Overview

In this lesson, we'll look at validating data that our users input. Any time your users type data into a form, you'll want to verify that it fits the format you need it to fit. That might be as simple as checking to see that it's not empty or it might mean checking length and contents.

We will look at two different methods of validating user input. Which one you use depends on how you take in the data. Each method has a different way of displaying on the page as well.

We are only looking at new tools to do validation. You can always set up a Service layer and validate there, sending the data up to the Controller and then sending data to the page to display the error. We will not be looking at that style of validation.

What Needs Validation?

We need to validate any user input that comes in. We often validate data based on the database. Let's say we define a field like this:

```
firstName VARCHAR(30) NOT NULL
```

We should validate at least two things here. First, because it is NOT NULL, we need to ensure that the first name is not empty. It is a required field and the user must type something there. Second, we should validate the length; specifically that it is not longer than 30 characters because anything longer than 30 characters will cause an error when we try to save it in the database.

Beyond what the database requires, we can validate based on our business requirements. For example, if we have a birthday field, we may want to verify that the date makes sense, that it's not too far in the past or in the future. If we have number values, we may want to verify that they are in a specific range. We can even validate against regular expressions.

Validation in Spring

From the Spring side of things, validation starts in the entity class; we add annotations onto fields to indicate how they should be validated. For example, here are the annotations for our first name field from above:

```
@NotBlank(message = "First name must not be empty.")
@Size(max = 30, message="First name must be less than 30 characters.")
private String firstName;
```

We've added two annotations, @NotBlank and @Size. @NotBlank will check that the firstName field is not null and that its length is greater than zero when the spaces are trimmed. We add the message property to indicate what message should be displayed when it doesn't pass this validation. @Size will check the length of the string and the max property indicates the maximum allowed length. We could add a min property if that was applicable for our check. We then add the message property as the feedback message when field this doesn't pass.

There are many different validations you can apply to fields. Here are a few common ones, and all of them take a message property to specify the feedback message:

`@NotBlank`	Verifies the field is not null and the trimmed size is greater than 0
`@Size(min=?, max=?)`	Verifies the size of the field with optional minimum and maximum checks; works on string length or collection/map size
`@Past/@Future`	Checks if a date or time is in the past or future
`@Digits(integer=X, fraction=X)`	Verifies the value is a number with X number of integral digits and X number of fractional digits; both properties are required
`@Email`	Verifies the field is a well-formed email address
`@Pattern(regexp=X)`	Verifies the field follows the regular expression pattern X; the regexp property is required

Once we have our entity annotated, we need to do the actual validation. There are a couple of ways we can do this depending on the situation.

If we pull in individual fields and build an object in our code, we need to create a Validator object to perform validation:

```
Validator validate = Validation.buildDefaultValidatorFactory().getValidator();
```

The imports for this come from the javax.validation package:

```
import javax.validation.Validation;
import javax.validation.Validator;
```

This Validator will look at the annotations we have added to the entity and check if the field matches them. Any data that does not match is added as an error to a list it gives back to us. This is what that looks like:

```
Set<ConstraintViolation<Teacher>> errors = validate.validate(teacher);
```

The ConstraintViolation object holds information about the error; specifically, each one will hold the message of a validation error it found. In this situation, we would send the full set to the page to process like a list, printing out the errors. We will see how that looks in the next section about validation in Thymeleaf.

The second way we can validate works when we pull in full objects from a form, something like this:

```
@PostMapping("editStudent")
public String performEditStudent(Student student) {
```

This ends up being much easier and cleaner, and we only have to make a few small changes to do the validation. To start, we add the @Valid annotation before the Student object:

```
public String performEditStudent(@Valid Student student)
```

This tells Spring to run validation against the Student object, checking the annotations in the entity. Next, we need a place to hold the errors, so we add another parameter, the BindingResult:

```
public String performEditStudent(@Valid Student student, BindingResult result)
```

The BindingResult holds the errors that were found. Spring automatically fills it in when it is paired with the @Valid annotation.

