Introduction to Spring MVC

Student Workbook

Chapter 1 - Spring MVC Overview

Objectives:

- Describe the problems that the Spring Framework is fixing.
- Create a simple Spring MVC web application.

What is Spring?

- The Spring Framework was originally created by Rod Johnson as an answer to the tedious coding model presented by Enterprise Java Beans (EJB).
    - Spring favors Plain Old Java Objects (POJOs) which do not extend classes or implement interfaces from Java EE packages.
        - This makes them easier to test and reuse since they can be utilized independently of the environment in which they were first deployed.
        - Spring developers typically refer to POJOs as “beans”.
- The Spring Framework provides a container (called Spring Core) which is primarily concerned with decoupling the responsibility for object creation from your code to the framework itself.
    - You can use XML to configure how you wish your beans to be created including what to pass to the constructor and which set() methods should be called to initialize them.
        - The process of populating object properties with values at creation is called *Dependency Injection*.
- Another key feature of Spring is integration.
    - Spring provides simple ways to integrate enterprise features like database access, transactions, and asynchronous messaging via additional configurations.
- See *http://projects.spring.io/spring-framework/* for more information.

Add support for the Spring Core by adding a dependency to your Maven *pom.xml* file.

```
<dependencies>
     <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-context</artifactId>
          <version>4.2.4.RELEASE</version>
     </dependency>
</dependencies>
```

Declare your Spring beans using XML (*SpringLotto/src/beans.xml*):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" … >

     <bean id="lottery" class="lotto.Hopper">
          <property name="name" value="Lottery Numbers"/>
          <property name="pingPongBalls">
               <list>
                    <bean class="lotto.PingPongBall">
                         <constructor-arg value="1" />
                    </bean>
                    <bean class="lotto.PingPongBall">
                         <constructor-arg value="2" />
                    </bean>
                    …
               </list>
          </property>
     </bean>
</beans>
```

Lookup your bean(s) with Spring (*SpringLotto/src/lotto/QuickPick.java*):

```
…
public class QuickPick {
  public static void main(String[] args) {
    ClassPathXmlApplicationContext context =
      new ClassPathXmlApplicationContext("beans.xml");

    Hopper h = context.getBean("lottery", Hopper.class);
    h.reset();
    System.out.println(h.getName());
    for (int i = 0; i < 6; i++) {
      System.out.print(h.drawBall().getValue() + " ");
    }
    System.out.println();
    context.close();
  }
}
```

What is Spring MVC?

- The goal of Spring MVC is to make it easier to build Java web applications.
    - You no longer extend **HttpServlet**, instead Spring MVC provides the servlet for you.
    - You will define your controller logic in a POJO annotated with **@Controller**.
    - Spring MVC calls methods in your **@Controller** class as requests are received.
    - Spring MVC also eliminates calls to **request.getParameter()** by replacing them with auto-populated method parameters.
        - You don't even have to convert request parameters to primitive data types, Spring MVC will do that for you.
    - Rather than using a **RequestDispatcher** to invoke a JSP, you will simply identify the JSP by name as part of the returned value of your method.
    - The full power of Spring is now available for you to take advantage of in your web applications.
        - You can still have Spring instantiate and configure objects for you.

Add support for Spring MVC by adding a dependency to your Maven *pom.xml* file. The **spring-webmvc** dependency can be used instead of **spring-context** since it provides a superset of the downloaded jar files.

