Desenvolvimento de Aplicações Empresariais – 2021-22-1S

Engenharia Informática – 3.º ano – Ramo SI

**Worksheet 6**

**Topics**: Client-side validation, Sending emails.

## Client-side validation

In a previous lesson, we validated the fields on the server-side. Now, let us implement the same in our client-side application (NUXT). As already mentioned in previous worksheets, it is important to validate inputs on each different layer, although the kind of validations should be distinct.

To validate the input fields on our NUXT application, you can use HTML and Javascript. For HTML, you can validate the fields using the properties that enforces the validations.

For example:

- `<input name="foo" required>` guarantees that the input field for "foo" must be required, and you can't submit the form while you don't give a value for this input.
- `<input name="course_type" pattern="graduation|master|PhD">`. The property "pattern" applies a regex to the value of the input. In this case, the value must be either "graduation", "master" or "PhD".
- `<input name="reason" minlength="10" maxlength="150">`. *minlength* and *maxlength* validates that the reason field must have, at least 10 characters and a maximum of 150 characters.
- `<input type="number" name="year" min="2010" max="2050">`. In the case of an input of type number, you can validate that the number must be between 2010 and 2050.
- `<input type="email" name="email">`, the field must be an e-mail. You don't need to use a pattern (regex). The input of type e-mail already validates that for you.

Read the documentation about form validations to better understand how to use the HTML input fields: https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Form_validation. Also, you can use other input types, like date, datetime-local, password (hides the characters), and others. Please, visit https://www.w3schools.com/html/html_form_input_types.asp to check what types you should use.

Besides the basic validations using HTML, you must complement your validations using Javascript. This gives you more power for complex cases. There are a lot of Javascript libraries that already do the job for you. You can google for libraries and implement on your project. For example, if you selected the framework "bootstrap-vue" for the web style, you can use https://bootstrap-validate.js.org/. This library uses jQuery. But you have a lot of even better validations libraries, that integrates with Vue.js. Please, check the most awesome packages for Vue here: https://github.com/vuejs/awesome-vue.

For this lesson, we will implement some validations using vanilla Javascript. Yet, we encourage you to use a Vue library instead, that saves you a lot of time! You can implement it right away in the academics NUXT project, or later in your workgroup project.

**1.** Implement the following example of Javascript validations for "pages/students/create.vue":

```
<template>
    <div>
        <h1>Create a new Student</h1>
        <form @submit.prevent="create" :disabled="!isFormValid">
            <b-input v-model.trim="username" :state="isUsernameValid" required
placeholder="Enter your username" />
            <b-input v-model="password" :state="isPasswordValid" required
placeholder="Enter your password" />
            <b-input v-model.trim="name" :state="isNameValid" required
placeholder="Enter your name" />
            <b-input ref="email" v-model.trim="email" type="email"
:state="isEmailValid" required pattern=".+@my.ipleiria.pt" placeholder="Enter
your e-mail" />
            <b-select v-model="courseCode" :options="courses"
:state="isCourseValid" required value-field="code" text-field="name">
                <template v-slot:first>
                    <option :value="null" disabled>-- Please select the Course --
</option>
                </template>
            </b-select>
            <p class="text-danger" v-show="errorMsg">{{ errorMsg }}</p>
            <nuxt-link to="/students">Return</nuxt-link>
            <button type="reset" @click="reset">RESET</button>
            <button @click.prevent="create"
:disabled="!isFormValid">CREATE</button>
        </form>
    </div>
</template>

<script>
export default {
    data() {
        return {
            username: null,
            password: null,
            name: null,
            email: null,
            courseCode: null,
            courses: [],
            errorMsg: false
        }
    },
```

```
    created() {
        this.$axios.$get('/api/courses').then(courses => { this.courses = courses
})
    },

    computed: {
        isUsernameValid () {
            if (!this.username) {
                return null
            }

            let usernameLen = this.username.length
            if (usernameLen < 3 || usernameLen > 15) {
                return false
            }

            return true
        },

        isPasswordValid () {
            if (!this.password) {
                return null
            }
            let passwordLen = this.password.length
            if (passwordLen < 3 || passwordLen > 255) {
                return false
            }
            return true
        },

        isNameValid () {
            if (!this.name) {
                return null
            }
            let nameLen = this.name.length
            if (nameLen < 3 || nameLen > 25) {
                return false
            }
            return true
        },

        isEmailValid () {
            if (!this.email) {
                return null
            }

            // asks the component if it's valid. We don't need to use a regex for
the e-mail. The input field already does the job for us, because it is of type
"email" and validates that the user writes an e-mail that belongs to the domain
of IPLeiria.
            return this.$refs.email.checkValidity()
        },

        isCourseValid () {
            if (!this.courseCode) {
                return null
            }
            return this.courses.some(course => this.courseCode === course.code)
        },

        isFormValid () {
            if (! this.isUsernameValid) {
```

```
                return false
            }

            if (! this.isPasswordValid) {
                return false
            }

            if (! this.isNameValid) {
                return false
            }

            if (! this.isEmailValid) {
                return false
            }

            if (! this.isCourseValid) {
                return false
            }

            return true
        }
    },

    methods: {
        reset () {
            this.errorMsg = false
        },

        create() {
            this.$axios.$post('/api/students', {
                username: this.username,
                password: this.password,
                name: this.name,
                email: this.email,
                courseCode: this.courseCode
             })
            .then(() => {
                this.$router.push('/students')
            })
            .catch(error => {
                this.errorMsg = error.response.data
            })
        }
    }
}
</script>
```

