Validating form data

Contents

Introduction

The current form will allow any data to be submitted. This means that invalid data will be processed and either our data will lose data quality, or the processing will fail. For example, since Spring Boot will try to convert the POST parameters to the type in the object, the application will fail when converting “1.23” to Long (and we may want to allow the user to enter “1.23” even if we convert to pence in our objects).

The main answer to this is validation, though we have to combine that with other forms of error handling to get a complete solution.

Add validation to our project

The validation library isn't included by default with the "web" library. We need to add validation to our project – and we do this by adding the required dependency to our Gradle build.

> One way of finding the right dependency, is to create a fresh project using the Spring Initializr and include the dependency that you require. Then, copy and paste the lines from the `build.gradle` file. Be careful to create the project with the same version as your current project!
>
> WARNING: Do not trust GitHub Co-Pilot here. It is known to add version-specific dependencies and add conflicting dependencies. This can introduce hard-to-debug issues.

Add the dependency to `build.gradle`.

```
dependencies {
//…add the line below anywhere in the 'dependencies' block.

    implementation 'org.springframework.boot:spring-boot-starter-validation'

//…
}
```

You will have to refresh the Gradle project for this to be loaded otherwise you may still get error highlighting in the IDE.

Convert the form to Thymeleaf

Copy the existing `menuItemForm.html` file to `menuItemFormTh.html`. We will update this new file.

Let's Thymeleaf-ise (not a real word) this file.

```
<form th:action="@{/menuItem}" th:method="post" th:object="${menuItem}">

    <input th:field="*{id}" type="hidden" th:value="0"/>


    <div class="mb-3">
        <label class="form-label" th:for="*{name}">Name:</label>
        <input class="form-control" th:field="*{name}" placeholder="Menu Item
Name" type="text"/>
    </div>

    <div class="mb-3">
```

```
        <label class="form-label" th:for="*{description}">Description:</label>
        <input class="form-control" th:field="*{description}"
placeholder="Description" type="text"/>
    </div>

    <div class="mb-3">
        <label class="form-label" th:for="*{priceInPence}">Price:</label>
        <input class="form-control" th:field="*{priceInPence}"
placeholder="Price" type="text"/>
    </div>

    <div class="mb-3">
        <label class="form-label" th:for="*{allergens}">Allergens:</label>
        <input class="form-control" th:field="*{allergens}" placeholder="Any
allergens" type="text"/>
    </div>

    <div class="mb-3">
        <label class="form-check-label"
th:for="*{isVegetarian}">Vegetarian?</label>
        <input class="form-check-input" th:field="*{isVegetarian}"
type="checkbox"/>
    </div>

    <div class="mb-3">
        <label class="form-check-label" th:for="*{isVegan}">Vegan?</label>
        <input class="form-check-input" th:field="*{isVegan}" type="checkbox"/>
    </div>


    <div class="mb-3">

        <button class="btn btn-primary">Submit</button>
    </div>

    <div>
        <div class="mb-3">
            <a class="btn btn-primary" href="/">Cancel</a>
        </div>
    </div>

</form>
```

Let’s walk through the elements containing new Thymeleaf attributes.

```
<form th:action="@{/menuItem}" th:method="post" th:object="${menuItem}">
```

In this element, we have 3 attributes;

th:action – this expands to an action attribute. Since it is now a Thymeleaf attribute, the value can be dynamic. This could be useful for when forms are updating the nth item in a collection.

th:method – this expands to a method attribute. It is unlikely that this will be a dynamic, but the possibility is now there.

th:object – this doesn't expand to a normal HTML attribute. Rather it acts as a scoping attribute. Imagine that the form is now bound to an object, and that when the form is submitted, the form elements are bound to the fields of that object. This makes it easy to have a form that updates an existing object as the values of the elements will be set from the object. In this case (where we are adding a new object), it does mean that we need to provide the form with an empty object (more on this later).

```
<label class="form-label" th:for="*{name}">Name:</label>
<input class="form-control" th:field="*{name}" placeholder="Menu Item Name"
type="text"/>
```

