FREE eBook

LEARNING spring-mvc

Free unaffiliated eBook created from **Stack Overflow contributors.**

#spring-mvc

Table of Contents

About ... 1
Chapter 1: Getting started with spring-mvc ... 2
Remarks ... 2
Versions ... 2
Spring ... 2
Servlet Spec ... 2
Examples ... 3
First Spring-MVC Project ... 3
Chapter 2: Exception Handling ... 8
Syntax ... 8
Examples ... 8
Controller-Based Exception Handling ... 8
Chapter 3: File Upload ... 10
Syntax ... 10
Parameters ... 10
Remarks ... 10
Examples ... 10
Uploading a single file ... 10
Uploading multiple files ... 11
Uploading multiple parts with different names ... 12
Marshaling a part into an object ... 13
Chapter 4: Global Exception Handling ... 15
Remarks ... 15
Examples ... 15
Global Exception Resolver ... 15
Chapter 5: Spring MVC Validation ... 17
Remarks ... 17
Examples ... 17
Spring MVC Form Validation with Bean Validation API ... 17
Add Dependencies ... 17

Create Model Class 17
Create FormController Class 19
Create JSP Input Form 20
Create JSP Success Page 21
Test Application 21
Chapter 6: Spring-MVC with annotations 24
Introduction 24
Parameters 24
Examples 24
dispatcher-servlet.xml 24
@Controller & @RequestMapping 25
@RequestParam 25
Credits 26

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: spring-mvc

It is an unofficial and free spring-mvc ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official spring-mvc.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with spring-mvc

Remarks

This section provides an overview of what spring-mvc is, and why a developer might want to use it.

It should also mention any large subjects within spring-mvc, and link out to the related topics. Since the Documentation for spring-mvc is new, you may need to create initial versions of those related topics.

Versions

Spring

Version	Release Date
4.3.x	2016-06-10
4.2.x	2015-07-31
4.1.x	2014-09-14
4.0.x	2013-12-12
3.2.x	2012-12-13
3.1.x	2011-12-13
3.0.x	2009-12-17
2.5.x	2007-12-25
2.0.x	2006-10-04
1.2.x	2005-05-13
1.1.x	2004-09-05
1.0.x	2003-03-24

Servlet Spec

Version	Release Date
3.1	2013-05-31
3.0	2009-12-31
2.5	2005-09-30
2.4	2003-11-30
2.3	2001-08-31
2.2	1999-08-31
2.1	1998-11-30
1.0	1997-07-31

Examples

First Spring-MVC Project

Create Dynamic Web project, provide following information's as stated below

1. Project name : DemoSpringMVCProject
2. Target runtime : set as Apache Tomcat v7.0 server

Click on finish, successfully we have created dynamic web project.

Now we have to setup Spring-MVC framework :

1. Create **web.xml** under **' WebContent\WEB-INF\ '** folder

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
<display-name>Demo9</display-name>

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

<servlet-mapping>
  <servlet-name>demo</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>

</web-app>
```

- Where DispatcherServlet class Intercepts incoming request and determines which controller

handles the request.

- We are going to use servlet-name ' **demo** ' while creating servlet.xml

2. Create **demo-servlet.xml** under **' WebContent\WEB-INF\ '** folder

```
<?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: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="com"></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>