**2.** Make sure your database and your Java EE application are running, run and test the NUXT `academics` client application.

**3.** (Optional) This code only outlines red or green around the field when the input is validated. It is better to show an error message to the user. Go further and try to use a Vue library for validations. When the user inputs a wrong value, you should display a `<span>` (or other HTML element) with text in red, under the input field, with the error message. You can use https://vuelidate.netlify.com/ or https://logaretm.github.io/vee-validate/.

Here is another example for the page "students/create.vue", that uses the the BootstrapVue framework to show error messages under an input field:

// NOTE: this code only shows an example for the "*username*" input

```
<template>
    <b-container>
        <div>
        <h1>Create a new Student</h1>

        <form @submit.prevent="create" :disabled="!isFormValid">
            <b-form-group
              id="username"
              description="The username is required"
              label="Enter your username"
              label-for="username"
              :invalid-feedback="invalidUsernameFeedback"
              :state="isUsernameValid"
            >
              <b-input id="username" v-model.trim="username" :state="isUsernameValid"
trim></b-input>
            </b-form-group>
            <b-input v-model="password" type="password" :state="isPasswordValid"
required placeholder="Enter your password" />
            <b-input v-model.trim="name" required :state="isNameValid"
placeholder="Enter your name" />
            <b-input ref="email" v-model.trim="email"
type="email" :state="isEmailValid" required pattern=".+@my.ipleiria.pt"
placeholder="Enter your e-mail" />
            <b-select v-model="courseCode" :options="courses" :state="isCourseValid"
required value-field="code" text-field="name">
                <template v-slot:first>
                    <option :value="null" disabled>-- Please select the Course --
</option>
                </template>
            </b-select>
            <p class="text-danger" v-show="errorMsg">{{ errorMsg }}</p>
            <nuxt-link to="/students">Return</nuxt-link>
            <button type="reset" @click="reset">RESET</button>
            <button @click.prevent="create" :disabled="!isFormValid">CREATE</button>
        </form>
    </div>
    </b-container>
</template>

<script>
export default {
    data() {
        return {
            username: null,
            password: null,
            name: null,
            email: null,
            courseCode: null,
            courses: [],
            errorMsg: false
        }
    },

    created() {
        this.$axios.$get('/api/courses').then(courses => { this.courses = courses })
    },
```

```
    computed: {
        invalidUsernameFeedback () {
            if (!this.username) {
                return null
            }
            let usernameLen = this.username.length
            if (usernameLen < 3 || usernameLen > 15) {
                return 'The username must be between [3, 15] characters.'
            }

            return ''
        },

        isUsernameValid () {
            if (this.invalidUsernameFeedback === null) {
                return null
            }
            return this.invalidUsernameFeedback === ''
        },

        isPasswordValid () {
            if (!this.password) {
                return null
            }
            let passwordLen = this.password.length
            if (passwordLen < 3 || passwordLen > 255) {
                return false
            }
            return true
        },

        isNameValid () {
            if (!this.name) {