```
     <dependencies>
          <dependency>
               <groupId>org.springframework</groupId>
               <artifactId>spring-webmvc</artifactId>
               <version>4.2.4.RELEASE</version>
          </dependency>
          <dependency>
               <groupId>javax.servlet</groupId>
               <artifactId>javax.servlet-api</artifactId>
               <version>3.1.0</version>
          </dependency>
          <dependency>
               <groupId>javax.servlet</groupId>
               <artifactId>jstl</artifactId>
               <version>1.2</version>
          </dependency>
     </dependencies>
```

Spring MVC Demo - web.xml

- You must add a *web.xml* file to the *WEB-INF* folder in your web application to configure the Spring MVC servlet.
    - The **<servlet-class>** entry must be set to *org.springframework.web.servlet.DispatcherServlet*.
    - The **<servlet-name>** can be anything, but must be the same in both the **<servlet>** and **<servlet-mapping>** sections.
        - Also, the name you choose will be used later as part of the name of the Spring configuration file.
    - The **<url-pattern>** element specifies what requests should be handled by the **DispatcherServlet**.
        - For the **<url-pattern>** it is typical to use _*.do_, but any pattern can be substituted.

SpringMVCLotto/WebContent/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="http://xmlns.jcp.org/xml/ns/javaee"
     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
          http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
     version="3.1">

     <servlet>
          <servlet-name>Lotto</servlet-name>
          <servlet-class>
               org.springframework.web.servlet.DispatcherServlet
          </servlet-class>
     </servlet>
     <servlet-mapping>
          <servlet-name>Lotto</servlet-name>
          <url-pattern>*.do</url-pattern>
     </servlet-mapping>
</web-app>
```

Spring MVC Demo - Spring Config File

- In addition to *web.xml* you must also create a Spring configuration file and save it in the same *WEB-INF* folder.
  - Name the file ***<servlet-name>*-servlet.xml** where the *servlet-name* is the one you specified in *web.xml*.
    - For example, if you used *WebApp* as the **<servlet-name>** in *web.xml*, then you should name the file *WebApp-servlet.xml*.
  - Add the **<context:component-scan base-package="*pkg*"/>** element, where *pkg* is the package that contains your **@Controller**.
  - Configure any Spring beans that you would like to use in your web application using **<bean>** declarations.
    - Add **id** and **class** attributes to specify an alias and fully qualified class name for your POJO.
    - Use **<constructor-arg>** child elements to call a constructor.
      - Identify which arg you are populating through the use of a zero based **index** attribute.
      - Use a **value** attribute to pass a string or primitive or a **ref** attribute to specify another bean.
    - You can also use **<property>** child elements to call **set()** methods on the newly created bean.
      - The property must specify which **set()** method to invoke by use of the **name** attribute - the name is derived from the **set()** method name without the prefix *set*.
      - Like **constructor-arg**, use a **value** attribute to pass a string or primitive or a **ref** attribute to specify another bean.

SpringMVCLotto/WebContent/WEB-INF/Lotto-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans … >

     <context:component-scan base-package="web"/>

     <bean id="lottery" class="lotto.Hopper">
          <property name="name" value="Lottery Numbers"/>
          <property name="pingPongBalls">
               <list>
                    <bean class="lotto.PingPongBall">
                         <constructor-arg value="1" />
                    </bean>
                    <bean class="lotto.PingPongBall">
                         <constructor-arg value="2" />
                    </bean>
                    …
               </list>
          </property>
     </bean>
</beans>
```

Spring MVC Demo - @Controller Class

- Annotate a POJO that encapsulates your controller logic with **@Controller**.
- Add methods that Spring's **DispatcherServlet** will call when requests are made.
  - Annotate each method with **@RequestMapping** specifying the URL pattern that the method is mapped to as the annotation's value.
  - Your method should return either a **String** or **ModelAndView** object.
    - Use a **String** to return the name of the JSP to forward to for presentation.
    - Use the **ModelAndView** object to encapsulate both the JSP view as well as additional model data that the JSP can present.
  - Add parameters that match the incoming request data from the HTML form that populates the request.
    - Precede each parameter with the **@RequestParam** annotation which identifies the name of the request parameter to map to.
    - Spring will handle data type conversions for primitive types.
- The **@Controller** class is automatically configured as a Spring bean based on its package being listed in the **<context:component-scan>** element in the Spring configuration file.
  - As a Spring bean it can take advantage of additional Spring features like auto wiring.
    - Any field that is annotated with **@Autowired** is injected with a Spring bean instance of the appropriate type as part of the instantiation process.

SpringMVCLotto/src/web/LottoController.java

```
package web;
…
@Controller
public class LottoController {
     @Autowired
     private Hopper hopper;

