|-----------------------------------------------------------------

| Complete Project Step By Step with

| Spring, Hibernate, Maven, JS and Bootstrap

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Prerequisite - Tools

|-----------------------------------------------------------------

1. Install JDK latest version and path setup along with JAVA

2. Install Git

3. Unzip eclipse

4. Apache Tomcat (I am going to use version 8.5)

5. H2 Database

We would also be needing a GitHub account to push the project from local machine to GitHub.

|-----------------------------------------------------------------

| Part 01 - Project Directory and uploading in GitHub repository

|-----------------------------------------------------------------

--------------------------------------------------------------------

1. Create an empty git repository on the Github. Select a repository name.

2. Check to add the README.md file and select your technology to add it inside the gitignore file.(i have choosed JAVA)

3. Copy the github repository url.

4. Open the eclipse IDE

5. Go to Window -> Show View -> Other -> Type Git on the filter and select Git Repositories

6. Now clone this repository at your desired location in which we are going to keep the project specific files (both front-end and back-end project)

7. Create the maven webapp project. Add it to the repository by right clicking on the project -> Team -> Share Project. Select the git repository which was created above.

8. Similary, create the maven quick start project and add it to the same git repository

9. Open the gitignore file and ignore the target folder of both the projects as shown. (for example target folder)

10. Open window -> Preference -> Filter for Git configuration select it then add two entries if it is not present

11. Click on Add Entry

Key - user.name

Value - Your Name

12. Again click on Add Entry

Key - user.email

Value - Your Email Id

13. Right click on the repository select Commit -> Git Staging view. Select all the files in the unstaged changes and drag it to the Staged Changes.

14. Enter your commit message. Click on Commit and Push. It will ask for you enter your github account username and password.

TO BE CONTINUED in PART 02 - Building the user interface in webapp project.

|-----------------------------------------------------------------

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 02 - Building the user interface in webapp project

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 01 - Solving Error and Updating for latest versions of servlet

|-----------------------------------------------------------------

1. We will get the error -

The superclass "javax.servlet.http.HttpServlet" was not found on the Java Build Path

2. We will be using all the latest configuration to work in our project so first we will be changing the version of JRE inside pom.xml

3. To work with the latest version of servlet change the web.xml file version

Right Click on Project -> Maven -> Update Project

4. You will get an error as version mismatch

Solution - Right Click on Project -> Properties -> Project Facets -> Uncheck Dynamic Web Module. Ensure java version is to the version you have.

Right Click on Project -> Maven -> Update Project

TO BE CONTINUED in PART 02 - 02 - Adding Spring Dependency and Printing a message using Spring WebMvc.

|-----------------------------------------------------------------

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 02 - Building the user interface in webapp project

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 02 - Adding Spring Dependency and Printing a message

|-----------------------------------------------------------------

1. Add the spring dependency in the pom.xml file

2. Configure the FrontController that will intercept all the incoming request first. It would be done inside web.xml

3. Create a directory with the following structure.

--src

----main

------java

4. Add a package in src/main/java directory keeping the following convention for naming it use the convention

groupid.artifactid.XXX (where XXX is the group of classes doing specific work)

5. We will be creating a controller class that will handle various url mapping like /, /index, /home all will be mapped to the same method handler. It will generate the data (greeting in our case) and along with it the logical view name.

6. Create a views folder inside WEB-INF and add a page.jsp file inside that folder.

--src

----main

------webapp

--------WEB-INF

----------views

------------page.jsp

TO BE CONTINUED in PART 02 - 03 - Creating a Spring Bean XML Configuration file to integrate everything together.

|-----------------------------------------------------------------

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 02 - Building the user interface in webapp project

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 03 - Creating a Spring Bean XML Configuration file.

|-----------------------------------------------------------------

1. We need to create a dispatcher-servlet file for creating the bean to resolve the view for every mapping and we are scanning the package where other controllers could be found.

NOTE: - The name of the file will be the servlet\_name-servlet.xml as per the convention else we have to do additional configuration that will be shown later.

2. Configure the ViewResolver inside the dispatcher-servlet to map the logical name to the actual physical filename.

3. Add the apache tomcat server.

4. Adding the jstl dependency.

TO BE CONTINUED in PART 02 - 04 - Understanding @RequestParam and @PathVariable

|-----------------------------------------------------------------

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 02 - Building the user interface in webapp project

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 04 - Understanding @RequestParam and @PathVariable

|-----------------------------------------------------------------

1. A query string is any string that is part of the URL that starts after the '?' where data is transferred through URL in the form of 'key=value' pair.

http://localhost:8080/onlineshopping/test?greeting=hi&key1=value1&key2=value2

http://localhost:8080/onlineshopping/test?greeting=hello

http://localhost:8080/onlineshopping/test?greeting=welcome

2. In order to keep the URL clean and make it more SEO friendly we need to access the value from the path for example: -

http://localhost:8080/onlineshopping/test/hi

http://localhost:8080/onlineshopping/test/hello

http://localhost:8080/onlineshopping/test/welcome

TO BE CONTINUED in PART 02 - 05 - Creating Landing Page and Loading Static Resources.

|-----------------------------------------------------------------

|

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 02 - Building the user interface in webapp project

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 05 - Creating Landing Page and Loading Static Resources.

|-----------------------------------------------------------------

1. Create a directory inside the

--src

----main

------resources

------webapp

--------assets

2. Get the bootstrap js, css and fonts files. Also create the directory for images and place it inside the assets directory created in above step.

3. Search for unstyled bootstrap starter template.

4. Bootstrap javascript code is dependent on jquery so we will copy that also.

5. In the dispatcher-servlet.xml file we have to add the mvc namespace.

6. Configure it to load the static resources.

7. Add the spring url tag library.

8. Create three variables name css, js and images using spring:url element.

9. Replace the css, js and images path with the variable name created above.

10. Commit the changes done till now.

TO BE CONTINUED in PART 02 - 06 - Creating Master Page

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 02 - Building the user interface in webapp project

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 06 - Creating Master Page

|-----------------------------------------------------------------

1. Identify the different sections of the page. i.e. the navigation bar, the main section and the footer section.

2. Separate out each section into different jsp file so that it becomes easier to modify later.

3. Include the files that we have separated out in the page.jsp

4. Now we will be adding the title and also make sure that the main content is only loaded when the user clicks on home by adding two objects in the model.

5. Modify the navigation bar to include the contact and about us page

6. Add the request mapping in the page controller.

7. Test the link.

TO BE CONTINUED in PART 02 - 07 - Solving Active Menu Problem using jQuery

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 02 - Building the user interface in webapp project

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 07 - Solving Active Menu Problem using jQuery

|-----------------------------------------------------------------

1. Let us add the id to the list of elements in the navbar.jsp

2. Add a small script element in the head section and create a new property with the name of menu in the window global object.

3. Create a myapp.js file in the js directory created earlier and add a script element at the end.

4. Add a function when the document is ready using the jquery short hand function.

5. Using a switch case element we would be adding a class of active to the menu list to which we have assigned the id in the first step.

