CST-339 Activity 2 Guide

Contents

Part 1: Creating Models, Views, and Controllers using Spring MVC

Overview

Goal and Directions:

In this activity you will code a number of Controllers, Models, and Views using Spring MVC and run the application as a Spring Boot application.

Execution

Execute this assignment according to the following guidelines:

1. Create a new Spring Boot Project by following steps in Appendix A from the Activity 1 Guide and naming your Group and Package Name as *com.gcu* and your Project Name *topic2-1*. Import the Spring Boot Project into the Spring Tool Suite by following steps from Appendix A from the Activity 1 Guide.
2. Update the Spring Component scanner package specification by updating the annotation in the *Topic21Application.java* source file. Resolve the error by hovering over the error and import the ComponentScan class into the code.

   @ComponentScan({ "com.gcu" })
3. Create a new file *HelloWorldController.java* in the *com.gcu.controller* package.
4. Add an annotation to the Controller so all requests get processed by the */hello* URI.
5. Add a Controller Request method named *printHello()* to handle a GET request to the */test1* URI and returns a String text response of *Hello World!*. Add the *@ResponseBody* to the method so raw text is returned from the Controller.

```java
@Controller
@RequestMapping("/hello")
public class HelloWorldController
{
    /**
     * Simple Hello World Controller that returns a String in the response body.
     * Invoke using /test1 URI.
     *
     * @return Hello World test string
     */
    @GetMapping("/test1")
    @ResponseBody
    public String printHello()
    {
        // Simply return a String in the response body (must use @ResponseBody annotation)
        return "Hello World!";
    }
```

6. Run the application. Open a browser and go to *localhost:8080/hello/test1*. The text response of *Hello World!* should be displayed in your browser. Take a screenshot.
7. Create a new HTML file named *hello.html* in the *src/main/resources/templates* directory.
8. Add the Tymeleaf *th* namespace to the beginning <*html*> tag.

```
<html xmlns:th="http://www.thymeleaf.org">
```

9. In the body add a <*h2*> tag with the Thymeleaf attribute set to print text from an attribute named *message*:

```
<h2 th:text="${message}">This is my default text</h2><br/>
```

10. Add a Controller Request method named *printHello(Model model)* to handle a GET request to the */test2* URI, which sets a model attribute of *message* to *Hello Spring MVC Framework!* and that returns a view of *hello*. This method should NOT have the *@ResponseBody* in the method.

```
/**
 * Simple Hello World Controller that returns a View Name along with a Model Attribute named message.
 * Invoke using /test2 URI.
 * @param model Model to bind to the View.
 *
 * @return View name hello
 */
@GetMapping("/test2")
public String printHello(Model model)
{
    // Simply return a Model with an attribute named message and return a View named hello using a passed in ModelMap
    model.addAttribute("message", "Hello Spring MVC Framework!");
    return "hello";
}
```

11. Run the application. Open a browser and go to *localhost:8080/hello/test2*. The text response of *Hello Spring MVC Framework!* should be displayed in your browser. Take a screenshot.
12. In the hello.html file add a new tag to print another message with an attribute of *message2*:

```
<h2 th:text="${message2}">This is more default text</h2><br/>
```

13. Add a Controller Request method named *printHello1()* to handle a GET request to the */test3* URI, in the method create an instance of a *ModelAndView* class, set model attributes of *message* to *Hello World from ModelAndView!* plus *message2* to *Another Hello World from ModelAndView!* plus set the view to *hello*, and that returns the instance of *ModelAndView*. This method should NOT have the *@ResponseBody* in the method.

```
/**
 * Simple Hello World Controller that returns a View Name along with a Model Attribute named message.
 * Invoke using /test3 URI.
 *
 * @return ModelAndView with both the Model and the View to render
 */
@GetMapping("/test3")
public ModelAndView printHello1()
{
   // Create a ModelAndView and then set an attribute named message and with a View named hello
   ModelAndView mv = new ModelAndView();
   mv.addObject("message", new String("Hello World from ModelAndView!"));
   mv.addObject("message2", new String("Another Hello World from ModelAndView!"));
   mv.setViewName("hello");
   return mv;
}
```

