

Table of Contents

Overview	2
Prepare Eclipse Project	3
Using Forms in Application	6
Validating Form using Build-in Validators.....	13
Uploading the Files in Spring MVC App	23
Uploading Using the Servlet Context Aware Controller	29
Uploading Multiple Files	34
Uploading Multiple Files Using Multiple Selections	38
Downloading Files from Application	45
Summary	51
Quiz	52

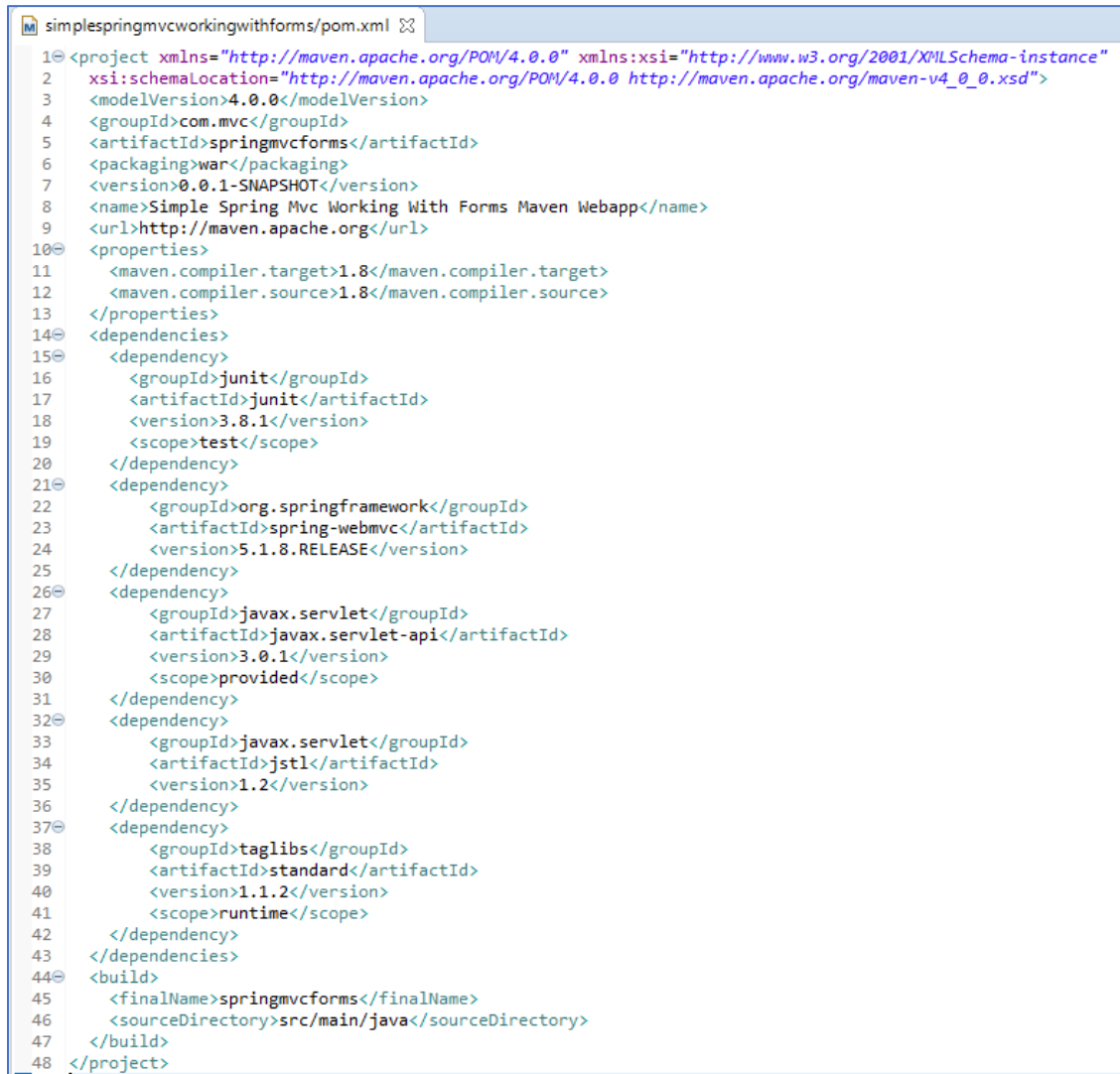
Overview

In the this course in this learning path, we will work with forms and files in Spring MVC. We'll see the use of built-in annotations in form validation. And also explore how files can be uploaded to and downloaded from our application.

Prepare Eclipse Project

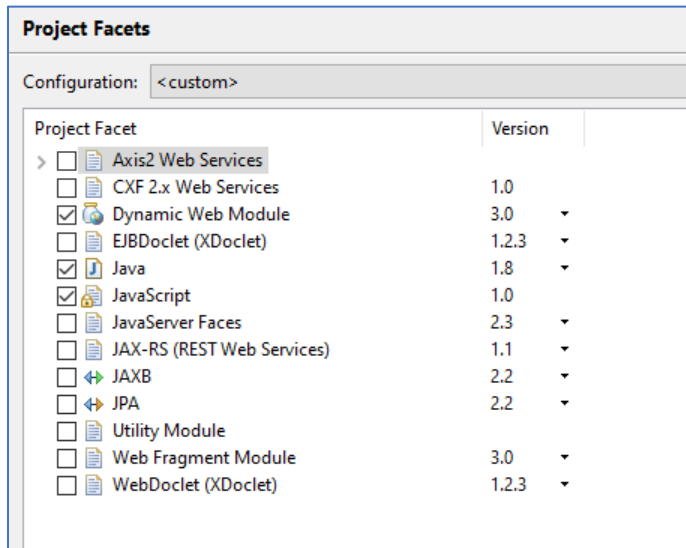
1. POM.xml

- JVM 1.8
- Spring Framework (org.springframework.spring-webmvc:5.1.8-RELEASE)
- Java Servlet (javax.servlet:javax.servlet-api: 3.0.1, javax.servlet.jstl:1.2, taglibs:1.1.2)

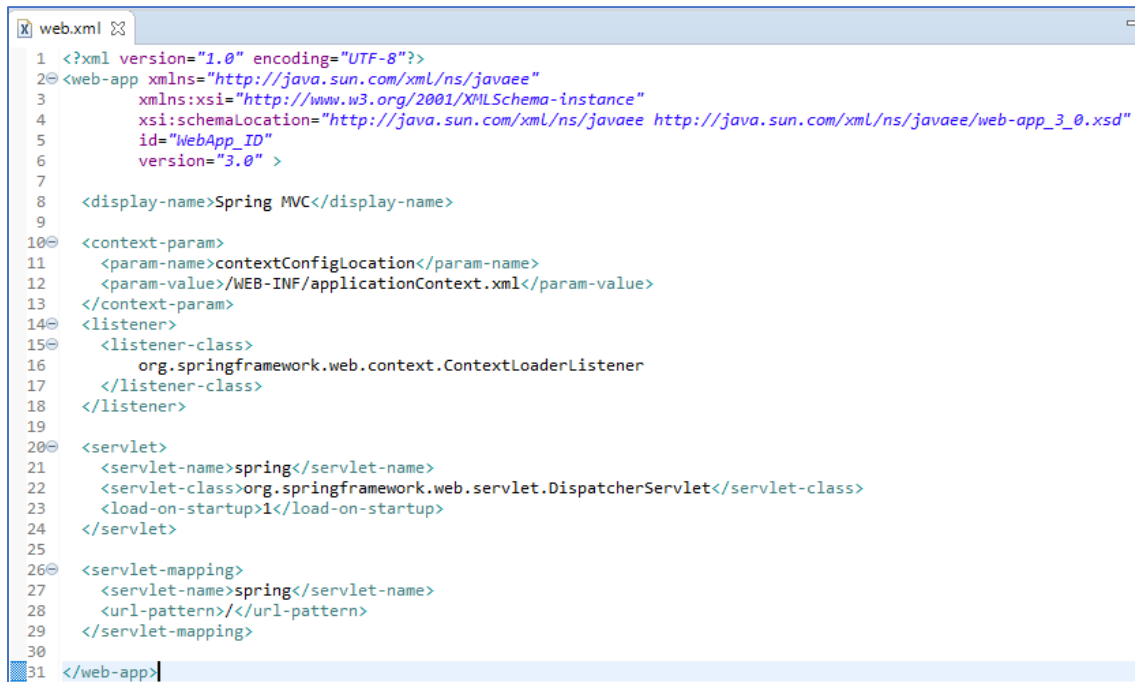


```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4   <groupId>com.mvc</groupId>
5   <artifactId>springmvcforms</artifactId>
6   <packaging>war</packaging>
7   <version>0.0.1-SNAPSHOT</version>
8   <name>Simple Spring Mvc Working With Forms Maven Webapp</name>
9   <url>http://maven.apache.org</url>
10  <properties>
11    <maven.compiler.target>1.8</maven.compiler.target>
12    <maven.compiler.source>1.8</maven.compiler.source>
13  </properties>
14  <dependencies>
15    <dependency>
16      <groupId>junit</groupId>
17      <artifactId>junit</artifactId>
18      <version>3.8.1</version>
19      <scope>test</scope>
20    </dependency>
21    <dependency>
22      <groupId>org.springframework</groupId>
23      <artifactId>spring-webmvc</artifactId>
24      <version>5.1.8.RELEASE</version>
25    </dependency>
26    <dependency>
27      <groupId>javax.servlet</groupId>
28      <artifactId>javax.servlet-api</artifactId>
29      <version>3.0.1</version>
30      <scope>provided</scope>
31    </dependency>
32    <dependency>
33      <groupId>javax.servlet</groupId>
34      <artifactId>jstl</artifactId>
35      <version>1.2</version>
36    </dependency>
37    <dependency>
38      <groupId>>taglibs</groupId>
39      <artifactId>standard</artifactId>
40      <version>1.1.2</version>
41      <scope>runtime</scope>
42    </dependency>
43  </dependencies>
44  <build>
45    <finalName>springmvcforms</finalName>
46    <sourceDirectory>src/main/java</sourceDirectory>
47  </build>
48 </project>
```

2. Java Build Properties



3. Web XML



4. Spring Servlet XML (under WEB-INF directory)

```
spring-servlet.xml
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:context="http://www.springframework.org/schema/context"
5       xmlns:mvc="http://www.springframework.org/schema/mvc"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7                           http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
8                           http://www.springframework.org/schema/mvc
9                           http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd
10                          http://www.springframework.org/schema/context
11                          http://www.springframework.org/schema/context/spring-context-4.0.xsd">
12
13   <bean id="viewResolver"
14         class="org.springframework.web.servlet.view.InternalResourceViewResolver" >
15       <property name="prefix" value="/WEB-INF/jsp/" />
16       <property name="suffix" value=".jsp" />
17     </bean>
18
19 </beans>
```

5. Application context xml (under WEB-INF directory)

```
applicationContext.xml
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:context="http://www.springframework.org/schema/context"
5       xmlns:mvc="http://www.springframework.org/schema/mvc"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7                           http://www.springframework.org/schema/beans/spring-beans.xsd
8                           http://www.springframework.org/schema/context
9                           http://www.springframework.org/schema/context/spring-context.xsd
10                          http://www.springframework.org/schema/mvc
11                          http://www.springframework.org/schema/mvc/spring-mvc.xsd
12       ">
13
14   <context:component-scan base-package="com.mvc"></context:component-scan>
15   <mvc:annotation-driven />
16
17   <bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
18     <property name="defaultErrorView" value="error" />
19   </bean>
20
21 </beans>
```

6. Error.jsp (under WEB-INF/jsp directory)

```
error.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6   <meta charset="ISO-8859-1">
7   <title>Error Message</title>
8 </head>
9 <body>
10   <h2>${exception}</h2>
11 </body>
12 </html>
```

Using Forms in Application

1. Prepare Model Layer



```
1 package com.mvc.model;
2
3 public class Registration {
4
5     private String firstName;
6     private String lastName;
7     private String gender;
8     private String department;
9     private String[] food;
10
11     public Registration() {
12     }
13
14     public String getFirstName() {
15         return firstName;
16     }
17
18     public void setFirstName(String firstName) {
19         this.firstName = firstName;
20     }
21
22     public String getLastName() {
23         return lastName;
24     }
25
26     public void setLastName(String lastName) {
27         this.lastName = lastName;
28     }
29
30     public String getGender() {
31         return gender;
32     }
33
34     public void setGender(String gender) {
35         this.gender = gender;
36     }
37 }
```

This object represents a registration for a summer picnic at the fantabulous university. Every student who registers is required to specify the first name, last name, the gender, the department that the student belongs to, and food references, what meals the student is interested in.

Make sure that you have a default no argument constructor for this object and setup getters and setters for all of the member variables in your object.