In these two elements, we have 2 new Thymeleaf attributes. Both of these use the *{} syntax. This is a shortcut to address the field of the scoped object (remember the th:object attribute?). It is saying that this element is bound to this field of the bound object.

th:for – this expands to a for attribute. It is unlikely that this will be dynamic, but it can be.

th:field – this expands to id, name and value attributes, based on the name and value of the field in the bound object. (For checkbox elements, it will control the checked attribute based on the value)

Route the request to the new form

To use this new Thymeleaf form, we need to route the request to it. To do this, we'll write a new method and we'll comment out the @GetMapping annotation on the existing method. Notice how the new method adds an empty MenuItem object to the ModelAndView object and how it uses the new copy of the template.

```
//@GetMapping("/menu/add")
public ModelAndView addMenuItem() {
    ModelAndView modelAndView = new ModelAndView("menu/menuItemForm");
    return modelAndView;
}

@GetMapping("/menu/add")
public ModelAndView addMenuItemTh() {
    ModelAndView modelAndView = new ModelAndView("menu/menuItemFormTh");
    modelAndView.addObject("menuItem", new MenuItem());
    return modelAndView;
}
```

Are your Java spidey-senses twitching? They should be. We don't have a default constructor on the MenuItem class, so let's add that.

```
public MenuItem() {
    this(0L, "", 0L, "none", "", false, false);
}
```

This constructor uses the existing constructor and sets default values on all fields.

Test the new code

Test the new code. There should be little difference in what you see, but the foundations are now in place to add validation.

Annotate the field to validate

Let's assume that we never want the name of the menu item to be empty.

To tell Spring Boot that this is what we want, add the `@NotEmpty` annotation to the `name` field in the `MenuItem` class.

```
@NotEmpty private String name;

//the imported package is jakarta.validation.constraints.NotEmpty.
```

Test

If you test after this change, there will be no change. We haven't told Spring Boot to validate the data. To do that, we need to annotate the parameter on the controller method.

Tell Spring Boot to validate the form data

This is a simple task. Add the `@Valid` annotation to the form data parameter.

```
@PostMapping("/menuItem")
public ModelAndView addMenuItem(@Valid MenuItem menuItem) {
```

Test

Now try testing again. Submit a new menu item with a blank name. You should get an ugly error message on the screen. If you read the stack trace, you should be able to see that the validation has picked up a field that breaks a stated constraint (i.e. something that should be "not empty" actually is).

Obviously, this isn't the best user interface, so we need to add some code to handle the error more gracefully.

Take the user back to the form

Our first baby step is to change the controller so that the user is taken back to the form if there are any validation errors. To do this, we need to do two things; we need to get hold of the errors to see if any exist and we need to get hold of the submitted data so that we can populate the form when we go back to it. Fortunately, Spring Boot gives us objects with this data pre-populated.

The `BindingResult` class provides us with the list of errors and set of useful methods. The `Model` class provides us with the key-value pairs that were passed into the previous rendering of the form.

All we have to do is add parameters of these types onto the controller method (noting that the order of parameters is important since Java doesn't support named parameters).

```
public ModelAndView addMenuItem(@Valid MenuItem menuItem, BindingResult
bindingResult, Model model)
```

Spring Boot will provide these objects for us.

Check for errors

We can now use the `BindingResult` object to check for any validation errors. They are called "binding" results because the process of taking the HTML parameters and mapping to the Java object is known as binding and that is when validation is done.

The object provides a useful `hasErrors`() method which returns true or false.

```
if (bindingResult.hasErrors()) {
```

If there are errors, we would like to take the user back to the form, retaining any values that they've entered. To do this, we need to set the view back to the form, and provide the model data.

```
if (bindingResult.hasErrors()) {
    ModelAndView modelAndView = new ModelAndView("menu/menuItemFormTh",
model.asMap());
    return modelAndView;
}
```

Note, the useful `asMap`() method on the `Model` object. We can create a `ModelAndView` object directly.