                return null
            }
            let nameLen = this.name.length
            if (nameLen < 3 || nameLen > 25) {
                return false
            }
            return true
        },

        isEmailValid () {
            if (!this.email) {
                return null
            }
            return this.$refs.email.checkValidity()
        },

        isCourseValid () {
            if (!this.courseCode) {
                return null
            }
            return this.courses.some(course => this.courseCode === course.code)
        },

        isFormValid () {
            if (! this.isUsernameValid) {
                return false
            }
            if (! this.isPasswordValid) {
                return false
            }
```

```
            if (! this.isNameValid) {
                return false
            }
            if (! this.isEmailValid) {
                return false
            }
            if (! this.isCourseValid) {
                return false
            }

            return true
        }
    },

    methods: {
        reset () {
            this.errorMsg = false
        },

        create() {
            this.$axios.$post('/api/students', {
                username: this.username,
                password: this.password,
                name: this.name,
                email: this.email,
                courseCode: this.courseCode
             })
            .then(() => {
                this.$router.push('/students')
            })
            .catch(error => {
                this.errorMsg = error.response.data
            })
        }
    }
}
</script>
```

You can visit the official documentation and check the given examples:

https://bootstrap-vue.js.org/docs/components/form

https://bootstrap-vue.js.org/docs/components/form-checkbox

https://bootstrap-vue.js.org/docs/components/form-file

https://bootstrap-vue.js.org/docs/components/form-group

https://bootstrap-vue.js.org/docs/components/form-input

https://bootstrap-vue.js.org/docs/components/form-radio

https://bootstrap-vue.js.org/docs/components/form-select

https://bootstrap-vue.js.org/docs/components/form-textarea

**4.** Validate the remaining inputs of your NUXT application (e.g.: pages to update a student, create/update a course, create/update a teacher, etc.), using HTML and Javascript or a library you may like.

## Sending emails

**5.** Download the fakeSMTP-2.0.jar file from Moodle. FakeSMTP simulates an email server on localhost (on your host machine), port 25.

**6.** Be sure to update your Java EE configuration files with the ones from the Github repo at: https://github.com/dmp593/docker-wildfly-postgres (either download and copy them from the website, or execute `git pull` from your local folder, if you've previously cloned this repo).
We added up a `setup_mail.sh` script and changed `startup_wildfly.sh` to execute it. This new script setup_mail.sh creates a new Mail Session in Wildfly named `fakeSMTP;`

**7.** Recreate your docker containers by executing `make down` and then `make up`, so that the new `setup_mail.sh` script gets executed and the new Wildfly container gets configured with the new `fakeSMTP` mail session. After the containers are up, you can check this new mail session on the Wildfly server administration console at http://localhost:9990, accessing *Configuration→Subsystems→Mail→fakeSMTP* and clicking "view";

**8.** Our code needs to use the `javax.mail.jar` library. Add the following dependency to your Maven pom.xml file:

```
<dependency>
    <groupId>javax.mail</groupId>
    <artifactId>mail</artifactId>
    <version>1.5.0-b01</version>
</dependency>
```

