**Topic:** SpringMVC

**SPRING MVC PROJECT**

**STEPS**

1. Create a Dynamic Web project and call it “SpringMVCProject”

2. Add the necessary Spring jar files into the lib folder, including the commons logging.

**Main Steps**

**1. Modify web.xml**

**2. Create spring-dispatcher-servlet.xml**

**3. Create the Controller.java class (Controller)**

**4. Create the JSP page (View)**

**3. Modify web.xml to include the dispatcher-servlet:**

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

**4. In Web-INF folder, create “****dispatcher-servlet.xml”.**

(use New, other, Spring Bean Configuration File)

**a). Insert the code for namespaces and schema in the top of the file.**

<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:tx="<http://www.springframework.org/schema/tx>"

xmlns:context="<http://www.springframework.org/schema/context>"

xsi:schemaLocation="<http://www.springframework.org/schema/mvc> <http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd>

<http://www.springframework.org/schema/tx> <http://www.springframework.org/schema/tx/spring-tx-3.1.xsd>

<http://www.springframework.org/schema/beans> <http://www.springframework.org/schema/beans/spring-beans-3.1.xsd>

<http://www.springframework.org/schema/context> <http://www.springframework.org/schema/context/spring-context-3.1.xsd> http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd">

</beans>

**SETUP USING ONLY XML**

**b).** **Underneath all the name-spaces declare three beans – HandlerMapping, Controller, and viewResolver:**

<bean id=”HandlerMapping” class=”org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping”/>

<bean name=”/welcome” class=”com.lab2.hellocontroller.HelloController” />

<bean id=”viewResolver”

class=”org.springframework.web.servlet.view.InternalResourceViewResolver”>

<property name=”prefix”>

<value>/WEB-INF/</value>

</property>

<property name=”suffix”>

<value>.jsp</value>

</property>

</bean>

**5. Create a package “com.lab2.hellocontroller” and create HelloController.java class** inside of it. HelloController needs to extend AbstractController class and override its ModelAndView method.

NOTE: make sure when you import classes to import them all from servlet, not portlet, otherwise your controller will error out.

@Override

Protected ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletResponse res) throws Exception {

ModelAndView modelandview = new ModelAndView(“hello-page”);

modelandview.addObject(“welcomeMessage”, “Hi User, welcome to the first Spring MVC Application”);

return modelandview;

}

**6. Create hello-page.jsp in WEB-INF folder, that will welcome the user by using the ${welcomeMessage} expression language in HelloController.**

**7. Run the application and include the /welcome.html path in the url in order to display the hello-page.jsp**

Your URL path should look like this:

http://localhost:8080/SpringMVCProject/welcome

**Reforming Using ANNOTATIONS**

**8. in dispatcher-servlet.xml file add the component scan:**

<context:component-scan base-package=”com.lab2.hellocontroller” />

(component scan will scan packages for controller classes and look at its other annotations, such as @RequestMapping and map the url endpoint to the provided JSP view).

NOTE: Make sure to add the name space for context in dispatcher-servlet.xml on top among xmlns declarations:

xmlns:context=”http://www.springframework.org/schema/context

**9. in spring-dispatcher-servlet.xml file ERASE the lines:**

<bean id=”HandlerMapping” class=”org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping”/>

<bean name=”/welcome.html” class=”com.lab2.hellocontroller.HelloController” />

**10. HelloController Class will now look like this:**

@Controller

public class HelloController {

@RequestMapping(“/”)

public ModelAndView helloWorld(){

ModelAndView mav = new ModelAndView(“hello-page”);

mav.addObject(“welcomeMessage”, “hello world”);

return mav;

}

}

**Using MULTI-ACTION CONTROLLER**

11. Add another @RequestMapping method to the HelloController for displaying another view for a different URL endpoint:

@RequestMapping(“/hi”)

public ModelAndView hiWorld(){

ModelAndView mav = new ModelAndView(“HelloPage”);

mav.addObject(“welcomeMessage”, “hi world”);

}

(Here we are using same JSP page view but the data is different if we use /hi URL endpoint).