     @RequestMapping("GetNumbers.do")
     public ModelAndView getNumbers(
                           @RequestParam("howmany") int count) {
          hopper.reset();

          List<String> nums = new ArrayList<>();
          for (int i = 0; i < count; i++) {
               nums.add(hopper.drawBall().getValue());
          }

          ModelAndView mv = new ModelAndView();
          mv.setViewName("form.jsp");
          mv.addObject("listOfNumbers", nums);
          return mv;
     }
}
```

SpringMVCLotto/WebContent/form.jsp

```
…
<body>
     <h3>Lottery Numbers</h3>
     <form action="GetNumbers.do" method="GET">
          <input type="text" name="howmany" value="6" size="2"/>
          <input type="submit" value="Get Numbers" />
     </form>
     <c:forEach var="number" items="${listOfNumbers}">
          ${number}
     </c:forEach>
     <br/>
</body>
</html>
```

Labs

1. Modify the existing SpringMVCLotto web application to display the lottery numbers in sorted order.

2. Create a new SpringMVC web application to implement simple Hello World functionality. Send a person's name from an HTML form to a Controller. The Controller should prepend the word "Hello" to the name, then have a JSP display the newly created string. Use the steps on the following pages to assist you in creating the new web application.

Steps for building a simple Spring MVC web application in Eclipse:

1. Create a new *Dynamic Web Project* in Eclipse.

2. Right Click on the project name in the *Project Explorer* and choose *Configure | Convert to Maven Project*. Click *Finish* on the *Maven POM* dialog.

3. Modify the newly created *pom.xml* file by adding the following just before the final *</project>* tag.

```
<dependencies>
      <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.2.4.RELEASE</version>
      </dependency>
      <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
      </dependency>
      <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
      </dependency>
</dependencies>
```

4. Create a web.xml file in your project's *WebContent/WEB-INF* folder. (If you have an existing file in another project it would be best to copy it.)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://xmlns.jcp.org/xml/ns/javaee"
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
            http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
      version="3.1">
      <servlet>
            <servlet-name>NAME</servlet-name>
            <servlet-class>
                  org.springframework.web.servlet.DispatcherServlet
            </servlet-class>
      </servlet>
      <servlet-mapping>
            <servlet-name>NAME</servlet-name>
            <url-pattern>*.do</url-pattern>
      </servlet-mapping>
</web-app>
```

5. Create another XML file in the *WebContent/WEB-INF* folder. Name this file *NAME-servlet.xml* where *NAME* is the name you used for the *<servlet-name>* in your *web.xml*. Make sure to modify the *base-package* attribute on the *context:component-scan* element to match the package you will use in step 7. (If you have an existing file in another project it would be best to copy it.)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.2.xsd">

       <context:component-scan base-package="controllers"/>

</beans>
```

6. Build an HTML file with an embedded form to gather data from the user. The *action* you specify will have to match your request mapping in the controller class you will be creating soon. In Eclipse, right click on your *WebContent* folder and choose *New | HTML File*.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Title</title>
</head>
<body>
       <form method="GET" action="process.do">
              <input type="text" name="data" /><br />
              <input type="submit" value="Submit" /><br />
       </form>
</body>
</html>
```

7. Create a new package to contain your Spring MVC controller class. Make sure to match the package name with what you placed in the Spring config file in step 5.

8. Create a new Java class within the newly created package to serve as your controller. Look for your newly created package and class within the *Java Resources/src* folder in Eclipse.

9. Add the *@Controller* annotation to your newly created class above the public class declaration. In Eclipse you can use *Source / Organize Imports* to automatically import the proper Spring library. If it fails to import properly, verify that your *pom.xml* was properly configured in step 3.

10. Add a method to your controller class with an *@RequestMapping* annotation set to the *action* you specified in your HTML file in step 6. The method should return a *ModelAndView* object and have parameters that match the request parameters from the HTML. Use the *@RequestParam* annotation to match request parameters with method parameters. The name of the method can be anything. Again, use *Source / Organize Imports* to import the necessary libraries.