To deal with any errors that were found, we need to add a section to the beginning of the body of the method:

```
if(result.hasErrors()) {
    return "editStudent";
}
```

This will check if the BindingResult has any errors. If it does, it returns immediately to the same page we were coming from. The errors will be sent to the page along with any data that was already there, so we don't have to fill in a new Model to return to the page. We'll see how we can display the BindingResult errors in the next section.

Validation in Thymeleaf

Thymeleaf can handle either method of validation, but the method you use will affect how error messages are displayed. Let's start by looking at the set of ConstraintViolations method.

To start, we need to make sure the set is added to the Model so that the page can see it. Once it is there, it is just like any other set or list and we can loop through it using th:each to pull data from it:

```
<div class="col-12 m-1 text-center"
  th:if="${!errors.isEmpty()}">
    <p class="alert alert-danger"
      th:each="error : ${errors}" th:text="${error.message}">Error message</p>
</div>
```

- We will first use th:if to see if the set is empty; if it is, we don't need to display anything. If it's not empty, we want this section to show up.
- We then use th:each to loop through the errors and th:text to pull the message out of the error to display.

One benefit of this method is that the Thymeleaf attributes aren't anything new; it's just th:if, th:each, and th:text working on a set. One drawback is that the errors are displayed as one block and not tied directly to a field, so the error message needs to indicate what field caused the issue.

Dealing with errors coming from a BindingResult works differently. We can pull out errors for specific fields from the BindingResult, allowing us to display the error right next to the field in question. For example, here is the div with an input for first name with the error code attached:

```
<div class="col-9">
    <input type="text" name="firstName" id="firstName"
        class="form-control" th:value="${student.firstName}"/>
    <div th:if="${#fields.hasErrors('student.firstName')}">
        <p th:each="error : ${#fields.errors('student.firstName')}"
            class="alert alert-danger" th:text="${error}"> Bad first name </p>
    </div>
</div>
```

- We have our input in place normally.
- Below the input, we add a div with th:if that calls to ${#fields.hasErrors('student.firstName')}. The #fields call is a Thymeleaf utility that helps us look into the BindingResult at specific fields, in this case to check if the Student first name has any errors. If it does, this section will display.
- Inside the div, we use a p tag with th:each to loop through all the errors for the Student first name. The #fields.errors call will give us a list of errors for the field.
- Once we have the individual error, we can display it using th:text.

This method keeps the error next to the field it ties to, but we have to do some extra Thymeleaf work to use it and we must take in full objects in the back-end to get the BindingResult. When possible, this is usually the preferred way to handle validation.

Implementation

We are going to walk through setting up validation for a few forms in the code-along class roster from the previous lesson. We won't take the time to add it to every form because a lot of that would just be repetition, but feel free to take the time to implement it into the others as practice.

We'll start by adding validation annotation to our entities; no matter how we validate, those are necessary. Let's start with Teacher:

```
@NotBlank(message = "First name must not be empty.")
@Size(max = 30, message = "First name must be less than 30 characters.")
private String firstName;

@NotBlank(message = "Last name must not be empty.")
@Size(max = 50, message = "Last name must be less than 50 characters.")
private String lastName;

@Size(max = 50, message = "Specialty must be less than 50 characters")
private String specialty;
```

- firstName needs to have data in it but must not be longer than 30 characters, so we add @NotBlank and @Size with the max set to 30.
- lastName also needs to have data in it but must not be longer than 50 characters, so we add @NotBlank and @Size with the max set to 50.
- specialty is valid if it's blank, so we don't need @NotBlank; we just care that's not longer than 50 characters, giving us @Size with a max of 50.

Next, we look at Student, which is similar to Teacher:

```
@NotBlank(message = "First name must not be empty.")
@Size(max = 30, message="First name must be less than 30 characters.")
private String firstName;

@NotBlank(message = "Last name must not be empty.")
@Size(max = 50, message="Last name must be less than 50 characters.")
private String lastName;
```

- firstName must have data in it and must not be longer than 30 characters, so we add @NotBlank and @Size with the max set to 30.
- lastName also needs to have data in it and not be longer than 50 characters, so we add in @NotBlank and @Size with the max set to 50.