**Path Variable**

**12. Inside HelloController add a path variable to RequestMapping to use helloWorld method’s logic.**

@RequestMapping(“/welcome/{countryName}/{userName}”)

public ModelAndView helloWorld(@PathVariable(“userName”) String name, @PathVariable(“countryName”) String country){

ModelAndView modelAndView = new ModelAndView(“HelloPage”);

modelAndView.addObject(“welcomeMessage”, “hello “ + name + “, you are from “ + country);

return model;

}

Run your application and type in your path: /welcome/MushroomKingdom/Mario

NOTE: for the use of many pathvariables in one url, lets refactor the code with a Map.

@RequestMapping(“/welcome/{countryName}/{userName}”)

public ModelAndView helloWorld(@PathVariable Map<String, String> pathVars){

String name = pathVars.get(“userName”);

String country = pathVars.get(“countryName);

ModelAndView modelAndView = new ModelAndView(“HelloPage”);

model.addObject(“msg”, “hello “ + name + “, you are from “ + country);

return model;

}

NOTE: make sure to place **<mvc:annotation-driven />** in your servlet-dispatcher.xml file for compatibility with more annotations in this project.

**HTML form and handling with @RequestParam annotation.**

**SETUP**

1. Create a new package “com.lab3.studentadmissioncontroller”.

2. Create a new class “StudentAdmissionController” inside that package.

3. Create JSPs “admission-form.jsp” and “admission-success.jsp”

4. Add component scan in “spring-dispatcher-servlet.xml” for the base package of “studentadmissioncontroller”.

**13. in StudentAdmissionController…**

(don’t forget to add @Controller annotation for StudentAdmissionController)

@RequestMapping(value=”/admission-form”, method=RequestMethod.GET)

public ModelAndView getAdmissionForm(){

ModelAndView mav = new ModelAndView(“admission-form”);

return model;

}

@RequestMapping(value=”submit-admission-form”, method = RequestMethod.POST)

public ModelAndView submitAdmissionForm(@RequestParam(“studentName”) String studentName, @RequestParam(“studentHobby”) String studentHobby){

ModelAndView model = ModelAndView(“admission-success”);

model.addObject(“studentName”, studentName);

model.addObject(“studentHobby”, studentHobby);

return model;

}

NOTE: just like in previous example, for many parameters we can use Map and only one @RequestParam like this:

public ModelAndView

submitAdmissionForm(@RequestParam Map<String, String> reqPar){

String name = reqPar.get(“studentName”);

String hobby = reqPar.get(“studentHobby”);

ModelAndView model = ModelAndView(“admission-success”);

model.addObject(“studentName”, studentName);

model.addObject(“studentHobby”, studentHobby);

return model;

}

**14. in admission-form.jsp…**

<h1>Student’s Admission Form for Coding Classes</h1>

<form action=”/SpringMVCProject/submit-admission-form” method=”post”>

Student’s Name: <input type=”text” name=”studentName” />

Student’s Hobby: <input type=”text” name=”studentHobby” />

<input type=”submit” value=”Submit” />

</form>

**15. in admission-success.jsp…**

<h2>${studentName}, you are now registered for the Coding Classes.</h2>

<h3>Thanks for letting us know about your ${studentHobby} hobby.</h3>

**MODEL ATTRIBUTE and its use**

**16. Create a Student class**

Create a Student class with private fields for studentName and studentHobby with getters and setters.

**17. Instantiate Student class in StudentAdmissionController and use its variables.**

Inside “submitAdmissionForm” method:

Student student01 = new Student();

student01.setStudentName(studentName);

student01.setStudentHobby(studentHobby);

ModelAndView mav = new ModelAndView(“admission-success”);

mav.addObject(“headerMessage”, “Welcome to Student Admission Section!”);

model.addObject(“student01”, student01);

return mav;

}

NOTE: don’t forget to change the variables in the expression language on “admission-success.jsp” accordingly in order for studentHobby and studentName to display its values.

**18. Refactor this code with @ModelAttribute**

(@ModelAttribute lets us instantiate Student class and add right away a new object of that class to the model object)

The new code for “submitAdmissionForm” method looks like this:

@RequestMapping(“/submit-admission-form/”)

public ModelAndView submitAdmissionForm(@ModelAttribute(“student01”) Student student01){

ModelAndView mav = new ModelAndView(“admission- success”);

return model;

}

NOTE: the way request parameters for studentName and studentHobby were mapped to the new student object that got passed by @ModelAttribute, was by matching the name attribute in the input fields on the initial HTML form. You have to make sure the form names and the field names in the Student class match.