</beans>
```

- context:component-scan is used scan all controllers defined under **'com'** package.
- ViewResolver Interface Use to manage mapping between logical and actual views. Predefined implementation of view resolver are available to map the views. Ex: InternalResourceViewResolver, VelocityViewResolver.
- To search all jsp pages we have defined **prefix** which is nothing but setter property , it's value is set as **'/WEB-INF/jsp/'**(folder path) . **Suffix** which is nothing but getter property , it's value is set as **'.jsp'** (search file with an extension .jsp)

3. Add Required Libraries:

Let us add Spring Framework and common logging API libraries in our project. To do this, right click on your project name DemoSpringMVCProject and then follow the following option available in context menu: Build Path -> Configure Build Path to display the Java Build Path window as follows:

Now use Add External JARs button available under Libraries tab to add the following core JARs from Spring Framework and Common Logging installation directories:

- commons-logging-1.1.1
- spring-aop-4.1.6.RELEASE
- spring-aspects-4.1.6.RELEASE
- spring-beans-4.1.6.RELEASE
- spring-context-4.1.6.RELEASE

- spring-context-support-4.1.6.RELEASE
- spring-core-4.1.6.RELEASE
- spring-expression-4.1.6.RELEASE
- spring-instrument-4.1.6.RELEASE
- spring-instrument-tomcat-4.1.6.RELEASE
- spring-jdbc-4.1.6.RELEASE
- spring-jms-4.1.6.RELEASE
- spring-messaging-4.1.6.RELEASE
- spring-orm-4.1.6.RELEASE
- spring-oxm-4.1.6.RELEASE
- spring-test-4.1.6.RELEASE
- spring-tx-4.1.6.RELEASE
- spring-web-4.1.6.RELEASE
- spring-webmvc-4.1.6.RELEASE
- spring-webmvc-portlet-4.1.6.RELEASE
- spring-websocket-4.1.6.RELEASE

Let's move towards controller and jsp pages :

1. Create a **com.demo.controller** package under **src** folder.
2. Create a **LoginController** class under **com.demo.controller** package

```
package com.demo.controller;

import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class LoginController {

@RequestMapping("/")
public String startPage(){
    return "login";
}

@RequestMapping("/signin")
public String handleRequest(HttpServletRequest request){
    String name = request.getParameter("name");
    String pass = request.getParameter("password");
```

```
    if(name.equals(pass))
    {
        return "welcome";
    }else{
        return "login";
    }

   }

}
```

3. Create a **login.jsp and welcome.jsp** page under **' WebContent\WEB-INF\jsp\ '**

login.jsp

```
   <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
   <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
   <html>
   <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Login Page</title>
    </head>
    <body>
      <form action="signin">
          <table>
            <tr>
                <td>User Name : </td>
                <td><input type="text" name="name" id="name"/> </td>
            </tr>
            <tr>
                <td>Password: </td>
                <td><input type="text" name="password"  id="password"/> </td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit" value="Login"/></td>
            </tr>
          </table>
      </form>
    </body>
    </html>
```

welcome.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
  <h1> Welcome to Spring MVC !!! </h1>
</body>
</html>
```

Add DemoSpringMVCProject in localTomcat server and run it on server.

Read Getting started with spring-mvc online: https://riptutorial.com/spring-mvc/topic/824/getting-started-with-spring-mvc

Chapter 2: Exception Handling

Syntax

- @ExceptionHandler(ExceptionToBeHandled.class)
- @ExceptionHandler({ExceptionToBeHandled.class, AnotherExceptionToBeHandled.class})

Examples

Controller-Based Exception Handling

In the scenario that a controller throws an exception, we can define exception handler methods to build and return specific responses. It is important to note that the defined exception handlers within the given controller will only apply to exceptions that occur within that controller.

```
@Controller
public class MyController {
    @GetMapping("/")
    public String somePage() throws Exception {
        // some set of code that can throw exceptions
    }

    @ExceptionHandler(Exception.class)
    public String genericErrorPage() {
        return "genericErrorView";
    }

    @ExceptionHandler(ChildException.class)
    public String childErrorPage(ChildException ex) {
        return "childErrorView with msg=" + ex.getMessage();
    }
}
```

If there are multiple exception handlers defined, the method with the most specific exception will be chosen. Take the above code as example, if a `ChildException` is thrown, then the `childErrorPage()` method will be invoked.

Suppose a `NullPointerException` is thrown. In this case, the `genericErrorPage()` method will be invoked. This is because there isn't a specific exception handler defined for `NullPointerException`, but `NullPointerException` is a descendant-class of `Exception`.