TO BE CONTINUED in PART 02 - 08 - Fixing the footer problem and changing the theme of the page.

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 02 - Building the user interface in webapp project

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 08 - Fixing the footer problem and changing the theme of the page.

|-----------------------------------------------------------------

1. Wrap the entire body code inside another div element.

2. Wrap the page content section inside another div element.

3. We will be using percentage so we need to specify the parent element i.e. body and html having 100% as the width.

4. Next we will be adding css styling properties to fix it.

5. To add a theme to the project we can download a theme from bootswatch.com

6. Commit the changes done till now.

TO BE CONTINUED in PART 03 - Linking Backend Project with Frontend Project.

|-----------------------------------------------------------------

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 03 - Linking Backend Project with Frontend Project.

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 01 - Creating DTO and DAO classes.

|-----------------------------------------------------------------

1. We will start designing the classes required for our online shopping website.

2. First create the packages to organize your classes

NOTE: if you are using the convention as I'm doing it so it would be

groupid.artifactid.dto (for classes that will be used to transfer data to database table where dto stands for Data Transfer Object)

groupid.artifactid.dao (for interfaces that will be useful in accessing the data where dao stands for Data Access Object)

groupid.artifactid.daoimpl (for classes that will implements interfaces which will be used for accessing the data)

3. We will start by creating the Category class in the backend project inside the dto package.

4. We are going to create the DAO (data access object) interface and DAOImpl classes to access the data.

5. Add some dummy data to a static list for testing purpose which would be replaced later by access to records in database tables.

TO BE CONTINUED in PART 03 - 02 - Linking Backend Project with UI.

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 03 - Linking Backend Project with Frontend Project.

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 02 - Linking Backend Project with UI

|-----------------------------------------------------------------

1. Add the backend dependency to the frontend project.

2. We are going to use the dependency injection feature of Spring framework and 'Autowired' the class which implements DAO interface.

3. Add the @Repository annotation. The @Repository annotation is a marker for class specifying that it fulfills the role of providing access to the data and will be managed by spring framework.

Error - Repository cannot be resolved

Solution - We will add the spring context dependency in the backend project as well

4. Modify the user interface to access the category in the sidebar.

5. Error - org.springframework.beans.factory.UnsatisfiedDependencyException

Solution -

Is to let the spring framework know where to get that categoryDAO dependency from. Add an context element in dispatcher servlet to scan the packages in the backend.

TO BE CONTINUED in PART 03 - 03 - Updating the URL to access Dynamic Page.

|-----------------------------------------------------------------

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 03 - Linking Backend Project with Frontend Project.

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 03 - Updating the URL to access Dynamic Page

|-----------------------------------------------------------------

1. Modify the URL in the sidebar.jsp and navbar.jsp

2. Add two request mapping one to display all the products and display products based on Category but we would be using the same view for both the mapping.

3. Add a new DAO method in CategoryDAO to fetch a single category based on its id.

4. Create a view called listProducts.jsp. This page will be further divided into two to display the category on the side and other to display the data table.

5. We will add the bootstrap component of breadcrumb to test our page.

Home/Category/Mobile

6. Modify page.jsp to load the view based on user click.

7. Activate the menu of View Products and rearrange the menu as per your requirement.

8. Activate the menu of category as well.

9. Commit the changes done till now.

TO BE CONTINUED in PART 03 - 04 - Getting H2 Database and Hibernate Dependency.

|-----------------------------------------------------------------

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 03 - Linking Backend Project with Frontend Project.

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| 04 - Getting H2 Database and Hibernate Dependency

|-----------------------------------------------------------------

1. Get the database from the given url and extract it to a location of your choice.

http://www.h2database.com

2. Select the Generic H2 (Server) and you can create a database of your choice.

NOTE:

driver : - org.h2.Driver

url: - jdbc:h2:tcp://localhost/~/onlineshopping

username: - sa

password: -

3. Add the following dependency in the backend project to work with database.

com.h2database -> h2 -> 1.4.193

org.hibernate -> hibernate-core -> 5.2.7.Final

org.springframework -> spring-orm -> ${spring.version}

4. To enable connection pooling we are going to use another dependency

org.apache.commons -> commons-dbcp2 -> 2.1.1

TO BE CONTINUED in PART 03 - 05 - Hibernate Configuration to work with H2 Database.

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 03 - Linking Backend Project with Frontend Project.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 05 - Hibernate Configuration to work with H2 Database

|--------------------------------------------------------------------

1. Create a package to keep all the application related configuration file inside the config package.

2. Create a HibernateConfig class and annotate it with the @Configuration, @ComponentScan and @EnableTransactionManagement

3. Create five static final variables to provide the database related information

DATABASE\_URL - jdbc:h2:tcp://localhost/~/onlineshopping

DATABASE\_DRIVER - org.h2.Driver

DATABASE\_DIALECT - org.hibernate.dialect.H2Dialect

DATABASE\_USERNAME - sa

DATABASE\_PASSWORD -

4. Three beans will be created i.e.

-- DataSource (providing the connection information of the database)

-- SessionFactory (configures application to use Hibernate and create a session object)

-- HibernateTransactionManager (to manage the hibernate transactions)

TO BE CONTINUED Part 03 - 06 - Creating our first @Entity class

|-----------------------------------------------------------------

| Part 03 - Linking Backend Project with Frontend Project.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 06 - Creating our first @Entity class

|--------------------------------------------------------------------

1. Keep the database queries in an sql file so that we dont have to write it again and again.

2. Write the create table query for Category and exceute it on the database.

3. Annotate the class with the @Entity annotation the entity name will be same as the class unless you want to change it.

4. All the entity class needs to have an id so we are going to annotate it using @Id

5. Use the auto generation strategy of IDENTITY

6. Change the column name as per the fields in table.

7. It always a good idea to add the toString method for checking purpose and solving errors.

8. Modify the DAO implementing class to add a new category to the database table using hibernate.

TO BE CONTINUED Part 03 - 07 - Writing JUnit Test Case

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 03 - Linking Backend Project with Frontend Project.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 07 - Writing JUnit Test Case

|--------------------------------------------------------------------

1. We will be using the latest version of JUnit test case. Changing version to 4.12

2. Create a CategoryTestCase class in the net.kzn.shoppingbackend.test

3. The class will be using the AnnotationConfigApplicationContext and CategoryDAO class.

4. Code the static init method and annotate it with @BeforeClass.

5. Write the test case method and annotate it with @Test.

6. Run the test case by right clicking on the class and Run As -> JUnit Test

TO BE CONTINUED Part 03 - 08 - Writing Complete JUnit Test Case for doing Category CRUD operation

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 03 - Linking Backend Project with Frontend Project.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 08 - CRUD Operation JUnit Test Case

|--------------------------------------------------------------------

1. Modify the DAO to include all the CRUD related methods

2. Modify the DAOImpl class to include all those methods.

3. Write the test case to get a single category.

4. Write the test case to update a single category.

5. Write a test case to delete a single category here deleting means to update the category is\_deleted fields as false.

6. Write a test case to fetch the list of active category

7. Write a single test case now integrating all the test cases together.

8. Check with the frontend project if categories are fetched properly now from the database as we have already removed the static list.

