**CHAPTER 2**

* 2.1.1 ESTABLISHING THE DOMAIN
  + An application’s domain is the subject area (ideas and concepts) that influence the understanding of the application
  + In taco cloud application, the domain includes taco designs, the ingredients that the designs use, customers and taco orders
  + We use Lombok to automatically generate the useful methods like setter,getter, equals(), hashcode(), toString() etc method at runtime
    - @Data annotation is provided by Lombok
    - Provides constructor that accepts all final properties as arguments
    - Add Lombok as dependency on pom.xml
      * Right click on pom.xml
      * Select spring🡪edit starters
        + Add Lombok
    - You also need to add Lombok as extension for IDE.
    - Ingredient class

package tacos;

import lombok.Data;

import lombok.RequiredArgsConstructor;

//generates getters, hashcode(), toString(),constructors, setters, equals()

@Data

//generates constructor with required arguments that has all final properties as arguments

*@RequiredArgsConstructor*

public class Ingredient {

private final String id;

private final String name;

private final *Type* type;

public static enum *Type* {

***WRAP***, ***PROTEIN***, ***VEGGIES***, ***CHEESE***, ***SAUCE***

}

}

* 2.1.2 CREATING A CONTROLLER CLASS
  + For Taco Cloud application, the controller must do following things
    - Handle HTTP GET requests where the request path is /design
    - Build a list of ingredients
    - Hand the request and the ingredient data off to a view template to be rendered as HTML and sent to the requesting web browser.

package tacos.web;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

import javax.validation.Valid;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.validation.Errors;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import lombok.extern.slf4j.Slf4j;

import tacos.Taco;

import tacos.Ingredient;

import tacos.Ingredient.Type;

/\*Lombok provided annotation that creates SLF4J(Simple logging facade for java) logger in the class at the runtime\*/

@Slf4j

/\* Identify this class as a controller and mark it for component scanning \*/

@Controller

/\*

\* Specifies that this class will handle requests whose path begin with /design

\*/

@RequestMapping("/design")

public class DesignTacoController {

/\*

\* specifies that when an HTTP GET request is received for design, this method will handle the

\* request

\*/

@GetMapping

public String showDesignForm(Model model) {

//creating new objects of Ingredient.java class and storing the objects in a list

List<Ingredient> ingredients = Arrays.asList(

new Ingredient("FLTO", "Flour Tortilla", Type.WRAP),

new Ingredient("COTO", "Corn Tortilla", Type.WRAP),

new Ingredient("GRBF", "Ground Beef", Type.PROTEIN),

new Ingredient("CARN", "Carnitas", Type.PROTEIN),

new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES),

new Ingredient("LETC", "Lettuce", Type.VEGGIES),

new Ingredient("CHED", "Cheddar", Type.CHEESE),

new Ingredient("JACK", "Monterrey Jack", Type.CHEESE),

new Ingredient("SLSA", "Salsa", Type.SAUCE),

new Ingredient("SRCR", "Sour Cream", Type.SAUCE));

//Getting all the values from the Type enum in the ingredient class and storing them in an array.

Type[] types = Ingredient.Type.values();

//for each elements in the types array

for (Type type : types) {

//add model attribute(wrap, new ingredient object with type wrap) etc which can directly be accessed by thymeleaf

model.addAttribute(type.toString().toLowerCase(),

filterByType(ingredients, type));

}

model.addAttribute("design", new Taco());

return "design";

}

private List<Ingredient> filterByType(List<Ingredient> ingredients, Type type) {

//returns all the objects from ingredients list that has type //equal to the type in the argument

return ingredients

.stream()

//ingredients.getType = type where getTYpe is automatically managed by Lombok annotation

