Spring MVC Tutorial – Contact List Application

Step 06: CRUD Functionality without Ajax – Form Validation

Step 06: CRUD Functionality without Ajax – Form Validation

Overview

Our application is shaping up pretty well, but we still have some rough edges. At this point, the application allows us to enter invalid data into our form — we could leave all information blank or we could enter fields that have more characters than the database can handle. We need a way to validate the data entered into the form when the form is submitted: form validation.

In this step, we will implement form input validation using Spring Form tags and JSR-303 annotations. We will specify data validation rules and error messages, and we will configure Spring to handle the validation for us behind the scenes.

Configuration

Because we used the SWCGuild Spring MVC Maven archetype to create our project, it is already configured for form validation. In case you work on a project in the future that doesn't use this archetype, here are the important configuration requirements:

You must include the following dependency entries in your POM file:

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.1.0.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.1.0.Final</version>
</dependency>
```

You must configure Spring MVC to be annotation-driven. This is specified in spring-dispatcher-servlet.xml file with the `<mvc:annotation-driven />` entry.

Annotating the DTO

Note: This section is exactly like the **Annotating the DTO** section of Step 18 in the Ajax version of this tutorial.

We start by adding Java Specification Request (JSR) 303 and custom Hibernate Validator annotations to our Contact DTO. These annotations allow us to specify which fields are required, the min and max size of fields, and in some cases (such as email) the format of the contents of the field. Furthermore, these annotations allow us to specify the error message that should be returned if the given field value does not meet the validation criteria. There are many annotations available beyond the examples contained here; please see the Hibernate Validator and JSR-303 documentation for details.

Validation annotations are located with the fields themselves, not the getters/setters. Add annotations to your Contact DTO so that the property declarations look like this:

```
private int contactId;
    @NotEmpty(message="You must supply a value for First Name.")
    @Length(max=50, message="First Name must be no more than 50 characters in length.")
    private String firstName;
    @NotEmpty(message="You must supply a value for Last Name.")
    @Length(max=50, message="Last Name must be no more than 50 characters in length.")
    private String lastName;
    @NotEmpty(message="You must supply a value for Company.")
    @Length(max=50, message="Company must be no more than 50 characters in length.")
    private String company;
    @NotEmpty(message="You must supply a value for Phone.")
    @Length(max=10, message="Phone must be no more than 10 characters in length.")
    private String phone;
    @Email(message="Please enter a valid email address.")
    @Length(max=50, message="Email must be no more than 50 characters in length.")
    private String email;
```

Modifying the Controller

As mentioned above, this form validation technique uses the Spring Forms taglib and JSR-303 annotations. In previous steps, we used both straight HTML and the Spring Forms taglib for pages that submit forms. If you recall, we also used the Spring Forms taglib on the page that allows us to update contact information. Since that page is already set up for Spring Forms, we will use it to demonstrate form validation.

The Controller endpoint that we are interested in is “/editContactNoAjax,” which maps to the **editContactNoAjax** method on our Controller. We need to modify the signature, annotations, and code for this method in order for it to work with form validation.

Modify your **editContactNoAjax** method so that it looks like this:

Notes:

1. Use the @Valid annotation to indicate that the incoming Contact object must be validated.
2. Include a BindingResult object parameter to check for validation errors.
3. If there are validation errors, show the form again. We will see how the validation errors (if present) are displayed in the next section.
4. If there were no validation errors, update the Contact and display the Contact list as normal.

```
@RequestMapping(value="/editContactNoAjax", method=RequestMethod.POST)
// #1 and #2 - New annotation and parameter
public String editContactNoAjax(@Valid @ModelAttribute("contact") Contact contact,
                                BindingResult result) {

    // #3 - Redisplay the edit form if we find any validation errors
    if (result.hasErrors()) {
        return "editContactFormNoAjax";
    }

    // #4 - If there are no errors, update the contact as normal
    dao.updateContact(contact);
    return "redirect:displayContactListNoAjax";
}
```

Modifying the View

We must modify the form used to submit Contact modifications so that it can display validation errors when they are present and so that it matches up with the changes we just made to the Controller. Modify the form in your **editContactFormNoAjax.jsp** so that it looks like the code on the next page.

Notes:

Use the errors tag to display validation errors when they are present. The **cssclass** attribute is used to specify a CSS class that you would like to use for the display of error messages.

```
<sf:form class="form-horizontal" role="form" modelAttribute="contact"
         action="editContactNoAjax" method="POST">
    <div class="form-group">
        <label for="add-first-name" class="col-md-4 control-label">First Name:</label>
        <div class="col-md-8">
        <sf:input type="text" class="form-control" id="add-first-name"
                  path="firstName" placeholder="First Name"/>
        <!-- #1 - error tag -->
        <sf:errors path="firstName" cssclass="error"></sf:errors>
        </div>
    </div>
    <div class="form-group">
        <label for="add-last-name" class="col-md-4 control-label">Last Name:</label>
        <div class="col-md-8">
           <sf:input type="text" class="form-control" id="add-last-name"
                     path="lastName" placeholder="Last Name"/>
           <sf:errors path="lastName" cssclass="error"></sf:errors>
        </div>
    </div>
    <div class="form-group">
        <label for="add-company" class="col-md-4 control-label">Company:</label>
        <div class="col-md-8">
            <sf:input type="text" class="form-control" id="add-company"
                      path="company" placeholder="Company"/>
            <sf:errors path="company" cssclass="error"></sf:errors>
        </div>
    </div>
    <div class="form-group">
        <label for="add-email" class="col-md-4 control-label">Email:</label>
        <div class="col-md-8">
            <sf:input type="email" class="form-control" id="add-email" path="email" placeholder="Email"/>
            <sf:errors path="email" cssclass="error"></sf:errors>
        </div>
    </div>
    <div class="form-group">
        <label for="add-phone" class="col-md-4 control-label">Phone:</label>
        <div class="col-md-8">
            <sf:input type="tel" class="form-control" id="add-phone"
                      path="phone" placeholder="Phone"/>
            <sf:errors path="phone" cssclass="error"></sf:errors>
            <sf:hidden path="contactId"/>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-4 col-md-8">
            <button type="submit" id="add-button" class="btn btn-default">Add New Contact</button>
        </div>
    </div>
</sf:form>
```

Wrap-up

In this step, we have done the following:

1. Used JSR-303 annotations to specify validation rules on a Model object
2. Used Spring Forms tags to display form validation errors