--------------------------------------------------------------------

TO BE CONTINUTED Part 04 - Working with DataTables to display Product Details

--------------------------------------------------------------------

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 04 - Working with DataTables to display Product Details.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 01 - Creating tables and classes for Products

|--------------------------------------------------------------------

1. First write the sql query for product and user table. Run in on the browser console to fill up some dummy data.

2. Next create the Product DTO class with the fields that we want to persist inside the database tables. Add a constructor to it that will generate a unique code for that table.

3. Let us create the ProductDAO interface along with the ProductDAOImpl class.

4. Write the test cases to ensure we can use the class without any problem in the ui project.

TO BE CONTINUED - Part 04 - 02 - Getting Product Data in JSON format.

|-----------------------------------------------------------------

| Part 04 - Working with DataTables to display Product Details.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 02 - Getting Product Data in JSON format

|--------------------------------------------------------------------

1. To get the product in JSON format we need to have a converter in the classpath for achieving this we will be adding new dependencies

in backend project

com.fasterxml.jackson.core -> jackson-databind -> 2.8.7

com.fasterxml.jackson.core -> jackson-annotations -> 2.8.7

2. Create a JsonDataController whose job will be to provide data in JSON format. At the class level we are going to add a RequestMapping of "/json/data" and for method we will add like all/products and category/{id}/products so the entire url will be

/json/data/all/products and

/json/data/category/{id}/products;

3. Install the Postman or RestClient extension for Google Chrome & Mozilla firefox respectively.

4. Open the extension and put the address of json/data in the address bar. Check for the JSON response.

5. Now we will add JsonIgnore annotation to ignore the field that we do not want to send in the form of JSON.

TO BE CONTINUED - Part 04 - 03 - Getting the jQuery DataTable plugin.

|-----------------------------------------------------------------

| Part 04 - Working with DataTables to display Product Details.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 03 - Getting the jQuery DataTable plugin

|--------------------------------------------------------------------

1. Go to the below website to download the jQuery DataTable plugin.

https://datatables.net/

2. Add the jquery.dataTable.js file to the proper location and link it in our project master page in proper order.

3. Create a new table element with some dummy fields inside the listProducts.jsp.

4. Code the javascript to fill the table with the some dummy data. Here the data will be a collection of some products. We will be using the data property to provide the dummy data. For more information on other property you can refer to the jQuery datatable documentation on how to use different data.

TO BE CONTINUED - Part 04 - 04 - Showing JSON data in jQuery DataTable plugin using Spring MVC.

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 04 - Working with DataTables to display Product Details.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 04 - Showing JSON data in jQuery DataTable plugin using Spring MVC

|--------------------------------------------------------------------

1. Add the other files in the project which is required for making dataTable boostrapish.

i.e. dataTables.bootstrap.js & dataTables.bootstrap.css

2. Modify the page.jsp file to add the contextRoot to window global object and also add categoryId to it in listProducts.jsp

3. We are going to prepare the jsonUrl in javascript to fetch the data in JSON format from the respective controller for this we will be using the contextRoot that was added previously.

contextRoot/json/data/all/products

contextRoot/json/data/category/{id}/products

4. Modify the jQuery dataTable code to display the data. We will be using the following properties:

ajax -> url and dataSrc

columns

5. We will use the following url for adding two anchor tags one to view the product and other to add the product to the cart.

The URL will be:

contextRoot/show/{id}/product

contextRoot/cart/add/{id}/product

6. We are also going to add the image in this dataTable.

TO BE CONTINUED - Part 04 - 05 - Showing the entire Product in Single Dynamic Page on click of View.

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 04 - Working with DataTables to display Product Details.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 05 - Showing the entire Product in Single Dynamic Page

| on click of view button.

|--------------------------------------------------------------------

1. Add the the request mapping to PageController to display the single product page.

2. Create a jsp page to display the product.

3. Include the single product page inside the master page that is already available.

TO BE CONTINUED - Part 04 - 06 - Using SLF4J Logging facility.

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 04 - Working with DataTables to display Product Details.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 06 - Using SLF4J Logging facility.

|--------------------------------------------------------------------

The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks, such as java.util.logging, logback and log4j. SLF4J allows the end-user to plug in the desired logging framework at deployment time.

1. We will be excluding the commomns-logging from the spring-core as well as from the commons-dbcp2 in backend project to use slf4j

2. We will be using the logback classic which implements slf4j directly and we have to use the jcl-over-slf4j dependency.

<!-- SLF4J Logging -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.1</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>jcl-over-slf4j</artifactId>

<version>1.7.24</version>

</dependency>

3. Create a separate logback.xml file in the src-->main-->resources directory.

It will consist of the following elements

|--appender (name, class)

| |----encoder

| |--- Pattern

|

configuration---|

|--logger (name, level, additivity)

| |--appender-ref (ref)

|

|--root (by default level is DEBUG so keep it simple to INFO)

|--appender-ref (ref)

4. We will add an appender for the console. Along with the following encoder pattern: %-5level %msg %d{HH:mm:ss.SSS} %logger{36} %n

%-5level – means the level of the logging event should be left justified to a width of five characters

%msg – outputs the application-supplied message associated with the logging event.

%d{HH:mm:ss.SSS} – Used to output the date of the logging event.

%logger{36} – outputs the name of the logger at the origin of the logging event. It takes an integer

%n – outputs the platform dependent line separator character or characters.

5. We will add the logger for displaying the DEBUG information.

6. Add the root element.

7. We will use it inside our controller let us first test it with the PageController.

TO BE CONTINUED - Part 04 - 07 - Global Exception Handling.

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 04 - Working with DataTables to display Product Details.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 07 - Global Exception Handling

|--------------------------------------------------------------------

1. We will create a package following our own standard i.e.

groupid.artifactid.exception

net.kzn.onlineshopping.exception

2. To handle the exception globally we will create a class annotate it using the ControllerAdvice annotation.

3. Let us start with handling the 404 error for our website. For this we will add an exception handler in our class first.

4. Create an error page to display the exception.

5. Lets create a user-defined exception for the Product class and throw that exception if product is null.

6. Add an exception handler and display the appropriate message.

7. We will also handle the generalize exception that would be displayed instead of the server error.

TO BE CONTINUED - Part 04 - 08 - Some of the bug fixes.

|-----------------------------------------------------------------

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 04 - Working with DataTables to display Product Details.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 08 - Bug fixes

|--------------------------------------------------------------------

1. We will tackle the similar issue in the single product page where we should not be adding the product to the cart whose quantity is zero and instead of that display a Out Of Stock.

2. Fixing the jQuery dataTable as well.

3. Commit and push project on GitHub.

TO BE CONTINUED - Part 05 - Spring form elements, Hibernate Validator and Multipart File Upload.

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 05 - Spring form elements, Hibernate Validator

| and Multipart File Upload.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 01 - Bootstrap Form Classes for Product Management

|--------------------------------------------------------------------

// Added ManagementController.java

1. One more controller class to do product management. In this we will add a RequestMapping on the class level that would be helpful for us for managing role based access to the application.