2. Prepare Presentation-Controller Class

```

RegistrationController.java
1 package com.mvc.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.ModelAttribute;
6 import org.springframework.web.bind.annotation.RequestMapping;
7
8 import com.mvc.model.Registration;
9
10 @Controller
11 @RequestMapping("/details")
12 public class RegistrationController {
13
14     @RequestMapping("/registration")
15     public String registrationForm(Model model) {
16         Registration registration = new Registration();
17         model.addAttribute("registration", registration);
18         return "registration-page";
19     }
20
21     @RequestMapping("/submission")
22     public String submitForm(@ModelAttribute("registration") Registration registration) {
23         return "confirmation-page";
24     }
25 }
26

```

All handler mappings specified in this controller have the path prefix /details.

Let's take a look at the first method registration form, which displays the form to a student, so that the student can fill the form in. The RequestMapping is for the path /registration. And we inject the model of our Spring MVC application to this method as an input argument. We instantiate a new Registration object. And we pass this registration object to the client by adding it as an attribute in our model. And we'll render out the registration-page. Having this registration object as the backing object of our form helps us avoid using request parameters to pass form values. Request parameters aren't really very safe because they are visible in the URL. You want those parameters to be hidden. The first method here corresponding to the /registration path is what displays the registration page with the form to the user.

The second method here is what is invoked when the student has filled up the form and then submits the form. This submitForm method accepts as an input argument the registration object. But this registration object is populated. The various member variables within this registration object have been initialized by the form value specified by the student. Notice the **@ModelAttribute** registration annotation on this registration object that we have injected into this method. This ModelAttribute is what links the registration object that is bound to the user interface form which the student fills in. And the same object is then injected into our controller method here.

Using this **@ModelAttribute** annotation also allows us to pass this registration object onward to the confirmation page. The confirmation page is rendered within the same request and the registration object is passed on to the confirmation page.

3. Prepare main view entry point, the index.jsp file. (Under WEB-INF directory)

```

index.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6   <meta charset="ISO-8859-1">
7   <title>Summer Picnic</title>
8 </head>
9 <body>
10   <h2>Fantabulous University Summer Picnic Program</h2>
11   <a href="details/registration">Click here to register.</a>
12 </body>
13 </html>

```

This is the Fantabulous University Summer Picnic. *Click here to register*. It's an anchor link that points to details/registration.

This is the path reference handled by our first RequestMapping that will display the registration page backed by the registration object that makes up the model for that page.

4. Prepare 2nd view layer, registration page. (under WEB-INF/jsp directory)

```

registration-page.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
4 <!DOCTYPE html>
5 <html>
6 <head>
7   <meta charset="ISO-8859-1">
8   <title>Summer Picnic Registration</title>
9 </head>
10 <body>
11
12   <h2>Fantabulous picnic registration form</h2>
13   <form:form action="submission" modelAttribute="registration">
14     <b>First Name</b><form:input path="firstName" /><br/><br/>
15     <b>Last Name</b><form:input path="lastName" /><br/><br/>
16     <b>Gender</b><br/>
17     <form:radio button path="gender" value="Male"/>Male<br/>
18     <form:radio button path="gender" value="Female"/>Female<br/>
19     <br/><br/>
20     <b>Department</b>
21     <form:select path="department">
22       <form:option value="Biology" label="Biology"/>
23       <form:option value="Mathematics" label="Mathematics"/>
24       <form:option value="Chemistry" label="Chemistry"/>
25       <form:option value="Arts" label="Arts"/>
26     </form:select>
27     <br/><br/>
28     <b>Meals</b><br/>
29     <form:checkbox path="food" value="Breakfast"/>BreakFast<br/>
30     <form:checkbox path="food" value="Lunch"/>Lunch<br/>
31     <form:checkbox path="food" value="Dinner"/>Dinner<br/>
32     <br/><br/>
33     <input type="submit" value="Submit"/>
34
35   </form:form>
36 </body>
37 </html>

```

Notice our taglib import at the very beginning.


```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>.
```

We use the form tags available in our JSTL taglib library with the prefix form. This form tag library is what allows us to bind our form data to a controller. It will also help with error handling and form validation later on.

We've defined the form starting on line 13 and the form action be specified as submission.
`<form:form action="submission" modelAttribute="registration">.`

Submitting this form will take us to the /details /submission handler mapping. That will render the confirmation-page with all of the registration details. Notice that the form action only says submission and not details /submission because this path is a relative to the path for this registration form which already has the details prefix. The next detail to note here is the modelAttribute that we have specified on the form. modelAttribute is equal to registration.

This modelAttribute will ensure that this particular form that is displayed is bound to the registration object that we pass down from the controller while rendering the registration page. The individual member variables within the registration object will map to form fields. And values that we fill in for those form fields will be used to populate the registration object. Now every input specified in this form corresponds to a member variable in the registration object.

Let's take a look. We have the first form:input for the first name path="firstName".

```
<b>First name:</b><form:input path="firstName" />.
```

firstName is also the member variable in that registration class. The next form:input is for the last name path="lastName".

```
<b>Last name:</b><form:input path="lastName" />.
```

And this path should match the lastName member variable in your registration class and it does. Let's take a look at gender.

```
<b>Gender: <b><br/>.
```

```
Male<form:radiobutton path="gender" value="Male" /> <br/>.
```

```
Female<form:radiobutton path="gender" value="Female" /> <br/>.
```

Gender we've specified as a radiobutton. You can have one of two possible values, Male or Female. Both of the radiobuttons have path set to gender. Depending on what radiobutton you select, the gender string in our registration object will be set to either Male or Female.

Let's scroll down further to see other form fields. The department is a drop-down menu with many options. A drop-down is specified using the form:select input path="department". And this path matches the department member variable in our registration class.

The form options are Biology, Mathematics, Chemistry and Arts. And the department string can take on one of any of these values.

And finally, we have the Meals that the students can pick for the picnic. This is in the form of a checkbox, which means multiple selections are possible.

For each checkbox, we've set path="food", which corresponds to the string array for food in our registration class. The value for each of the elements in the array can be Breakfast, Lunch, or Dinner.

Any combination and any number of meals can be checked. And of course, at the very bottom, we have the Submit button of the form.

5. Prepare 3rd view layer, the confirmation page. (under /WEB-INF/jsp)

```
confirmation-page.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4 <!DOCTYPE html>
5 <html>
6 <head>
7   <meta charset="ISO-8859-1">
8   <title>Summer Picnic Confirmation</title>
9   <style type="text/css">
10     div, li {
11       color: green;
12       padding: 6px;
13     }
14   </style>
15 </head>
16 <body>
17   <div><b>Your registration is confirmed!. Please make sure your details are correct</b></div>
18   <br/><br/>
19   <div>First Name : ${registration.firstName}</div>
20   <div>Last Name : ${registration.lastName}</div>
21   <div>Gender : ${registration.gender}</div>
22   <div>Department : ${registration.department}</div>
23   <div>Meals opted in : </div>
24   <ul>
25     <c:forEach var="meal" items="${registration.food}">
26       <li>${meal}</li>
27     </c:forEach>
28   </ul>
29
30 </body>
31 </html>
```

A confirmation page receives the registration object and prints out the registration details that the student has entered. We can access the member variables of the registration object. This is achieved using reflection in Java, registration.firstName, .last name, .gender, .department.

And finally at the bottom, we use a forEach loop to iterate over the food preferences that the student has specified.

6. Build and Run the App on Tomcat Server

With our application all set up, it's time to build this application and deploy it to our Tomcat server.

Make sure you run the server and kill any previous server versions that have been running.

Here is what our main page looks like,

Fantabulous University Summer Picnic Program
[Click here to register.](#)

Here is the registration page mapping to the request path /details /registration.

Fantabulous picnic registration form
First Name
Last Name
Gender
☒ Male
☐ Female
Department
Meals
☐ BreakFast
☒ Lunch
☐ Dinner

The registration object that we pass down from our controller is the model back in this form. So let's go ahead and fill in all the different form fields, James Watson, first name, last name, male. Here is the department drop-down. I'm going to choose Mathematics as a department for James Watson.

The next step is to select the meals that the student would prefer. I'm just going to select Lunch, go ahead and hit Submit. This takes us directly to the confirmation page.

Your registration is confirmed!. Please make sure your details are correct

First Name : James

Last Name : Watson

Gender : Male

Department : Mathematics

Meals opted in :

- Lunch

The registration object which was the model backing our form was populated when we filled up the form. And the same registration object was injected into our submitForm method, thanks to the input registration argument that we have annotated with the @ModelAttribute annotation.

The same registration object is also passed as a model to the confirmation page. And we use this registration object in the confirmation page to render out the registration details for James Watson.

Validating Form using Build-in Validators

Our previous form had no validation at all. We just accepted whatever the user input. And if the user left some fields blank, there was nothing that we did about it. Here in this demo, we'll add some validation to our form

1. Adding new dependency on POM.xml for validation

And this requires a few additional dependencies that I've specified in pom.xml.

javax.validation.validation-api:2.0.1.Final
org.hibernate.hibernate-validator:6.0.13.Final
jakarta.xml.bind-api:2.3.2

I want to use Java annotations to specify validations that I want performed on my form fields.

For this, I need to include two specific dependencies.

The validation API artifact from Java X validation and the hibernate validator as a dependency, that is the implementation of the validation API

```
43<dependency>
44    <groupId> javax.validation</groupId>
45    <artifactId>validation-api</artifactId>
46    <version>2.0.1.Final</version>
47</dependency>
48<dependency>
49    <groupId>org.hibernate</groupId>
50    <artifactId>hibernate-validator</artifactId>
51    <version>6.0.13.Final</version>
52</dependency>
53<dependency>
54    <groupId>jakarta.xml.bind</groupId>
55    <artifactId>jakarta.xml.bind-api</artifactId>
56    <version>2.3.2</version>
57</dependency>
```

2. Update Model Registration Class

Let's take a look at our registration object which is the model object that is bound to our form.

```

1 package com.mvc.model;
2
3 import javax.validation.constraints.Email;
4 import javax.validation.constraints.Max;
5 import javax.validation.constraints.NotEmpty;
6 import javax.validation.constraints.Size;
7
8 public class Registration {
9
10     @NotEmpty
11     @Size(min = 6, max = 50, message = "Your name should not be empty and should be between 6 and 50 characters")
12     private String name;
13
14     @NotEmpty(message = "Please enter your email")
15     @Email(message = "Your email address is not valid")
16     private String email;
17
18     @Max(value = 2, message = "You are only allowed a maximum of 2 guests")
19     private Integer numGuests;
20
21     private String gender;
22     private String department;
23     private String[] food;
24
25     public Registration() {
26     }
27
28     public String getName() {
29         return name;
30     }
31
32     public void setName(String name) {
33         this.name = name;
34     }
35

```

This registration object now looks a little different, I've specified a few different fields. I've also added annotations in order to validate individual fields in this object. Instead of having a separate first name and last name field, I have a single name field on which I've specified some validations. The name field should not be empty as specified by the **@NotEmpty** annotation. The name field should also be within certain size limits.

@NotEmpty

@Size(min = 6, max = 50, message = "Your name should not be empty and should be between 6 and 50 characters")

I indicate that using the **@Size** annotation. These annotations are part of the Java X validation API and these are implemented using the hibernate validator. These are built-in validations that are available to us.

The size validation that we have specified is that the name should be a minimum of 6 characters, a maximum of 50. And if the validation fails, the message that should be displayed is Your name should not be empty and should be between 6 and 50 characters. The next field is the field for email specification.

This is a string as well. It has the **@NotEmpty** annotation and if the field is left empty the message will be Please enter your email.

@NotEmpty(message = "Please enter your email")

There is also an **@Email** annotation which validates to see whether the email is in a valid form. And if it's an invalid email, you'll get a corresponding message,

```
@Email(message = "Your email address is not valid")
Your email address is not valid.
```

And finally, every student who attends this picnic can bring a maximum of 2 guests. I have an integer numGuest to which I've added the **@Max** validation annotation. Max value is 2, the method says you are only allowed a maximum of 2 guests.

```
@Max(value = 2, message = "You are noly allowed a maximum of 2 guests")
```

Once we specify these validations, the built-in implementations will ensure that these validations are applied to the fields of our registration object. The rest of the code remains the same.