**@ModelAttribute at Method Level**

If you write @ModelAttribute above a method, you can add data to its attribute which will be available to all types of request handler methods that are present within that class.

a).

@ModelAttribute

public void addingCommonObjects(Model model){

model.addAttribute(“headerMessage”, “Our SpringMVC App Header!”);

}

b). Add “headerMesage” to every single jsp page that we have so far with Expression Language.

**DATA BINDING and its usage with Date**

**19. In “admission-form.jsp” add input fields for “Student’s Mobile”, “Date of Birth” and checkbox fields for “Student’s Skills” with values “Java Core”, JSP/Servlets”, “Spring MVC” and “Oracle SQL”.**

**20. Add corresponding fields in Student class with getters and setters.**

private Long studentMobile;

private Date studentDOB;

private ArrayList<String> studentSkills;

NOTE: make sure the names of the fields in the class match with the names inside the HTML form of “admission-form.jsp”.

**21. Update the “admission-succes.jsp”.**

It should look something like this:

<h1>${headerMessage}</h1>

<h3>Congratulations ${student01.studentName}, you are enrolled in the course!</h3>

<table>

<tr>

<td>Student name :</td>

<td>${student01.studentName}</td>

</tr>

<tr>

<td>Student hobby :</td>

<td>${student01.studentName}</td>

</tr>

<tr>

<td>Student Mobile :</td>

<td>${student01.studentMobile}</td>

</tr>

<tr>

<td>Student DOB :</td>

<td>${student01.studentDOB}</td>

</tr>

<tr>

<td>Student Skills :</td>

<td>${student01.studentSkills}</td>

</tr>

</table>

**Data Binding with a User-Defined Type**

Here we are going to create another class “Address” which will be instantiated as “studentAddress” in student class. Then, in HTML form, we will bind the Address’s properties to the “studentAddress” instance.

**22. Create “Address” class with fields and getters and setters.**

private String country;

privateString city;  
 private String street;

private int postalCode;

private String aptNo;

// getters/setters

**23. Instantiate Address class inside Student class**

private Address studentAddress;

// getters/setters

**24. Update the “admission-form.jsp” with the input fields for studentAddress and data-bind them.**

<table>

<tr>

<td>country :</td>

<td><input type=”text” name=”studentAddress.country”</td>

</tr>

<tr>

<td>city :</td>

<td><input type=”text” name=”studentAddress.city”</td>

</tr>

<tr>

<td>street :</td>

<td><input type=”text” name=”studentAddress.street”</td>

</tr>

<tr>

<td>postal code :</td>

<td><input type=”text” name=”studentAddress.postalCode</td>

</tr>

<tr>

<td>Apartment No. :</td>

<td><input type=”text” name=”studentAddress.aptNo</td>

</tr>

</table>

**25. Update the “admission-success.jsp” to display studentAddress values:**

country: ${student01.studentAddress.country}

city: ${student01.studentAddress.city}

street: ${student01.studentAddress.street}

postal code: ${student01.studentAddress.postalCode}

apartment number: ${student01.studentAddress.aptNo}

(find out more on @Init Binder and its methods)

**@INIT BINDER annotation**

**what if after submitting the form we wanted to bind all the input values except for Student’s phone number? That’s where @Init Binder comes in.**

**26. In StudentAdmissionController, add initBinder method for Student’s mobile.**

// add on very top right after the class declaration

@InitBinder

public void initBinder(WebDataBinder binder){

binder.setDisallowedFields(new String[] {“studentMobile”});

}

**27. Create a rule which will modify the date format submission.**

// inside initBinder method

SimpleDateFormat dateFormat = new SimpleDateFormat(“yyyy-mm- dd);

binder.registerCustomEditor(Date.class, “studentDOB”, new CustomDateEditor(dateFormat, false));

NOTE: Spring provides Property Editor classes (CustomDateEditor is one of them). In order to customize data-binding for a particular type, you have to use one of Property Editor classes, such as ClassEditor, CustomBooleanEditor and so on.

If one of those Property Editors isn’t able to help you with your requirements, you can create your own PropertyEditor class.

In this example we want to create one that will check if userName input field has gender displayed, such as Ms. or Mr.