11. Within the method, process the request appropriately, then build a new *ModelAndView* object to return. Call the *setViewName()* method on the object to specify the name of the JSP file (the view) that will display the results. Call the *addObject()* method for each item you wish to place in the model that the view needs to access.

```
package controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class MyController {
	@RequestMapping("process.do")
	public ModelAndView processData(@RequestParam("data") String s) {
		String allCaps = s.toUpperCase();
		ModelAndView mv = new ModelAndView();
		mv.setViewName("view.jsp");
		mv.addObject("result", allCaps);
		return mv;
	}
}
```

12. Build a JSP as your view, using JSP EL to retrieve values from the Model to display. In Eclipse, you can right click on the *WebContent* folder and choose *New | JSP File*.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
<head>
<title>View</title>
</head>
<body>
	<p>The result is: ${result}.</p>
</body>
</html>
```

13. Run your program by right clicking on your project name and choosing *Run As | Run on Server*. The newly deployed application should be available on *http://localhost:8080/ProjectName* where *ProjectName* is the name of your project. If you do not specify an *index.html* file, then you should manually navigate to the location of your html file. *http://localhost:8080/ProjectName/Form.html*

Chapter 2 - Handling Requests

Objectives:

- Build a controller class to handle HTTP requests.

- Use the various elements within the **@RequestMapping** and **@RequestParam** annotations.

@Controller

- Use the **@Controller** annotation on a POJO class to ask Spring to automatically load it as a Spring bean.

```
package web;

import org.springframework.stereotype.Controller;

@Controller
public class MyController { … }
```

  - For this auto-discovery to take place, your Spring XML configuration file must identify the package that your controller belongs to using the **<context:component-scan>** element.

```
<context:component-scan base-package="web">
```

  - If you have more than one package that contains controllers, comma separate the names within the **base-package** attribute.

```
<context:component-scan
   base-package="web,com.code.controllers">
```

States/src/controllers/StateController

```
…
@Controller
public class StateController {
     @Autowired
     private StateDAO stateDao;
     …
}
```

@RequestMapping

- Your controller class typically has one or more methods annotated with **@RequestMapping** whose purpose is to handle HTTP requests.

- At a minimum, identify the URL pattern that the method should map to by embedding a string within the parenthesis of the annotation.

```
@RequestMapping("getData.do")
```

  - If you wish to have the same method invoked for multiple URL patterns, set the **path** element to an array of strings.

```
@RequestMapping(path={"getData.do", "data.do"})
```

- To identify which HTTP methods should be mapped to your Java method, add the **method** element.

```
@RequestMapping(path="getData.do",
  method=RequestMethod.GET)
```

  - This, too, can be an array.

```
@RequestMapping(path="getData.do",
  method={RequestMethod.GET, RequestMethod.POST})
```

  - If you don't specify a **method** element, then your code will respond to all HTTP methods at the specified path.

- You can use the **params** element to narrow the choice of which method to invoke to those that include a particular request parameter.

```
@RequestMapping(path="getData.do",
  params="name")
```

States/src/controllers/StateController

```
…
@Controller
public class StateController {
     @Autowired
     private StateDAO stateDao;

     @RequestMapping(path="GetStateData.do",
               params="name",
               method=RequestMethod.GET)
     public ModelAndView getByName(
          @RequestParam("name") String n) {
          …
     }

     @RequestMapping(path="GetStateData.do",
               params="abbrev",
               method=RequestMethod.GET)
     public ModelAndView getByAbbrev(
          @RequestParam("abbrev") String a) {
          …
     }

     @RequestMapping(path="NewState.do",
               method=RequestMethod.POST)
     public ModelAndView newState(State state) {
          …
     }
}
```

You can also use the **@RequestMapping** annotation at the class level. If you do, then all mappings at the method level are relative to the URL pattern defined on the class.

@RequestParam

- Have Spring load data directly into your request handling method's parameters with the **@RequestParam** annotation.

  - Specify the name of the HTTP request param that you used in your HTML form within the parenthesis of the annotation.