3. Update Controller Class, RegistrationClass.java

```
RegistrationController.java
1 package com.mvc.controller;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import javax.validation.Valid;
7
8 import org.springframework.stereotype.Controller;
9 import org.springframework.ui.Model;
10 import org.springframework.validation.BindingResult;
11 import org.springframework.web.bind.annotation.ModelAttribute;
12 import org.springframework.web.bind.annotation.RequestMapping;
13 import org.springframework.web.bind.annotation.RequestMethod;
14
15 import com.mvc.model.Registration;
16
17 @Controller
18 @RequestMapping("/details")
19 public class RegistrationController {
20
21     @RequestMapping(value = "/registration", method = RequestMethod.GET)
22     public String registrationForm(Model model) {
23         Registration registration = new Registration();
24         model.addAttribute("registration", registration);
25
26         List<String> departmentList = new ArrayList<>();
27         departmentList.add("Biology");
28         departmentList.add("Mathematics");
29         departmentList.add("Chemistry");
30         departmentList.add("Arts");
31         model.addAttribute("departmentList", departmentList);
32
33         return "registration-page";
34     }
35
36     @RequestMapping(value = "/submission", method = RequestMethod.POST)
37     public String submitForm(@Valid @ModelAttribute("registration") Registration registration,
38                             BindingResult result,
39                             Model model)
40     {
41         if (result.hasErrors()) {
42             return "registration-page";
43         } else {
44             return "confirmation-page";
45         }
46     }
47
48 }
```

The first handler method, that we have mapped within this controller, is the method that displays the registration form to the student.

```
@RequestMapping(value = "/registration", method = RequestMethod.GET)
```

This corresponds to the request mapping for /registration. I've explicitly specified here that, this is a controller method that responds to GET HTTP requests.

It's good practice to have explicit request method specifications on your request mapping so that your method doesn't respond to HTTP requests that don't make sense. Now, within this registration form method, I instantiate a registration object on line 23 and add that as an attribute to my model.

This is the object that will be bound to the form and this object will be populated when a student fills in the form fields. Another change that I've made to this method is that the list of departments that are available in the drop-down are now populated on the server. I set up a department list with biology, mathematics, chemistry, and arts, and add the department list as an attribute to my model. We then render the registration page.

When a student fills up the form and hits submit, this submitForm method will be called, map to the path/details/submission.

```
@RequestMapping(value = "/submission", method = RequestMethod.POST)
```

This request is a POST request. Form submissions are usually POST requests as we have discussed before and I've explicitly mapped this handler method to the HTTP request type post. Notice, the registration object that we inject into this method. There are two annotations. The first is the **@ModelAttribute** annotation, we are familiar with this. This is what picks up the model backing the form with its fields filled in and injects that registration object to this method so that it's accessible in our controller.

The **@Valid** annotation basically tells Spring MVC that there are validators that we have specified in this registration object. And that Spring MVC should validate the form fields to make sure that they are all valid.

Now, any errors in the validation will be available in the binding result that I've also passed in as an input argument to this method.

Now, within this method I check the binding result to see whether the form has errors. This is in result.hasErrors.

If yes, we'll send the user back to the registration-page so that the user has an opportunity to fix those errors.

If there are no errors, we'll display the confirmation page to the user.

Any error messages associated with the form will be automatically sent back to the registration page.

Spring MVC does this completely transparently, you're abstracted away from this detail.

4. Update registration-page.jsp page

```

1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
4 <!DOCTYPE html>
5 <html>
6 <head>
7   <meta charset="ISO-8859-1">
8   <title>Summer Picnic Registration</title>
9   <style type="text/css">
10    .error {
11      color: red;
12      font-weight: bold;
13    }
14  </style>
15 </head>
16 <body>
17
18   <h2>Fantabulous picnic registration form</h2>
19   <form:form action="submission" modelAttribute="registration">
20     <b>Name</b><form:input path="name" /><form:errors path="name" cssClass="error" /><br/><br/>
21     <b>Email</b><form:input path="email" /><form:errors path="email" cssClass="error" /><br/><br/>
22     <b>Number of Guests : </b><form:input path="numGuests" /><form:errors path="numGuests" cssClass="error" /><br/><br/>
23     <b>Gender</b><br/>
24     <form:radio button path="gender" value="Male"/>Male<br/>
25     <form:radio button path="gender" value="Female"/>Female<br/>
26     <br/><br/>
27     <b>Department</b>
28     <form:select path="department" items="${departmentList}" />
29     <br/><br/>
30     <b>Meals</b><br/>
31     <form:checkbox path="food" value="Breakfast"/>BreakFast<br/>
32     <form:checkbox path="food" value="Lunch"/>Lunch<br/>
33     <form:checkbox path="food" value="Dinner"/>Dinner<br/>
34     <br/><br/>
35     <input type="submit" value="Submit"/>
36
37   </form:form>
38 </body>
39 </html>
40

```

The registration page looks a little different because the form fields that we are filling in are different. Remember that the path specification for every form input has to correspond to the name of a member variable in our Registration object. For all form fields where we have specified the validation checks, I've associated with those fields a form errors tag. On line 20, I have form errors path is equal to name.

```

<b>Name</b><form:input path="name" /><form:errors path="name" cssClass="error" /><br/><br/>

```

Any validation errors with the Name field will show up in this tag. I have form: errors path=email, any email validation failures, the message will be displayed within this form errors. And finally, I have a form errors for the path numGuests as well.

By default, Spring MVC will display the error message within a span tag and to that span tag we'll have the CSS class error applied. This is true for all three error messages. The rest of the form is pretty self explanatory. Just one thing to note is that the drop-down gets the list of items to display from the department list variable.

```

<form:select path="department" items="${departmentList}" />

```

This department list is what we had populated in our controller.

5. Update Confirmation Page JSP

```
confirmation-page.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4 <!DOCTYPE html>
5 <html>
6 <head>
7   <meta charset="ISO-8859-1">
8   <title>Summer Picnic Confirmation</title>
9   <style type="text/css">
10     div, li {
11       color: green;
12       padding: 6px;
13     }
14   </style>
15 </head>
16 <body>
17   <div><b>Your registration is confirmed!. Please make sure your details are correct</b></div>
18   <br/><br/>
19   <div>Name : ${registration.name}</div>
20   <div>Email : ${registration.email}</div>
21   <div>Number of guest : ${registration.numGuests}</div>
22   <div>Gender : ${registration.gender}</div>
23   <div>Department : ${registration.department}</div>
24   <div>Meals opted in : </div>
25   <ul>
26     <c:forEach var="meal" items="${registration.food}">
27       <li>${meal}</li>
28     </c:forEach>
29   </ul>
30
31 </body>
32 </html>
```

Now, that we have set up our form, let's take a quick look at the confirmation page. Confirmation page receives the registration object and prints out the details of the registration name, email, numGuests, gender, department, and food.

6. Build and Run the app

Time to build our WAR file and deploy to our Tomcat server and run our server and see how this form works.

Here's an application running on the Tomcat server. Let's click here to register for the Fantabulous University Summer Picnic Program.

Fantabulous University Summer Picnic Program

[Click here to register.](#)

Here's our form, I'm going to go ahead and fill in the form name, email, number of guests, gender, male, department, mathematics, and I'm going to opt for lunch for James Watson.

Fantabulous picnic registration form

Name

Email

Number of Guests :

Gender
☒ Male
☐ Female

Department

Meals
☒ Breakfast
☐ Lunch
☒ Dinner

Click on Submit,

Your registration is confirmed!. Please make sure your details are correct

Name : James Watson
Email : james@my-mail.com
Number of guest : 2
Gender : Male
Department : Mathematics
Meals opted in :

- Breakfast
- Dinner

this is a perfectly valid form. There are absolutely no errors, which means when the form is submitted the handler method that is invoked, that is the /submission, will check to see whether the binding result has errors. If there are no errors, then we are taken to the confirmation page and the registration object is passed on to the confirmation page. We get to the confirmation page only if the form is completely valid.

Now, let's go back to our form once again and this time around, we'll fill in the form with a few invalid values.

I fill in the details, but notice that I've left the Name field in the form empty.

Fantabulous picnic registration form
Name
Email
Number of Guests :
Gender
☒ Male
☐ Female
Department
Meals
☒ BreakFast
☐ Lunch
☒ Dinner

Now, when I try to submit this form, the validation annotations that we've applied to the registration object will kick in. When you hit submit, you'll immediately see a few error messages. Next to name there are two error messages, because we have two separate validators. The first error method says, must not be empty. And that corresponds to the `@NotEmpty` annotation on our string name member variable.

Fantabulous picnic registration form
Name **Your name should not be empty and should be between 6 and 50 characters must not be empty**
Email **Please enter your email**
Number of Guests :
Gender
☒ Male
☐ Female
Department
Meals
☒ BreakFast
☐ Lunch
☒ Dinner

Must not be empty is the default message for the `@NotEmpty` annotation. The second error message that says **your name should not be empty and should be between 6 and 50 characters** comes from our `@Size` validation on the name field. The message displayed is the custom error message that we have configured on the `@Size` annotation.

Please enter your email. This is the message that corresponds to the **@NotEmpty** validation on the email member variable in our registration object.

I'm going to fill in the name field of the form, but I'll just type in James, which does not meet our size requirement. In that case, you only see one error message.

Fantabulous picnic registration form
Name **Your name should not be empty and should be between 6 and 50 characters**
Email
Number of Guests :
Gender
☒ Male
☐ Female
Department
Meals
☒ BreakFast
☐ Lunch
☒ Dinner

The name field is not empty, so the first error message has disappeared, but we still get the error message from the **@Size** validation annotation.

Let's put in an invalid email. I'll only type in *james@*, this is clearly not valid. When I hit Submit now my error message has changed.

Fantabulous picnic registration form
Name
Email **Your email address is not valid**
Number of Guests :
Gender
☒ Male
☐ Female
Department
Meals
☒ BreakFast
☐ Lunch
☒ Dinner

It says *Your email address is not valid*. This is the validation performed by the **@Email** annotation that we've applied to the email string field. The **@Email** validator checks for a valid email address, *James@* isn't valid, that's why we get our custom message.

I'm going to quickly re-enter the data in this form and perform one last validation check. I'm going to enter the number of guests as 6, which is clearly above the limit.

When I hit Submit,

Fantabulous picnic registration form
Name
Email
Number of Guests : *You are noly allowed a maximum of 2 guests*
Gender
☒ Male
☐ Female
Department
Meals
☒ BreakFast
☐ Lunch
☒ Dinner

I get the message, *you are only allowed a maximum of 2 guests*. This is from the **@Max** annotation that we've applied on the *numGuests* variable in the registration object.

Uploading the Files in Spring MVC App

In this demo, we'll see how you can perform file uploads in your Spring MVC application. File uploads in any app are multipart requests, multipart requests combine one or more sets of data into a single body separated by boundaries. Multipart requests are commonly used for uploading files, as we'll do now, or for transferring large chunks of a data from one location to another. Multipart requests allows you to transfer several kinds of data as a part of the same request.

1. Adding new dependency on POM.xml for handling multipart request (upload file) implementation

commons-fileupload:1.3.1

```
58 <dependency>
59   <groupId>commons-fileupload</groupId>
60   <artifactId>commons-fileupload</artifactId>
61   <version>1.3.1</version>
62 </dependency>
63 </dependencies>
```

This is what allows developers to set up their own multipart implementation if they want to. There are several multipart resolvers out there which work well with Spring MVC and we'll use one of them. The most common of all is the commons-fileupload. Notice that I've added this as a dependency in my pom.xml file.

2. Adding Multipart resolver bean in spring-servlet.xml

Multipart resolver needs to be injected into our Spring MVC application as a bean.

```
spring-servlet.xml
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:context="http://www.springframework.org/schema/context"
5   xmlns:mvc="http://www.springframework.org/schema/mvc"
6   xsi:schemaLocation="http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
8     http://www.springframework.org/schema/mvc
9     http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd
10    http://www.springframework.org/schema/context
11    http://www.springframework.org/schema/context/spring-context-4.0.xsd">
12
13   <bean id="viewResolver"
14     class="org.springframework.web.servlet.view.InternalResourceViewResolver" >
15     <property name="prefix" value="/WEB-INF/jsp/" />
16     <property name="suffix" value=".jsp" />
17   </bean>
18
19   <bean id="multipartResolver"
20     class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
21     <property name="maxUploadSize" value="20971520" />
22     <property name="maxInMemorySize" value="1048576" />
23   </bean>
24
25 </beans>
```

The bean with ID multipartResolver which references the class CommonsMultipartResolver from the commons-fileupload jar. The Spring MVC allows you to plug in your own implementation of the multipart resolver.

I've specified two properties, the maximum upload size in bytes and the max size of the file in memory also in bytes. Just set these to some sensible values which makes sense for your use case.

3. Update index.jsp to reference a new link for upload page

```

1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6   <meta charset="ISO-8859-1">
7   <title>Summer Picnic</title>
8 </head>
9 <body>
10  <h2>Fantabulous University Summer Picnic Program</h2>
11  <a href="details/registration">Click here to register.</a>
12
13  <h2>File Upload</h2>
14  <br/>
15  <a href="uploadForm">click to upload</a>
16 </body>
17 </html>

```

The UI entry point of our application, index.jsp. This is very straightforward, there is a single href a link which says Click to Upload Image.