14. Run the application. Open a browser and go to *localhost:8080/hello/test3*. The text response with both messages should be displayed in your browser. Take a screenshot.
15. Add an anchor tag to the *hello.html* file that navigates the user back to the *test4* URI passing it an HTTP parameter named *message* that is set to the value of *Hello World from a Tymeleaf template!*.

```
<a th:href="@{test4?message=Hello World from a Tymelead template!}">Go to /test4</a><br/>
```

16. Add a Controller Request method named printHello(@RequestParam("message") String message, Model model) to handle a GET request to the */test4* URI with a HTTP request parameter, set model attribute of *message* to the message method argument (i.e. the HTTP Request Parameter), and that returns a view of *hello*. This method should NOT have the *@ResponseBody* in the method.

```
/**
 * Simple Hello World Controller that returns a View Name along with a Model Attribute named message.
 * Invoke using /test4 URI.
 * @param message HTTP Parameter named message to add to the Model for rendering.
 * @param model Model to bind to the View.
 *
 * @return View name hello
 */
@GetMapping("/test4")
public String printHello(@RequestParam("message") String message, Model model)
{
   // Simply return a Model attribute named message and return a View named hello
   model.addAttribute("message", message);
   return "hello";
}
```

17. Run the application. Open a browser and go to *localhost:8080/hello/test2*. Test the anchor tag in the hello view. Take a screenshot
18. Add additional anchor tags to the hello.html file that navigates the user back to the test3 URI and test2 URI.

```
<a th:href="@{test3}">Go to /test3</a><br/>
<a th:href="@{test2}">Go to /test2 </a><br/>
```

19. Run the application. Open a browser and go to *localhost:8080/hello/test2*. The text responses with the proper messages and a links to the '/test2', '/test3', and '/test4' URI's should be displayed. Validate all of the anchor tags work as expected. Take a screenshot.
20. Code the following Controller and View:
    a. Create a new Controller named *HomeControler.java* that is mapped to the root URI (i.e. '/').
    b. Create a new View named *home.html* that simply displays a *Welcome to CST-339 Topic 2 Activity* and an anchor tag to the */hello/test2* URI for a GET request to the *HelloWorldController*.
    c. Run the application. Open a browser and go to *localhost:8080*. The *Welcome to CST-339 Topic 2 Activity* message and a link to the */hello/test2* URI should be displayed. Validate the anchor tag works as expected. Take a screenshot.

d. Update the POM file to name the output JAR file *cst339activity*. Run a Maven build and test the JAR file out from a terminal window. Reference the steps as needed from the Activity 1 Guide.

Deliverables:

The following needs to be submitted as this part of the Activity:

a. Screenshots of the Views for the /hello, /test1, /test2, /test3, /test4, and / Controller Routes.

Part 2: Creating Forms with Data Validation using Spring MVC

Overview

Goal and Directions:

In this activity you will create a form that posts its data to a Controller Route. The form data will also be validated using the JSR-303 Data Validation framework

Execution

Execute this assignment according to the following guidelines:

1. Create a new Spring Boot Project by following steps in Appendix A from the Activity 1 Guide and naming your Group and Package Name as *com.gcu* and your Project Name *topic2-2*. Import the Spring Boot Project into the Spring Tool Suite by following steps from Appendix A from the Activity 1 Guide.
2. Update the Spring Component scanner package specification by updating the annotation in the *Topic22Application.java* source file. Resolve the error by hovering over the error and import the ComponentScan class into the code.

   @ComponentScan({ "com.gcu" })
3. Create a new file *LoginModel.java* in the *com.gcu.model* package. Add two private class member variables named *username* and *password* of type String both with getter and setter methods.
4. Create a new file *LoginController.java* in the *com.gcu.controller* package.
5. Add an annotation to the Controller so all requests get processed by the */login* URI.
6. Add a Controller Request method named *display(Model model)* to handle a GET request to the root / URI, that sets model attribute named *title* with a value of *Login Form*, sets another model attribute named *loginModel* with a value of an instance of a *LoginModel* class, and that returns a view named *login*.
7. Create a new HTML file named *login.html* in the *src/main/resources/templates* directory.
8. Add the Tymeleaf *th* namespace to the beginning <*html*> tag.

   ```
   <html xmlns:th="http://www.thymeleaf.org">
   ```
9. Put a *test* message in the body of the *login.html* page.