```
<form action="getData.do" method="GET">
    <input type="text" name="age"/>
```

    maps to

```
@RequestMapping("getData.do")
public String getData(@RequestParam("age") int a)
{…}
```

  - Spring will automatically map the request parameter to your method parameter without need for the annotation if you use the same name for both.

```
public String getData(int age)
```

  - Use the **defaultValue** element to have Spring load the given value in the parameter if it is not present.

```
public String getData(
 @RequestParam(name="age", defaultValue="5") int a)
```

  - Add the **required=false** element to the annotation to indicate that the parameter is optional and will be loaded with a **null** value if not provided.

```
public String getData(@RequestParam(name="age",
    required=false) Integer a)
```

- Spring will automatically convert the data from the request to the datatype identified by the parameter.

While Spring allows you to provide both the **required=true** and **defaultValue** elements, it will not honor the required instruction if the default value is present.

Avoid using primitives when you specify **required=false** since there is no way to store a **null** value to a primitive. Use their wrapper equivalents to avoid a 500 error in your browser.

States/WebContent/nameForm.html

```
…
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>States</title>
</head>
<body>
     <h3>States</h3>
     <form action="GetStateData.do" method="GET">
          Name:
          <input type="text" name="name"/>
          <input type="submit" value="Get State Data" />
     </form>
</body>
</html>
```

States/src/controllers/StateController

```
…
@Controller
public class StateController {
     @Autowired
     private StateDAO stateDao;

     @RequestMapping(path="GetStateData.do",
               params="name",
               method=RequestMethod.GET)
     public ModelAndView getByName(
          @RequestParam("name") String n) {

          ModelAndView mv = new ModelAndView();
          mv.setViewName("result.jsp");
          mv.addObject("state", stateDao.getStateByName(n));
          return mv;
     }
     …
}
```

Method Return Types

- Return a **String** from your request handler method that identifies the name of the JSP or HTML file to display after the method completes.

- Or, return a **ModelAndView** object to identify the view to display but also load data into the model for presentation.

    - Create the **ModelAndView** object using the no-arg constructor and call the **setViewName()** method on the newly created object passing the filename that represents the view.

    - Call the **addObject(*key,value*)** method repeatedly for each data item you wish to make available to the view.

```
ModelAndView mv = new ModelAndView();
mv.setViewName("result.jsp");
mv.addObject("name","Joe");
mv.addObject("age", 12);
return mv;
```

States/src/controllers/StateController

```
…
@Controller
public class StateController {
     @Autowired
     private StateDAO stateDao;

     @RequestMapping(path="GetStateData.do",
               params="name",
               method=RequestMethod.GET)
     public ModelAndView getByName(
          @RequestParam("name") String n) {

          ModelAndView mv = new ModelAndView();
          mv.setViewName("result.jsp");
          mv.addObject("state", stateDao.getStateByName(n));
          return mv;
     }
     …
}
```

States/WebContent/result.jsp

```
…
<body>
     <c:choose>
          <c:when test="${! empty state}">
               <ul>
                    <li>${state.abbreviation}</li>
                    <li>${state.name}</li>
                    <li>${state.capital}</li>
               </ul>
          </c:when>
          <c:otherwise>
               <p>No state found</p>
          </c:otherwise>
     </c:choose>
</body>
</html>
```

Command Objects

- If you have multiple request parameters you may be able to use a command object to simplify your controller.
  - A *command object* is a JavaBean which you list as a parameter to your controller method whose properties match the names of the request parameters.
  - A property name is derived from a setter method by removing the word "set" and lowercasing the first letter of the remaining string.
    - For example, the method **setLastName()** would result in a property **lastName**.