Next, we'll look at Course:

```
@NotBlank(message = "Name must not be blank")
@Size(max = 50, message="Name must be fewer than 50 characters")
private String name;

@Size(max = 255, message = "Description must be fewer than 255 characters")
private String description;

private Teacher teacher;
private List<Student> students;
```

- name needs to have data and a character limit of 50, so we use @NotBlank and @Size with max of 50.
- description can be empty but has a character limit of 255, so we only have @Size with the max of 255.
- We will not validate teacher and students. We could possibly put @NotNull onto teacher but we will trust the system to handle that. With student, we have nothing to validate because we don't have any defined limits on the size of the course.

We are choosing not to validate the Teacher or list of Students. Because we take those from the form as IDs, it would be difficult to use our existing validation to check them. Our setup should guarantee that we have a Teacher, but it is valid for no Students to come through, so we have nothing to validate there.

Now that we have the objects annotated, we can start working on the Controllers and Thymeleaf templates. We'll start by updating the Add Teacher form.

We have to make a few changes in the Teacher Controller to support this. First, we need to add a class variable to hold the set of ConstraintViolations from our Validator:

```
Set<ConstraintViolation<Teacher>> violations = new HashSet<>();
```

With that in place, we need to add some code to the addTeacher POST mapping. This code should replace the teacherDao.addTeacher(teacher); line:

```
Validator validate = Validation.buildDefaultValidatorFactory().getValidator();
violations = validate.validate(teacher);

if(violations.isEmpty()) {
    teacherDao.addTeacher(teacher);
}
```

- We instantiate our Validator object.
- We then pass the full Teacher object into the Validator and save the results in a “violations” class variable.
- We then check if we found any validation errors; if not, we add the Teacher.

The result of this is when there are validation errors, we return to the main page and they are displayed. When there are no validation errors, it saves the new Teacher and returns to the main page.

Before we can update the HTML, we need to be sure the violations variable is added to the Model for the main Teachers page. Add this line into the displayTeachers method, which is the GET mapping for teachers:

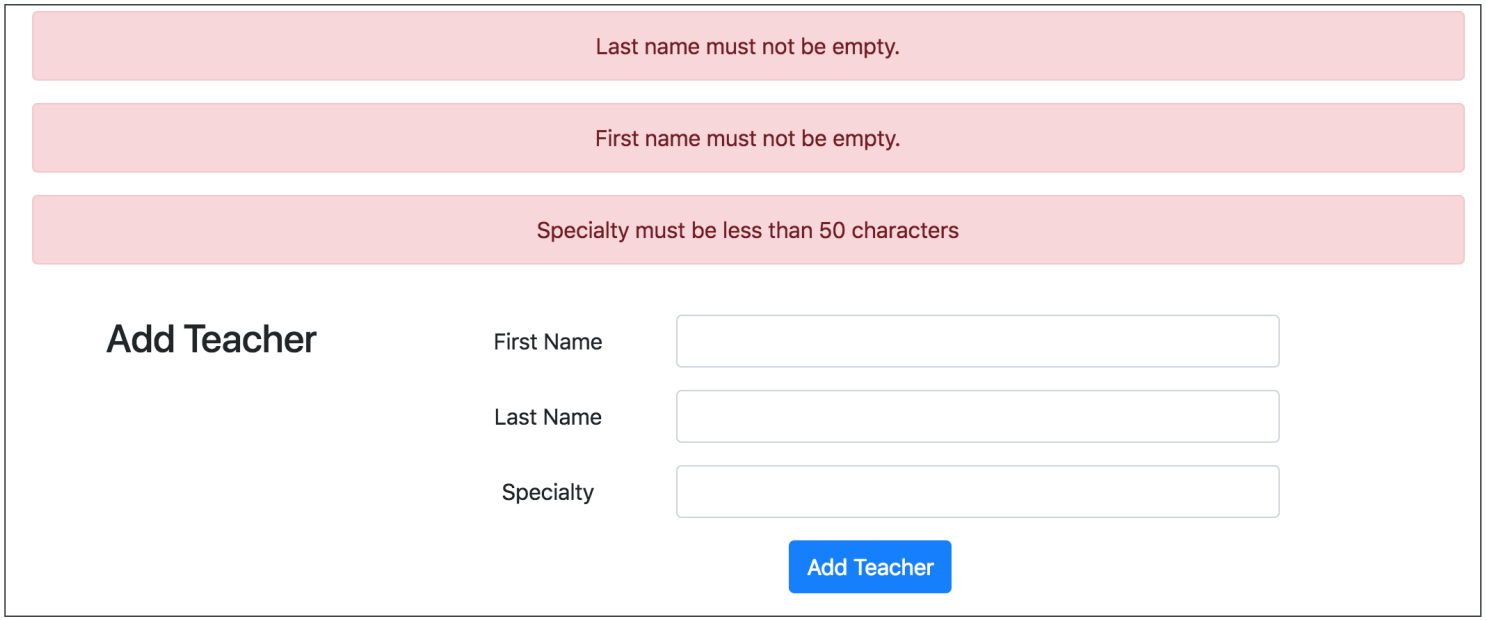
```
model.addAttribute("errors", violations);
```