2. Create a request mapping to manage the products in the application this will display the jsp page that we will create in the next step for managing the products.

// Added manageProduct.jsp

3. Create a separate jsp file to manage the product details.

4. Add a link to manage products in the navbar.jsp and modify the page.jsp to show the manageProduct.jsp. (Test it!)

5. For designing form we will be using form-horizontal that allows us to avoid row class and we can use the column classes inside for designing the form.

form -> form-horizontal

div -> form-group

label -> control-label + column classes

div -> column classes

input -> form-control

span -> help-block

TO BE CONTINUED - Part 05 - 02 - Adding Spring Form and Linking with the Model

|-----------------------------------------------------------------

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 05 - Spring form elements, Hibernate Validator

| and Multipart File Upload.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 02 - Adding Spring Form and Linking with the Model

|--------------------------------------------------------------------

1. Add the spring form tags library in the page.jsp using taglib directive.

2. Modify the management controller handler method to return a new product model for form.

3. Modify the form designed previously to link with product model send in previous step. The form will now have elements based on the Product fields i.e.

name,

brand,

description,

unitPrice,

quantity,

categoryId,

4. For category we are going to create a method level @ModelAttribute and fetch it.

TO BE CONTINUED - Part 05 - 03 - Storing Product from Spring form to Database

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 05 - Spring form elements, Hibernate Validator

| and Multipart File Upload.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 03 - Storing Product from Spring form to Database

|--------------------------------------------------------------------

1. To handle the form submission add an action to the form and map it to a handler method in the ManagementController.

2. Add a logger object to log the product object for testing the value of product object.

3. Add hidden elements for the fields inside product which we are not modifying inside the form through form elements.

4. Autowire the productDAO bean in the ManagementController and in the handler method call the add method of it.

5. Check for the product just added.

TO BE CONTINUED - Part 05 - 04 - Validating Product Details using Hibernate Validator

\

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 05 - Spring form elements, Hibernate Validator

| and Multipart File Upload.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 04 - Validating Product Details using Hibernate Validator

|--------------------------------------------------------------------

1. We will add the hibernate validator dependency for our project to implement server side validation.

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>5.4.1.Final</version>

</dependency>

<dependency>

<groupId>javax.validation</groupId>

<artifactId>validation-api</artifactId>

<version>1.1.0.Final</version>

</dependency>

2. Modify the Product class and we can use the different annotation to add server side validation for the field.

List of annotations are :

@NotNull - Checks that the annotated value is not null.

@NotEmpty - Checks whether the annotated element is not null nor empty.

@NotBlank - Checks that the annotated character sequence is not null and the trimmed length is greater than 0. The difference to @NotEmpty is that this constraint can only be applied on strings and that trailing white-spaces are ignored.

@Size(min=, max=) - Checks if the annotated element’s size is between min and max (inclusive)

@Max(value=) - Checks whether the annotated value is less than or equal to the specified maximum

@Min(value=) - Checks whether the annotated value is higher than or equal to the specified minimum

@Email - Checks whether the specified character sequence is a valid email address. The optional parameters regexp and flags allow to specify an additional regular expression (including regular expression flags) which the email must match.

@Range(min=, max=) - Checks whether the annotated value lies between (inclusive) the specified minimum and maximum. Works with string and numbers.

@Length(min=, max=) - Validates that the annotated character sequence is between min and max included. Works with string.

3. Modify the manageProduct.jsp file to display the error message if the user do not give valid input for all the elements.

4. Modify the POST mapping for managing the product by annotating it using the @Valid annotation.

TO BE CONTINUED - Part 05 - 05 - Multipart File Upload

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 05 - Spring form elements, Hibernate Validator

| and Multipart File Upload.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 05 - Multipart File Upload

|--------------------------------------------------------------------

What is a Multipart?

A HTTP multipart request is a HTTP request that HTTP clients construct to send files and data over to a HTTP Server. It is commonly used by browsers and HTTP clients to upload files to the server.

1. Add a transient field along with setter and getter for uploading a file in the Product class. It would be transient as we don't have to persist it.

2. Change the product form enctype to multipart/form-data which indicates that when this form will be posted it consists of the form data along with the binary file. Also add the file element and link with the field created above.

3. Spring uses MultipartResolver interface to handle the file uploads in web application, two of the implementation :

StandardServletMultipartResolver – Servlet 3.0 multipart request parsing.

CommonsMultipartResolver – Classic commons-fileupload.jar

So, we are going to add a bean in the dispatcher-servlet.xml file

4. Configure the web.xml for uploading file.

fileSizeThreshold: We can specify the size threshold after which the file will be written to disk. The size value is in bytes, so 1024\*1024\*10 is 10 MB.

location: Directory where files will be stored by default, it’s default value is “”.

maxFileSize: Maximum size allowed to upload a file, it’s value is provided in bytes. It’s default value is -1L means unlimited.

maxRequestSize: Maximum size allowed for multipart/form-data request. Default value is -1L that means unlimited.

5. Create a separate class for uploading the file.

TO BE CONTINUED - Part 05 - 06 - Adding a new Category

|-----------------------------------------------------------------

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 05 - Spring form elements, Hibernate Validator

| and Multipart File Upload.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 06 - Adding a new Category

|--------------------------------------------------------------------

1. Add a new button that will target a modal dialog box in which we are going to provide an option to add a new category.

data-toggle

data-target

2. Design the modal dialog box to give an option for creating a category.

3. Add an ModelAttribute for category to link it with the category form.

4. Download the jQuery validate plugin and will work with client side validation for working with category.

5. Will require to write the rules and messages to validate the category form.

6. Add a handler to handle form submission of category.

7. Modify the message in manage/product handler were we are passing the message for displaying the alert message

TO BE CONTINUED - Part 05 - 07 - Editing a product

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 05 - Spring form elements, Hibernate Validator

| and Multipart File Upload.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 07 - Editing a product

|--------------------------------------------------------------------

1. Create a table to be displayed to the admin for all the products available in the website.

2.

TO BE CONTINUED - Part 05 - 08 - Activating and De-activating a product

|-----------------------------------------------------------------

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 05 - Spring form elements, Hibernate Validator

| and Multipart File Upload.

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 08 - Activating and De-activating a product

|--------------------------------------------------------------------

Removed @JsonIgnore from the active field.

TO BE CONTINUED - Part 06 - User Registration using Spring Webflow

TO BE CONTINUED - Part 06 - 01 - Entity classes for Registration Module

TO BE CONTINUED - Part 06 - 02 - Understanding Hibernate Mapping

TO BE CONTINUED - Part 06 - 03 - Understanding Spring Webflow

TO BE CONTINUED - Part 06 - 04 - Configuring Spring Webflow

TO BE CONTINUED - Part 06 - 05 - Preparing the views based on flow

TO BE CONTINUED - Part 06 - 06 - Configuring Membership flow file

TO BE CONTINUED - Part 06 - 07 - Writing a handler for flow

TO BE CONTINUED - Part 06 - 08 - Validating fields in the flow.

