Spring MVC Tutorial – Contact List Application

Step 18: Form Validation

Step 18: Form Validation

Overview

So far, our code only handles the happy path for the Contact List application. Of course, we know that users will put bad data into our system, either by accident or with malicious intent. This is where form validation comes in. In this step of the tutorial, we will use JSR 303 annotations, custom server-side error handling code, and client-side JavaScript to enforce validation on the Add New Contact form.

Configuration

Because we used the SWCGuild Spring MVC Maven archetype to create our project, things are 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:

1. 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>
```

2. 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

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;
```

Annotating the Controller

Our next step is to add the @Valid annotation to the **createContact** method of our controller. @Valid tells the Spring Framework to use the annotations on our DTO to validate the field values on the incoming Contact object. If all fields pass validation, everything works (this is the definition of the happy path). If one or more fields do not pass validation, an error is thrown. We will write code in the next section to handle these errors.

Add the @Valid annotation (shown in **bold red** below) to the createContact method of your Controller:

```
@RequestMapping(value="/contact", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED)
@ResponseBody public Contact createContact(@Valid @RequestBody Contact contact) {
    // persist the incoming contact
    dao.addContact(contact);
    // the addContact call to the dao assigned a contactId to the incoming
    // Contact and set that value on the object.  Now we return the updated
    // object to the caller
    return contact;
}
```

Creating the Validation Error and Validation Error Container

In this section, we will create a DTO (called ValidationError) to hold validation error information for a single field. We will then create a container (called ValidationErrorContainer) to hold all of the errors that occur when validating a given incoming Contact. Create a new Java class called **ValidationError** in the com.swcguild.contactlistmvc.validation package and insert the following code:

```
package com.swcguild.contactlistmvc.validation;

public class ValidationError {
    private String fieldName;
    private String message;

    public ValidationError(String fieldName, String message) {
        this.fieldName = fieldName;
        this.message = message;
    }

    public String getFieldName() {
        return fieldName;
    }

    public String getMessage() {
        return message;
    }
}
```

Now, create a new Java class called **ValidationErrorContainer** (put this in the **validation** package as well) and insert the following code (this class is just a thin wrapper around a List of ValidationErrors):

```
package com.swcguild.contactlistmvc.validation;

import java.util.ArrayList;
import java.util.List;

public class ValidationErrorContainer {
    private List<ValidationError> validationErrors = new ArrayList<>();

    public void addValidationError(String field, String message) {
        ValidationError error = new ValidationError(field, message);
        validationErrors.add(error);
    }

    public List<ValidationError> getFieldErrors() {
        return validationErrors;
    }
}
```

Creating the REST Validation Handler

We will now create the class that will handle validation errors when they occur. This class is advice that will be applied to our controller. Whenever a **MethodArgumentNotValidException** is thrown during validation, the **processValidationErrors** method on this class will be called. This method converts each validation error into a ValidationError object and adds it to the **ValidationErrorContainer**. After all errors have been processed, the **ValidationErrorContainer** is returned to the caller.

Create a new class called **RestValidationHandler** in the **validation** package. Add the following code:

Notes:

1. We must mark this class as ControllerAdvice to let the SpringFramework know that the code in this class should be applied to our controller
2. Mark this as an exception handler and specify which exception it handles
3. Specify the HTTP Status code to return after this class processes the validation errors
4. Mark our return type with the @ReponseBody annotation so that the returned ValidationErrorContainer will be included in the response body
5. Get the binding result from the exception
6. Put each validation error into the ValidationErrorContainer

```java
package com.swcguild.contactlistmvc.validation;

import java.util.List;
import org.springframework.http.HttpStatus;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

/**
 *
 * @author warde
 */
// #1: Mark this class as advice that should be applied to Controller components
@ControllerAdvice
public class RestValidationHandler {

    // #2: Specify which exception this handler can handle
    @ExceptionHandler(MethodArgumentNotValidException.class)
    // #3: Specify the HTTP Status code to return when an error occurs
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    // #4: Let Spring know that we have something to return in the body of the
    //     response.  In this case it will be a ValidationErrorContainer containing
    //     a ValidationError object for each field that did not pass validation.
    @ResponseBody
    public ValidationErrorContainer processValidationErrors(MethodArgumentNotValidException e) {
        // #5: get the Binding Result and all field errors
        BindingResult result = e.getBindingResult();
        List<FieldError> fieldErrors = result.getFieldErrors();

        // #6: Create a new ValidationError for each fieldError in the Binding Result
        ValidationErrorContainer errors = new ValidationErrorContainer();
        for(FieldError currentError : fieldErrors) {
            errors.addValidationError(currentError.getField(),
                                          currentError.getDefaultMessage());
        }
        return errors;
    }
}
```

Adding a Validation Error Div to the Home Page

We need a placeholder HTML element on the Home Page where we can display any validation errors returned from the server. Add the following div to home.jsp, placed just after the New Contact Form:

```
<div id="validationErrors" style="color: red"/>
```

Adding an Error Callback to the Add Button Ajax Call

At this point, we have added code to our server-side components that will return errors if and when they occur and we have a place (just below the New Contact form) where we can display these errors. We now need to add an error callback to the Ajax call that is triggered with the Create Contact button is clicked. The error callback is very similar to the success callback that we already have in place for the Ajax, except that it processes and displays validation error information rather than processing and displaying Contact information. Change the code for your #add-button click handler so it looks like the following:

Notes:

1. We are adding code to the .success callback that clears all error messages
2. The .error callback just goes through each validationError in fieldErrors and appends the error message to the #validationErrors div

```
    // on click for our add button
    $('#add-button').click(function (event) {
        // we don't want the button to actually submit
        // we'll handle data submission via ajax
        event.preventDefault();
        $.ajax({
            type: 'POST',
            url: 'contact',
            data: JSON.stringify({
                firstName: $('#add-first-name').val(),
                lastName: $('#add-last-name').val(),
                company: $('#add-company').val(),
                phone: $('#add-phone').val(),
                email: $('#add-email').val()
            }),
            headers: {
                'Accept': 'application/json',
                'Content-Type': 'application/json'
            },
            'dataType': 'json'
        }).success(function (data, status) {
            $('#add-first-name').val('');
            $('#add-last-name').val('');
            $('#add-company').val('');
            $('#add-phone').val('');
            $('#add-email').val('');
            // #1 - Remove all validation error messages
            $('#validationErrors').empty();
            loadContacts();
            //return false;
        }).error(function (data, status) {
            // #2 - Go through each of the fieldErrors and display the associated error
            // message in the validationErrors div
            $.each(data.responseJSON.fieldErrors, function (index, validationError) {
                var errorDiv = $('#validationErrors');
                errorDiv.append(validationError.message).append($('<br>'));
            });
        });
    });
```

With all of this code in place, that application will prevent the form from being submitted and will display any error messages when validation errors occur. When the errors are corrected and the form is resubmitted successfully, the error messages are cleared.

Wrap-up

In this step we covered the following techniques:

1. Using JSR 303 annotations to specify validation rules on DTOs
2. Using the @Valid annotation on our Controller methods to tell the Spring Framework to validate incoming DTOs using the JSR 303 validation rules
3. Creating and using validation error objects, validation error container objects, and a validation error handler to process validation errors when they occur
4. Using the jQuery .error Ajax callback to process errors on the client