Now we can update our teachers.html page to display the errors. We'll add code to the beginning of the Add Teacher div:

```
<div class="col-12 m-1 text-center"
      th:if="${!errors.isEmpty()}">
    <p class="alert alert-danger" th:each="error : ${errors}"
        th:text="${error.message}">Error message</p>
</div>
```

- We first have the th:if to check if we have any errors to display. We only display this div if there are errors.
- Inside the div, we loop through the errors with th:each and display the message of each one with th:text.
- We assign the class ‘alert alert-danger' to each message so it will display as a red alert box to make it clear there were errors.

With this in place, we should be able to see errors from the Add Teacher form displayed like this:



The data that was typed in is not preserved with this method. We only get the errors.

Next, let's look at the Edit Student form. Because the POST mapping for editStudent takes in the full Student object, we don't need to create a Validator to validate the data – we can just modify the method so it looks like this:

```
@PostMapping("editStudent")
public String performEditStudent(@Valid Student student, BindingResult result) {
    if(result.hasErrors()) {
        return "editStudent";
    }
    studentDao.updateStudent(student);
    return "redirect:/students";
}
```

- In our parameter list, we add the @Valid annotation before the Student object to indicate it should be validated.
- We add the BindingResult to the parameter list to hold the results of the validation.
- Inside the method, we check if the BindingResult has any errors in it. If it does, we go back to the editStudent page directly.
- If no error was found, we update the Student and redirect back to the main Student page like normal.

Returning directly to the Edit Student page will let us display the error results and keep the Student data that was submitted in the form. First, we will modify the section for the first name input:

```
<div class="form-group row">
  <label for="firstName" class="col-3 col-form-label">
            First Name</label>
  <div class="col-9">
    <input type="text" name="firstName" id="firstName" class="form-control"
      th:value="${student.firstName}" />
    <div th:if="${#fields.hasErrors('student.firstName')}">
      <p th:each="error : ${#fields.errors('student.firstName')}" class="alert alert-danger"
        th:text="${error}"> Bad first name </p>
    </div>
  </div>
</div>
```