This example also shows how you can access the exception. In the `childErrorPage` handler we have the `ChildException` passed as a parameter. It is then available to be used in the body of the handler, as shown. Similarly, you can define that handler like this:

```
    @ExceptionHandler(ChildException.class)
    public String childErrorPage(HttpServletRequest req, ChildException ex) {
        // Both request and exception objects are now available
        return "childErrorView with msg=" + ex.getMessage();
```

```
    }
```

This allows you to access the request that raised the exception as well as the exception that was raised.

Read Exception Handling online: https://riptutorial.com/spring-mvc/topic/6202/exception-handling

Chapter 3: File Upload

Syntax

- `@RequestPart(String, String, boolean)`

Parameters

Parameter	Details
`@RequestPart`	This annotation specifies that a parameter should be mapped to a given request part. The part name must match the name of the method parameter, *unless* you choose to provide it as an argument to `@RequestPart`. If the part name is not expressible as a Java name (e.g. `123`), then you can use the `value` attribute of the `@RequestPart` to specify the actual name. e.g. `@RequestPart("123") String _123`.

Remarks

If you are running on an older version of Java (pre 1.7), or are compiling *without* debug information, then Java will replace the name of the parameter with `arg0`, `arg1`, etc, which will prevent Spring from being able to match them up with the part names. If that is the case, then you will need to set the name of the part in the `@RequestPart` annotation, as documented in Parameters.

Examples

Uploading a single file

To receive a file uploaded via an HTTP Post, you need to do the following:

```
@RequestMapping(
    value = "...",
    method = RequestMethod.POST,
    consumes = MediaType.MULTIPART_FORM_DATA_VALUE
)
public Object uploadFile(
    @RequestPart MultipartFile file
) {
    String fileName = file.getOriginalFilename();
    InputStream inputStream = file.getInputStream();
    String contentType = file.getContentType();
    .
    .
    .
}
```

Note that the name of the `@RequestPart` parameter needs to match up with the name of the part in the request.

As HTML:

```
<form action="/..." enctype="multipart/form-data" method="post">
       <input type="file" name="file">
   </form>
```

As HTML (Spring TagLibs):

```
<form action="/..." enctype="multipart/form-data" method="post">
    <form:input type="file" path="file">
</form>
```

As a raw HTTP request:

```
POST /... HTTP/1.1
Host: ...
Content-Type: multipart/form-data; boundary=----------287032381131322

------------287032381131322
Content-Disposition: form-data; name="file"; filename="r.gif"
Content-Type: image/gif

GIF87a.............,...........D..;
------------287032381131322--
```

That request would mean the following:

```
fileName == "r.gif"
contentType == "image/gif"
```

In Spring MVC

Need to add the mentioned bean for accessing multipart functionality

```
    <!-- max size of file in memory (in bytes) -->
    <property name="maxInMemorySize" value="1048576" /> <!-- 1MB -->

</bean>
```

Uploading multiple files

To receive multiple files uploaded via a single HTTP Post, you need to do the following:

```
@RequestMapping(
    value = "...",
    method = RequestMethod.POST,
    consumes = MediaType.MULTIPART_FORM_DATA_VALUE
)
public Object uploadFile(
```

```
    @RequestPart MultipartFile[] files
) {
    for (file : files) {
        String fileName = file.getOriginalFilename();
        InputStream inputStream = file.getInputStream();
        String contentType = file.getContentType();
        .
        .
        .
    }
}
```

Note that the name of the `@RequestPart` parameter needs to match up with the name of the part in the request.