4. Create new upload-page.jsp file under (/WEB-INF/jsp)

```

1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
4 <!DOCTYPE html>
5 <html>
6 <head>
7   <meta charset="ISO-8859-1">
8   <title>File Upload Page</title>
9 </head>
10 <body>
11
12   <h3 style="color:green">${filesuccess}</h3>
13   <form:form method="post" action="saveFile" enctype="multipart/form-data">
14     <p><label for="image">Choose file to upload</label></p>
15     <p><input name="file" id="fileToUpload" type="file" /></p>
16     <p><input type="submit" value="Upload" /></p>
17   </form:form>
18
19 </body>
20 </html>

```

A filesuccess message is populated only if the upload has gone through successfully, otherwise no message will be displayed.

This contains the form that allows us to upload our file to our Spring MVC app. Notice the form definition on line 14, **method="post"** here explicitly specifies that the submission of this form is a post request made to our server. The action performed is saveFile, this is the handler mapping that we'll set up in our controller file.

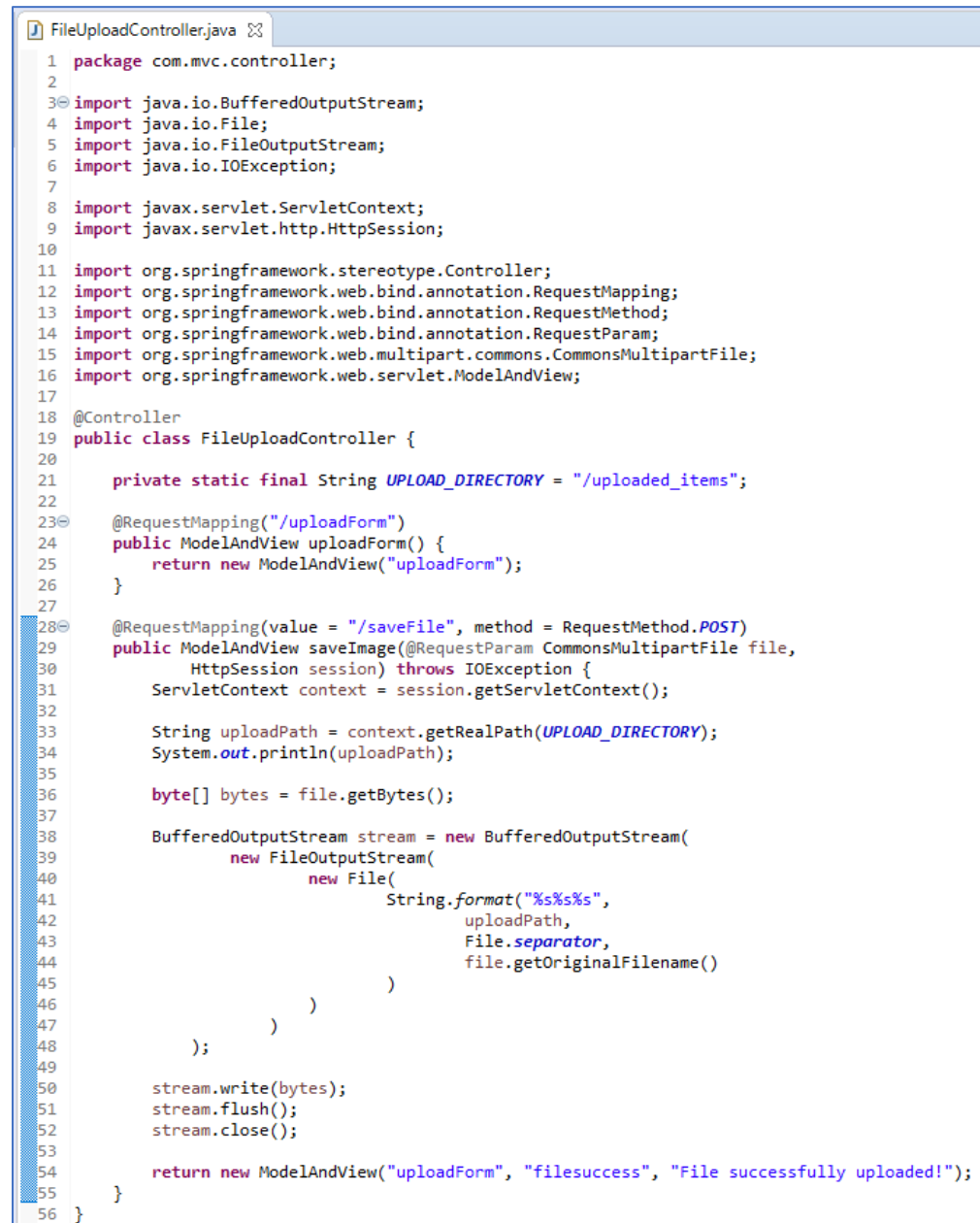
```
<form:form method="post" action="saveFile" enctype="multipart/form-data">
```

Notice that the encoding type specified on the form is **multipart/form-data**. This is what tells the form that we are planning to transfer files using this form. There is multipart data involved which needs to be transferred to the server. On line 14, we have a label which simply says Choose File to Upload. On line 15 is the actual input that accepts a file.

Notice the input type="file", forget the name and the ID that's not really important here. Setting the input type to file will display a button that will allow us to bring up an explorer or finder window on our machine to locate the files that we want to upload.

And finally, we have the submit button, which we'll hit to actually submit the form and upload the file to our server.

5. Create new Controller File to Handle upload process



```
1 package com.mvc.controller;
2
3 import java.io.BufferedOutputStream;
4 import java.io.File;
5 import java.io.FileOutputStream;
6 import java.io.IOException;
7
8 import javax.servlet.ServletContext;
9 import javax.servlet.http.HttpSession;
10
11 import org.springframework.stereotype.Controller;
12 import org.springframework.web.bind.annotation.RequestMapping;
13 import org.springframework.web.bind.annotation.RequestMethod;
14 import org.springframework.web.bind.annotation.RequestParam;
15 import org.springframework.web.multipart.commons.CommonsMultipartFile;
16 import org.springframework.web.servlet.ModelAndView;
17
18 @Controller
19 public class FileUploadController {
20
21     private static final String UPLOAD_DIRECTORY = "/uploaded_items";
22
23     @RequestMapping("/uploadForm")
24     public ModelAndView uploadForm() {
25         return new ModelAndView("uploadForm");
26     }
27
28     @RequestMapping(value = "/saveFile", method = RequestMethod.POST)
29     public ModelAndView saveImage(@RequestParam CommonsMultipartFile file,
30         HttpSession session) throws IOException {
31         ServletContext context = session.getServletContext();
32
33         String uploadPath = context.getRealPath(UPLOAD_DIRECTORY);
34         System.out.println(uploadPath);
35
36         byte[] bytes = file.getBytes();
37
38         BufferedOutputStream stream = new BufferedOutputStream(
39             new FileOutputStream(
40                 new File(
41                     String.format("%s%s%s",
42                         uploadPath,
43                         File.separator,
44                         file.getOriginalFilename()
45                     )
46                 )
47             )
48         );
49
50         stream.write(bytes);
51         stream.flush();
52         stream.close();
53
54         return new ModelAndView("uploadForm", "filesuccess", "File successfully uploaded!");
55     }
56 }
```

The FileUploadController tag using the @Controller annotation. I want all of my files to be uploaded to a specific directory which is the uploaded_items directory, just under my root directory.

The first handler method that I have mapped is to the path /uploadForm, this just displays the uploadForm.jsp file.

The second handler method that we have mapped is what handles the submission from our form containing the file upload. The RequestMapping is to the /saveFile path and the method is RequestMethod.POST.

Remember submitting form data is always a POST request. There are two input arguments to this method that accepts an uploaded file and saves it out on our server.

```
@RequestMapping(value = "/saveFile", method = RequestMethod.POST)  
public ModelAndView saveImage(@RequestParam CommonsMultipartFile file,  
                               HttpSession session) throws IOException {
```

The first is the file itself, it's of type CommonsMultipartFile, and I inject it using the @RequestParam annotation.

The second input argument is the current HttpSession. An HttpSession is typically used to maintain information about a client across multiple requests that a client makes. We'll use this HttpSession to get a servlet context. session.getServletContext gives us the current servlet context of this request. The ServletContext you can think of as containing methods that allows you to communicate with the servlet container. And we'll use the servlet context to get a path to which the file needs to be saved.

```
ServletContext context = session.getServletContext();
```

Now, once we have the servlet context, we can call getRealPath and specify the directory where we want the files to be uploaded and saved.

```
String uploadPath = context.getRealPath(UPLOAD_DIRECTORY);
```

This getRealPath will give us a complete path on our server. Remember, this files need to be saved on the server. And if you want to debug this code, you can always use a system.out.println statement to print out the upload path.

This path will then be printed out on the terminal window where you run your Apache server. Next, I convert the file to bytes, and then

```
byte[] bytes = file.getBytes();
```

I use a BufferedOutputStream to write out this file. We'll use the original name of the file where we save it out to our upload directory. File.getOriginalFilename is what I use to access the original name of the file. We'll write out the buffered output stream, flush it, close it, and then return a file success message.

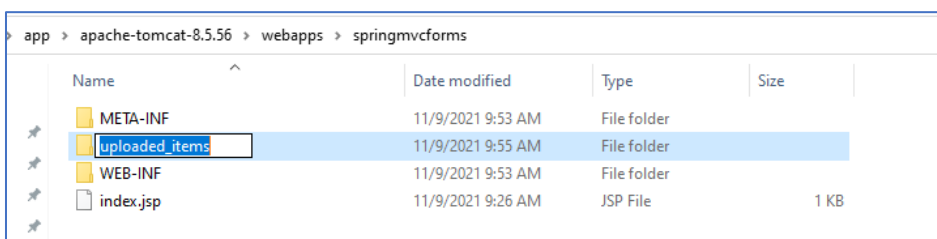
This controller sends us back to the upload form as you can see from the return statement. And it also passes back a file success message saying file successfully uploaded.

6. Build And Run application

Build the Maven WAR file and deploy this WAR to your Tomcat server. Once your application is running, you should see this page this is our index.jsp file. Click to upload image.

Fantabulous University Summer Picnic Program
[Click here to register.](#)
File Upload
[click to upload](#)

Before I start uploading files, I need to create the folder that will hold my uploaded files, the uploaded items folder that my controller code expects. For this, I'll go to my Tomcat installation and go to the webapps folder under which there is my SpringMvcSimpleApp.



This is the folder corresponding to the application that I've just deployed. Within this folder, I'm going to create an **uploaded_items** folder. You run an ls command here, you'll see that the uploaded_items folder is at the same level as the web inf folder right next to the index.jsp. I'll cd into this folder and show you that there are no items here.

Back to our browser, and let's upload a few files. I'm going to click through and the upload form will be displayed.

Choose file to upload

No file chosen

I'm going to select the Choose file button and this will bring up a finder window on my local machine. I'll select the file that I want uploaded, an image of this cute dog. Once I've selected my file, I have to explicitly hit the Upload button.

So this file is uploaded to my server. I get the message which says file successfully uploaded. Let's head over to our uploaded_items folder and you can see our file is present there. This is part of our server. Here I am within my Tomcat server installation.

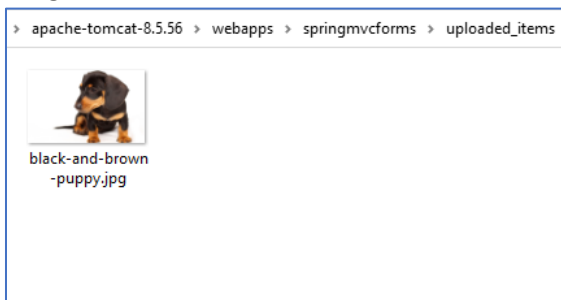
On my finder window, I'm going to expand all of the sub folders here, till I find the folder corresponding to my SpringMvcSimpleApp.

File successfully uploaded!

Choose file to upload

No file chosen

Under the webapps directory, I have an uploaded_items directory. And there I can find my black-and-brown-puppy.jpg file. Double-click on this file. And you'll see that this is the same image that we downloaded from our local machine to our server application.