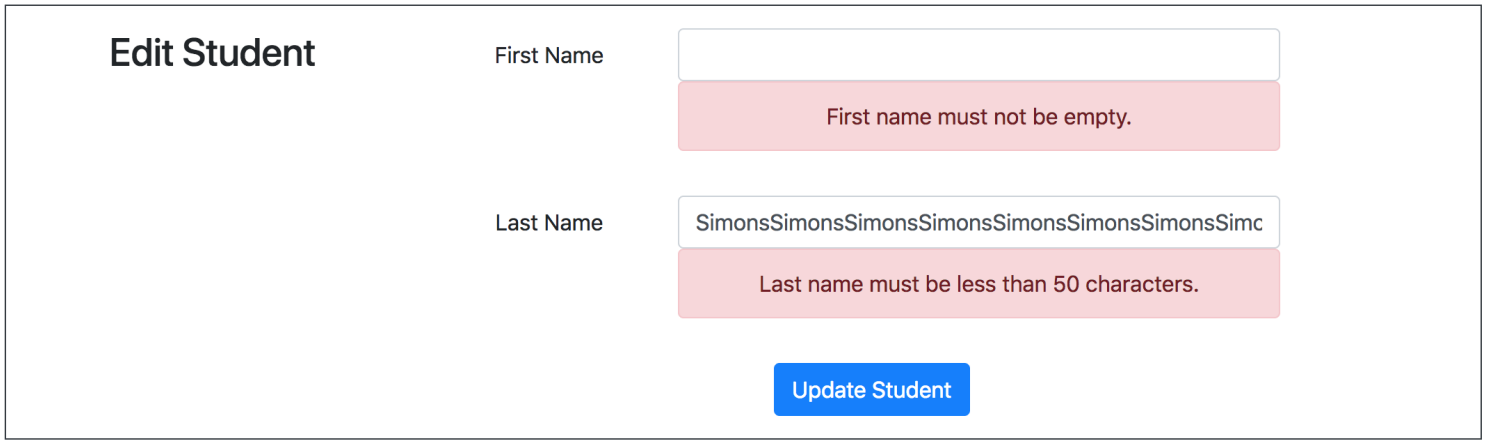
- We add in a new div after the input, with a th:if attribute that checks if the student.firstName has any errors. The #fields.hasErrors call looks at the BindingResult to check if the passed field has any errors. This div will only display when there are errors for that field.
- We get the list of errors for the field using the #fields.errors call and loop through it, displaying each one in an alert.

Now, we make the same change to the last name input section:

```
<div class="form-group row">
  <label for="lastName" class="col-3 col-form-label">
            Last Name</label>
  <div class="col-9">
    <input type="text" name="lastName" id="lastName" class="form-control"
      th:value="${student.lastName}" />
    <div th:if="${#fields.hasErrors('student.lastName')}">
      <p th:each="error : ${#fields.errors('student.lastName')}" class="alert alert-danger"
        th:text="${error}"> Bad last name</p>
    </div>
  </div>
</div>;
```

- The th:if checks for errors in student.lastName, so this div will display only when we have errors. The #fields.hasErrors will again check the BindingResult specifically for the field we pass it.
- We then loop through the actual errors, picking them up with #fields.errors and displaying each one in an alert.

With this method of display, the errors are right next to the field that caused them. We can pick up an error specifically for a field as opposed to picking them all up at once. The display will then look like this:



Next, let's look at Edit Course. We will do something a little different with Edit Course in that we will validate that we have selected at least one Student for the class. We do not want to save the data without a Student in the class. This makes our editCourse POST mapping more complicated because we pull in that data through the HttpServletRequest. Here is what it will look like now:

```
@PostMapping("editCourse")
    public String performEditCourse(@Valid Course course, BindingResult result, HttpServletRequest req
uest, Model model) {
        String teacherId = request.getParameter("teacherId");
        String[] studentIds = request.getParameterValues("studentId");

        course.setTeacher(teacherDao.getTeacherById(Integer.parseInt(teacherId)));

        List<Student> students = new ArrayList<>();
        if(studentIds != null) {
            for(String studentId : studentIds) {
                students.add(studentDao.getStudentById(Integer.parseInt(studentId)));
            }
        } else {
            FieldError error = new FieldError("course", "students", "Must include one student");
            result.addError(error);
        }

        course.setStudents(students);

        if(result.hasErrors()) {
            model.addAttribute("teachers", teacherDao.getAllTeachers());
            model.addAttribute("students", studentDao.getAllStudents());
            model.addAttribute("course", course);
            return "editCourse";
        }

        courseDao.updateCourse(course);
```

```
        return "redirect:/courses";
    }
```