**9.** Create the stateless `EmailBean` EJB with the following code:

```
@Stateless
public class EmailBean {
    @Resource(name = "java:/jboss/mail/fakeSMTP")
    private Session mailSession;

    public void send(String to, String subject, String text) throws
MessagingException {
        Message message = new MimeMessage(mailSession);
        try {
            // Adjust the recipients. Here we have only one recipient.
            // The recipient's address must be an object of the InternetAddress
class.
            message.setRecipients(Message.RecipientType.TO,
InternetAddress.parse(to, false));

            // Set the message's subject
            message.setSubject(subject);

            // Insert the message's body
            message.setText(text);

            // Adjust the date of sending the message
            Date timeStamp = new Date();
            message.setSentDate(timeStamp);
```

```
            // Use the 'send' static method of the Transport class to send the
message
            Transport.send(message);
        } catch (MessagingException e) {
            throw e;
        }
    }
}
```

**10.** Create the `EmailDTO` class, which should have a `subject` and a `message` as attributes.

**11.** On the `StudentService` class, create a service method with the following signature that allows to send an email to some student:

```
@POST
@Path("/{username}/email/send")
public Response sendEmail(@PathParam("username") String username, EmailDTO email)
throws MyEntityNotFoundException, MessagingException {

    Student student = studentBean.findStudent(username);
    if (student == null) {
        throw new MyEntityNotFoundException("Student with username '" + username
+ "' not found in our records.");
    }
    emailBean.send(student.getEmail(), email.getSubject(), email.getMessage());
    return Response.status(Response.Status.OK).entity("E-mail sent").build();
}
```

**12.** Run the FakeSMTP application (double click on the `fakeSMTP-2.0.jar` file) and click in the "Start server" button.

**13.** Run your application (`make monitor`).

**14.** Write an HTTP request to test the new service;

**15.** (Optional) As you can see, if you call this endpoint, you need to wait a little to get the response "email sent". If you want to get an immediate response, you need to transform the code to be asynchronous, using a thread that handles the job. To do it, change the code in the class `EmailBean` to:

```
package ejbs;

import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
@Stateless(name = "EmailEJB")
public class EmailBean {

    @Resource(name = "java:/jboss/mail/fakeSMTP")
    private Session mailSession;

    private static final Logger logger = Logger.getLogger("EmailBean.logger");

    public void send(String to, String subject, String text) {
        Thread emailJob = new Thread(() -> {
            Message message = new MimeMessage(mailSession);
            Date timestamp = new Date();

            try {
                message.setRecipients(Message.RecipientType.TO,
InternetAddress.parse(to, false));

                message.setSubject(subject);

                message.setText(text);

                message.setSentDate(timestamp);

                Transport.send(message);
            } catch (MessagingException e) {
                logger.log(Level.SEVERE, e.getMessage());
            }
        });

        emailJob.start();
    }
}
```

However, there are better approaches as, for example, queues and events that notify the client when the e-mail is really sent. This approach runs the code after the response is sent to the client. By doing so, the client is notified that the e-mail was sent, when in reality, you are still processing the job. If you had an error sending the e-mail, you couldn't notify the user. You are always sending the OK status. A better approach is the use of queues that do the job in background and using events that fire when the e-mail is sent, and when the e-mail throws an error.

The client subscribes to the events using sockets. This approach is more complex and heavy, so we will opt for the approach that sends an e-mail and responds to the client, while still sending the e-mail using a background thread. If, for some reason, the e-mail server is down or simply didn't work, we just assume that this is not critical to our application, and the user can still click again on the `students` page and write a new e-mail. Many applications use this optimistic approach, when the e-mail notification is not critical for the user, because it improves the server performance. Imagine this scenario: You have 100 clients sending e-mails at the same time. If you send a synchronous email, at the second e-mail request, the server is already blocked and does not respond to anybody, because it's stuck sending the first e-mail. You can try this, by sending 3 requests at the same time, and a fourth request to create a new student. You need to wait the response to the student creating until all the 3 e-mails were sent.

**16.** Let's create a new form page to send an e-mail to a student. For that, we want to know to whom we want to send the email. The email page form must also receive, like the details page (_username.vue), the username as a route parameter. To do so, create a new directory, under *pages* folder, called "*_username*" and move the file "*pages/students/_username.vue*" to "*pages/students/_username/index.vue*".