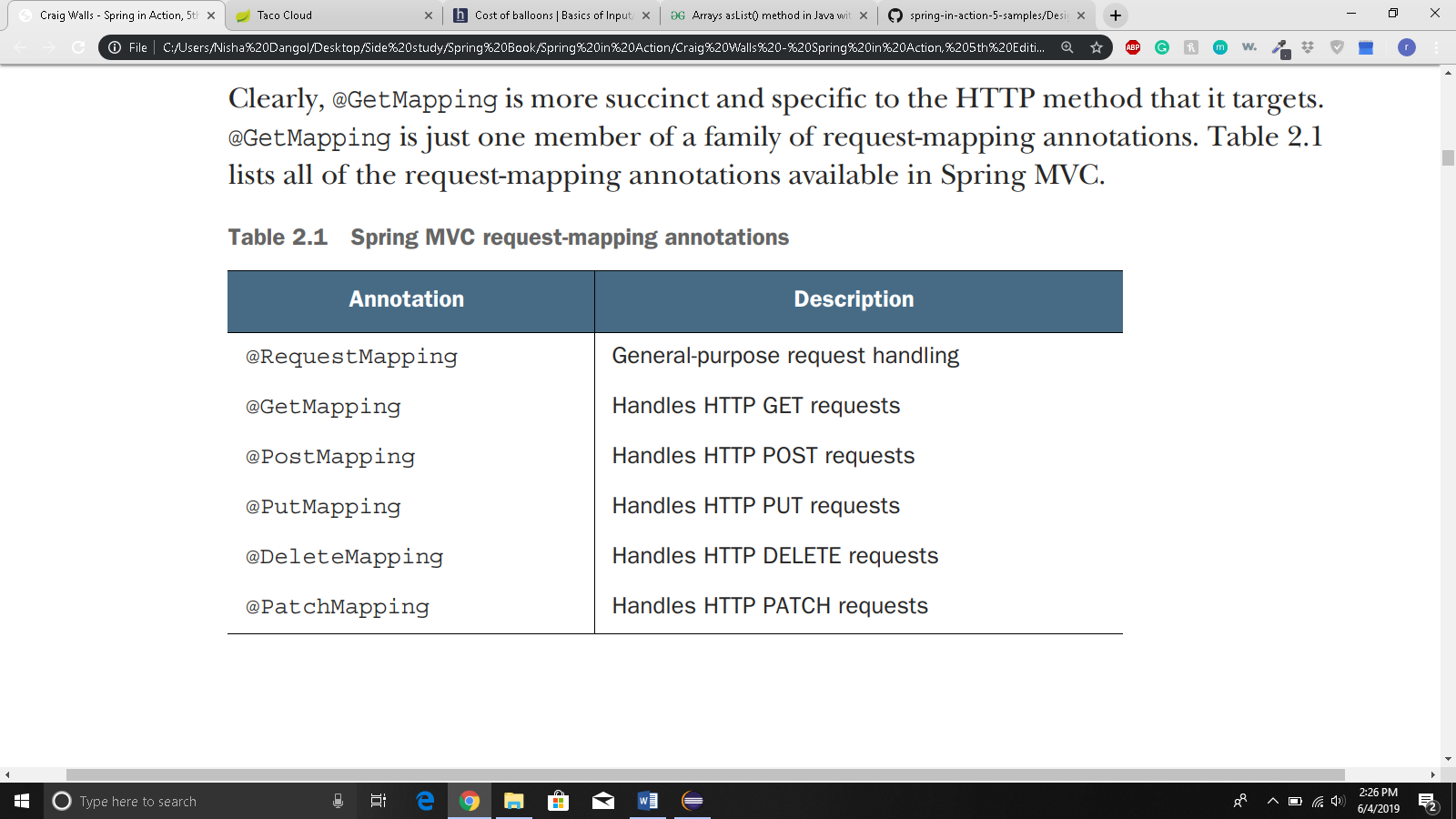
.filter(x -> x.getType().equals(type))

//converts stream into a list

.collect(Collectors.toList());

}

}}



* 2.1.3 DESIGNING THE VIEW
  + Thymeleaf:
    - Thymeleaf are designed to be decoupled from any particular web framework
    - Unable to work with data that controller places to the model
    - Therefore, before handling the request to the view, spring copies the model data into request attribute that thymeleaf and other view templating options have ready access to.
    - Lets study a thymeleaf code

<h3>Designate your wrap</h3>

<div th:each = “ingredient:${wrap}”>

<input name = “ingredients” type = “checkbox” th:value = “${ingredient.id}”/>

<span th:text = “${ingredient.name}”>INGREDIENT</span>

<br/>

</div>

* + - * th:each repeats rendering of the <div> once for each item in the collection found in the wrap request attribute. On each iteration, the ingredient item is bound to a thymeleaf variable named ingredient.
      * the checkbox input uses thymeleaf’s th:value to set rendered <input> element’s value attribute to the value found in the ingredient’s id property
      * <span> element uses th:text to replace “INGREDIENT” placeholder text with the value of the ingredient’s name property.
    - The above code if written in simple html will look like this for a single iteration:

<div>

<input name = “ingredients” type = “checkbox” value = “FLTO”/>

<span> Flour Tortilla</span><br/>

</div>

* + - Here is the list of complete code:
* <!DOCTYPE html>
* <html xmlns="http://www.w3.org/1999/xhtml"
* xmlns:th="http://www.thymeleaf.org">
* <head>
* <title>Taco Cloud</title>
* <!-- Getting the css file -->
* <!-- @ is used to access location of thymeleaf -->
* <link rel = "stylesheet" th:href = "@{/styles.css}"/>
* </head>
* <body>
* <h1>Design your taco!</h1>
* <!-- accessing image for a thymeleaf view -->
* <img th:src = "@{/images/TacoCloud.png}"/>
* <form method = "POST" th:object = "${design}">
* <div class = "grid">
* <div class = "ingredient=group" id = "wraps">
* <h3>Designate your wrap:</h3>
* <!-- ${wrap} is a request attribute which can be directly accessed by thymeleaf-->
* <!-- $ sign is used to access an attribute in thymeleaf -->
* <!-- for each wrap request attribute (in this case 2) -->
* <div th:each = "ingredient: ${wrap}">
* <!-- setting the value of the input to the id of the ingredient in current loop -->
* <input name = "ingredients" type = "checkbox" th:value = "${ingredient.id}"/>
* <!-- Writing text in html with current ingredient's name by replace INGREDIENT -->
* <span th:text = "${ingredient.name}">INGREDIENT</span><br/>
* </div>
* </div>
* <div class = "ingredient-group" id = "proteins">
* <h3>Pick your protein:</h3>
* <div th:each = "ingredient: ${protein}">
* <input name = "ingredients" type = "checkbox" th:value = "${ingredient.id}"/>
* <span th:text = "${ingredient.name}">INGREDIENT</span><br/>
* </div>
* </div>
* <div class = "ingredient-group" id = "cheeses">
* <h3>Choose your cheese:</h3>
* <div th:each = "ingredient: ${cheese}">
* <input name = "ingredients" type = "checkbox" th:value = "${ingredient.id}"/>
* <span th:text = "${ingredient.name}">INGREDIENT</span><br/>
* </div>
* </div>
* <div class = "ingredient-group" id = "veggies">
* <h3>Pick your veggies:</h3>
* <div th:each = "ingredient: ${veggies}">
* <input name = "ingredients" type = "checkbox" th:value = "${ingredient.id}"/>
* <span th:text = "${ingredient.name}">INGREDIENT</span><br/>
* </div>
* </div>
* <div class = "ingredient-group" id = "sauces">
* <h3>Choose your sauce</h3>
* <div th:each = "ingredient: ${sauce}">
* <input name = "ingredients" type = "checkbox" th:value = "${ingredient.id}"/>
* <span th:text = "${ingredient.name}">INGREDIENT</span>
* </div>
* </div>
* </div>
* <div>
* <h3>Name your taco creation</h3>
* <input type = "text" th:field = "\*{name}"/>
* <br/>
* <button>Submit your taco</button>
* </div>
* </form>
* </body>
* </html>
* 2.2 PROCESSING FOR SUBMISSION
  + You use post method
  + @PostMapping annotation will be used
  + When the form is submitted, the fields in the form are bound to properties of a taco object(whose class is as below)

package tacos;

import java.util.List;

import lombok.Data;

//automatically generates essential JavaBean methods at runtime

@Data

public class Taco {

private String name;

private List<String> ingredients;

* + In the full html file above, there is input field = “\*{name}” which is associated with name field in Taco class
  + Similarly, there is input name = “ingredients” which is a checkbox that can be selected more than one or zero. It is associated with ingredients in the Taco class so, we have List in the Taco class.
  + Here is the processDesign method:

//Handles post request

*@PostMapping*

public String processDesign(*@Valid* *@ModelAttribute*("design") Taco design, Errors errors, Model model) {

if(errors.hasErrors()) {

return "design";

}

//Save the taco design

//We'll do this is chapter 3

***log***.info("Processing design: " + design);

//redirects the user to the relative path /order/current

return "redirect:/order/current";