- To start, we indicate that we are validating the simple Course fields with the @Valid annotation.
- We then add in the BindingResult directly after the Course; the BindingResult must follow whatever we are validating in order to work.
- At the end, we also add in a Model. This page needs extra information to display, the list of Teachers and the list of Students, so we need the Model to get the data back out there if we have validation errors.
- We pull in our Teacher and Student IDs like usual and set the Teacher, but before we create the students we need to check if studentIds is null. If no Students were selected, that field will come in as null and our enhanced for loop will throw an exception.
- If it's not null, we proceed like normal; but if it is null, we create a new type of object, a FieldError. The BindingResult uses FieldErrors to keep track of which field has errors in our object. Lucky for us, we can create our own FieldError and add it to the BindingResult. We create one for our students field and give it a message to print out on the page.
- We now put our list of Students into the Course; if it was null, it should just be an empty list.
- Now we can check if our BindingResult has any errors. If it does, we need to put some data into our Model: the list of Teachers, the list of Students, and finally the Course that was passed in. Normally we don't need to do this – the object we are validating goes back out automatically – but because we are putting data into the Model ourselves, we need to include it here. Once the data is in, we return to the editCourse page.
- Finally, if there were no errors, we finish the update and go back to the main Courses page.

The big addition here is that we can add FieldErrors ourselves. They must be fields that exist in our object, which is why we used “students” instead of “studentIds.” This way, we can still have validation in place for fields we can't validate normally.

On the page, displaying the errors looks the same as the Courses page. First, we have the name field:

```
<div class="form-group row">
  <label for="name" class="col-3 col-form-label">Name</label>
  <div class="col-9">
    <input type="text" name="name" id="name" class="form-control"
      th:value="${course.name}" />
    <div th:if="${#fields.hasErrors('course.name')}">
      <p th:each="error : ${#fields.errors('course.name')}"
        class="alert alert-danger" th:text="${error}">
        Bad name </p>
    </div>
  </div>
</div>
```

- We add in the div with th:if on whether our course.name has any errors, using #fields.hasErrors to check the BindingResult.
- If we do have errors, we then loop through them with th:each, picking up the errors with #fields.errors and displaying them in an alert.

We next have description:

```
<div class="form-group row">
    <label for="description" class="col-3 col-form-label">
        Description</label>
    <div class="col-9">
        <input type="text" name="description" id="description" class="form-control"
            th:value="${course.description}"/>
        <div th:if="${#fields.hasErrors('course.description')}">
            <p th:each="error : ${#fields.errors('course.description')}"
                class="alert alert-danger" th:text="${error}">
                    Bad description</p>
        </div>
    </div>
</div>
```