```
public void setLastName(String ln) {
     this.lastNm = ln;
}
```

    - The name of the field does NOT matter. In the above example, the property is **lastName** even though the field is **lastNm**.
  - Your command object will be automatically loaded with data by Spring MVC when the request is submitted.
- The command object is automatically stored in the request so it is available to your view for presentation.
  - You don’t need to add it to the **ModelAndView** object like you would other objects that you wish to pass to your view.

States/WebContent/newState.html

```
…
<form action="NewState.do" method="POST">
     Abbrev:
     <input type="text" name="abbreviation" value="PR"/><br/>
     Name:
     <input type="text" name="name" value="Puerto Rico"/><br/>
     Capital:
     <input type="text" name="capital" value="San Juan"/><br/>
     <input type="submit" value="Add State" />
</form>
…
```

States/src/data/State

```
…
     public void setAbbreviation(String abbreviation) {…}
     public void setName(String name) {…}
     public void setCapital(String capital) {…}
…
```

States/src/controllers/StateController

```
…
     @RequestMapping(path="NewState.do",
               method=RequestMethod.POST)
     public ModelAndView newState(State state) {
          stateDao.addState(state);
          ModelAndView mv = new ModelAndView();
          mv.setViewName("result.jsp");
          return mv;
     }
}
```

You can add additional parameters to your methods that Spring MVC will automatically inject with appropriate values. Available objects include:

- HttpServletRequest
- HttpServletResponse
- HttpSession
- Model

Labs

1. Each state object in the model of the States web application has an additional property for the population of the capital. Modify the view so that this data is displayed as well.

2. Each state object also contains properties for the latitude and longitude of the state capital. Search the Internet for information on how you can use this extra data to embed a map in your view.

3. Study *State.java*, *StateDAO.java*, and *StateFileDAO.java* to see how the model data is loaded from *WebContent/WEB-INF/states.csv*. Add an additional column of data in the csv file for something else about the state that you wish to display (bird, flower, motto, flag, etc.) and modify the application to also display this additional information.

4. Use any HTML or CSS skills you have learned to improve the look and feel of the application.

5. Build another Spring MVC web application based on the data stored in *presidents.csv.* The user should be able to type in a presidents term number and pull up information about that president.

6. Add to your Presidents application so that the user can also search for presidents using additional criteria: (last name, party, term length, etc.).

Chapter 3 - Sessions

Objectives:

- Use an **HttpSession** object in a Spring MVC program.
- Store data to the session using **@SessionAttributes**.

Storing an Object in the Session

- Use an **HttpSession** object to store data on behalf of a user across requests.
  - Inject the **HttpSession** object into your controller method by simply listing it as a parameter.

```
@RequestMapping("processData.do")
public String processData(HttpSession session) {
   …
}
```

  - Store data to the session like you would in a servlet by calling the **setAttribute()** method on the session object passing a key/value pair.

LottoSessions/src/web/LottoController.java

```
…
@Controller
public class LottoController {
     @Autowired
     private Hopper hopper;

     @RequestMapping("GetNumbers.do")
     public String lotto(@RequestParam("howmany") int count,
               HttpSession session) {

          List<Integer[]> lottoHistory = (List<Integer[]>)
               session.getAttribute("lottoHistory");
          if (lottoHistory == null) {
               // first time
               lottoHistory = new ArrayList<>();
               session.setAttribute("lottoHistory",
                    lottoHistory);
          }

          lottoHistory.add(getLottoNumbers(count));
          return "form.jsp";
     }

     private Integer[] getLottoNumbers(int howMany) {
          Integer[] drawing = new Integer[howMany];
          hopper.reset();
          for (int i = 0; i < howMany; i++) {
               drawing[i] = Integer.parseInt(
                    hopper.drawBall().getValue());
          }
          Arrays.sort(drawing);
          return drawing;
     }
}
```

Storing an Object in a Session the Spring MVC Way

- Spring MVC advocates an alternative way to store data in the session.
- Initialize the object you intend to store in the session by returning it from a method annotated with **@ModelAttribute**.

```
@ModelAttribute("emp")
public Employee initEmployee() {
  return new Employee("John", "Doe", 50);
}
```

  - For session scoped data, this method is called once when the user first accesses this controller.
  - The name you specify in the **@ModelAttribute** annotation is what you will use for further lookups on this object.
  - The name is the key, the returned object is the value.

- Identify the object as one that should be stored in the session by listing its name within the @**SessionAttributes** annotation on the class.

```
@Controller
@SessionAttributes("emp")
public class MyController{…}
```

- Inject the session scoped object into your request method using the **@ModelAttribute** annotation.