}

* + It returns a string “redirect:/order/current”
    - After processDesign() method executes, the user’s browser should be redirected to relative path /order/current.
    - This is because once the user selects the taco, he must order it so we need to go to the order page.
    - When the browser redirects to /current, orderform() method from OrderController.java class is executed
  + Let’s create OrderController class:

package tacos.web;

import javax.validation.Valid;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.validation.Errors;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import lombok.extern.slf4j.Slf4j;

import tacos.Order;

/\*creates SLF4J logger object at runtime\*/

@Slf4j

*@Controller*

/\* specifies that this class handles requests whose path begins with /orders \*/

*@RequestMapping*("/orders")

public class OrderController {

/\*

\* specifies that orderForm() method will handle get requests for

\* /orders/current

\*/

*@GetMapping*("/current")

public String orderForm(Model model) {

//adding "order" attribute to model which is a new Order() object

model.addAttribute("order", new Order());

/\* orderForm is a viewName \*/

return "orderForm";

}

/\*

\* specifies that processOrder() method will handle post requests for

\* /orders which is done in orderForm.html, th:action = "@{/orders}"

\*/

*@PostMapping*

public String processOrder(Order order) {

***log***.info("Order Submitted: " + order);

return "redirect:/";

* + We also have order.java class:

package tacos;

import javax.validation.constraints.Digits;

import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;

import org.hibernate.validator.constraints.~~NotBlank~~;

import lombok.Data;

//Automatically defines the basic getter, setter, equal, hash and tostring() method

@Data

public class Order {

private String name;

private String street;

private String city;

private String state;

private String zip;

private String ccNumber;

private String ccExpiration;

private String ccCVV;

* + We also need view for our order. It’s name is orderform.html
* <!DOCTYPE html>
* <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
* <head>
* <title>Taco Cloud</title>
* <link rel = "stylesheet" th:href = "@{/styles.css}"/>
* </head>
* <body>
* <!-- th:action to /orders request mapping from OrderController.java -->
* <!-- ${order} is a model attribute containg new order() object that was set in orderForm() method in OrderController.java-->
* <form method = "POST" th:action = "@{/orders}" th:object = "${order}">
* <h1>Order your taco creations!</h1>
* <!-- getting the image tacoCloud.png -->
* <img th:src = "@{/images/TacoCloud.png}"/>
* <!-- link to design url -->
* <a th:href = "@{/design}" id = "another">Design another taco</a></br>
* <div th:if = "${#fields.hasErrors()}">
* <span class = "validationError">
* Please correct the problems below and resubmit.
* </span>
* </div>
* <!-- since we have ${order} object, we are linking our input to the order objects properties -->
* <h3>Deliver my taco masterpieces to...</h3>
* <label for = "name">Name: </label>
* <input type = "text" th:field = "\*{name}"/>
* <br/>
* <label for = "street">Street address: </label>
* <input type = "text" th:field = "\*{street}"/>
* <br/>
* <label for = "city">City: </label>
* <input type = "text" th:field = "\*{city}"/>
* <br/>
* <label for = "state">State: </label>
* <input type = "text" th:field = "\*{state}"/>
* <br/>
* <label for = "zip">Zip code: </label>
* <input type = "text" th:field = "\*{zip}"/>
* <br/>
* <h3>Here's how I'll pay...</h3>
* <label for = "ccNumber">Credit Card #: </label>
* <input type = "text" th:field = "\*{ccNumber}"/>
* <br/>
* <label for = "ccExpiration">Expiration: </label>
* <input type = "text" th:field = "\*{ccExpiration}"/>
* <br/>
* <label for = "ccCVV">CVV: </label>
* <input type = "text" th:field = "\*{ccCVV}"/>
* <br/>
* <input type = "submit" value = "Submit order"/>
* </form>
* </body>
  + - </html>
  + 2.3VALIDATING FORM INPUT
    - Spring supports Java’s Bean Validation API(JSR-303)
    - To apply validation in spring MVC, you need to
      * Declare validation rules on the class that is to be validated: specifically the Taco class
      * Specify that the validation should be performed in the controller methods that require validation: specifically the DesignController’s processDesign() method and OrderController’s processOrder() method.
      * Modify the form views to display validation errors.
    - 2.3.1 DECLARING VALIDATION RULES
      * For taco class, name should not be empty or null and the selected ingredients must atleast have one item.