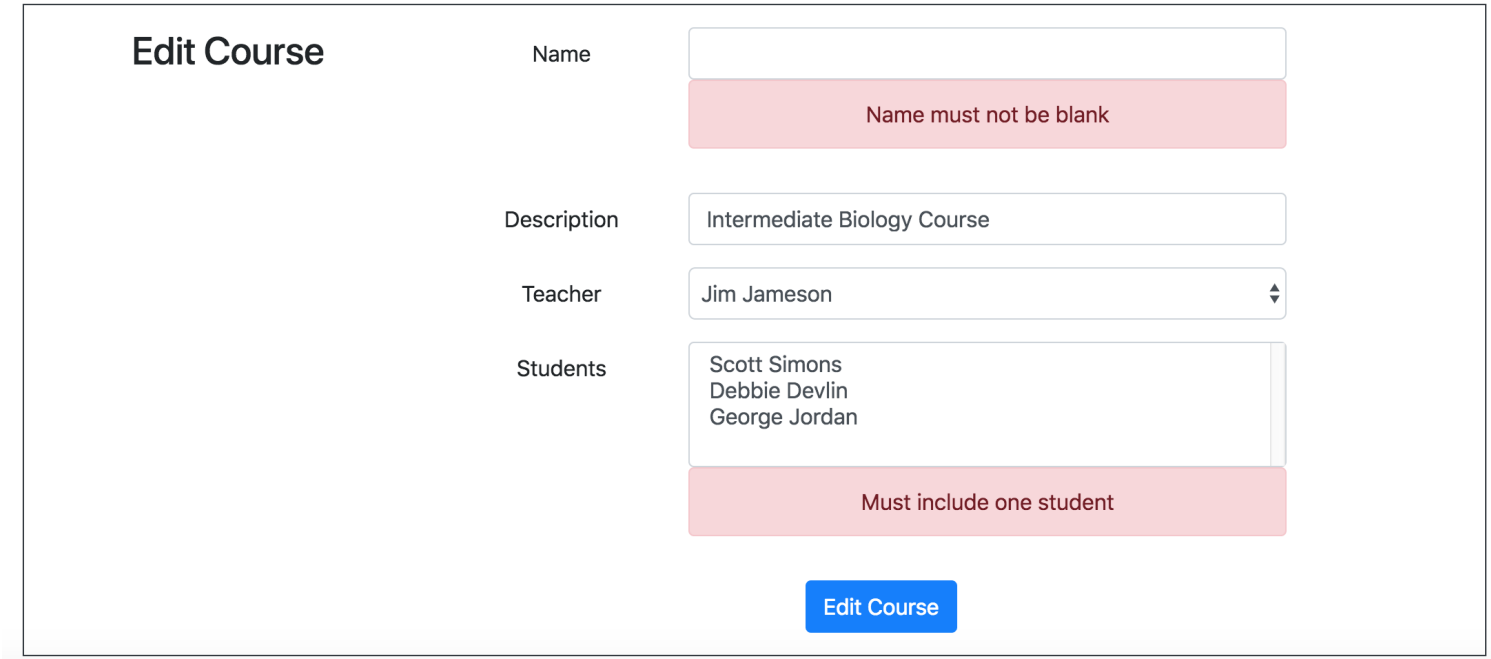
- We add in the div with th:if on whether our course.description has any errors, using #fields.hasErrors to check the BindingResult.
- If we do have errors, we then loop through them with th:each, picking up the errors with #fields.errors and displaying them in an alert.

Lastly, we have the students field:

```
<div class="form-group row">
    <label for="students" class="col-3 col-form-label">Students</label>
    <div class="col-9">
        <select multiple id="students" name="studentId" class="form-control">
            <option th:each="student : ${students}"
                    th:value="${student.id}"
                    th:text="${student.firstName + ' ' + student.lastName}"
                    th:selected="${course.students.contains(student)}">
                Student Name</option>
        </select>
        <div th:if="${#fields.hasErrors('course.students')}">
            <p th:each="error : ${#fields.errors('course.students')}"
               class="alert alert-danger" th:text="${error}">
                Bad student </p>
        </div>
    </div>
</div>
```

- In the BindingResult, it's just another field error, so we pick it up the same way.
- We add in the div with th:if on whether our course.students has any errors, using #fields.hasErrors to check the BindingResult.
- If we do have errors, we loop through them with th:each, picking up the errors with #fields.errors and displaying them in an alert.

When we have errors, the Edit Course page should now look like this:



Those are the only forms we are going to apply validation to, but you should be able to use these same techniques to validate any data coming into the application.

Summary

We went through two techniques to validate data: using the Validator class or using the @Valid annotation and BindingResults. Both work – you just have to decide which one makes the most sense for your project. This lesson included the most common methods, but there are many other types of validations you can use as well.

◄ Previous activity

Jump to...

Next activity ►