**28. Create Custom Property Editor class and call it “StudentNameEditor.**

**SKIP THIS**

public class StudentNameEditor extends PropertyEditorSupport {

// When you will submit the admission form

// Spring MVC will run setAsText function of this class

// before performing data binding task for studentName // property of student object

@Override

public void setAsText(String studentName) throws IllegalArgumentException {

if(studentName.contains(“Mr.”) || studentName.contains(“Ms.”)){

setValue(studentName);

} else {

studentName = “Ms.” + studentName;

setValue(studentName);

}

}

**29. Register CustomEditor in StudentAdmissionController class.**

// right underneath the line of registering customEditor for studentDOB add…

binder.registerCustomEditor(String.class, “studentName”, new StudentNameEditor());

**FORM VALIDATIONS**

(review of the whole app - watch if needed)

JSR-303 (Java Specification Request) is a Java Bean Validation framework that has been approved by the JCP (Java Community Process) as of 16th of November 2009 and is accepted as part of the Java EE 6 specification.

**30. Add Jars for the validation.**

Go to <http://hibernate.org/validator/downloads/>

(classmate-1.0.0, hibernate-validator-5.1.2.Final, jboss-logging-3.1.1.GA, validation-api-1.1.0Final)

**31. Add @Size validation for Student hobby in Student class.**

@Size(min=2, max=30)

private String studentHobby;

**32. Add @Valid and check the look of “submitAdmissionForm” in StudentAdmissionController.**

@RequestMapping(value=”/submitAdmissionForm.html”, method = RequestMethod.POST)

public ModelAndView submitAdmissionForm(@Valid @ModelAttribute(“student01”) Student student01, BindingResult result){

if (result.hasErrors()) {

ModelAndView model1 = new ModelAndView(“admission- form”);

return model1;

}

ModelAndView model = new ModelAndView(“admission- success”);

return model;

}

To display errors in your JSP page when the validation fails, you have to add the taglib directive for the spring’s form on top of the page.

<@taglib class=”org.springframework/spring/tag” />

Also, you need to add spring forms’ tag for errors to display a line of error message.

<form:errors path=”student01.\*” />

That code will display error messages for all the fields that fail the validation. If you want to create individual error message for each failed class field, you will have to use multiple lines and specifically declare the field of the student object like this:

<form:errors path=”student01.studentName” />

**33. Add custom message for @Valid for Student Hobby in Student class**

@Size(min=2, max=30, message = “please, enter a value for studentHobby field between {min} and {max} characters”)

private String studentHobby;

**SPRING MESSAGE SOURCE with Properties file**

**34. Add bean for properties file in “dispatcher-servlet.xml”**

// below ViewResolver

<bean id=”messageSource”

class=”org.springframework.context.support.ReloadableResourceBun dleMessageSource”>

<property name=”basename” value=”/WEB- INF/studentmessages” />

</bean>

**35**. **Create “studentmessages.properties” file inside WEB-INF folder.**

// inside properties file, add…

Size.student01.studentHobby = “please enter a value for studentHobby field between 2 and 30 characters”

//better to rewrite it though variables that link to the actual data in the class, in case it gets changed.

Size.student01.studentHobby = “please enter a value for {0} field between {2} and {1} characters.

**EXPLANATION:**

1. Placeholder {0} is always replaced dynamically by Spring MVC with the name of field for which violation occurs.

2. Placeholders {1}, {2} etc would be replaced with the arguments values passed to the constraint annotation.

Arguments are ordered alphabetically, that’s why “max” is {1} and “min” {2}.

NOTE: Spring MVC would search for a key matching with this key pattern - [ValidationAnnotationName].[ObjectReferenceName].[FieldName]

(Size.student01.studentHobby)

If above key pattern is not found, then Spring MVC would again search for a key matching with this key pattern - [ValidationAnnotationName].[FieldName]

(Size.studentHobby)

If above key pattern is not found, then Spring MVC would again search for a key matching with this key pattern - [ValidationAnnotationName].[FieldName]

(Size.java.lang.String)

If above key pattern is still not found, then Spring MVC would finally search for a key matching with this key pattern - [ValidationAnnotationName]

(Size)

(default error message that Spring provides).

**More Form­­­ VALIDATIONS**

**36. Add @Pattern annotation with regular expression to studentName field in Student class to make sure it doesn’t contain any digits.**

@Pattern(regexp=”[^0-9]\*”)

private String studentName;

NOTE: to learn more about patterns (regular expressions) which you can use as an argument in the @Pattern annotation, visit:

<http://en.wikipedia.org/wiki/Regular_expression>

**37. Add @Past annotation to “studentDOB” field to make sure the date of birth is in the past.**

@Past

private Date studentDOB;

**EXERCISE:**

Add a couple more Validation rules for the fields.