As HTML:

```
<form action="/..." enctype="multipart/form-data" method="post">
    <input type="file" name="files">
    <input type="file" name="files">
</form>
```

As a raw HTTP request:

```
POST /... HTTP/1.1
Host: ...
Content-Type: multipart/form-data; boundary=----------287032381131322

------------287032381131322
Content-Disposition: form-data; name="files"; filename="r.gif"
Content-Type: image/gif

GIF87a.............,...........D..;
------------287032381131322
Content-Disposition: form-data; name="files"; filename="banana.jpeg"
Content-Type: image/jpeg

GIF87a.............,...........D..;
------------287032381131322--
```

That request would mean the following:

```
files[0].getOriginalFilename() == "r.gif"
files[0].getContentType() == "image/gif"
files[1].getOriginalFilename() == "r.jpeg"
files[1].getContentType() == "image/jpeg"
```

Uploading multiple parts with different names

It is possible to upload multiple parts, each with a different name. For each part name, you will need one parameter annotated with `@RequestPart`, whose name matches the part name.

To receive a file uploaded via an HTTP Post, you need to do the following:

```
@RequestMapping(
    value = "...",
    method = RequestMethod.POST,
    consumes = MediaType.MULTIPART_FORM_DATA_VALUE
)
public Object uploadFile(
    @RequestPart MultipartFile profilePicture,
    @RequestPart MultipartFile companyLogo,
) {
    .
    .
    .
}
```

As HTML:

```
<form action="/..." enctype="multipart/form-data" method="post">
    <input type="file" name="profilePicture">
    <input type="file" name="companyLogo">
</form>
```

As a raw HTTP request:

```
POST /... HTTP/1.1
Host: ...
Content-Type: multipart/form-data; boundary=----------287032381131322

------------287032381131322
Content-Disposition: form-data; name="profilePicture"; filename="r.gif"
Content-Type: image/gif

GIF87a............,...........D..;
------------287032381131322
Content-Disposition: form-data; name="companyLogo"; filename="banana.jpeg"
Content-Type: image/jpeg

GIF87a............,...........D..;
------------287032381131322--
```

Marshaling a part into an object

If you want to convert the content of a part into a domain object (e.g. a `User` or `Account` or `Address`), then the process is very simple:

It is possible to upload multiple parts, each with a different name. For each part name, you will need one parameter annotated with `@RequestPart`, whose name matches the part name.

To receive a file uploaded via an HTTP Post, you need to do the following:

```
@RequestMapping(
    value = "...",
    method = RequestMethod.POST,
    consumes = MediaType.MULTIPART_FORM_DATA_VALUE
)
public Object uploadFile(
```

```
    @RequestPart Address address,
) {
    .
    .
    .
}
```

As a raw HTTP request:

```
POST /... HTTP/1.1
Host: ...
Content-Type: multipart/form-data; boundary=----------287032381131322

------------287032381131322
Content-Disposition: form-data; name="address"; filename="address.json"
Content-Type: application/json

{"houseNumber": "10/A", "streetName": "Dumbldore Road", "town": "Hogsmede"}
------------287032381131322--
```

The most important things are:

- The name of the part must match the name of the variable.
- The `Content-Type` of the part must be one that Spring would be able to handle if you had sent it as a regular request. That is, if you could perform a `POST` to an endpoint with a `Content-Type` of `foo/bar`, and Spring is able to turn that into an object, then it will also be able to marshal a part into an object.
- You *must* be able to set the `Content-Type` of the part. If you cannot, this approach will not work - Spring will *not* attempt to guess the `Content-Type` of the part.