If there are no errors, then we carry on and process the form. The full method looks like this:

```
@PostMapping("/menu/add")
public ModelAndView addMenuItem(@Valid MenuItem menuItem,
BindingResult bindingResult, Model model) {
    ModelAndView modelAndView;
    if (bindingResult.hasErrors()) {
        modelAndView = new ModelAndView("menu/menuItemFormTh",
model.asMap());
    } else {
        MenuService menuService = MenuService.getInstance();
        menuService.addMenuItem(menuItem);
```

```
          modelAndView = new ModelAndView("redirect:/menu");
     }
     return modelAndView;
}
```

Notice that the URL has been changed on the `@PostMapping` annotation. The matching change on the template is:

```
<form th:action="@{/menu/add}" th:method="post" th:object="${menuItem}">
```

This is done so that the URL stays the same after a validation error sends the user back. It might be better to use a URL of "/menu/newItem" as this would be more RESTful.

Test

At this point, test the solution and check that the errors are being caught and the user isn't being allowed to continue until a name is provided.

Provide Useful Error Messages

At this point, we're preventing bad data from reaching the list of menu items, but we're not telling the user what the issue actually is. This is relatively straightforward. Due to the integration that exists between Spring Boot and Thymeleaf, we only have to update the template. Spring Boot passes the errors to Thymeleaf automatically.

```
<div class="mb-3">
    <label class="form-label" th:for="*{name}">Name:</label>
    <input class="form-control" placeholder="Menu Item Name" th:field="*{name}"
type="text"/>
    <div class="alert alert-warning" th:errors="*{name}"
th:if="${#fields.hasErrors('name')}">Name Error</div>
</div>
```

The new `div` element contains two Thymeleaf attributes.

`th:errors` gives us access to the errors. The "`*{name}`" syntax will give access to the errors associated with the "name" field.

`th:if` is a normal "if" statement. `#fields` is an object provided by Thymeleaf and we can ask if the "name" field has errors. If it does, the "if" is true and the element will be shown. We can do this for all the fields.

Test

Test the solution and observe the error message that is played when the name field is left blank.

Improve the error message

At the moment, the standard annotation error message is played. We can customise that by adding an error message to the annotation.

```
@NotEmpty(message = "The name cannot be empty")
private String name;
```

Test

Test again and observe the message being shown.

Exercise

Think about this approach to error messages. Is there a better approach? What if we wanted to support the ability to change the messages without updating the code or wanted to support internationalisation?

Consider form object design

At the moment, we are using the `MenuItem` class for a number of purposes. It represents a menu item in our application and it is what we "store" in the collection of items. We also use it as the object that we bind the form to.

This presents a problem. The priceInPence field is a Long. This is a reasonable decision because we want to do financial calculations with integer values rather than floating point values to reduce the chance of rounding errors. However, this means that the user has to enter an integer value into the form which is not intuitive.

There are a range of solutions to this.

We could handle this on the client-side with JavaScript by having another form field that accepts floating point numbers and have it populate a hidden priceInPence form element that gets submitted. This would work as long as JavaScript is turned on (which it might not be for strict accessibility).

We could add a field to the `MenuItem` class and use that only for form binding. On submission, we would convert to integer and then use that integer field for processing. However, this would pollute the class with an additional field that might confuse future developers.

We could have a separate class that we use purely for form processing. This can have whatever field types that we want. The disadvantage is that we now have an "almost" duplicate class that will need to be kept in sync with changes to the core `MenuItem` class. This seems like a bad thing to do, but quite often the data that we input is a subset of the data we keep. We might want to retain a timestamp of change on the `MenuItem` objects, or a history of prices – so having a separate class that simply deals with new data isn't such a better idea.

We'll implement this third option.