10. Run the application. Open a browser and go to *localhost:8080/login/*. The *test* message should be displayed.
11. In the body add a <*form*> tag with a Thymeleaf *action* attribute set to @{doLogin}, a Thymeleaf *object* attribute set to @{loginModel}, and the form *method* attribute set to *post*. Layout the form elements in a two-column table using the following form elements:
    a. Row 1 – Column 1 label set to User Name and Column 2 as an HTML input text field with a Thymeleaf *field* attribute set to **{username}*.
    b. Row 2 – Column 1 label set to Password and Column 2 as an HTML input password field with a Thymeleaf *field* attribute set to **{password}*.
    c. Row 3 – Column span set to 2 with a HTML form submit button.

```
<form action="#" th:action="@{doLogin}" th:object="${loginModel}" method="post">
    <table>
        <tr>
            <td>User Name:</td><td><input type="text" th:field="*{username}" ></td>
        </tr>
        <tr>
            <td>Password:</td><td><input type="password" th:field="*{password}" ></td>
        </tr>
        <tr>
            <td colspan="2"><input type="submit" value="Submit" /></td>
        </tr>
    </table>
</form>
```

12. Add a Controller Request method named *doLogin(LoginModel loginModel, BindingResult bindingResult, Model model)* to handle a POST request to the */doLogin* URI that prints the *LoginModel* username and password properties to the console and that returns a view named *login.*

```
@Controller
@RequestMapping("/login")
public class LoginController
{
    @GetMapping("/")
    public String display(Model model)
    {
        // Display Login Form View
        model.addAttribute("title", "Login Form");
        model.addAttribute("loginModel", new LoginModel());
        return "login";
    }

    @PostMapping("/doLogin")
    public String doLogin(LoginModel loginModel, BindingResult bindingResult, Model model)
    {
        // Print the form values out
        System.out.println(String.format("Form with Username of %s and Password of %s", loginModel.getUsername(), loginModel.getPassword()) );

        // Navigate back to the Login View
        return "login";
    }
}
```

13. Run the application. Open a browser and go to *localhost:8080/login/*. The *Login Form* should be displayed. Validate the Login Form works as expected. Take a screenshot of the Login Form displayed in the browser and the *LoginModel* values beings displayed in the Console window.
14. Create a new file *OrderModel.java* in the *com.gcu.model* package. Add five private class member variables: *id* of type Long, *orderNo* and *productName* both of type String, *price*

of type float, and *quantity* of type int, all with getter and setter methods, and with a non-default constructor that initializes all class member variables.

15. Update the *doLogin()* Controller method to create a List of *OrderModel* using some default values, set model attribute of *title* to the value of *My Orders*, set model attribute of *orders* to the instance of the *OrderModel* list, and that returns a view of *orders.*

```
// Create some Orders
List<OrderModel> orders = new ArrayList<OrderModel>();
orders.add(new OrderModel(0L, "0000000001", "Product 1", 1.00f, 1));
orders.add(new OrderModel(1L, "0000000002", "Product 2", 2.00f, 2));
orders.add(new OrderModel(2L, "0000000003", "Product 3", 3.00f, 3));
orders.add(new OrderModel(3L, "0000000004", "Product 4", 4.00f, 4));
orders.add(new OrderModel(4L, "0000000005", "Product 5", 5.00f, 5));
```

16. Create a new HTML file named *orders.html* in the *src/main/resources/templates* directory.
17. Add the Tymeleaf *th* namespace to the beginning <*html*> tag.

```
<html xmlns:th="http://www.thymeleaf.org">
```

18. Put a *test* message in the body of the *orders.html* page.
19. Run the application. Open a browser and go to *localhost:8080/login/*. Test the form submit navigates to the Orders View properly.
20. Update the *orders.html* page by displaying the List of *OrderModel* in a four column HTML table using the following table layout:
    a. Header Row: Labels of Order Number, Product Name, Price, and Quantity.
    b. Tbody Row: a Thymeleaf *if* attribute set to *${orders.empty}* with a 4 column span set to a message of *No Orders Available*.
    c. Tbody Row: a Thymeleaf *each* attribute set to *$order : ${orders}* with each table data column using a Thymeleaf *text* attribute set ${order.[ORDER PROPERTY]} where ORDER PROPERTY are *orderNo, productName, price,* and *quantity* respectively.

```
<table style="width:50%">
    <thead>
        <tr>
            <th style="text-align:center">Order Number</th>
            <th style="text-align:center">Product Name</th>
            <th style="text-align:center">Price</th>
            <th style="text-align:center">Quantity</th>
        <th>
    </thead>
    <tbody>
        <tr th:if="${orders.empty}">
            <td style="text-align:center" colspan="4"> No Orders Available </td>
        </tr>
        <tr th:each="order : ${orders}">
            <td style="text-align:center"><h5 th:text="${order.orderNo}">0000000001</h5></td>
            <td style="text-align:center"><h5 th:text="${order.productName}">Product 1</h5></td>
            <td style="text-align:center"><h5 th:text="${order.price}">1.00</h5></td>
            <td style="text-align:center"><h5 th:text="${order.quantity}">1</h5></td>
        </tr>
    </tbody>
</table>
```

21. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form to display the Orders View. Take a screenshot.
22. Add JSR-303 Data Validation support to the application by adding the following entry to list of dependencies in the POM file. Right click on your project and select the Maven->Update Project menu option to update the project.

```
<!--  For Data Validation -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

23. Update the *LoginModel* class to add *@NotNull* and *@Size* annotation data validation rules to the username and password properties. The annotations should be imported from the *javax.validation.constraints* package.

```
@NotNull(message="User name is a required field")
@Size(min=1, max=32, message="User name must be between 1 and 32 characters")
private String username;

@NotNull(message="Password is a required field")
@Size(min=1, max=32, message="Password must be between 1 and 32 characters")
private String password;
```

24. Update the *LoginController* class to enable data validation on the *LoginModel* parameter by adding the *@Valid* annotation and then checking for data validation errors by calling the *hasErrors()* method on the *bindingResult* parameter and if errors navigating back to the Login View.

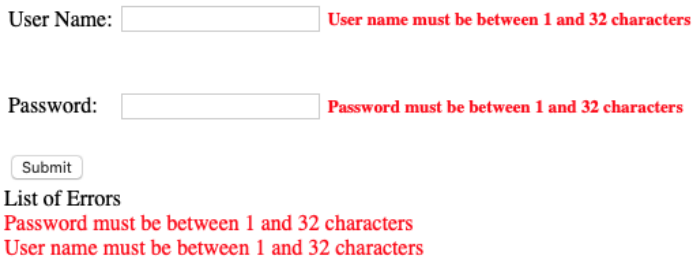
```
@PostMapping("/doLogin")
public String doLogin(@Valid LoginModel loginModel, BindingResult bindingResult, Model model)
{
    // Check for validation errors
    if (bindingResult.hasErrors())
    {
        model.addAttribute("title", "Login Form");
        return "login";
    }

    // Create some Orders
    List<OrderModel> orders = new ArrayList<OrderModel>();
    orders.add(new OrderModel(0L, "0000000001", "Product 1", 1.00f, 1));
    orders.add(new OrderModel(1L, "0000000002", "Product 2", 2.00f, 2));
    orders.add(new OrderModel(2L, "0000000003", "Product 3", 3.00f, 3));
    orders.add(new OrderModel(3L, "0000000004", "Product 4", 4.00f, 4));
    orders.add(new OrderModel(4L, "0000000005", "Product 5", 5.00f, 5));

    // Display the Orders View
    model.addAttribute("title", "My Orders");
    model.addAttribute("orders", orders);
    return "orders";
}
```

25. Update the Login View page to add field level data validation error messages for the username and password form fields. The data validation errors should be displayed in a new table data column. The field level error message can be accessed by a Thymeleaf *errors* attribute set to **{username}* and **{password}*for each table data column along with a Thymeleaf *if* attribute set to *${#fields.hasErrors('username')}* and *${#fields.hasErrors('password')}*. A list of errors can be displayed by a Thymeleaf *each*

26. attribute set to a value of *err : ${#fields.errors('*')}* and a <*span*> tag displaying each error using a Thymeleaf *text* attribute set to a value of *err : ${err}*.