Read File Upload online: https://riptutorial.com/spring-mvc/topic/3050/file-upload

Chapter 4: Global Exception Handling

Remarks

1. Don't forget to create custom exceptions if you have to
2. Both the resolver and handler must be beens discovered by Spring
3. If you are on Spring 3.2 or higher, you can use `@ContrllerAdvice`

Source

Examples

Global Exception Resolver

```
@Component
public class RestExceptionResolver extends ExceptionHandlerExceptionResolver {

    @Autowired
    //If you have multiple handlers make this a list of handlers
    private RestExceptionHandler restExceptionHandler;
    /**
     * This resolver needs to be injected because it is the easiest (maybe only) way of
getting the configured MessageConverters
     */
    @Resource
    private ExceptionHandlerExceptionResolver defaultResolver;

    @PostConstruct
    public void afterPropertiesSet() {
        setMessageConverters(defaultResolver.getMessageConverters());
        setOrder(2); // The annotation @Order(2) does not work for this type of component
        super.afterPropertiesSet();
    }

    @Override
    protected ServletInvocableHandlerMethod getExceptionHandlerMethod(HandlerMethod
handlerMethod, Exception exception) {
        ExceptionHandlerMethodResolver methodResolver = new
ExceptionHandlerMethodResolver(restExceptionHandler.getClass());
        Method method = methodResolver.resolveMethod(exception);
        if (method != null) {
            return new ServletInvocableHandlerMethod(restExceptionHandler, method);
        }
        return null;
    }

    public void setRestExceptionHandler(RestExceptionHandler restExceptionHandler) {
        this.restExceptionHandler = restExceptionHandler;
    }

    public void setDefaultResolver(ExceptionHandlerExceptionResolver defaultResolver) {
        this.defaultResolver = defaultResolver;
    }
}
```

Then an example handler will look like this

```
@Component
public class RestExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ResponseBody
    public Map<String, Object> handleException(ResourceNotFoundException e,
HttpServletResponse response) {
        Map<String, Object> error = new HashMap<>();
        error.put("error", e.getMessage());
        error.put("resource", e.getResource());
        return error;
    }
 }
```

Of course you will not forget to register your beens

Read Global Exception Handling online: https://riptutorial.com/spring-mvc/topic/7648/global-exception-handling

Chapter 5: Spring MVC Validation

Remarks

In Spring MVC it is possible to validate form fields using Bean Validation API (JSR 303 for Bean Vaidation 1.0 and JSR 349 for Bean Validation 1.1) that is used to define validation constraints of the JavaBean object.

Hibernate Validator is Bean Validation API reference implementation. Hibernate Validator offers additional value on top of the features required by Bean Validation. For example, a programmatic constraint configuration API as well as an annotation processor which plugs into the build process and raises compilation errors whenever constraint annotations are incorrectly used.

Examples

Spring MVC Form Validation with Bean Validation API

This example shows how to validate forms in Spring MVC using **Bean Validation API** using Java Annotations, without any `xml`. User will be proposed to input their registration data and validator will check it for validity.

Add Dependencies

First of all add the following dependencies in your project:

- Bean Validation API and
- Hibernate Validator Engine

```
dependencies {
    compile group: 'javax.validation', name: 'validation-api', version: '1.1.0.Final'
    compile group: 'org.hibernate', name: 'hibernate-validator', version: '5.2.4.Final'
}
```

Create Model Class

Create the model class `User` as below:

```
import org.hibernate.validator.constraints.Email;
import org.springframework.format.annotation.DateTimeFormat;

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Size;
import java.util.Date;
```

```
public class User {

    @NotNull(message = "Please input your email.")
    @Email(message = "Email format is wrong.")
    private String email;

    @NotNull(message = "{user.password.notNull}")
    @Size(min = 8, max = 16, message = "{user.password.size}")
    private String password;

    @NotNull(message = "{user.age.notNull}")
    @Min(18)
    @Max(100)
    private Integer age;

    @NotNull(message = "{user.birthday.notNull}")
    @DateTimeFormat(pattern = "dd.MM.yyyy")
    @Past(message = "{user.birthday.past}")
    private Date birthday;

    // getters, setters
}
```

Here is using some of the JSR 303 annotations: `@NotNull`, `@Size`, `@Min`, `@Max` and `@Past` as well as some additional annotations provided by hibernate validator implementation: `@Email`, `@DateTimeFormat`.