Create new menu item form class

Create a new class in the `menu` package called `MenuItemForm`.

```
@Data
@AllArgsConstructor
public class MenuItemForm {
    private Long id;
    @NotEmpty(message = "The name cannot be empty")
    private String name;
    @Pattern(regexp = "^[0-9]+(\\.[0-9]{1,2})?$", message = "The price must be 
a number with up to 2 decimal places")
    private String price;
    private String allergens;
    private String description;
    private Boolean isVegetarian;
    private Boolean isVegan;

    public MenuItemForm() {
        this(0L, "", "0", "none", "", false, false);
    }
}
```

Note that in this class, the `price` field is a String. This means that the HTML parameters can always be bound because there is no conversion required. The `price` field has a regular expression on it. This regex checks that the string contains a number with up to 2 decimal places. Therefore, we can validate the format of the data and convert it in code later on.

Change the template

We now need to change the template to use the new `price` field rather than the `priceInPence` field.

```
<div class="mb-3">
    <label class="form-label" th:for="*{price}">Price:</label>
    <input class="form-control" placeholder="Price" th:field="*{price}" 
type="text"/>
</div>
```

Use the new class in the controller

Finally, we need to work with an instance of the new class and we need to map the data from the new class to the old class.

In the GET method

```
@GetMapping("/menu/add")
public ModelAndView addMenuItemTh() {
    ModelAndView modelAndView = new ModelAndView("menu/menuItemFormTh");
    MenuItemForm emptyMenuItem = new MenuItemForm();
    modelAndView.addObject("menuItem", emptyMenuItem);
    return modelAndView;
}
```

In the POST method

```
@PostMapping("/menu/add")
public ModelAndView addMenuItem(@Valid MenuItemForm menuItem, BindingResult
bindingResult, Model model) {

    if (bindingResult.hasErrors()) {
        ModelAndView modelAndView = new ModelAndView("menu/menuItemFormTh",
model.asMap());
        return modelAndView;
    } else {
        MenuService menuService = MenuService.getInstance();
        MenuItem newMenuItem = new MenuItem(menuItem.getId(),
menuItem.getName(), Double.valueOf(menuItem.getPrice()).longValue()*100,
menuItem.getAllergens(), menuItem.getDescription(), menuItem.getIsVegetarian(),
menuItem.getIsVegan());
        menuService.addMenuItem(newMenuItem);
        ModelAndView modelAndView = new ModelAndView("redirect:/menu");
        return modelAndView;
    }
}
```

In the POST method Correction

The above code contains an error. Since we have changed the type of the form to `MenuItemForm`, we need to tell Spring that the key to that object remains as “menuItem” otherwise it will assume it should be “menuItemForm” (deduced from the class name). The `@ModelAttribute` annotation tells Spring to change the key.

```
@PostMapping("/menu/add")
public ModelAndView addMenuItem(@Valid @ModelAttribute(“menuItem) MenuItemForm
menuItem, BindingResult bindingResult, Model model) {

    if (bindingResult.hasErrors()) {
        ModelAndView modelAndView = new ModelAndView("menu/menuItemFormTh",
model.asMap());
        return modelAndView;
    } else {
        MenuService menuService = MenuService.getInstance();
        MenuItem newMenuItem = new MenuItem(menuItem.getId(),
menuItem.getName(), (long) Double.valueOf(menuItem.getPrice())*100,
menuItem.getAllergens(), menuItem.getDescription(), menuItem.getIsVegetarian(),
menuItem.getIsVegan());
        menuService.addMenuItem(newMenuItem);
        ModelAndView modelAndView = new ModelAndView("redirect:/menu");
        return modelAndView;
    }
}
```

Test

Try out the new code. Try invalid prices and see the validation fire.

Exercise

- Add validations to other fields.
- What other validations might you need to do? Consider how they might be done. Might there be value in having a more specific class (rather than a String) sometimes? If having a more specific class, research how conversion errors can be trapped and handled.
- Consider the code that maps the `MenuItemForm` object to a `MenuItem` object. How would you improve this code?