Uploading Using the Servlet Context Aware Controller

In this demo, we'll see another example of file upload in a Spring MVC application using a ContextAware controller. So rather than extracting the CurrentServletContext from the HTTP session that we inject, we will use a ContextAware controller which will have the ServletContext injected in automatically. We'll continue to depend on the commons-fileupload library, which is what we'll use for our file upload.

pom.xml

```
58 <dependency>
59   <groupId>commons-fileupload</groupId>
60   <artifactId>commons-fileupload</artifactId>
61   <version>1.3.1</version>
62 </dependency>
```

at our SpringServlet XML file, we have set the multipartResolver bean to the CommonsMultipartResolver, which is part of Apache Commons.

```
19 <bean id="multipartResolver"
20   class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
21   <property name="maxUploadSize" value="20971520" />
22   <property name="maxInMemorySize" value="1048576" />
23 </bean>
```

1. Update index.jsp file as follows

```
<h2>Upload Single File</h2><br/>
<input type="button" onclick="location.href='uploadSingleFile'"
style="background-color: #008080;
color: black;
width: 150px; height: 40px; top: 40px;
"
value="Single File Upload">
```

2. Create uploadSingleFile.jsp under WEB/INF/jsp page

```
uploadSingleFile.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4 <%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
5 <!DOCTYPE html>
6 <html>
7 <head>
8   <meta charset="ISO-8859-1">
9   <title>Upload file</title>
10 </head>
11 <body>
12
13   <form method="POST" action="uploadSingleFile" enctype="multipart/form-data">
14     <p>Choose File. <input type="file" name="file"></p>
15     <p>File Name : <input type="text" name="name">
16     <input type="submit" value="upload">
17   </form>
18
19 </body>
20 </html>
```

Method is equal to post form submission actions are always POST requests made to our controller. Action is the same uploadSingleFile that accepts a POST request. The encoding type is multipart/form-data. I have an input type= file, which says Choose File.

3. Create Controller File

```

1 package com.mvc.controller;
2
3 import java.io.BufferedOutputStream;
4 import java.io.File;
5 import java.io.FileOutputStream;
6 import javax.servlet.ServletContext;
7
8 import org.springframework.stereotype.Controller;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RequestMethod;
11 import org.springframework.web.bind.annotation.RequestParam;
12 import org.springframework.web.bind.annotation.ResponseBody;
13 import org.springframework.web.context.ServletContextAware;
14 import org.springframework.web.multipart.MultipartFile;
15 import org.springframework.web.servlet.ModelAndView;
16
17 @Controller
18 public class SingleFileUploadController implements ServletContextAware {
19
20     private static final String UPLOAD_DIRECTORY = "/uploaded_items/";
21
22     private ServletContext servletContext;
23
24     @Override
25     public void setServletContext(ServletContext servletContext) {
26         this.servletContext = servletContext;
27     }
28
29     @RequestMapping(value = "/uploadSingleFile", method = RequestMethod.GET)
30     public ModelAndView uploadSingleFile() {
31         return new ModelAndView("uploadSingleFile");
32     }
33
34     @RequestMapping(value = "/uploadSingleFile", method = RequestMethod.POST)
35     public @ResponseBody String uploadSingleFileHandler(
36         @RequestParam("name") String fileName,
37         @RequestParam("file") MultipartFile file
38     ) {
39         if (!file.isEmpty()) {
40             try {
41                 byte[] bytes = file.getBytes();
42                 String pathToFile = servletContext.getRealPath("/") + UPLOAD_DIRECTORY + fileName;
43
44                 BufferedOutputStream stream =
45                     new BufferedOutputStream(new FileOutputStream(new File(pathToFile)));
46
47                 stream.write(bytes);
48                 stream.flush();
49                 stream.close();
50
51                 return String.format("File successfully uploaded %s!", fileName);
52             } catch (Exception ex) {
53                 return String.format("Failed to upload %s", fileName);
54             }
55         } else {
56             return String.format("Failed to upload, File %s is empty", fileName);
57         }
58     }
59 }
60

```

Notice that our `SingleFileUploadController` has the `@Controller` annotation and it also implements the `ServletContextAware` interface.

```

@Controller
public class SingleFileUploadController implements ServletContextAware

```

Implementing this interface requires that you override the `setServletContext` method, and provide an implementation.

```
@Override
public void setServletContext(ServletContext servletContext) {
    this.servletContext = servletContext;
}
```

setServletContext accepts as an input argument the servletContext which I've assigned to the servletContext member variable in this class. When your controller implements the servletContextAware interface, and has a setServletContext implementation, the servletContext is automatically injected by Spring and becomes part of this controller. So you can access the servletContext directly instead of getting it from the HTTP session.

```
@RequestMapping(value = "/uploadSingleFile", method = RequestMethod.GET)
public ModelAndView uploadSingleFile()
```

The first handler method that I have mapped within this controller. The RequestMapping is to the path, uploadSingleFile, and this handler mapping response to the GET HTTP request. The name of the method is uploadSingleFile, and all this method does is display the uploadSingleFile view. The JSP file which contains the button, allowing us to upload files. the method responded to GET HTTP requests, and it only displays the form which we use for file upload.

```
@RequestMapping(value = "/uploadSingleFile", method = RequestMethod.POST)
public @ResponseBody String uploadSingleFileHandler( ... )
```

The second handler mapping, and this request mapping is also to the same path uploadSingleFile. How is it possible to have two mappings to the same path? our request mappings are both to the uploadSingleFile path. The difference is, that both of our handler mapping methods respond to different kinds of HTTP requests. method responds to a POST request. This is the method invoked when we submit our form. You can have the same path mapping for multiple methods, so long as the methods respond to different kinds of HTTP requests. The second handler method is what is invoked when we actually hit Submit on our form to upload the file that we have selected, the uploadSingleFileHandler. Notice that I have tagged its return type with **@ResponseBody**. The **@ResponseBody** annotation on the return value from this method, binds whatever we return from this method as a web response that will be displayed to the user. The return value from this method is of type String as you can see just next to the **@ResponseBody** annotation.

Whatever string we return from this method will be displayed in the web page to the user, that is the ResponseBody. You can see that the uploadSingleFileHandler method takes in two input arguments.

```
@RequestParam("name") String fileName,
@RequestParam("file") MultipartFile file
```

The first is the name of the file that we want to use to store the file on our server. And the second is the file itself, which is of type MultipartFile.

The MultipartFile is an interface specification for any implementation. And as you know, we've chosen to use the Apache Commons implementation. Both of these are tagged using the

@RequestParam annotation, indicating that this information needs to be extracted from the request parameters, name, and file respectively.

The code for the actual fileUpload and saving out on our server, is the same as what we've seen before. More or less, we check whether the file is empty, if the file is not empty,

we perform the upload within a try catch loop. We convert the file to its byte format using `file.getBytes`, and we get the path to where the file needs to be uploaded using the `servletContext`. We call `servletContext.getRealPath` to get the path of the root directory of the application.

```
String pathToFile = servletContext.getRealPath("/") + UPLOAD_DIRECTORY +  
fileName;
```

We concatenate to this the `UPLOAD_DIRECTORY` plus the name that we want to assign the file when we save it out, that is in the filename variable. Once we have the path where we want the file to be written out, we simply use a `BufferedOutputStream` called `stream.write`, close the stream, and then return the string you have successfully uploaded this filename.

If there is an exception, we return the string You failed to upload this filename, and we also print out the message of the exception.

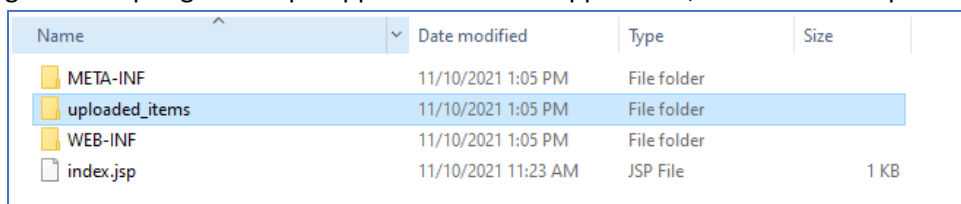
Or if the file was not uploaded because the file was empty, this is where we move to the else block of our code, we'll return,

You failed to upload filename because the file was empty. Now all of these string responses that you can see in our return statements, these string responses will be rendered as a web response to the user, thanks to our `@ResponseBody` annotation on the return type.

4. Build and Run the App

Let's go ahead and build our WAR file and deploy this application to the Tomcat server.

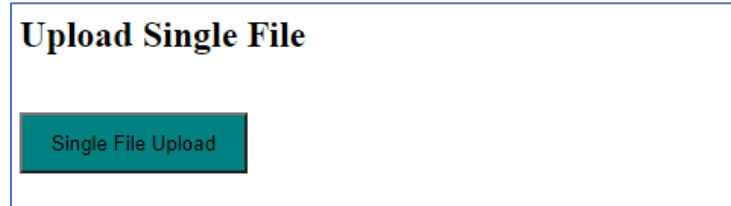
Now after you've deployed the application, but before you use the application, make sure you go to the `SpringMvcSimpleApp` under the `webapps` folder, and create an `uploaded_items` folder.



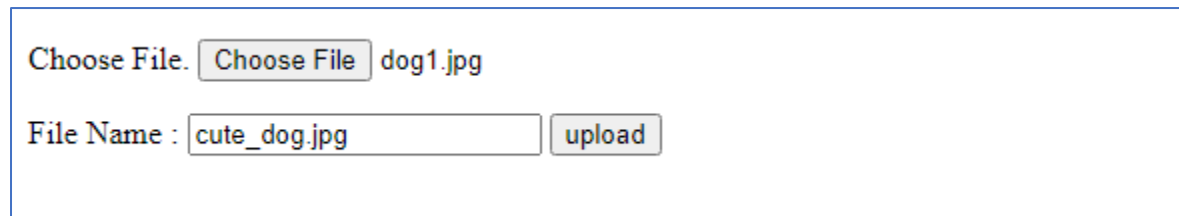
Name	Date modified	Type	Size
META-INF	11/10/2021 1:05 PM	File folder	
uploaded_items	11/10/2021 1:05 PM	File folder	
WEB-INF	11/10/2021 1:05 PM	File folder	
index.jsp	11/10/2021 11:23 AM	JSP File	1 KB

This `uploaded_items` folder is where you will save the uploaded files. This `uploaded_items` folder needs to exist on your server, and files will be placed there. This folder is currently empty, but we'll soon fix that. Let's

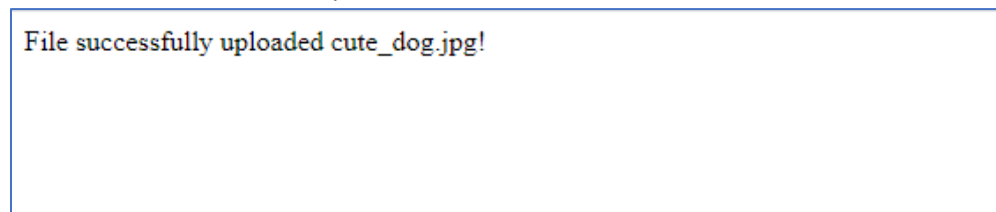
head over to our application. And here is our index.jspFile with the button for Single File Upload. Clicking on this button takes us to our Upload page.



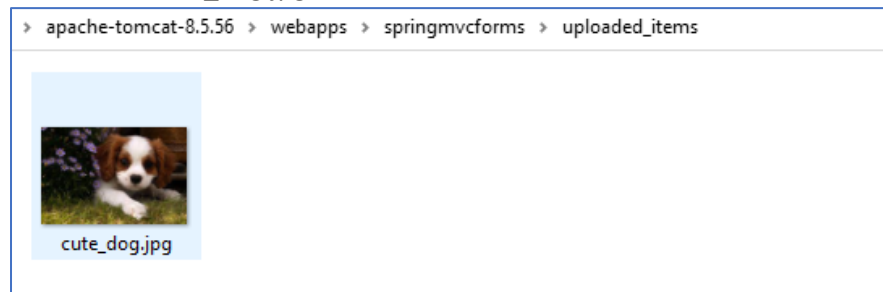
Here we have a button that allows us to choose the file that we want to upload. It'll also specify a name for this file.



I'm going to select this dog1.jpg file I have on my machine, click on Open, and I'm going to specify a name cute_dog.jpg. This is the name that will be used to save this file on my server. I get this string message here, You successfully uploaded cute_dog.jpg. If you remember, this was the return string from our controller after we had completed upload, this return string is now available to us as a web response.



Let's confirm that our file is available on the server. Open uploaded_items folder. You can see cute_dog.jpg is present here. I'm going to head over to the finder window, in order to view this image, I'm going to in the folder where you have apache-tomcat installed and running, you should find a folder for your web app. And under that, we have the uploaded items folder, and there is our cute_dog.jpg.



We've successfully uploaded a single file using a servletContextAware controller. Along the way we also understood the use of the @ResponseBody annotation.

Uploading Multiple Files

In this demo, we'll add a little bit of functionality to the application that we had built in the previous demo for single file upload. We'll add the ability to upload multiple files at the same time.

1. Update index.jsp

```
<h2>Upload Multiple File</h2><br/>
<input type="button" onclick="location.href='uploadMultipleFiles'"
style="background-color: #008080;
      color: black;
      width: 150px; height: 40px; top: 40px;
      "
value="Multiple File Upload">
```

The button is to upload multiple files onClick, it'll go to the location href=uploadMultipleFiles.