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 06 - User Registration using Spring Webflow

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 01 - Entity classes for Registration Module

|--------------------------------------------------------------------

1. Understanding the relationship among the various classes.

2. Designing the user class.

3. Designing the address class.

4. Designing the cart class.

5. Create the DAO and DAOImpl for the User entity.

6. Copy the logback.xml file from the front-end project to use the similar setting for logging

7. Create separate test class to write test cases for User.

user = new User() ;

user.setFirstName("Hrithik");

user.setLastName("Roshan");

user.setEmail("hr@gmail.com");

user.setContactNumber("1234512345");

user.setRole("USER");

user.setPassword("123456");

address = new Address();

address.setAddressLineOne("101/B Jadoo Society, Krissh Nagar");

address.setAddressLineTwo("Near Kaabil Store");

address.setCity("Mumbai");

address.setState("Maharashtra");

address.setCountry("India");

address.setPostalCode("400001");

address.setBilling(true);

address = new Address();

address.setAddressLineOne("201/B Jadoo Society, Kishan Kanhaiya Nagar");

address.setAddressLineTwo("Near Kudrat Store");

address.setCity("Mumbai");

address.setState("Maharashtra");

address.setCountry("India");

address.setPostalCode("400001");

address.setShipping(true);

8. Add the hbm2ddl.auto property in the HibernateConfig file to understand the mappings and let the hibernate create the tables for us.

update - create the table is not there or update if any changes are there to existing one.

create - drop the exisiting tables and re-create it.

TO BE CONTINUED - Part 06 - 02 - Understanding Hibernate OneToOne Uni-directional Mapping

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 06 - Working with Registration Module using Spring Web flow

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 02 - Understanding Hibernate OneToOne Uni-directional Mapping

|--------------------------------------------------------------------

1. Remove the userId field and create a user field of type User. Annotate it with OneToOne to work with relationship mapping.

2. Add a user and see the result.

TO BE CONTINUED - Part 06 - 03 - Understanding Hibernate OneToOne Bi-directional

Mapping

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 06 - Working with Registration Module using Spring Web flow

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 03 - Understanding Hibernate OneToOne Bi-directional Mapping

|--------------------------------------------------------------------

1. Let us understand the bi-directional relationship so that we can fetch the cart of the user. For this we will add a cart field in the User class.

NOTE: if we don't define who is the owner of the relationship we get the fields created in both the tables. To avoid it we can use the mappedBy attribute to define the relationship owner.

2. Remove the addCart method from the DAO and DAOImpl which is not required as we will creating the cart for the user if the user is a USER.

TO BE CONTINUED - Part 06 - 04 - Completing the Address Part Using ManyToOne

|-----------------------------------------------------------------

| Part 06 - Working with Registration Module using Spring Web flow

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 04 - Hibernate ManyToOne Mapping on Address

|--------------------------------------------------------------------

1. We will add the ManyToOne annotation on the Address class on the user field and remove the existing userId field.

2. Add methods to the UserDAOImpl related to address.

3. Write the test cases for Address to check if the list of addresses is fetched or not.

4. Analyze the query executed.

5. Make your choice!

TO BE CONTINUED - Part 06 - 05 - Understanding Spring Webflow

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 06 - Working with Registration Module using Spring Web flow

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 05 - Understanding Spring Webflow

|--------------------------------------------------------------------

TO BE CONTINUED - Part 06 - 06 - Configuring Spring Webflow

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 05 - Working with Registration Module using Spring Web flow

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 06 - Configuring Spring Webflow

|--------------------------------------------------------------------

1. First add the webflow dependency.

<dependency>

<groupId>org.springframework.webflow</groupId>

<artifactId>spring-webflow</artifactId>

<version>2.4.4.RELEASE</version>

</dependency>

2. Add the webflow namespace to dispatcher-servlet.xml file.

3. Add the flow-executor that will handle the flow indicating the entry point in the project.

<wf:flow-executor id="flowExecutor" />

NOTE:

We have to also specify the flow-registry attribute and specify the id of flow-registry element if not provided it would automatically look for element with id of flowRegistry

4. Add the flow-registry for finding the flows of the application.

<wf:flow-registry id="flowRegistry" base-path="/WEB-INF/views/flows">

<wf:flow-location-pattern value=“/\*\*/\*-flow.xml” />

</wf:flow-registry>

NOTE:

All flows are given IDs (which are ultimately used to reference the flow). The flow ID will be the directory path containing the flow definition file relative to the base path (the part represented with the double-asterisk).

Path to the flow file Flow Id

------------------------------------------------------------------------

/WEB-INF/views/flows/membership/membership-flow.xml membership

/WEB-INF/views/flows/register/signup-flow.xml register

/WEB-INF/views/flows/cart/checkout/checkout-flow.xml cart/checkout

In case if you do not want to use the base path you can use following for indicating the flow file and its id.

<wf:flow-registry id="flowRegistry">

<wf:flow-location path="/WEB-INF/views/flows/signup-flow.xml" />

</wf:flow-registry>

NOTE: the rules change for flow id when we are not using base-path for locating the flow file it is the file name itself i.e. "signup-flow"

Or you can be even more explicit about the flow ID by specifying an id attribute of the <wf:flow-location> element:

<wf:flow-registry id="flowRegistry">

<wf:flow-location id="register" path="/WEB-INF/views/flows/signup-flow.xml" />

</wf:flow-registry>

5. To customize the flow-registry and provide additional configuration we are going to use the flow-builder-services but first we would be adding

flow-builder-services="flowBuilderServices" in the registry.

<wf:flow-builder-services id="flowBuilderServices"

view-factory-creator="viewFactoryCreatory"

/>

6. Spring Web Flow provides a Spring MVC handler adapter called FlowHandlerAdapter. This handler adapter is the bridge between DispatcherServlet and the flow executor, handling requests and manipulating the flow based on those requests.

<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">

<property name="flowExecutor" ref="flowExecutor" />

</bean>

7. DispatcherServlet knows how to dispatch requests by consulting with one or more handler mappings. For web flows, FlowHandlerMapping helps DispatcherServlet know to send flow requests to the FlowHandlerAdapter:

<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">

<property name="flowRegistry" ref="flowRegistry" />

</bean>

Here for any incoming request, the Spring Webflow registry will be checked to see whether any guided flows are configured. That is why we make the order for Spring Webflow ‘-1’. The flow registry contains all the flows available in the application. If a guided flow is found then the flow executor is initiated.

8. For resolving views for web flow we will add one more bean which will point to the view resolver -

<bean id="viewFactoryCreator"

class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator">

<property name="viewResolvers">

<list>

<ref bean="viewResolver" />

</list>

</property>

</bean>

TO BE CONTINUED - Part 06 - 07 - Creating membership flow file.