**17.** Run the `academics-management` NUXT application. Everything must work as before.

**18.** Create a new page "*pages/students/_username/send-email.vue*".

**19.** Create a form page that asks for the *subject* and the *message* for the e-mail to send. When the user submits the form, it calls the endpoint designed to send e-mails. We encourage you to try to code on your own and only check the code below just as a reference. If you have the time, you can improve a lot the web design.

```
<template>
    <div>
        <h1>Send an E-mail to Student {{ student.name }}</h1>
        <form @submit.prevent="send">
            <div>
                subject:
                <input v-model="subject" type="text">
            </div>
            <div>
                message:
              <textarea v-model="message" name="message"></textarea>
            </div>
            <nuxt-link :to="`/students`">
              Go to Students
            </nuxt-link>
             
            <nuxt-link :to="`/students/${username}`">
              Go to Student Details
            </nuxt-link>
             
            <button @click.prevent="send">
              SEND
            </button>
       </form>
     </div>
</template>
<script>
export default {
    data() {
        return {
            student: {},
            subject: null,
            message: null,
        }
    },
    created() {
        this.$axios.$get(`/api/students/${this.username}`)
            .then((student) => { this.student = student })
    },
```

```
    computed: {
        username() {
            return this.$route.params.username
        }
    },
    methods: {
        send() {
            this.$axios.$post(`/api/students/${this.username}/email/send`, {
                subject: this.subject,
                message: this.message
             })
            .then(msg => {
                this.$toast.success(msg).goAway(1500)
            })
            .catch(error => {
                this.$toast.error('error sending the e-mail').goAway(3000)
            })
        }
    }
}
</script>
```

**Note:** to use the *toast* feature (this.$toast) you must add this module. Run the

`npm i --save @nuxtjs/toast`

command. Add this module in the file "nuxt.config.js", in the *modules* array.

```
/*
** Nuxt.js modules
*/
modules: [
  // Doc: https://bootstrap-vue.js.org/docs/
  'bootstrap-vue/nuxt',
  '@nuxtjs/axios',
  '@nuxtjs/toast'
],
...
```

**20.** Add a `<nuxt-link>` to go to the `send-email.vue` page. You can add it in the details page and in the students table, in the "Actions" column.

```
// pages/students/_username/index.vue ("details page")
<template>
  <b-container>
    <h4>Student Details</h4>
    <p>Username: {{ student.username }}</p>
    <p>Name: {{ student.name }}</p>
    <p>Email: {{ student.email }}</p>
    <p>Course: {{ student.courseName }}</p>
    <h4>Subjects</h4>
    <b-table v-if="subjects.length" striped over :items="subjects"
:fields="subjectFields" />
      <p v-else>No subjects enrolled.</p>
     <nuxt-link to="/students">Go back</nuxt-link>
      
     <nuxt-link :to="`/students/${this.username}/send-email`">Send e-mail</nuxt-
link>
  </b-container>
</template>

// pages/students/index.vue
```

```
<template>
  <div>
    <b-container>
      <b-table striped over :items="students" :fields="fields">
        <template v-slot:cell(actions)="row">
          <nuxt-link class="btn btn-link"
:to="`/students/${row.item.username}`">Details</nuxt-link>
          <nuxt-link :to="`/students/${row.item.username}/send-email`">Send e-
mail</nuxt-link>
        </template>
      </b-table>
      <nuxt-link to="/">Back</nuxt-link>
       
      <nuxt-link to="/students/create">Create a New Student</nuxt-link>
    </b-container>
  </div>
</template>
```

**21.** Run and test the application, by sending an e-mail to a student.

**Bibliography**

Java EE Tutorial 8 (all of it)

NUXT

NUXT | Module Toast

Bootstrap Vue

JavaMail

HTML Forms - Mozilla Developer Network