2. Create new JSP Page, uploadMultipleFiles.jsp under /WEB-INF/jsp

```
uploadMultipleFiles.jsp
1  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2    pageEncoding="ISO-8859-1"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6    <meta charset="ISO-8859-1">
7    <title>Upload Multiple File</title>
8
9    <form method="POST" action="uploadMultipleFiles" enctype="multipart/form-data">
10      <p>choose 1st file <input type="file" name="file"></p>
11      <p>1st file name : <input type="text" name="name"></p>
12
13      <p>choose 2nd file <input type="file" name="file"></p>
14      <p>2nd file name : <input type="text" name="name"></p>
15
16      <p>choose 3th file <input type="file" name="file"></p>
17      <p>3th file name : <input type="text" name="name"></p>
18
19      <input type="submit" value="upload">
20    </form>
21  </head>
22  <body>
23
24  </body>
25  </html>
```

This is the form that allows us to upload multiple files.

The form method is POST. The form action is maps to upload multiple files. The encoding type is of course multipart/form-data. This form allows us to upload three files at a time. So I have set up three file chooser dialog buttons, and three input boxes, allowing us to specify the names of the file when it's uploaded and stored on our server.

3. Create MultipleFileUploadController.java

```

1 package com.mvc.controller;
2
3 import java.io.BufferedOutputStream;
4 import java.io.File;
5 import java.io.FileOutputStream;
6 import javax.servlet.ServletContext;
7
8 import org.springframework.stereotype.Controller;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RequestMethod;
11 import org.springframework.web.bind.annotation.RequestParam;
12 import org.springframework.web.bind.annotation.ResponseBody;
13 import org.springframework.web.context.ServletContextAware;
14 import org.springframework.web.multipart.MultipartFile;
15 import org.springframework.web.servlet.ModelAndView;
16
17 @Controller
18 public class MultipleFileUploadController implements ServletContextAware {
19
20     private static final String UPLOAD_DIRECTORY = "/uploaded_items/";
21     private ServletContext servletContext;
22
23     @Override
24     public void setServletContext(ServletContext servletContext) {
25         this.servletContext = servletContext;
26     }
27
28     @RequestMapping(value = "/uploadMultipleFiles", method = RequestMethod.GET)
29     public ModelAndView uploadMultipleFiles() {
30         return new ModelAndView("uploadMultipleFiles");
31     }
32
33     @RequestMapping(value = "/uploadMultipleFiles", method = RequestMethod.POST)
34     @ResponseBody String uploadMultipleFilesHandler(
35         @RequestParam("name") String[] fileNames,
36         @RequestParam("file") MultipartFile[] files
37     ) {
38         if (files.length != fileNames.length)
39             return "Required information missing";
40         String message = "";
41         for (int i=0; i<files.length; i++) {
42             MultipartFile file = files[i];
43             String filename = fileNames[i];
44             try {
45                 byte[] bytes = file.getBytes();
46                 String pathToFile = servletContext.getRealPath("/") + UPLOAD_DIRECTORY + filename;
47                 BufferedOutputStream stream =
48                     new BufferedOutputStream(new FileOutputStream(new File(pathToFile)));
49
50                 stream.write(bytes);
51                 stream.flush();
52                 stream.close();
53
54                 message += String.format("file %s is successfully uploaded<br/>", filename);
55             } catch (Exception ex) {
56                 return String.format("Failed to upload file %s, %s", filename, ex.getMessage());
57             }
58         }
59         return message;
60     }
61 }

```

The file upload controller implements the `ServletContextAware` interface and we have overridden the `setServletContext` method. The `ServletContext` will be injected into this controller.

The `UPLOAD_DIRECTORY` continues to be uploaded items.

```

@RequestMapping(value = "/uploadMultipleFiles", method = RequestMethod.GET)
public ModelAndView uploadMultipleFiles()

```

The first method is mapped to the path `uploadMultipleFiles`, and it responds to a get HTTP request as well. This methods display our upload forms to screen.

```
@RequestMapping(value = "/uploadMultipleFiles", method = RequestMethod.POST)  
public @ResponseBody String uploadMultipleFilesHandler( ... )
```

The 2nd method that will be invoked when we want to upload multiple files. This is the uploadMultipleFiles handler. And the request mapping is to the path uploadMultipleFiles and it responds to the HTTP POST method. There are two request parameters that we inject, but both of these request parameters are of type array.

```
@RequestParam("name") String[] fileNames,  
@RequestParam("file") MultipartFile[] files
```

So we have a string array of file names, and a multi part file array of files. And really this is the main difference between the two methods. We have an array of files and corresponding filenames, we'll iterate over this array and write out one file at a time in this method. We'll first do a sanity check to see if the files length is the same as the length of the filenames, otherwise we'll return required information missing.

```
if (files.length != fileNames.length)  
    return "Required information missing";
```

We then run a for loop starting from 0, up to the length of files that we have uploaded to the server.

```
for (int i=0; i<files.length; i++) {  
    MultipartFile file = files[i];  
    String filename = fileNames[i];
```

For each iteration of the for loop, we'll extract the file and the corresponding file name. The actual process of writing out each file with the new file name specified remains the same. We get the bytes representation of the file and use a BufferedOutputStream in order to perform this write operation.

```
byte[] bytes = file.getBytes();  
String pathToFile = servletContext.getRealPath("/") + UPLOAD_DIRECTORY +  
filename;  
BufferedOutputStream stream =  
    new BufferedOutputStream(new FileOutputStream(new File(pathToFile)));  
  
stream.write(bytes);  
stream.flush();  
stream.close();
```

For every file that is successfully written out without exceptions, we'll print out, you successfully uploaded this file name to a message variable that is then returned.

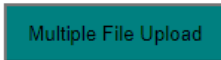
4. Build and Run The App

Build the WAR file and deploy the WAR file to your Tomcat server.

Here within my web apps and SpringMvcSimpleApp folder, I'm going to create an uploaded_items directory where my files uploaded will be stored. Remember to create this directory upfront, this directory is initially empty.

Here is our application allowing multiple file upload.

Upload Multiple File



Let's quickly test out multiple file upload.




choose 1st file dog1.jpg
1st file name :
choose 2nd file dog2.jpg
2nd file name :
choose 3th file dog3.jpg
3th file name :

We'll check all of the uploaded files in one go. It's pretty clear what you have to do here, click on the Choose File button, select a different file from your local machine and give each file a name. I do this for three different files. So I have three files that I'll upload as a part of a single request. The three files are dog1, dog2, and the dog3. And the names for these three files on my server will be cute_dog, adorable_dog, and lovely_dog. Go ahead and hit the Upload button and I see this gratifying message indicating that I've successfully uploaded all three files.

file cute_dog.jpg is successfully uploaded
file adorable_dog.jpg is successfully uploaded
file lovely_dog.jpg is successfully uploaded

Let's head over to our server directory in order to confirm that these files have been uploaded.

» > apache-tomcat-8.5.56 > webapps > springmvcforms > uploaded_items

Name	Date	Type	Size	Tags
 adorable_dog.jpg	11/11/2021 1:38 PM	JPG File	5 KB	
 cute_dog.jpg	11/11/2021 1:38 PM	JPG File	10 KB	
 lovely_dog.jpg	4/14/2019 3:27 PM	JPG File	208 KB	

In the uploaded items folder shows us that there are three JPEG files here. And if you want to ensure that these files are present, you can head over to your Explorer window and double click on these JPEG files to ensure that the upload has taken place successfully.

Uploading Multiple Files Using Multiple Selections

In this demo, we'll continue working with multiple file upload, but we'll learn a couple of new things. The first thing is, we'll use a form input type that allows selection of multiple files. And all of those files will be uploaded onto our server. Instead of having our controller handle just a fixed number of files for upload, we'll see how we can upload any number of files, either one file, two files, or n files. We'll do this by having a Java object acting as the model bound to our form.

1. Create Model Bound Class for our form

Our model object is called upload data. And this is the class for upload data in the package `com.mvc.model`.

```
UploadData.java
1 package com.mvc.model;
2
3 import java.util.List;
4
5
6
7 public class UploadData {
8
9     private List<MultipartFile> images;
10
11     public List<MultipartFile> getImages() {
12         return images;
13     }
14
15     public void setImages(List<MultipartFile> images) {
16         this.images = images;
17     }
18
19 }
```

This upload data contains a member variable that is a list of images. This is a list of multipart file objects. The code for the model object is very simple. We have a getter method, `getImages`, that returns a list of images, and a setter method, `setImages`, which we can use to set a list of images that is multipart files on this object.

2. Create Multiple Selection File Upload Controller

We'll now need to update our controller in order to use this model object. Here is the Controller.

```

MultipleSelectionFileUploadController.java
1 package com.mvc.controller;
2
3 import java.io.File;
4 import java.io.IOException;
5 import java.util.ArrayList;
6 import java.util.List;
7
8 import javax.servlet.http.HttpServletRequest;
9
10 import org.springframework.stereotype.Controller;
11 import org.springframework.ui.Model;
12 import org.springframework.web.bind.annotation.ModelAttribute;
13 import org.springframework.web.bind.annotation.RequestMapping;
14 import org.springframework.web.bind.annotation.RequestMethod;
15 import org.springframework.web.multipart.MultipartFile;
16
17 import com.mvc.model.UploadData;
18
19 @Controller
20 public class MultipleSelectionFileUploadController {
21
22     @RequestMapping(value = "/save-upload", method = RequestMethod.POST)
23     public String uploadResource(
24         HttpServletRequest servletRequest,
25         @ModelAttribute UploadData upload,
26         Model model
27     ) {
28
29         List<MultipartFile> files = upload.getImages();
30         List<String> fileNames = new ArrayList<>();
31
32         if (null != files && files.size() > 0) {
33             for (MultipartFile multipartFile : files) {
34                 String fileName = multipartFile.getOriginalFilename();
35                 fileNames.add(fileName);
36                 File imageFile = new File (servletRequest.getServletContext().getRealPath("/images"), fileName);
37                 try {
38                     multipartFile.transferTo(imageFile);
39                 } catch (IOException e) {
40                     e.printStackTrace();
41                 }
42             }
43             model.addAttribute("upload", upload);
44
45             return "viewUploadDetail";
46         }
47     }
48
49     @RequestMapping(value = "/uploadMultipleSelectionForm", method = RequestMethod.GET)
50     public String displayForm(Model model) {
51         model.addAttribute("upload", new UploadData());
52         return "uploadMultipleSelectionForm";
53     }
54 }
55

```

I've gotten rid of the implementation of the ServletContextAware interface. This no longer a ServletContextAware. And that's totally fine, we can work with this as well. Here is the method uploadResources, which is invoked when multiple files have to be uploaded onto our server. The @RequestMapping annotation specifies the path /save-upload.

```
@RequestMapping(value = "/save-upload", method = RequestMethod.POST)
```

This is the path to which a request is made when we hit Submit on our form in the UI, we'll see that in a bit. And the HTTP request type that this method responds to is the HTTP POST request, request method is POST. This method which handles the upload of any number of files, one plus files takes in three input arguments.

```
public String uploadResource(  
    HttpServletRequest servletRequest,  
    @ModelAttribute UploadData upload,  
    Model model  
)
```

The first input argument is the HttpServletRequest, which we'll use to access the servlet context. The second input argument is the UploadData object which I've tagged using the **@ModelAttribute annotation**. This is the upload data, which is the model that holds the files that need to be uploaded.

The third input argument is the model that we'll use to return information to the view.

Let's now take a look at the code. We first access all of the files, which need to be uploaded using upload.getImages.

```
List<MultipartFile> files = upload.getImages();
```

This will return a list of multipart file objects which contains all of the files for upload. Notice that this is a list, which means the number of elements in this list is not bounded. You can upload as many files as our server will support. Every file will be saved on the server using the original file name. I'll extract the file names from the individual files and place it in a file names array list which I instantiate on next line.

```
List<String> fileNames = new ArrayList<>();
```

I then have an if check to ensure that files is not equal to null, and the number of files that is files.size is greater than 0. If there are indeed files to upload, we'll run a for each loop, for multipartFile in files. For every file, we'll get the original file name and store it in the fileName string. And we'll also add it to the list of file names, this I do on this line.

```
for (MultipartFile multipartFile : files) {  
    String fileName = multipartFile.getOriginalFilename();
```

Then for every file,

```
fileNames.add(fileName);
```

I instantiate a file object using the servlet context from the HttpRequest.

```
File imageFile = new File  
(servletRequest.getServletContext().getRealPath("/images"), fileName);
```

In this demo, I'm going to save all uploaded files to the **/images** path. If you're on a Windows machine, the path separator that you specify will be a little different. So just be mindful of that. You might want to use the backslash instead. This time around instead of using the buffered output stream in order to write out the contents of the file, I can use the multipartFile.transferTo method.