Make sure the mobile phone number is 11 values (including country code).

Use @NotNull and @NotEmpty on couple more fields in Student Class.

**Custom Form VALIDATIONS**

**38. Create a custom form validation which would check student’s hobby and make sure it’s either “music”, “sports”, “gaming”, “yoga” or “gameofthrones”.**

**a). Create an Interface “IsValidHobby” having complete definition of this custom annotation.**

@Documented

@Constraint(validateBy = HobbyValidator.class)

@Target( { ElementType.FIELD })

@Retention(RetentionPolicy.RUNTIME)

public @interface IsValidHobby {

String message() default “Please provide a valid Hobby; “ +

“accepted hobbies are - Music, Sports, Gaming, Yoga, GameOfThrones (choose any one)”;

Class<?>[] groups() default {};

Class<? extends Payload>[] payload() default {};

**b). Create validator class used by a custom annotation. It has to implement ConstraintValidator interface and override two of its methods (with the name- initialize and isValid).**

public class HobbyValidator implements ConstraintValidator<IsValidHobby, String> {

@Override

public void initialize(IsValidHobby isValidHobby) {

}

@Override

public Boolean isValid(String studentHobby, ConstraintValidatorContext ctx) {

if(studentHobby == null) {

return false;

}

if(studentHObby.matches(“Music|Sports|Gaming|Yoga|GameOfThron es”)){

return true;

} else {

return false;

}

}

**c). Add custom annotation to studentHobby.**

@Size(min=2, max=30) @IsValidHobby

private String studentHobby;

**More Custom Form VALIDATIONS**

**39. Lets REFACTOR @IsValidHobby so it accepts arguments as allowed hobbies when adding it in Student class.**

**skipped**

**INTERCEPTOR**

Intercepts request on particular mapping and checks for certain rules before handing over the requested path. For instance:

If we want to make sure that on Sundays our website gives a message that website is “taking a break on Sunday”, Interceptor will check for the day of the week, and will print out the message if the request is made on Sunday.

**Steps:**

**1.** Include a Java class which extends HandlerInterceptorAdapter class and overrides one of its methods with the name preHandle.

**2.** Write the code in the preHandle method which you want Spring MVC to execute before handling the request.

**3.** Put an entry of this newly added Java class in the Spring’s configuration file.

**39. Include a Java class SundayAccessInterceptor.**

public class SundayAccessInterceptor extends HandlerInterceptorAdapter {

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

Calendar cal = Calendar.getInstance();

// getting the day on which request is made

int dayOfWeek = cal.get(cal.DAY\_OF\_WEEK);

if(dayOfWeek == 1) { // 1 is Sunday

response.getWriter().write(“Yo, it’s Sunday, I am resting! Come back tomorrow!”);

return false;

}

return true;

}

NOTE: except for “preHandle” HandlerInterceptorAdapter provides also two important methods - “postHandle” and “afterCompletion”.

**40. Add the bean to spring-dispatcher.xml**

// under annotation-driven add:

<mvc:interceptors>

<bean class=”com.lab2.studentadmissioncontroller.SundayAccessInterceptor” />

</mvc:interceptors>

NOTE: Here we are asking Spring MVC to consider this Interceptor class to execute before the application for every single url request. If we wanted the Interceptor to execute only for a specific url path, the following code modification has to be done.

<mvc:interceptors>

<mvc:interceptor>

<mvc:mapping path=”/admission-form.jsp” />

<bean class=”com.lab2.studentadmissioncontroller.SundayAccessInterceptor” />

</mvc:interceptor>

</mvc:interceptors>

**41. Add postHandle and afterCompletion methods to SundayAccessInterceptor class.**

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modeAndView) throws Exception {

// this method is called after Spring MVC executes the request handler method for the request.

System.out.println(“HandlerInterceptorAdapter : Spring MVC called postHandle method for “ + request.getRequestURI().toString());

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) thows Exception {

// this method is called after the response object is produced by the view for the request

System.out.println(“HandlerInterceptorAdapter : Spring MVC called afterCompletion method for “ + request.getRequestURI().toString());

}

**INTERNATIONALIZATION and LOCALIZATION**

We are going to modify our application to display everything in French if the user requests so.