package tacos;

import java.util.List;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

import lombok.Data;

//automatically generates essential JavaBean methods at runtime

@Data

public class Taco {

//javax validation annotation to make sure that the name field is not null

*@NotNull*

//javax validation annotation to make sure that the name field must have at least 5 characters

*@Size*(min = 5, message = "Name must be at least 5 characters long")

private String name;

//javax validation annotation to make sure that the user choose at least one ingredient for the taco

*@Size*(min = 1, message = "You must choose at least 1 ingredient")

private List<String> ingredients;

}

* + - * To validate the submitted taco orders, you must apply annotation to order class

package tacos;

import javax.validation.constraints.Digits;

import javax.validation.constraints.Pattern;

import javax.validation.constraints.NotBlank;

import org.hibernate.validator.constraints.CreditCardNumber;

import lombok.Data;

//Automatically defines the basic getter, setter, equal, hash and tostring() method

@Data

public class Order {

//javax validation to make sure that name field is not blank

*@NotBlank*(message = "Name is required")

private String name;

*@NotBlank*(message = "Street is required")

private String street;

*@NotBlank*(message = "City is required")

private String city;

*@NotBlank*(message = "State is required")

private String state;

*@NotBlank*(message = "zip code is required")

private String zip;

//passes Luhn's algorithm check to validate the entered credit card number

*@CreditCardNumber*(message = "Not a valid credit card number")

private String ccNumber;

//passes regex to make sure that user inputs the value in mm/yy format

*@Pattern*(regexp = "^(0[1-9]|1[0-2])([\\/])([1-9][0-9])$", message = "Must be formatted MM/YY")

private String ccExpiration;

//ensures that the value contains exactly three numeric digits

*@Digits*(integer = 3, fraction = 0, message = "Invalid CVV")

private String ccCVV;

}

* + - 2.3.2 PERFORMING VALIDATION AT FOR BINDING
      * We need to revisit each of the controllers, to make sure that the validations are performed when the forms are posted to their respective handler methods
      * To validate a submitted Taco, you need to add Java Bean Validation API’s @Valid annotation to the Taco argument of DesignTacoController’s processDesign() method.

public String processDesign(*@Valid* Taco design, Errors errors, Model model) {

//if there is an error then just return the 'design' view

if(errors.hasErrors()) {

return "design";

}

//Save the taco design

//We'll do this is chapter 3

***log***.info("Processing design: " + design);

//redirects the user to the relative path /orders/current i.e in the browser http:8080/orders/current

return "redirect:/orders/current";

* + - * + @Valid annotation tells Spring MVC to perform validation on the submitted Taco object after it’s bound to the submitted form data and before the processDesign() method is called.
        + If there are any validation errors, the details of the errors will be captured in the Errors object.
        + The first few lines consults the Errors object asking its hasErrors() method if there are any validation errors.
        + If there are any errors, the method concludes without processing the taco and return “design” view name so that the form is re-displayed
      * To validate the submitted order, some changes must be done in the processOrder() method of OrderController:

/\*

\* specifies that processOrder() method will handle post requests for

\* /orders which is done in orderForm.html, th:action = "@{/orders}"

\*/

*@PostMapping*

//@Valid will check if there are any validation errors

public String processOrder(*@Valid* Order order, Errors errors) {

***log***.info("Order Submitted: " + order);

return "redirect:/";

}

* + - * + If there are validation errors, the request will be forwarded to the form view to give the user a chance to correct their mistakes.
      * Not we need to inform users what to correct if there are any mistakes
    - 2.3.3 DISPLAYING VALIDATION ERRORS
      * In thymeleaf we can access Errors object via the fields property and with th:errors attribute.