```
<form action="#" th:action="@{doLogin}" th:object="${loginModel}" method="post">
    <table>
        <tr>
            <td>User Name:</td><td><input type="text" th:field="*{username}" ></td><td><h5 style="color:red" th:if="${#fields.hasErrors('username')}" th:errors="*{username}">User Name Error</h5></td>
        </tr>
        <tr>
            <td>Password:</td><td><input type="text" th:field="*{password}" ></td><td><h5 style="color:red" th:if="${#fields.hasErrors('password')}" th:errors="*{password}">Password Error</h5></td>
        </tr>
        <tr>
            <td colspan="3"><input type="submit" value="Submit" /></td>
        </tr>
    </table>
    <div th:if="${#fields.hasErrors('*')}">List of Errors</div>
    <div th:each="err : ${#fields.errors('*')}">
        <span style="color:red" th:text="${err}" class="fieldError"></span>
    </div>
</form>
```

27. Run the application. Open a browser and go to *localhost:8080/login/*. Submit the form with validation errors and verify that the field level data validation errors and list of data validation errors are displayed properly. Take a screenshot.



Deliverables:

The following needs to be submitted as this part of the Activity:

a. Screenshot of Login Form with no data validation.
b. Screenshot of Login Form posted values in the Console window.
c. Screenshot of Login Form with data validation errors.

Part 3: Creating Layouts using Thymeleaf

Overview

Goal and Directions:

In this activity you will create reusable page fragments and display pages using a common layout using Thymeleaf Layouts.

Execution

Execute this assignment according to the following guidelines:

1. Add Thymeleaf Layout support to the application by adding the following entry to list of dependencies in the POM file. Right click on your project and select the Maven->Update Project menu option to update the project.

```
<!--  For Thymelead Layout Dialect (Apache Tiles like behavior -->
<dependency>
    <groupId>nz.net.ultraq.thymeleaf</groupId>
    <artifactId>thymeleaf-layout-dialect</artifactId>
</dependency>
```

2. Create a directory named *layouts* in the *src/main/resources/templates* directory.
3. Create a new HTML file named *common.html* in the *src/main/resources/templates/layouts* directory.
4. Add the Tymeleaf *th* namespace and *layout* namespace to the beginning <*html*> tag.

```
<html xmlns:th="http://www.thymeleaf.org" xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
```

5. Add the following common page fragments to support a page header and footer.
    a. Header: a <*div*> tag with a Thymeleaf *fragment* attribute set to *header* and the contents of the <*div*> tag with the following HTML elements:
        i. *Welcome to my CST-339 Spring Boot Activity Application* text in a *h1* tag that is centered on the screen.
        ii. Bootstrap NavBar that has a navigation button to the root */login/* URI.
        iii. Page title that is displayed with *h2* tag and a Thymeleaf *text* attribute set to a value of ${title}.
        iv. Image logo that is displayed with a *img* tag and a Thymeleaf *src* attribute set to a value of @{/images/gcu-logo.jpg}. Copy an image file named *gcu-logo.jpg* to the *src/main/resources/static/images* directory.
    b. Foot: a <*div*> tag with a Thymeleaf *fragment* attribute set to *footer and* the contents of the <*div*> tag with the following HTML elements.
    c. *Copyright 2020 Grand Canyon University* text in a *h5* tag that is centered and pinned to the bottom of the screen.

```
<!-- Common Header Fragment -->
<div th:fragment="header">
    <div align="center">
        <h1>Welcome to my CST-339 Spring Boot Activity Application</h1>
    </div>
    <div class="bs-example">
        <nav class="navbar navbar-default">
            <!-- Brand and toggle get grouped for better mobile display -->
            <div class="navbar-header">
                <button type="button" data-target="#navbarCollapse" data-toggle="collapse" class="navbar-toggle">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a href="#" class="navbar-brand">GCU</a>
            </div>
            <!-- Collection of nav links and other content for toggling -->
            <div id="navbarCollapse" class="collapse navbar-collapse">
                <ul class="nav navbar-nav">
                    <li><a href="https://www.gcu.edu" target="_blank">Home</a></li>
                    <li><a href="https://lopeshops.gcu.edu" target="_blank">Shop</a></li>
                    <li><a href="http://news.gcu.edu" target="_blank">News and Events</a></li>
                </ul>
                <ul class="nav navbar-nav navbar-right">
                    <li><a href="/">Start Page</a></li>
                </ul>
            </div>
        </nav>
    </div>

    <!-- Page Title and Log -->
    <h2 th:text="${title}">Default Message Text</h2><br/>
    <img th:src="@{/images/gcu-logo.jpg}"/><br/><br/>
</div>

<!--  Common Footer Fragment -->
<div th:fragment="footer"
    ><h5 style="color:purple; text-align: center;position: fixed;bottom: 0;width: 100%;">Copyright 2020 Grand Canyon University</h5>
</div>
```

6. Create a new HTML file named *defaultTemplate.html* in the *src/main/resources/templates/layouts* directory.
7. Add the Tymeleaf *th* namespace and *layout* namespace to the beginning <*html*> tag.