```
public void method(@ModelAttribute("emp") Employee e)
```

- Access it in JSP EL using the optional **sessionScope** implicit object:

```
${sessionScope.emp}
```

LottoSessions/src/web/LottoController2.java

```
…
@Controller
@SessionAttributes("lottoHistory")
public class LottoController2 {
     @Autowired
     private Hopper hopper;

     @ModelAttribute("lottoHistory")
     public List<Integer[]> initSessionObject() {
          List<Integer[]> list = new ArrayList<>();
          return list;
     }

     @RequestMapping("GetNumbers2.do")
     public String lotto(@RequestParam("howmany") int count,
         @ModelAttribute("lottoHistory") List<Integer[]> list) {

          list.add(getLottoNumbers(count));
          return "form2.jsp";
     }

     private Integer[] getLottoNumbers(int howMany) {
          Integer[] drawing = new Integer[howMany];
          hopper.reset();
          for (int i = 0; i < howMany; i++) {
               drawing[i] = Integer.parseInt(
                    hopper.drawBall().getValue());
          }
          Arrays.sort(drawing);
          return drawing;
     }
}
```

LottoSessions/src/web/LottoController2.java

```
…
<c:forEach var="drawing" items="${sessionScope.lottoHistory}">
     <tr>
          <c:forEach var="number" items="${drawing}">
               <td>${number}</td>
          </c:forEach>
     </tr>
</c:forEach>
…
```

Labs

1. Modify the States web application so that a user can browse through all 50 states one at a time using Next/Previous buttons.

2. Modify the Presidents web application that you built in a previous lab to also allow for navigation using Next/Previous buttons. Make this functionality work not only for browsing all of the presidents, but also for situations where the user has requested all of the presidents from a particular party, or all of the presidents who had a particular term length, etc.

3. Add to your Presidents application so that the user can also search for presidents using additional criteria: (last name, party, term length, etc.).

4. If you still have time, build a Spring MVC web application that allows browser based clients to access your Pig Latin program, or your Bingo program, or your Caesar Cipher program, or …

Chapter 4 - Validation

Objectives:

- Use Java EE validation annotations to validate web application inputs.
- Display validation errors with the form:errors tag.

Validation Annotations

- You can take advantage of Java EE annotations to simplify the validation logic in your web applications.
    - This validation is not intended to replace client-side validation, but rather supplement it.
- Add the annotations above the fields in your JavaBean to indicate your validation preferences.
    - Available annotations include:

```
@Min(0)
private int age;
@Max(500)
private int weight;
@Size(min=2, max=100)
private String firstName;
@NotNull
private String lastName;
@Pattern(regexp="\\d{5}(-\\d{4})?")
private String zipCode;
```

    - The annotations can also be combined together on the same field:

```
@Min(0)
@Max(120)
private int age;
```

Validation/src/data/Book.java

```
package data;

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Book {
	@NotNull
	@Size(min=13, max=13, message="ISBN must be of length 13")
	private String isbn;
	@NotNull
	private String title;
	@NotNull
	private String author;
	@Min(0)
	private int totalSold;
	@Min(1400)
	@Max(2020)
	private int yearPublished = 2016;
	…
}
```

Controller Support for Validation

- In your controller annotate a parameter whose datatype matches your bean with the @Valid annotation.

```
@RequestMapping("process.do")
public String add(@Valid Item item, Errors errors) {
   …
}
```

  - This instructs Spring MVC to validate the object when the method is called.
  - This is a command object whose properties are automatically populated by Spring MVC based on matching request parameters.

- Add an **Errors** object as a parameter to your method so you can determine if any validation problems occurred.