<label for = “ccNumber”>Credit Card #: </label>

<input type = “text” th:field = “\*{ccNumber}”/>  
<span class = “validationError”

th:if = “${#fields.hasErrors(‘ccNumber’)}”

th:errors = “\*{ccNumber}”>CCNumError</span>

* + - * + The <span> element uses th:if attribute to decide whether or not to display the <span>
        + The fields property’s hasErrors() method checks if there are any errors in the ccNumber Field. If so, the <span> will be rendered
        + The th:errors attribute references the ccNumber field and, assuming that there are errors for that field, it will replace the placeholder content of the <span> element with the validation message
      * This the complete Design view

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml"

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

<head>

<title>Taco Cloud</title>

<!-- Getting the css file -->

<!-- @ is used to access location of thymeleaf -->

<link rel = "stylesheet" th:href = "@{/styles.css}"/>

</head>

<body>

<h1>Design your taco!</h1>

<!-- accessing image for a thymeleaf view -->

<img th:src = "@{/images/TacoCloud.png}"/>

<!-- no th:action so redirect to same url as get request i.e /design -->

<!-- ${design} is a model attribute with value of new Taco() -->

<!-- new Taco() has name and list of ingredients -->

<form method = "POST" th:object = "${design}">

<!-- If there are any validation error in the ingredients field from Taco class, then display the error message that was written in the Taco class -->

<span class = "validationError"

th:if = "${#fields.hasErrors('ingredients')}"

th:errors = "\*{ingredients}">Ingredient Error</span>

<div class = "grid">

<div class = "ingredient=group" id = "wraps">

<h3>Designate your wrap:</h3>

<!-- ${wrap} is a request attribute copied automatically from model attribute which can be directly accessed by thymeleaf-->

<!-- $ sign is used to access an attribute in thymeleaf -->

<!-- for each wrap request attribute (in this case 2) -->

<div th:each = "ingredient: ${wrap}">

<!-- setting the value of the input to the id of the ingredient in current loop -->

<input name = "ingredients" type = "checkbox" th:value = "${ingredient.id}"/>

<!-- Writing text in html with current ingredient's name by replace INGREDIENT -->

<span th:text = "${ingredient.name}">INGREDIENT</span><br/>

</div>

</div>

<div class = "ingredient-group" id = "proteins">

<h3>Pick your protein:</h3>

<div th:each = "ingredient: ${protein}">

<input name = "ingredients" type = "checkbox" th:value = "${ingredient.id}"/>

<span th:text = "${ingredient.name}">INGREDIENT</span><br/>

</div>

</div>

<div class = "ingredient-group" id = "cheeses">

<h3>Choose your cheese:</h3>

<div th:each = "ingredient: ${cheese}">

<input name = "ingredients" type = "checkbox" th:value = "${ingredient.id}"/>

<span th:text = "${ingredient.name}">INGREDIENT</span><br/>

</div>

</div>

<div class = "ingredient-group" id = "veggies">

<h3>Pick your veggies:</h3>

<div th:each = "ingredient: ${veggies}">

<input name = "ingredients" type = "checkbox" th:value = "${ingredient.id}"/>

<span th:text = "${ingredient.name}">INGREDIENT</span><br/>

</div>

</div>

<div class = "ingredient-group" id = "sauces">

<h3>Choose your sauce</h3>

<div th:each = "ingredient: ${sauce}">

<input name = "ingredients" type = "checkbox" th:value = "${ingredient.id}"/>

<span th:text = "${ingredient.name}">INGREDIENT</span>

</div>

</div>

</div>

<div>

<h3>Name your taco creation</h3>

<input type = "text" th:field = "\*{name}"/>

<span class = "validationError"

th:if = "${#fields.hasErrors('name')}"

th:errors = "\*{name}">Name Error</span>

<br/>

<button>Submit your taco</button>

</div>

</form>

</body>

</html>

* + - * This is the complete orderForm view:

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">

<head>

<title>Taco Cloud</title>

<link rel = "stylesheet" th:href = "@{/styles.css}"/>

</head>

<body>

<!-- th:action to /orders request mappting from OrderController.java -->

<!-- ${order} is a model attribute containg new order() object that was set in orderForm() method in OrderController.java-->

<form method = "POST" th:action = "@{/orders}" th:object = "${order}">

<h1>Order your taco creations!</h1>

<!-- getting the image tacoCloud.png -->

<img th:src = "@{/images/TacoCloud.png}"/>

<!-- link to design url -->

<a th:href = "@{/design}" id = "another">Design another taco</a></br>

<!-- #fields attribute accesses the Errors object. After that we check if it has any errors -->

<div th:if = "${#fields.hasErrors()}">

<span class = "validationError">

Please correct the problems below and resubmit.