```
<html xmlns:th="http://www.thymeleaf.org" xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
```

8. Add the following HTML content to support a default page layout that includes a header and footer.
    a. <*head*> tag should import the Bootstrap and jQuery libraries.
    b. <*div*> tag in the HTML body with a Thymeleaf *replace* attribute set to a value of *layouts/common :: header*. Add a closing </*div*> tag. The content within the <*div*> tag can just be set to *My Header* text.
    c. <*section*> tag in the HTML body with an attribute set to a value of *layout: fragment=”content”*. Add a closing </*section*> tag. The content within the <*section*> tag can just be set to *My Body Contents* text.
    d. <*div*> tag in the HTML body with a Thymeleaf *replace* attribute set a value of *layouts/common :: footer*. Add a closing </*div*> tag. The content within the <*div*> tag can just be set to *My Footer* text.
    e. <*div*> tag in the HTML body that centers all content in the body.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>CST-339 Spring Boot Activity Application</title>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap-theme.min.css">
        <link rel="stylesheet" href="https://cdn.datatables.net/1.10.16/css/jquery.dataTables.min.css">
        <link rel="stylesheet" href="https://cdn.datatables.net/responsive/2.2.0/css/responsive.dataTables.css">
        <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
        <script src="https://cdn.datatables.net/1.10.16/js/jquery.dataTables.min.js"></script>
        <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
        <script src="//cdn.datatables.net/responsive/2.2.0/js/dataTables.responsive.js"></script>
        <style type="text/css">.bs-example { margin: 20px; }</style>
    </head>

    <body>
        <div  align="center">
            <!-- Insert Header Fragment -->
            <div th:replace="layouts/common :: header">My Header</div>

            <!-- Main Body Content -->
            <section layout:fragment="content">
                <p>My Body Contents</p>
            </section>

            <!-- Insert Footer Fragment -->
            <div th:replace="layouts/common :: footer">My Footer</div>
        </div>
    </body>
</html>
```

9. Update the *login.html* page to use the default page layout by adding the *layout:decorate* attribute to the <*html*> tag and enclosing the form with an opening (and closing) <*div*> tag with an attribute of *layout:fragment* set to a value of *content*.

```
<html xmlns:th="http://www.thymeleaf.org" xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
                                          layout:decorate="layouts/defaultTemplate">
```

```
    <body>
        <div layout:fragment="content">
```

10. Update the *orders.html* page to use the default page layout by adding the *layout:decorate* attribute to the <*html*> tag and enclosing the table with an opening (and closing) <*div*> tag with an attribute of *layout:fragment* set to a value of *content*.

```
<html xmlns:th="http://www.thymeleaf.org" xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
                                          layout:decorate="layouts/defaultTemplate">
```

```
    <body>
        <div layout:fragment="content">
```

11. Run the application. Open a browser and go to *localhost:8080/login/*. Take a screenshot of the Login page with the new layout. Submit the form without validation errors. Take a screenshot of the Orders page with the new layout.

12. Update the POM file to name the output JAR file *cst339activity*. Run a Maven build and test the JAR file out from a terminal window. Reference the steps as needed from the Activity 1 Guide.

Deliverables:

The following needs to be submitted as this part of the Activity:

a. Screenshot of Login page using Layouts.
b. Screenshot of Orders page using Layouts.

Research Questions

1. Research Questions: For traditional ground students, answer the following questions in a Word document:
    a. How does Spring MVC support the MVC design pattern? Draw a diagram that supports the answer to this question.
    b. Research and identify 2 MVC Frameworks other than Spring MVC. What are the frameworks and how do they differ from Spring MVC?

Final Activity Submission

1. In a Microsoft Word document complete the following for the Activity Report:
    a. Cover Sheet with the name of this assignment, date, and your name.
    b. Section with a title that contains all the screenshots for each part of the Activity.
    c. Section with a title that contains the answers to the Research Questions (traditional ground students only).
2. Submit the Activity Report to the Learning Management System (LMS).