что используется тип button, а не submit. Если не поменять тип кнопки, то будет все равно происходить отправка формы при нажатии. Избежать этого можно если в конце обработчика вернуть false:

```
<input type="submit" name="cancelButton" value="Cancel"

        onclick="location.href='${browseUrl}'; return false">
```

Добавим обработчик запроса. Форма будет передаваться как POST-запрос следовательно замапим обработчик следующим образом:

```
@RequestMapping(value = "/add.html", method = RequestMethod.POST)
public String save(User user) throws Exception {
        userDao.create(user);
```

```
        return "redirect:browse.html";
    }
```

Это еще один пример какие входные и выходные данные может содержать обработчик. Сигнатура обработчика включает в себя входной параметр типа User. Spring попытается создать и наполнить этот объект данными, которые пришли в запросе. Для этого имена параметров запроса должны совпадать с именами полей объекта. Так как мы использовали спринговые теги на форме все имена будут совпадать. Полученный таким образом объект мы сохраним в базу данных используя DAO.
Метод возвращает строку "redirect:browse.html", которая указывает, что мы должны перейти на страницу browse.html (именно на страницу, а не на вид с именем browse). Этим мы выполним все действия необходимые для отображения страницы browse, а именно перечитать из базы всех пользователей.

Если мы попытаемся добавить пользователя через страницу add то скорее всего увидим ошибку, о том что сервер не смог проинициализировать поле Date. Это связано с тем, что спринг не смог преобразовать строку, которую мы ввели в поле даты рождения в объект типа java.util.Date. Мы можем настроить формат строки, из которой нужно парсить дату. Для этого нужно написать метод с аннотацией @org.springframework.web.bind.annotation. InitBinder и параметром типа org.springframework.web.bind.WebDataBinder

```
@InitBinder
public void initBinder(WebDataBinder binder) {
        binder.registerCustomEditor(
        Date.class,
        new CustomDateEditor(
        new SimpleDateFormat("dd.MM.yyyy"), false));
   }
```

Тут мы зарегистрировали editor для типа java.util.Date и указали что формат поля будет <день>.<месяц>.<год> и поле не может быть пустым.

Теперь добавление должно пройти успешно. А для того, чтобы дата отображалась так же на странице browse, изменим вывод поля dateOfBirth используя тег форматирования:

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsp/jstl/fmt" %>
        ….
<f:formatDate value="${user.dateOfBirth}" pattern="dd.MM.yyyy"/>
```

Задания на самостоятельную работу

Ознакомиться как передать в обработчик параметры запроса и реализовать CRUD (редактирование, удаление и просмотр полной информации об объекте).