</span>

</div>

<!-- since we have ${order} object, we are linking our input to the order objects properties -->

<!-- th:if checks if the name field has any error. -->

<!-- th: errors replaces Name Error with validation message if there are any errors -->

<h3>Deliver my taco masterpieces to...</h3>

<label for = "name">Name: </label>

<input type = "text" th:field = "\*{name}"/>

<span class = "validationError"

th:if = "${#fields.hasErrors('name')}"

th:errors = "\*{name}">Name Error</span>

<br/>

<label for = "street">Street address: </label>

<input type = "text" th:field = "\*{street}"/>

<span class = "validationError"

th:if = "${#fields.hasErrors('street')}"

th:errors = "\*{street}">Street Error</span>

<br/>

<label for = "city">City: </label>

<input type = "text" th:field = "\*{city}"/>

<span class = "validationError"

th:if = "${#fields.hasErrors('city')}"

th:errors = "\*{city}">City Error</span>

<br/>

<label for = "state">State: </label>

<input type = "text" th:field = "\*{state}"/>

<span class = "validationError"

th:if = "${#fields.hasErrors('state')}"

th:errors = "\*{state}">State Error</span>

<br/>

<label for = "zip">Zip code: </label>

<input type = "text" th:field = "\*{zip}"/>

<span class = "validationError"

th:if = "${#fields.hasErrors('zip')}"

th:errors = "\*{zip}">Zip Error</span>

<br/>

<h3>Here's how I'll pay...</h3>

<label for = "ccNumber">Credit Card #: </label>

<input type = "text" th:field = "\*{ccNumber}"/>

<span class = "validationError"

th:if = "${#fields.hasErrors('ccNumber')}"

th:errors = "\*{ccNumber}">CC Number Error</span>

<br/>

<label for = "ccExpiration">Expiration: </label>

<input type = "text" th:field = "\*{ccExpiration}"/>

<span class = "validationError"

th:if = "${#fields.hasErrors('ccExpiration')}"

th:errors = "\*{ccExpiration}">CC Expiration Error</span>

<br/>

<label for = "ccCVV">CVV: </label>

<input type = "text" th:field = "\*{ccCVV}"/>

<span class = "validationError"

th:if = "${'ccCVV'}"

th:errors = "\*{ccCVV}">CC CVV Error</span>

<br/>

<input type = "submit" value = "Submit order"/>

</form>

</body>

</html>

* + 2.4 WORKING WITH VIEW CONTROLLERS
    - When a controller is simple enough that it doesn’t populate a model or process input, you can declare a view controller that does nothing but forward the request to a view.
    - Here is the view controller class