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 06 - Working with Registration Module using Spring Web flow

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 07 - Creating flow file

|--------------------------------------------------------------------

1. We will create the directory and add flow file to it. Using the inbuild flow id we will trigger the flow.

Path to the flow file Flow Id

------------------------------------------------------------------------

/WEB-INF/views/flows/register/signup-flow.xml register

2. Add a menu to the navbar.jsp for registering a new user. We are going to use the flow id in the href along with our contextRoot to start the flow.

3. To execute the triggered flow we will we need to provide details about the state and view for the flow. It requires the namespace to use different flow elements. For getting the namespace we can refer webflow documentation.

http://docs.spring.io/spring-webflow/docs/2.4.4.RELEASE/reference/html/

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

<flow xmlns="http://www.springframework.org/schema/webflow"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/webflow

http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

</flow>

4. We will add a single view state for now to the flow and add the corresponding signup-personal.jsp view page and

5. We are going to add simple navbar for the flow

<nav class="navbar navbar-inverse navbar-fixed-top" role="navigation">

<div class="container">

<div class="navbar-header">

<a class="navbar-brand" href="">Home</a>

</div>

</div>

</nav>

6. Add the global transition to move to the home and end the state.

7. Add the end state for home so that we can cancelled the guided navigation and redirect the user to home.

8. Test everything!

9. Now when everything works well separate out the header and footer inside shared directory of flows.

WEB-INF

----views

--------flows

------------register

----------------signup-personal.jsp ../shared/flow-header.jsp &

../shared/flow-footer.jsp

------------shared

----------------flow-header.jsp ./flow-navbar.jsp

----------------flow-footer.jsp ../../shared/footer.jsp

----------------flow-navbar.jsp

TO BE CONTINUED - Part 06 - 08 - Spring webflow with model and navigation

|-----------------------------------------------------------------

| Part 06 - Working with Registration Module using Spring Web flow

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 08 - Spring webflow with model and all navigation

|--------------------------------------------------------------------

1. Create all the views for the flow and add the view-state element inside the flow.

signup-personal.jsp (view-state)

signup-billing.jsp (view-state)

signup-confirm.jsp (view-state)

signup-success.jsp (end-state)

2. Complete the signup-personal.jsp page by adding the spring form element as it requires modelAttribute we are going to use the var element of flow that will create a flow instance variable.

REFERENCE : PART 05 - 01 & 02 videos for working with Spring Form

<div class="row">

<div class="col-md-6 col-md-offset-3">

<div class="panel panel-primary">

<div class="panel-heading">

<h4>Sign Up - Personal</h4>

</div>

<div class="panel-body">

<sf:form

method="POST"

class="form-horizontal"

id="registerForm"

>

<div class="form-group">

<label class="control-label col-md-4">First Name</label>

<div class="col-md-8">

<sf:input type="text" path="firstName" class="form-control"

placeholder="First Name" />

</div>

</div>

<div class="form-group">

<label class="control-label col-md-4">Last Name</label>

<div class="col-md-8">

<sf:input type="text" path="lastName" class="form-control"

placeholder="Last Name" />

</div>

</div>

<div class="form-group">

<label class="control-label col-md-4">Email</label>

<div class="col-md-8">

<sf:input type="text" path="email" class="form-control"

placeholder="abc@zyx.com" />

</div>

</div>

<div class="form-group">

<label class="control-label col-md-4">Contact Number</label>

<div class="col-md-8">

<sf:input type="text" path="contactNumber" class="form-control"

placeholder="XXXXXXXXXX" maxlength="10" />

</div>

</div>

<div class="form-group">

<label class="control-label col-md-4">Password</label>

<div class="col-md-8">

<sf:input type="password" path="password" class="form-control"

placeholder="Password" />

</div>

</div>

<!-- radio button using bootstrap class of radio-inline -->

<div class="form-group">

<label class="control-label col-md-4">Select Role</label>

<div class="col-md-8">

<label class="radio-inline">

<sf:radiobutton path="role" value="USER" checked="checked"/> User

</label>

<label class="radio-inline">

<sf:radiobutton path="role" value="SUPPLIER"/> Supplier

</label>

</div>

</div>

<div class="form-group">

<div class="col-md-offset-4 col-md-8">

<!-- submit button inside the form -->

</div>

</div>

</sf:form>

</div>

</div>

</div>

</div>

3. Similarly we are going to complete the signup-billing.jsp view along with the billing address model.

<div class="row">

<div class="col-md-6 col-md-offset-3">

<div class="panel panel-primary">

<div class="panel-heading">

<h4>Sign Up - Address</h4>

</div>

<div class="panel-body">

<sf:form

method="POST"

class="form-horizontal"

id="billingForm"

>

<div class="form-group">

<label class="control-label col-md-4" for="addressLineOne">Address Line One</label>

<div class="col-md-8">

<sf:input type="text" path="addressLineOne" class="form-control"

placeholder="Enter Address Line One" />

</div>

</div>

<div class="form-group">

<label class="control-label col-md-4" for="addressLineTwo">Address Line Two</label>

<div class="col-md-8">

<sf:input type="text" path="addressLineTwo" class="form-control"

placeholder="Enter Address Line Two" />

</div>

</div>

<div class="form-group">

<label class="control-label col-md-4" for="city">City</label>

<div class="col-md-8">

<sf:input type="text" path="city" class="form-control"

placeholder="Enter City Name" />

</div>

</div>

<div class="form-group">

<label class="control-label col-md-4" for="postalCode">Postal Code</label>

<div class="col-md-8">

<sf:input type="text" path="postalCode" class="form-control"

placeholder="XXXXXX" />

</div>

</div>

<div class="form-group">

<label class="control-label col-md-4" for="state">State</label>

<div class="col-md-8">

<sf:input type="text" path="state" class="form-control"

placeholder="Enter State Name" />

</div>

</div>

<div class="form-group">

<label class="control-label col-md-4" for="country">Country</label>

<div class="col-md-8">

<sf:input type="text" path="country" class="form-control"

placeholder="Enter Country Name" />

</div>

</div>

<div class="form-group">

<div class="col-md-offset-4 col-md-8">

<!-- submit button for moving to the personal -->

<!-- submit button for moving to the confirm -->

</div>

</div>

</sf:form>

</div>

</div>

</div>

</div>

4. Complete the signup-confirm.jsp page where we would be able to see the details. It requires data from two different models.