Simply pass in the file object and the file will be written out.


```
multipartFile.transferTo(imageFile);
```

Using the transferTo method to write out files on our server makes our code much cleaner and easier to work with. Let's scroll down a bit. Finally, after upload is complete, the upload data I'm going to send back to the client. So we can display a confirmation message. So model.addAttribute upload, and the view that we render out is the viewUploadDetail.

```
model.addAttribute("upload", upload);
```

At the bottom we have the get method which displays the upload form. The method is called displayForm, it responds to the path /uploadForm.

```
@RequestMapping(value = "/uploadForm", method = RequestMethod.GET)
public String displayForm(Model model)
```

We add to the model injected an instance of the uploaded data form, because this object is the model backing our form. And once that's done, we simply render out the upload form view.

3. Update Index.jsp File

Here is the index.jsp file which contains a single button that directs us to the upload form page.

```
<h2>Upload Multiple Selection File</h2><br/>
<input type="button" onclick="location.href='uploadMultipleSelectionForm'"
style="background-color: #008080;
color: black;
width: 150px; height: 40px; top: 40px;
"
value="Multiple Selection File Upload">
```

4. Create Upload Multiple Selection Form Page under /WEB-INF/jsp

```
uploadMultipleSelectionForm.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2 pageEncoding="ISO-8859-1"%>
3 <%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
4 <!DOCTYPE html>
5 <html>
6 <head>
7 <meta charset="ISO-8859-1">
8 <title>Upload Multiple Selection Form</title>
9 </head>
10 <body>
11 <h2>Multiple File Upload</h2>
12 <form:form modelAttribute="upload" action="save-upload" method="post" enctype="multipart/form-data">
13 <table>
14 <tr><td>Please Select one or more files to upload</td></tr>
15 <tr><td>Enter name:</td></tr>
16 <tr><td><input type="file" name="images" multiple="multiple" /></td></tr>
17 <tr><td>&nbsp;</td></tr>
18 <tr>
19 <td><input type="submit" id="submit" value="Upload"
20 style="background-color: #DE1515; color: white; width: 100px; height: 40px; top: 250px;" /></td>
21 <td><input type="reset" id="reset" value="Reset"
22 style="background-color: #DE1515; color: white; width: 100px; height: 40px; top: 250px;" /></td>
23 </tr>
24 </table>
25 </form:form>
26 </body>
27 </html>
```

This is the upload form page where we'll upload multiple files.

A couple of details to note here. Notice that I've set the modelAttribute on the form, this is on line 12. The modelAttribute is equal to upload,

```
<form:form modelAttribute="upload" action="save-upload" method="post">
```

```
enctype="multipart/form-data">
```

this is the upload data object that this form will send to the server. Just a reminder that this model attribute has to match the name of the variable that we passed into our controller object.

Notice in the controller object the name of the variable with the @ModelAttribute annotation is upload, which means on the form the model attribute should be upload as well. The action taken on form submit will be save-upload, which is the path to our handler mapping. A method is equal to post and the encoding type is multipart/form-data. All of these attributes you're familiar with. The interesting bit to allow multiple file selection is here on line 26. Notice the input type is equal to file indicating this is a file upload button. Name= images and multiple= multiple.

```
<input type="file" name="images" multiple="multiple" />
```

Just specifying the multiple attribute here will allow you to select multiple files in the file dialog that pops up when this button is clicked.


And below at the bottom, I have two buttons, a submit button allowing me to submit the form.

```
<input type="submit" id="submit" value="Upload" style="background-color: #DE1515; color: white; width: 100px; height: 40px; top: 250px;" />
```

The submit action will be invoked. And a reset button which will simply reset the form if you want to select a different set of files.

```
<input type="reset" id="reset" value="Reset" style="background-color: #DE1515; color: white; width: 100px; height: 40px; top: 250px;" />
```

5. Create viewUploadDetail Page under /WEB-INF/jsp



```

1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
4 <!DOCTYPE html>
5 <html>
6 <head>
7 <meta charset="ISO-8859-1">
8 <title>Upload Success</title>
9 </head>
10 <body>
11 <div id="global">
12 <h3>Following files are uploaded successfully</h3>
13 <ol>
14 <c:forEach items="${upload.images}" var="image">
15 <li>${image.originalFilename}</li>
16 </c:forEach>
17 </ol>
18 </div>
19 </body>
20 </html>

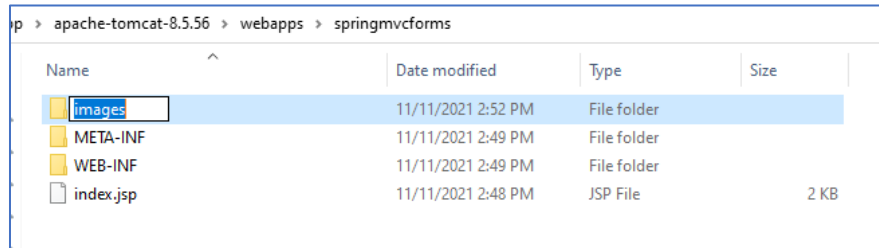
```

Remember, the upload data object is sent as an input to this page. I run a for each command using JSTL. This is on this line, iterate over upload.images.

```
<c:forEach items="${upload.images}" var="image">
  <li>${image.originalFilename}</li>
</c:forEach>
```

6. Build And Run the App

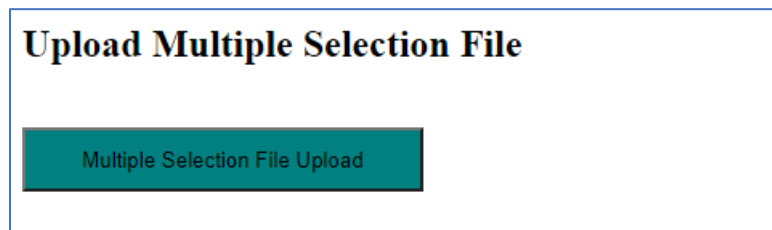
Time to build our WAR file and deploy to our Tomcat server. After you've deployed the WAR file and brought up your Tomcat server, head over to your /webapps/SpringMvcSimpleApp/ folder for your Tomcat server and create a new directory called images.



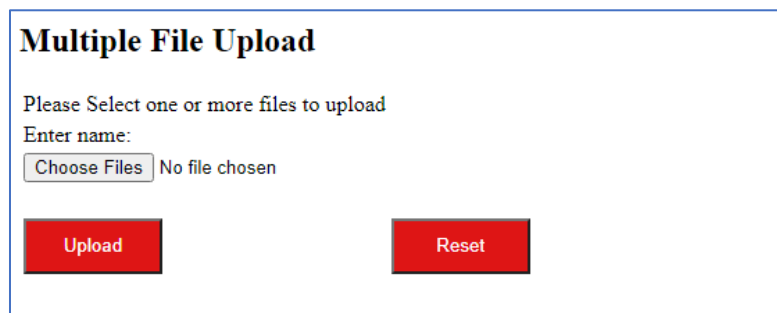
Name	Date modified	Type	Size
images	11/11/2021 2:52 PM	File folder	
META-INF	11/11/2021 2:49 PM	File folder	
WEB-INF	11/11/2021 2:49 PM	File folder	
index.jsp	11/11/2021 2:48 PM	JSP File	2 KB

This is the directory relative to the root of our application which will hold the images that we upload. If you run an ls command, you can see the new folder has been created, that is images.

I'm going to cd into that folder and run an ls command to show you that the folder is completely empty to start off with. Here is the main page of our web application with the Upload Files button.



Click on this button, and this will take you to a form that allows upload of multiple files.



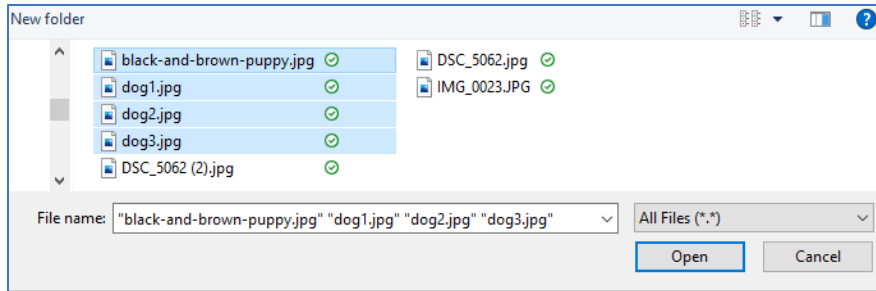
Multiple File Upload

Please Select one or more files to upload

Enter name:

No file chosen

Once you've selected your files, you can choose to upload those files or reset your form. I'm going to click on the Choose Files button in order to bring up a file dialog that allows for multiple selection.



Hold down the Shift, or the Cmd key or the Ctrl key for multiple selection. Click on Open and you can see that three files have been chosen. These are the four files I want to upload,

Multiple File Upload

Please Select one or more files to upload

Enter name:

Choose Files 4 files

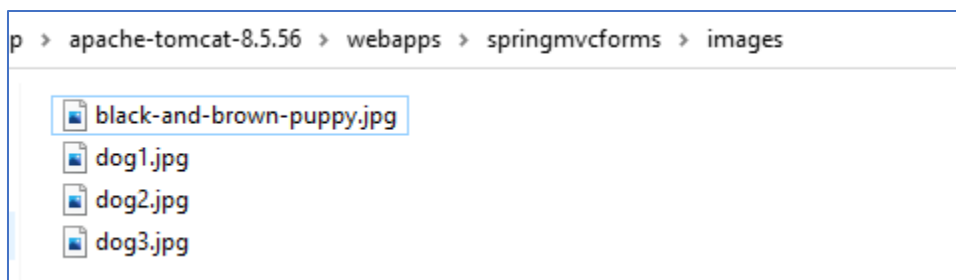
UploadReset

click on the Upload button. And these files have been uploaded successfully. This is the message that I get from the server.

Following files are uploaded successfully

1. black-and-brown-puppy.jpg
2. dog1.jpg
3. dog2.jpg
4. dog3.jpg

One last sanity check to see that these files are on our server. Head over to your Tomcat server folder under webapps, SpringMvcSimpleApp.



You'll find our images directory and you can see that dog1, dog2, dog3, all JPG files are present here. You can double-click on one of these files and make sure that the file contents are what you expect.

Downloading Files from Application

At this point, we have a lot of experience using the upload button to upload files to our Spring MVC application running on our server. Now we'll see how we can perform file download in Spring MVC. And in order to perform file download, I'm going to include a few additional dependencies in my pom.xml file.

1. Add commons-io dependency in pom.xml

In pom.xml, add artifactId **commons-io**

```
<dependency>
  <groupId>commons-io</groupId>
  <artifactId>commons-io</artifactId>
  <version>2.6</version>
</dependency>
```

This commons-io dependency is a JAR file that has common file handling utilities. And we'll use some of those utilities when we download files from our server down to our local machine.

The basic components of the application, including the XML configuration files, web.xml, spring-servlet.xml, index.jsp, all are familiar to you. There are a few changes, but nothing that you haven't seen before.

2. Update Index.jsp

index.jsp

```
<h2>File Download</h2><br/>
<a href="download/downloadForm">Click To Download File</a>
```

3. Prepare Download Page JSP

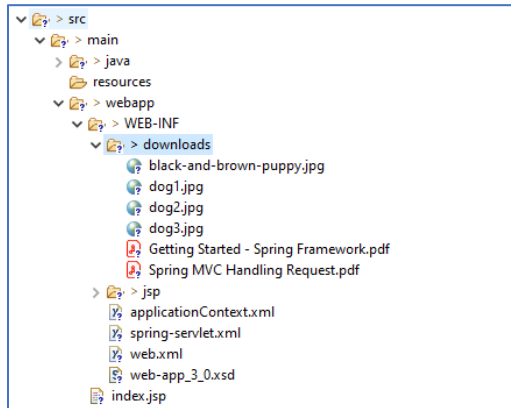
```
downloadForm.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="ISO-8859-1">
7 <title>Download Form</title>
8 </head>
9 <body>
10
11 <form action="performDownload">
12   Enter file name here : <input type="text" name="filename"/><br/><br/>
13   <input type="submit" value="Download" />
14 </form>
15
16 </body>
17 </html>
```

Let's just quickly take a look at the view file, which gives us the form allowing us to specify the file that we want to download. This is the downloadForm.jsp file. The form action here is performDownload, and this is the path that we'll set up in our controller mapping.

All we have here is an input text box, which contains the filename and a submit button. So we simply type in a filename in this input text box, hit submit, and if the file is available, it will be downloaded from the server down to our client.