Notice that error messages for `email` field is specified inside its annotations. Whereas the error messages for the `password`, `age` and `birthday` fields is specified in a `messages.properties` file in order to demonstrate the *externalization of validation error messages*. This files should be put under `resources` folder:

```
user.password.notNull = Password field cannot be empty.
user.password.size = Password must be between {min} and {max} characters in length.
user.age.notNull = Please enter your age.
user.birthday.notNull = Please enter your birthday.
user.birthday.past = That's impossible.

typeMismatch=Please use dd.MM.yyyy format
```

For this ability `messageSource()` with `bean.setBasename("classpath:messages");` code and `validator()` beans must be also configured as well as annotation:

```
@Configuration
@PropertySource("application.properties")
public class AppConfig extends WebMvcConfigurerAdapter {

    @Bean
    public MessageSource messageSource() {
        ReloadableResourceBundleMessageSource bean = new
ReloadableResourceBundleMessageSource();
        bean.setBasename("classpath:messages");
        bean.setDefaultEncoding("UTF-8");
        return bean;
    }
```

```
    @Bean
    public LocalValidatorFactoryBean validator() {
        LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
        bean.setValidationMessageSource(messageSource());
        return bean;
    }

    @Override
    public Validator getValidator() {
        return validator();
    }
}
```

Also configuration class mu st be annotated with `@PropertySource("application.properties")` and the path to `jsp` pages must be added to this file as below:

```
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
```

Create FormController Class

Now in the controller class, annotate the model object that is backing the form by the `@Valid` annotation from `javax.validation` package.

Spring MVC will validate the model object annotated by the `@Valid` annotation after binding its properties with inputs from JSP form that uses Spring's form tags. Any constraint violations will be exposed as errors in the `BindingResult` object, thus we can check the violation in the controller's method.

```
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 javax.validation.Valid;
import java.util.HashMap;
import java.util.Map;

@Controller
public class FormController {

    private Map<String, User> users = null;

    public FormController() {
        users = new HashMap<String, User>();
    }

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String viewRegister(Map<String, Object> model) {
        User user = new User();
        model.put("user", user);
        return "register";
    }
```

```
    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public String doRegister(@Valid User user, BindingResult result, Model model) {
        if (result.hasErrors()) {
            return "register";
        }
        model.addAttribute("user", user);
        users.put(user.getEmail(), user);
        return "registerSuccess";
    }
}
```

Create JSP Input Form

Add `register.jsp` file with the following content:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>User Form Page</title>
<style>
.error {
    color: #ff0000;
    font-weight: bold;
}
</style>
</head>
<body>
    <form:form method="POST" commandName="user" action="register">
        <table>
            <tr>
                <td>Email:</td>
                <td><form:input path="email" placeholder="Email"/></td>
                <td><form:errors path="email" cssClass="error" /></td>
            </tr>
            <tr>
                <td>Password:</td>
                <td><form:password path="password" placeholder="Password"/></td>
                <td><form:errors path="password" cssClass="error" /></td>
            </tr>
            <tr>
                <td>Age:</td>
                <td><form:input path="age" placeholder="Age"/></td>
                <td><form:errors path="age" cssClass="error" /></td>
            </tr>
            <tr>
                <td>Birthday:</td>
                <td><form:input path="birthday" placeholder="dd.MM.yyyy"/></td>
                <td><form:errors path="birthday" cssClass="error" /></td>
            </tr>
            <tr>
                <td colspan="3"><input type="submit" value="Register"></td>
            </tr>
        </table>
```