<div class="row">

<!-- column to display the personal details -->

<div class="col-sm-6">

<div class="panel panel-primary">

<div class="panel-heading">

<h4>Personal Details</h4>

</div>

<div class="panel-body">

<!-- code to display the personal details -->

</div>

<div class="panel-footer">

<!-- anchor to move to the edit of personal details -->

</div>

</div>

</div>

<!-- column to display the address -->

<div class="col-sm-6">

<div class="panel panel-primary">

<div class="panel-heading">

<h4>Billing Address</h4>

</div>

<div class="panel-body">

<!-- code to display the communication address -->

</div>

<div class="panel-footer">

<!-- anchor to move to the edit of address -->

</div>

</div>

</div>

</div>

<!-- to provide the confirm button after displaying the details -->

<div class="row">

<div class="col-sm-4 col-sm-offset-4">

<div class="text-center">

<!-- anchor to move to the success page -->

</div>

</div>

</div>

5 The user will finally submit the details and it would be forwarded to success page.

<div class="row">

<div class="col-sm-offset-4 col-sm-4">

<div class="text-center">

<h1>Welcome!</h1>

<h3>onlineshopping.com</h3>

<h6>You can use your email address as username to login!</h6>

<div>

<a href="${contextRoot}/login" class="btn btn-lg btn-success">Login Here</a>

</div>

</div>

</div>

</div>

TO BE CONTINUED - Part 06 - 09 - Spring Webflow Using Composite Model And Handler.

|-----------------------------------------------------------------

| Part 06 - Working with Registration Module using Spring Web flow

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 09 - Spring Webflow Using Composite Model And Handler.

|--------------------------------------------------------------------

1. Create a RegisterModel class which will hold both the user and billing address.

2. Create a RegisterHandler class to provide the above model at the start of the flow along with both the flow instance variable.

3. On exit of each state store the flow instance inside the register model.

4. Use this model to display the data entered by the user inside the signup-confirm page.

5. Create an action state that will be called once the user submit the details.

6. Code the method to handle the user registration details.

TO BE CONTINUED - Part 06 - 10 - Spring Webflow With Validation

|-----------------------------------------------------------------

| Part 06 - Working with Registration Module using Spring Web flow

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| Part 06 - 10 - Spring Webflow With Validation - https://youtu.be/2x31awTOvRM

|--------------------------------------------------------------------

1. Add the hibernate validator annotation on the dto classes based on the validation you want to perform.

2. Add the sf:errors element for the respective field in the spring form of personal and address views.

REF: PART 05 - 04 - Validating Product Details using Hibernate Validator

3. Add the validator attribute to our flow-builder-services element to use the validation service. The bean with id "validator" will provide validation service to spring form in views specified through spring webflow configuration

<bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>

4. Test everything!

5. Write custom validation for checking uniqueness of the email id and also provide an additional field for confirm password. For this we would be creating an action state with an id of validate for personal view state.

TO BE CONTINUED - Part 07 - Authentication & Authorization Using Spring Security

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 07 - Authentication & Authorization Using Spring Security

|-----------------------------------------------------------------

Authentication - Who am I ? (username and password)

Authorization - Where can I go ? (has right authority)

|--------------------------------------------------------------------

| 01 - Spring Security Dependency and Configuration

|--------------------------------------------------------------------

1. Add the following dependency to the front-end project.

<spring.security.version>4.2.2.RELEASE</spring.security.version>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-config</artifactId>

<version>${spring.security.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-web</artifactId>

<version>${spring.security.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-taglibs</artifactId>

<version>${spring.security.version}</version>

</dependency>

2. Configure the web.xml file to load the spring security configuration file.

https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/

3. Create the spring-security.xml file in the WEB-INF directory and add the namespace to it.

<beans:beans xmlns="http://www.springframework.org/schema/security"

xmlns:beans="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/security

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

...

</beans:beans>

4. To enable web security all we need to add is the http element.

/\* -----> /abc

/\*\* -----> /abc/pqr or /abc/pqr/mno or /abc/pqr/mno/xyz or ...

Similarly,

/abc/\* ----> /abc/pqr or abc/mno or abc/xyz

/abc/\*\* ----> /abc/pqr or /abc/mno or /abc/mno/pqr or /abc/pqr/mno/ or ...

5. Provide the authentication manager and authentication provider which uses difference in built services to authenticate and authorize a user.

6. Understand the difference between hasRole and hasAuthority

TO BE CONTINUED - Part 07 - 02 - Spring Security Using jdbc-user-service

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 07 - Authentication & Authorization Using Spring Security

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 02 - Spring Security Using jdbc-user-service

|--------------------------------------------------------------------

1. We will add a new jdbc-user-service element and assign values to the three important attributes of it that is

data-source-ref

authorities-by-username-query

users-by-username-query

NOTE: we will get an error it is not able to find the dataSource bean.

2. Add the context:component-scan of our backend application to the spring-security.xml file and assign a name to the dataSource bean created before.

TO BE CONTINUED - Part 07 - 03 - Spring Security With Custom Login Page

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 07 - Authentication & Authorization Using Spring Security

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 03 - Spring Security With Custom Login Page

|--------------------------------------------------------------------

1. We are going to provide a custom login page at a request mapping of /login for this we will have to use the login-page attribute.

2. Add a custom login page mapping inside the page controller.

3. Create the login.jsp page with simple navigation bar.

<!-- Navigation -->

<nav class="navbar navbar-inverse navbar-fixed-top" role="navigation">

<div class="container">

<!-- Brand and toggle get grouped for better mobile display -->

<div class="navbar-header">

<a class="navbar-brand" href="${contextRoot}/home">Online Shopping</a>

</div>

</div>

</nav>

4. We are going to design a simple html form with the following two input elements textboxes having the names as

username (default name) to change it we have to use username-parameter

password (default name) to change it we have to use password-parameter

<div class="container">

<div class="row">

<div class="col-md-offset-3 col-md-6">

<div class="panel panel-primary">

<div class="panel-heading">

<h4>Login</h4>

</div>

<div class="panel-body">

<form action="${contextRoot}/login" method="POST" class="form-horizontal"

id="loginForm"

>

<div class="form-group">

<label for="username" class="col-md-4 control-label">Email: </label>

<div class="col-md-8">

<input type="text" name="username" id="username" class="form-control"/>

</div>

</div>

<div class="form-group">

<label for="password" class="col-md-4 control-label">Password: </label>

<div class="col-md-8">

<input type="password" name="password" id="password" class="form-control"/>

</div>

</div>

<div class="form-group">

<div class="col-md-offset-4 col-md-8">

<input type="submit" value="Login" class="btn btn-primary"/>

</div>

</div>

</form>

</div>

<div class="panel-footer">

<div class="text-right">

New User - <a href="${contextRoot}/register">Register Here</a>

</div>

</div>

</div>

</div>

</div>

</div>

5. Add a separate intercept-url pattern for displaying the login page.

6. Add a separate http element to allow loading of static resources which does not require any form of security.

7. Add jQuery validation for the login form elements.

TEST EVERYTHING!

TO BE CONTINUED - Part 07 - 04 - Handling CSRF In Spring Security

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 07 - Authentication & Authorization Using Spring Security

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 04 - Handling CSRF Using Spring Security

|--------------------------------------------------------------------

1. After clicking on the submit we get an error related to CSRF. By default csrf is enabled in spring security and it provides us with protection from CSRF attack right out of the box.

More can be read at:

https://www.incapsula.com/web-application-security/csrf-cross-site-request-forgery.html

To solve this we either

a. disable the csrf protection (not recommended)

b. or provide an hidden field that will keep the csrf token that only server recognize when the form is submitted.

<input type="hidden" name="${\_csrf.parameterName}" value="${\_csrf.token}"/>

2. To Handle invalid credentials inside the /login request handler. For this we will use the @RequestParam annotation on the login mapping.