```
@RequestMapping("process.do")
public String add(@Valid Item item, Errors errors) {
   if (errors.getErrorCount() != 0) {
        return "itemForm.jsp";
   }
   itemDAO.addItem(item);
   return "results.jsp";
}
```

  - Spring MVC will load this object with errors, if they occur.
  - A non-zero result from **errors.getErrorCount()** indicates validation problems.
    - Redisplay the page that failed validation to have the user try again.

Validation/src/controllers/BookController.java

```
@Controller
public class BookController {
     @Autowired
     private BookDAO bookDAO;
     …
     @RequestMapping(path="AddBook.do",
                    method=RequestMethod.GET)
     public ModelAndView addBook() {
          // Prime the model with an empty book object so that
          // the form can populate it with values
          Book b = new Book();
          return new ModelAndView("newbook.jsp", "book", b);
     }

     @RequestMapping(path="AddBook.do",
                    method=RequestMethod.POST)
     public String addBook(@Valid Book book, Errors errors) {
          if (errors.getErrorCount() != 0) {
               return "newbook.jsp";
          }
          bookDAO.addBook(book);
          return "results.jsp";
     }
}
```

form:form

- Spring MVC provides its own custom tags to simplify form building and automate error reporting.
  - Declare the tag library with the **@taglib** directive.

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

- Use the **form:form** tag to create a form.

```
<form:form action="AddItem.do" modelAttribute="item">
```

  - Use the **action** attribute like you normally would to identify which request method to bind to.
  - Use **modelAttribute** to specify which command object the form maps to.
    - This object must be placed into the model with the same name before the form is displayed.
- **form:input** will result in an HTML text input box.

```
<form:input path="name" />
```

  - Rather than a **name** attribute, this tag has a **path** which maps to a property on the command object you previously identified with **modelAttribute**.
- The **form:errors** tag will display any validation errors that correspond to the property identified by the **path** attribute.

```
<form:errors path="name" />
```

  - If there are no corresponding errors, then this tag outputs nothing.

Validation/src/controllers/BookController.java
```
@Controller
public class BookController {
     @Autowired
     private BookDAO bookDAO;
     …
     @RequestMapping(path="AddBook.do",
                    method=RequestMethod.GET)
     public ModelAndView addBook() {
          // Prime the model with an empty book object so that
          // the form can populate it with values
          Book b = new Book();
          return new ModelAndView("newbook.jsp", "book", b);
     }
     …
}
```

Validation/WebContent/newbook.jsp
```
…
<%@ taglib uri="http://www.springframework.org/tags/form"
prefix="form"%>
…
<body>
     <h3>Add Book</h3>
     <form:form action="AddBook.do" modelAttribute="book">
          <table>
               <tr>
                    <td>ISBN:</td>
                    <td><form:input path="isbn" /></td>
                    <td><form:errors path="isbn" /></td>
               </tr>
               <tr>
                    <td>Title:</td>
                    <td><form:input path="title" /></td>
                    <td><form:errors path="title" /></td>
               </tr>
               …
          </table>
          <input type="submit" value="Add Book" />
     </form:form>
</body>
</html>
```

Other form tags include **form:password**, **form:textarea**, **form:checkbox**, **form:select**, **form:radiobutton**, and more.

Other Considerations

- In order for Spring MVC to recognize the validation annotations, you must enable annotation processing in the Spring config file with the **mvc:annotation-driven** element.

- Make sure to have Maven download the necessary jar files by adding the **hibernate-validator** dependency.

Validation/WebContent/WEB-INF/Books-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans …>

     <context:component-scan base-package="controllers"/>
     <mvc:annotation-driven/>  <!-- So validation works -->

     <bean id="dao" class="data.BookFileDAO"/>
</beans>
```

Validation/pom.xml

```
<project …>
     …
     <dependencies>
          <dependency>
               <groupId>org.springframework</groupId>
               <artifactId>spring-webmvc</artifactId>
               <version>4.2.4.RELEASE</version>
          </dependency>
          <dependency>
               <groupId>javax.servlet</groupId>
               <artifactId>jstl</artifactId>
               <version>1.2</version>
          </dependency>
          <!-- For validation to work -->
          <dependency>
               <groupId>org.hibernate</groupId>
               <artifactId>hibernate-validator</artifactId>
               <version>5.2.2.Final</version>
          </dependency>
     </dependencies>
</project>
```

Labs

1. Modify the States web application so that the add state form uses validation.