```
    </form:form>

</body>
</html>
```

Typically, we would return the input form back to the user when any validation errors occurred. And in the JSP form, we can show validation error messages using the Spring's form errors tag like `<form:errors path="email"/>`.

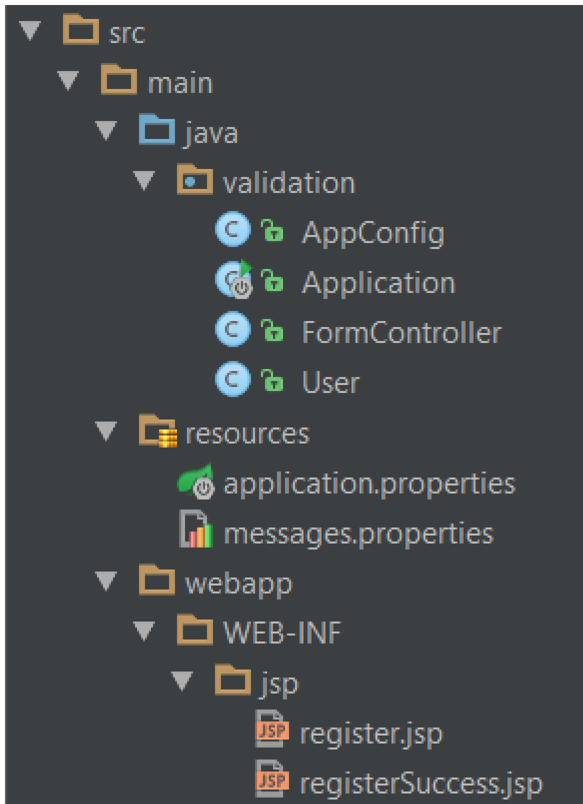
Create JSP Success Page

The `registerSuccess.jsp` page will be displayed in case the user enters all data valid. Here's the code:

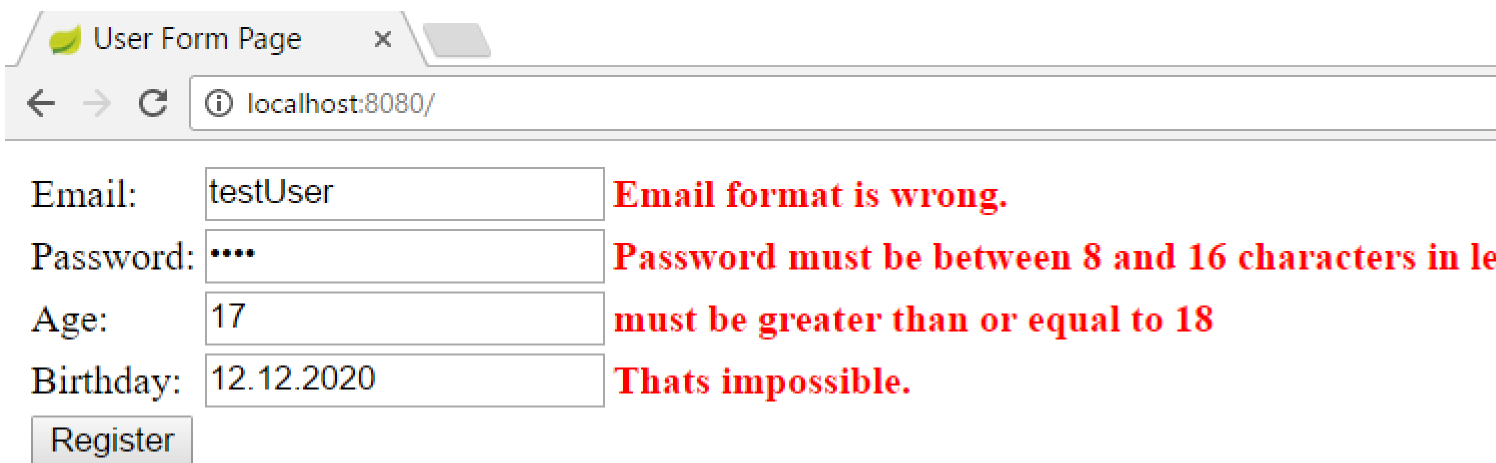
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<%@ page session="false" %>
<html>
<head>
    <title>Success</title>
</head>
<body>
<h3>User Registered Successfully.</h3>

<strong>User Email: ${user.email}</strong><br>
<strong>User Age: ${user.age}</strong><br>
<strong>User Birthday: <fmt:formatDate value="${user.birthday}" type="date"
pattern="dd.MM.yyyy"/></strong><br>

</body>
</html>
```