package tacos.web;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

*@Configuration*

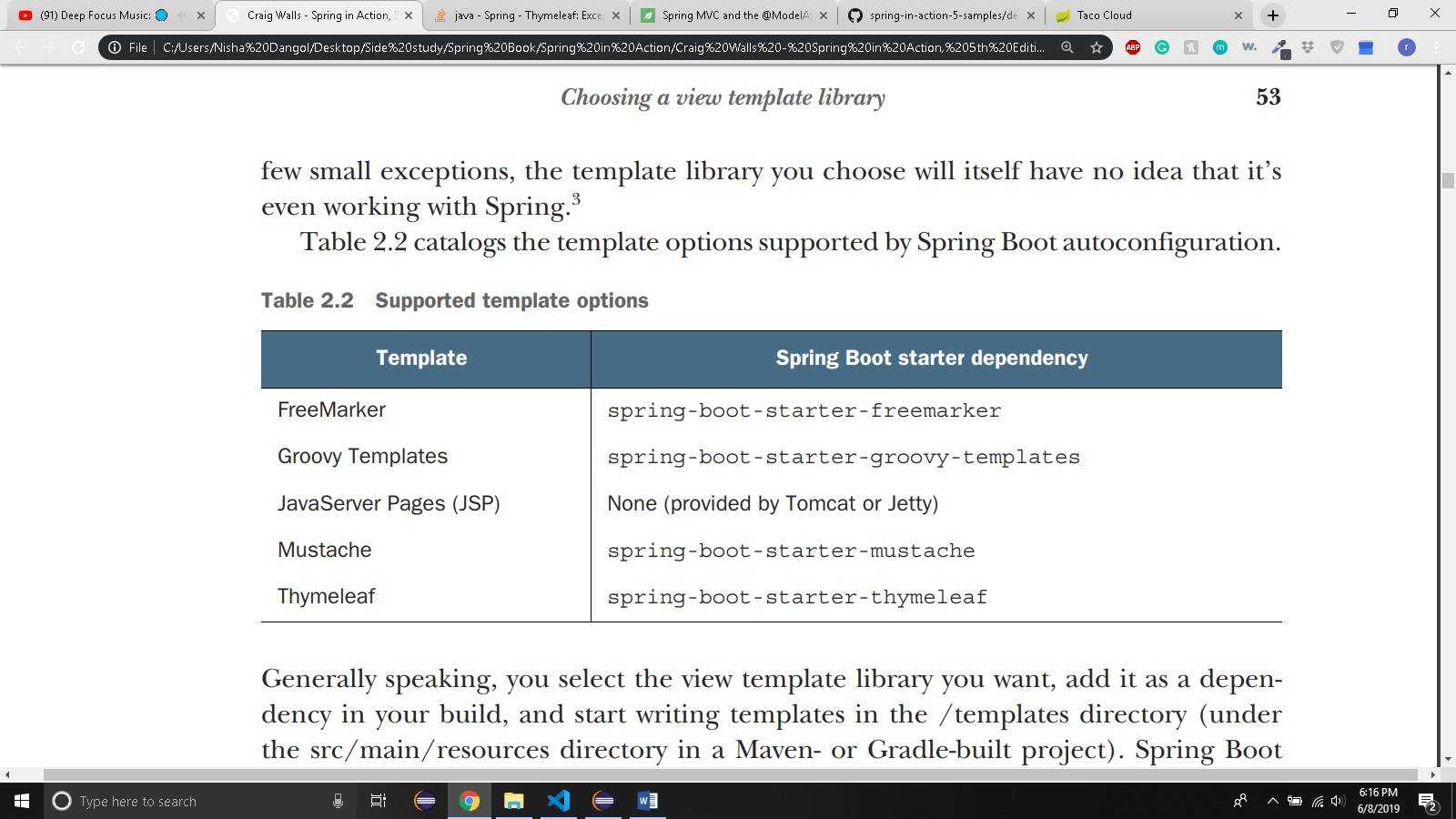
public class WebConfig implements WebMvcConfigurer {

*@Override*

public void addViewControllers(ViewControllerRegistry registry) {

registry.addViewController("/").setViewName("home");

}

* + - * WebMvcConfigurer defines several methods for configuring spring mvc.
      * Even thought it’s interface, it provides default implementations of all the methods, so you only need to override the methods you need. In this case you override addViewControllers
      * The addViewControllers() has ViewControllerRegistry as arguments that you can use to register one or more view controllers.
        + Here “/” is the path for which your view controller will handle GET requests
        + That method returns the viewControllerRegistry object on which you immediately call setViewName() to specify home as the view that a request for “/” should be forwarded to.
  + 2.5 CHOOSING A VIEW TEMPLATE LIBRARY
    - The template library you choose will have no idea that it’s working with spring
    - Here are the few template options supported by spring boot:
    - 
    - Generally, you select the view template library you want, add it as a dependency in your build, and start writing templates in /templates directory
    - Spring boot will detect your chosen template library and automatically configure the components required for it to serve views for your spring MVC controllers.
    - If we want to use Mustache instead of thymeleaf we can do one of the following:
      * Select Mustache during project initialization or,
      * In pom.xml remove thymeleaf starter and add:

<dependency>

<groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-mustache</artifactId>

</dependency>

* + - * You will have to write mustache specific syntax
    - You can also use JSP but there is a problem with it:
      * Java servlet containers look for JSPs somewhere under /WEB-INF. But if you’re building your application as an executable JAR, there is no way to satisfy this requirement.
      * So, use JSP only if you are building WAR file and deploying in a traditional servlet container
    - 2.5.1 CACHING TEMPLATES
      * By default, templates are only parsed once, when they’re first used and the results of that parse are cached for subsequent use. This reduces redundant template parsing on each request and improves performance.
      * But during development phase, you will make a lot of changes. The only way to remove that cache is restarting the application.
      * Fortunately, there is a way to disable caching. Set template appropriate caching property to false
        + In thymeleaf: **spring.thymeleaf.cache = false**
      * But when you deploy the application, remove the above line or set it to true.
      * However, devtools will disable caching during development and reenable the caching during deployment.