**42. Transfer all the text fields labels from “admission-form.jsp” to “studentmessages.properties” file.**

In studentmessages.properties file we are going to add:

label.studentName = Student’s Name

label.studentHobby = Student’s Hobby

label.studentMobile = Student’s Mobile

label.studentDOB = Student’s DOB

label.studentSkills = Student’s Skills

label.studentAddress = Student’s Address

label.country = country

label.city = city

label.street = street

label.pincode = Postal Code

label.submit.admissionForm = Submit

label.admissionForm = Student Admission Form for Coding Courses

label.headerMessage = Platform Coding Program

**43. Add spring tag inside “admission-form.jsp” to support place holders.**

On top of the page underneath the previous tag add:

<%@ taglib prefix=”spring” uri=”http.//www.springframework.org/tags” %>

**44. Replace all the label fields in “admission-form.jsp” with placeholder code.**

Examples:

<tr><td><spring:message code=”label.studentName” /></td></tr>

<tr><td><spring:message code=”label.studentHobby” /></td></tr>

Do the same for the rest of the fields.

**45. Create “studentmessages\_fr.properties” file**

Before you do that, RENAME “studentmessages.properties” file to “studentmessages\_en.properties”.

Insert into “studentmessages\_fr.properties” French translation.

Size.student01.studentHobby = s’il vous plait entre rune valeur pour champ {0} entre {2} et {1} caractéres.

label.studentName = Nom de l'étudiant

label.studentHobby = Loisir de l’étudiant

label.studentMobile = Telephoné de l’étudiant

label.studentDOB = Le DOB de l’étudiant

label.studentSkills = Competences Set de l’étudiant

label.studentAddress = Adresse de l’étudiant

label.country = pays

label.city = ville

label.street = rue

label.pincode = code PIN

label.submit.admissionForm = Soumettre

label.admissionForm = Formulaire étudiant d’admission pour cours de codage

label.headerMessage = Programme de codage de Platform

**46. Test your application with language preferences in Chrome.**

Go to Chrome Settings, go to Languages and input settings, add French language and bring it on top above English. Check your application.

**CREATE** links so the user can change the language preference from English to French on top of the Admission form.

STEPS:

1. include two links - English and French on top of the AdmissionForm.jsp page.

2. Register an interceptor with the name “localeChangeInterceptor” in the spring’s configuration file.

3. Include an entry of cookieLocaleResolver bean in the spring’s configuration file.

**47. Provide English and French links on top of “admission-form.jsp”**

<a href=”/admission-form.jsp?siteLanguage=en”>English</a> |

<a href=”/admission-form.jsp?siteLanguage=fr”>English</a>

**48. Register the interceptor in the configuration file (spring-dispatcher.xml)**

Inside <mvc:interceptors> tag underneath first interceptor declaration add:

<mvc:interceptor>

<mvc:mapping = path”/admission-form” />

<bean class=”org.springframework.web.servlet.i18n.LocaleChangeInterceptor”>

<property name=”paramName” value=”siteLanguage” />

</bean>

</mvc:interceptor>

**49. Add cookieLocaleResolver to the configuration file.**

Underneath the bean “messageSource” add:

<bean id=”LocaleResolver” class=”org.springframework.web.servlet.i18n.CookieLocaleResolver” />

**MULTIPLE THEMES**

**skipped**

**EXCEPTION HANDLING**

**50. Create a JSP page “NullPointerException.jsp”**

<body>

<h1>Platform Coding</h1>

<h3>Application has encountered a Null Pointer error Please contact support by sending an email to willneveranswer@goodluck.com.</h3>

</body>

**51. Add throws Exception to “getAdmissionForm” method inside StudentAdmissionController class.**

right after “getAdmissionForm()” add:

throws Exception {

String exceptionOccured = “NULL\_POINTER”;

if(exceptionOccured.equalsIngnoreCase(“NULL\_POINTER”)) {

throw new NullPointerException(“Null Pointer Exception”);

}

ModelAndView model = new ModelAndView(“admission-form”);

return model;

}

**52. Include a method with @ExceptionHandler annotation inside “StudentAdmissionController” class**

at the bottom of the file add:

@ExceptionHandler(value = NullPointerException.class)

public String handleNullPointerException(Exception e){

//logging Null Pointer Exception

System.out.println(“Null Pointer Exception Occurred: “ + e);

return “NullPointerException”;

}