Test Application

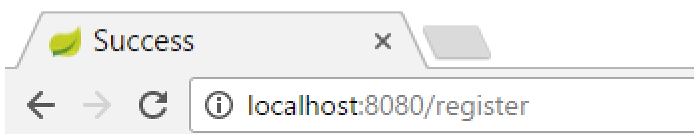
After all project structure should looks like this:



Start the application, go to `http://localhost:8080/` and try to enter invalid data:



When valid data be entered, user will redirect to success page:



User Registered Successfully.

User Email: testUser@email.com
User Age: 18
User Birthday: 12.12.2015

Read Spring MVC Validation online: https://riptutorial.com/spring-mvc/topic/7127/spring-mvc-validation

Chapter 6: Spring-MVC with annotations

Introduction

In this topic you'll read about annotations mainly related to Spring MVC. Some of the related annotations are as follows: `@Controller`, `@RequestMapping`, `@RequestParam`, `@RequestBody`, `@ResponseBody`, `@RestController`, `@ModelAttribute`, `@ControllerAdvice`, `@ExceptionHandler`, `@ResponseStatus`.

Of course there're more annotations which are extremly important as well but not belong directly to Spring MVC. Such as: `@Required`, `@Autowired`, `@Resource`, and many more.

Parameters

Annotation	Explanation
`@Controller`	With `@Controller` annotation you mark a Java Class as a Class that holds HTTP handlers, in other words, HTTP access points to your application.
`@RequestMapping`	The `@RequestMapping` annotation is the one that you'll use to mark HTTP handlers (HTTP access points to your application) within your `@Controller` Class
`@RequestParam`	Use the `@RequestParam` annotation to bind request parameters to a method parameter in your controller.

Examples

dispatcher-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:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
        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-4.3.xsd">


    <mvc:annotation-driven/>
    <context:component-scan base-package="your.base.package.to.scan" />
</beans>
```

With these two lines of configuration, you'll enable the usage of MVC annotations.

@Controller & @RequestMapping

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {

//your handlers here, for example:

@RequestMapping(path = "/new", method = RequestMethod.GET)
public AppointmentForm getNewForm() {
    return new AppointmentForm();
}

@RequestMapping(method = RequestMethod.POST)
public String add(@Valid AppointmentForm appointment, BindingResult result) {
    if (result.hasErrors()) {
        return "appointments/new";
    }
    appointmentBook.addAppointment(appointment);
    return "redirect:/appointments";
}
```

}

With `@Controller` annotation you'll mark a Java Class as a Class that holds several HTTP handlers, in other words, HTTP access points to your application.

The `@RequestMapping` annotation is the one that you'll use to mark HTTP handlers (HTTP access points to your application) within your @Controller Class

@RequestParam

```
@Controller
public class EditPetForm {

    @RequestMapping("/pets")
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }
}
```

Important to mention, but pretty obvious, is that `@RequestParam` is intended to work when using HTTP GET method only because only with GET you can send a query string with parameters, and `@RequestParam` you can bind parameters in the query string to your controller handler parameters.

Read Spring-MVC with annotations online: https://riptutorial.com/spring-mvc/topic/10662/spring-mvc-with-annotations

Credits

S. No	Chapters	Contributors
1	Getting started with spring-mvc	Community, Flash, ipsi, Slava Semushin
2	Exception Handling	Nathaniel Ford, Tim Tong
3	File Upload	akhilsk, ipsi
4	Global Exception Handling	Amanuel Nega
5	Spring MVC Validation	DimaSan
6	Spring-MVC with annotations	Moshe Arad