The name of the input is filename. This will be sent as the request parameter to our server, which we'll extract on the controller end.

In order to be able to perform download, we need to store some files on our server. And we'll store these files under the WEB-INF folder. Under WEB-INF, choose the New option after right clicking, and select Folder. We'll create a new folder that will hold files.



This will be deployed with our web application, and these are the files that we have the option of downloading. This new folder is called downloads. Keep track of the location of this downloads folder. It's under the WEB-INF subfolder, which is under webapp. I'm going to populate this downloads folder with a few files.

4. Prepare DownloadController

My DownloadController is capable of allowing file downloads for JPEG files as well as PDF files. Let's take a look at our controller where we have the code to handle the actual download operation.

```
DownloadController.java
1 package com.mvc.controller;
2
3 import java.io.IOException;
4
17
18 @Controller
19 @RequestMapping("/download")
20 public class DownloadController {
21
22     @Autowired
23     ServletContext context;
24
25     @RequestMapping("/performDownload")
26     public void downloader(HttpServletRequest request, HttpServletResponse response, Model model) {
27         String fileName = request.getParameter("filename");
28         System.out.println(String.format("Downloading file : %s", fileName));
29         String downloadFolder = context.getRealPath("/WEB-INF/downloads/");
30         Path file = Paths.get(downloadFolder, fileName);
31         if (Files.exists(file)) {
32             String extension = FilenameUtils.getExtension(fileName);
33             if ("pdf".equalsIgnoreCase(extension)) {
34                 response.setContentType("application/pdf");
35             } else {
36                 response.setContentType("image/jpeg");
37             }
38
39             response.addHeader("Content-Disposition", "attachment; filename=" + fileName);
40             try {
41                 Files.copy(file, response.getOutputStream());
42                 response.getOutputStream().flush();
43             } catch (IOException e) {
44                 e.printStackTrace();
45             }
46         }
47     }
48
49     @RequestMapping("/downloadForm")
50     public String displayDownloadForm() {
51         return "downloadForm";
52     }
53
54 }
```

Here is the DownloadController, it's tagged using the @Controller annotation, as well as the @RequestMapping for forward slash download.

```
@Controller
@RequestMapping("/download")
public class DownloadController
```

Forward slash download will be the path prefix for handler mappings in this controller. I need the ServletContext, but instead of setting up a context aware controller implementation, I've simply autowired the ServletContext so that it's injected into my controller when it's created.

```
@Autowired
ServletContext context;
```

Let's take a look at the downloader method, which is what handles the forward slash performDownload operation.

```
@RequestMapping("/performDownload")
public void downloader(HttpServletRequest request, HttpServletResponse response, Model model)
```

We inject into this method, an HttpServletRequest as well as the HttpServletResponse along

with the model. you can see that we use the `HttpServletRequest` to extract the request parameter sent to the controller. We'll use `request.getParameter` to extract the filename.

```
String fileName = request.getParameter("filename");
```

This is the file that the user wants to download. I then print out a debugging message `System.out.println`, Downloading file, and the `fileName`.

It's useful to have these debug messages while developing your code. This debug message will be printed out as a part of the Apache logs. I'll show you that when we run our server.

```
String downloadFolder = context.getRealPath("/WEB-INF/downloads/");
```

Now, let's get the download folder. The real path of the download folder can be accessed using the `ServletContext`, `context.getRealPath`, forward slash `WEB-INF` forward slash `downloads`. Remember, the `WEB-INF` folder is just under the root folder of the application. If you're working on a Windows machine, rather than forward slash, you might need to use backslash in order to get a path to your folder.

Just keep that in mind. Let's get a path object to this download folder and append the filename,

```
Path file = Paths.get(downloadFolder, fileName);
```

`Paths.get` downloadFolder, `fileName`, this gives us a path object. Now, it's only possible to download this file if the file actually exists, and this we check using `Files.exists`.

```
if (Files.exists(file)) {
```

If this file does exist, we'll get into the if block. Within the if block, we'll use the `FilenameUtils` utility class,

```
String extension = FilenameUtils.getExtension(fileName);
```

which is a class present in the `commons-io` library. We'll use that to get the extension associated with this `fileName`. And we store this in the String `extension`. If we find that the extension is equal to `pdf`, we know that it's a PDF file that the user wants to download onto his local machine.

```
if ("pdf".equalsIgnoreCase(extension)) {
```

We call `response.setContentType` and set it to `application` forward slash `pdf`, so that the response knows how to deal with that file.

```
response.setContentType("application/pdf");
```

Otherwise, we'll assume that the file is a JPEG file, it's an image file. We'll call `response.setContentType` and set it to `image` forward slash `jpeg`.

```
    } else {  
        response.setContentType("image/jpeg");  
    }
```


The response is what will contain the file that will be downloaded onto the local machine. We need to add a header to the response, indicating that it contains an attachment.

```
response.addHeader("Content-Disposition", "attachment; filename=" +  
fileName);
```

addheader Content-Disposition is the header name, and the value is attachment, filename is equal to the name of your file. The actual download is performed by copying the contents of the file to the output stream of the response, and we'll do this within a try catch block. We'll use Files.copy, take the file which we know exist,

```
Files.copy(file, response.getOutputStream());
```

and copy it out to the response.getOutputStream. We'll then flush this output stream, so that the file contents are downloaded, response.getOutputStream.flush.

```
response.getOutputStream().flush();
```

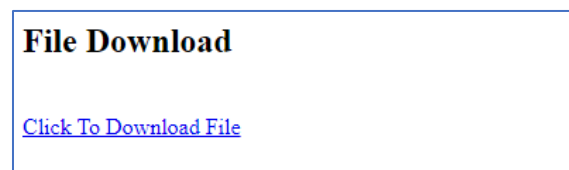
And this bit of code takes care of the actual downloading of the file. And finally, we have one last method that we have mapped here using @RequestMapping downloadForm.

```
@RequestMapping("/downloadForm")  
public String displayDownloadForm()
```

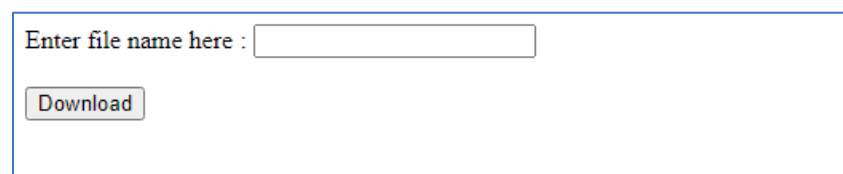
This is the method that responds to the HTTP GET request to display the DownloadForm onto Stream, where the user can type in the name of the file to download.

5. Build and Run the App

Now, it's time for us to build the Maven WAR file, and once that is done, deploy the file to our Apache Tomcat server. Here is what the index.jsp looks like, the entry point of our application. (http://localhost:8080/springmvcforms/)

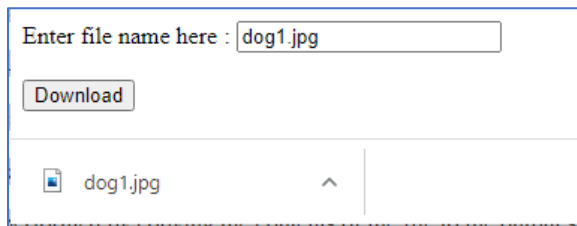


I'll click on this link here, which takes us to the form allowing us to download a file. (http://localhost:8080/springmvcforms/download/downloadForm)

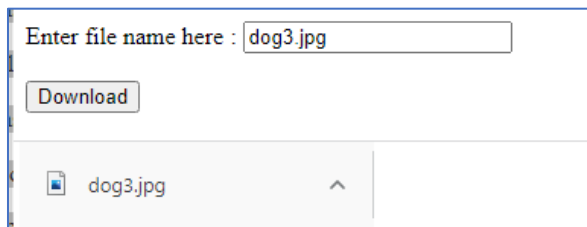
A screenshot of a web form. It has a label "Enter file name here :" followed by a text input field. Below the input field is a button labeled "Download".

Notice the URL path to this form. It is the root path of our application, forward slash download, forward slash downloadForm. The first file that I want to download is dog1.jpg, we know that it exists in the Downloads folder. When I hit Submit, this file will be downloaded onto my local

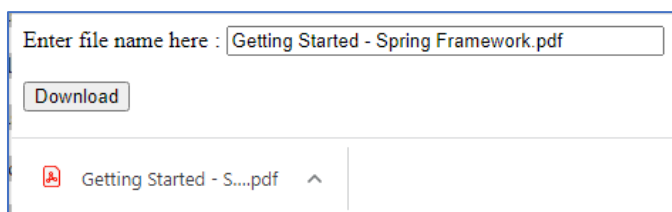
machine. And if you head over to the terminal window where I have the Tomcat server running, you can see our debug log message there, Downloading file dog1.jpg. Let's head over to our local machine Downloads folder, and here is the dog1.jpg file successfully downloaded.



Let's try downloading a couple of more files. Here I download dog3.jpg, hit Submit, and the dog3.jpg file has also been successfully downloaded. On my Apache terminal window, you can see that extra log message, Downloading dog3.jpg.



Back to our form here, I'm going to download one last file, irregular-verbs.pdf. This is a PDF file and not an image file, hit Submit, and this file will also be successfully downloaded. Remember, our DownloadController handles JPEG files and PDF files.



Here is the log message on the terminal window, and here is the “**Getting Started - Spring Framework.pdf**” on my local machine. The file is now available for me to use.

```
12-Nov-2021 10:29:23.331 INFO [localhost-startStop-6] org.springframework.web.context.ContextLoader.initWebApplicationContext Root WebApplicationContext: initialization started
12-Nov-2021 10:29:24.549 INFO [localhost-startStop-6] org.hibernate.validator.internal.util.Version.<clinit> HV000001: Hibernate Validator 6.0.13.Final
12-Nov-2021 10:29:25.216 INFO [localhost-startStop-6] org.springframework.web.context.ContextLoader.initWebApplicationContext Root WebApplicationContext initialized in 1884 ms
12-Nov-2021 10:29:25.224 INFO [localhost-startStop-6] org.springframework.web.servlet.FrameworkServlet.initServletBean Initializing Servlet 'spring'
12-Nov-2021 10:29:25.307 INFO [localhost-startStop-6] org.springframework.web.servlet.FrameworkServlet.initServletBean Completed initialization in 83 ms
12-Nov-2021 10:29:25.323 INFO [localhost-startStop-6] org.apache.catalina.startup.HostConfig.deployWAR Deployment of web application archive C:\app\apache-tomcat-8.5.56\webapps\s
pringmvcforms.war has finished in 4,034 ms
Downloading file : dog1.jpg
Downloading file : dog3.jpg
Downloading file : dog3.jpg
Downloading file : dog3.jpg
Downloading file : Getting Started - Spring Framework.pdf
Downloading file : Getting Started - Spring Framework.pdf
```

Summary

In this course, we explored the use of forms to send information from the browser to our server. And saw how we could validate these form elements. We also dealt with file uploads and downloads. We started off by setting up a form to capture user input using specialized form elements available as a part of the JSTL library.

We explored form elements for text boxes, dropdowns, checkboxes, and password fields. We then moved on to using built-in validations for our form values using the validator APIs available as a part of the Spring Framework. We enable these annotations on form fields using Java annotations on the model object bound to the form. We saw how errors and forms could be identified within our controller.

And how these errors could be displayed to the user, prompting a resubmit of the form. We rounded off our exploration of Spring MVC features by working with files. We saw how we could use form elements to upload a single file as well as multiple files to our application. We also saw how it was possible to download files from our application.

Quiz

1. What is the attribute on a JSP taglib form tag which binds the form input fields to a Java object?
object
command
✓ **modelAttribute**
model
2. Which object contains information about validation errors on your form?
FormResult
✓ **BindingResult**
PathVariable
ModelAttribute
3. What is a multipart request?
Multiple files merged together in the JSON format
✓ **Combination of one or more sets of data, separated by boundaries**
A special kind of file request
Multiple portions of data from a single file rather than the entire file
4. What does the @ResponseBody tag do?
✓ **Converts a return value from a handler to a web response**
Injects a web response into a handler
Converts string responses to JSON format
Indicates that a particular handler has no response to send to the client
5. Can you have two methods mapped to the same path in your controller?
No this is not possible, every method has to be mapped to a different path
Yes, but each method has to have a different name
Yes, the application will figure out which method to invoke based on the servlet context
✓ **Yes, but each method has to respond to a different HTTP request**
6. Identify the attribute that you would set on the input tag to allow selection of more than one file in the file dialog.
No special attribute
multiUpload
✓ **multiple**
files
7. which operation performs the actual download of the file to the user's local machine?
Setting the content type of the file on the response
✓ **Setting the file contents as the response body**
Setting the file contents on the header of the response
Copying the file to the output stream of the response