3. For ajax request we need to send csrf token along with the request for that we will save the token inside the meta tag.

<meta name="\_csrf" content="${\_csrf.token}">

<meta name="\_csrf\_header" content="${\_csrf.headerName}">

4. Using the token stored we will construct the header for ajax request.

TO BE CONTINUED - Part 07 - 05 - Role Based Access Control

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 07 - Authentication & Authorization Using Spring Security

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 05 - Role Based Access Control

|--------------------------------------------------------------------

1. Specify the roles for accessing the specific part of the application.

2. Add the access-denied-handler element with the error-page attribute

3. Create the request mapping and handler method to display the view.

TO BE CONTINUED - Part 07 - 06 - Spring Security Password Encryption

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 07 - Authentication & Authorization Using Spring Security

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 06 - Spring Security Password Encryption

|--------------------------------------------------------------------

1. Modify the database table to incorporate hashed password using bcrypt mechanism.

2. Add a bean element to the spring-security.xml file with the fully qualified name as given below:

<beans:bean id="passwordEncoder" class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"/>

3. Autowired the bean inside the RegisterHandler that we created for registering a user and call the encode method on user entered password.

4. Since we are using hashed password now we have to decode the password that was encoded for this we will use the pasword-encoder element inside the authentication provider.

5. Test everything!

TO BE CONTINUED - Part 07 - 07 - Global Controller To Fetch User Model

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 07 - Authentication & Authorization Using Spring Security

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 07 - Global Controller To Fetch User Model

|--------------------------------------------------------------------

1. Create the menus to display the details.

2. Create a UserModel in which we are going to store fields that we require once the user logs in.

3. Create a GlobalController and a method that will return the UserModel with the details that is required for display.

TO BE CONTINUED - Part 07 - 08 - Displaying Menus and Buttons based on Role

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 07 - Authentication & Authorization Using Spring Security

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| Part 07 - 08 - Displaying Menus and Buttons based on Role

|--------------------------------------------------------------------

1. Open the navbar.jsp and add security tag library to it.

2. Use the authorize element along with its access attribute to provide access to a particular menu.

3. We will store the role value from the userModel inside window object of javascript and access it inside myapp.js to display the appropriate buttons.

TO BE CONTINUED - Part 07 - 09 - Spring Security Logout Handling

|

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 07 - Authentication & Authorization Using Spring Security

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| Part 07 - 09 - Spring Security Logout Handling

|--------------------------------------------------------------------

1. Since CSRF protection is enabled for our application we cannot use the logout filter that we have added inside the spring security file as per the documentation.

2. We are going to add our own custom logout handler by adding a request mapping and redirecting it again to the login page.

3. Display the logout success message to the user.

TO BE CONTINUED - Part 07 - 10 - Fixing Bug And Making DataTable Responsive

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 07 - Authentication & Authorization Using Spring Security

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| Part 07 - 10 - Fixing Bug For DataTable and Making it Responsive

|--------------------------------------------------------------------

1. Wrap the datatable inside a div with class of container-fluid.

2. Create another div with class of table-responsive.

3. Code the table-responsive class with the overflow-x property

TO BE CONTINUED - Part 08 - Cart Module for User

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 08 - Cart Module for User

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 01 - Creating DTO, DAO, DAOImpl and Test Class for CartLine

|--------------------------------------------------------------------

1. We will create a cart\_line table inside our database.

2. Let us see how our CartLine dto class looks like with the different relationship mapping that exist between other dto.

3. Inside our CartLineDAO interface apart from basic methods like list, get, add, update, delete we are going to have business method like getByCartAndProduct, listAvailable.

4. Let us implement our dao method apart from the common ones.

5. Also move the updateCart method from the UserDAO to CartLineDAO

6. Let us test the method of CartLineDAO by writing a test case.

TO BE CONTINUED - Part 08 - 02 - Creating the Cart View

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 08 - Cart Module for User

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 02 - Creating the Cart View.

|--------------------------------------------------------------------

1. Create a CartController that will handle the request of user related to cart.

2. Create a jsp file in the views directory with the name of cart.jsp.

3. Visit bootsnipp.com. It consist of various bootstrap html template code for displaying the cart table you can choose your own. I have choosen one from the below link. Copy the html and css content from it.

https://bootsnipp.com/snippets/featured/responsive-shopping-cart

4. Update the icons to use glyphicon.

5. Modify the navbar.jsp and page.jsp to display the cart view.

6. Display a cart is empty message when there are no cart lines.

TO BE CONTINUED - Part 08 - 03 - Creating a Service Layer for Cart

|-----------------------------------------------------------------

| Part 08 - Cart Module for User

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 03 - Creating a Service Layer for Cart.

|--------------------------------------------------------------------

1. Create a CartService class in the online shopping project within a service package that will handle the cart logic for us and using the CartLineDAO it will interact with the database.

2. We will create a method that will fetch the cart of the user from the session.

3. Another method to fetch the cart lines using the cart line dao.

4. Update the controller to use the newly created CartService.

5. Update the cart.jsp table element to display the cartLine content.

TO BE CONTINUED - Part 08 - 04 - Updating CartLine Product Count

|-----------------------------------------------------------------

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 08 - Cart Module for User

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 04 - Updating CartLine Product Count jQuery.

|--------------------------------------------------------------------

1. Update the product count input field by adding id and not allowing user to increase count to more than 3 of similar product.

2. Update the refresh button to handle the update of product count.

3. Handle the button click event through jQuery.

/cart/{cartLineId}/update?count=1|2|3

TO BE CONTINUED - Part 08 - 05 - CartLine Update through CartService.

|-----------------------------------------------------------------

|-----------------------------------------------------------------

| Part 08 - Cart Module for User

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 05 - CartLine Update through CartService.

|--------------------------------------------------------------------

1. Handle the request submitted by the user in the CartController through the url which has been used in the jQuery i.e.

/cart/{cartLineId}/update?count=value

2. Add the method in the cart service class to handle the update of product count.

3. Update the cart controller show method to handle the response by cart service.

4. Update the cart.jsp page to display the message.

TO BE CONTINUED - Part 08 - 06 - Removing the CartLine

|-----------------------------------------------------------------

| Part 08 - Cart Module for User

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 06 - Removing the CartLine

|--------------------------------------------------------------------

1. Update the button to an anchor tag in cart.jsp that will handle the deletion of cartline.

2. Based on the url write a method inside the CartController to handle it.

3. Write a method inside the CartService class to handle deletion of cart line.

4. Handle the response given back by the cart line in the showCart method.

TO BE CONTINUED - Part 08 - 07 - Adding new item to the CartLine

|-----------------------------------------------------------------

| Part 08 - Cart Module for User

|-----------------------------------------------------------------

|--------------------------------------------------------------------

| 07 - Adding New CartLine Case One

|--------------------------------------------------------------------

When the product is not there inside cart.

1. Add a request mapping inside the CartController with the following url that we have already used in our project

/add/{productId}/product

2. Create a method inside the CartService class to add that product inside the CartLine.

3. Handle the response in the showCart method of CartController.

4. Update the url to continue with the shopping