Project Part 8 - Client-Side Validation

20 Possible Points

11/8/2021

Attempt 1

IN PROGRESS
Next Up: Submit Assignment

Add Comment

Unlimited Attempts Allowed

Details

Due: Monday, November 8
Points: 20 points
Deliverables:

- [Name]BookstoreValidate.war uploaded to the CS5244 server
- URL for your CS5244 site uploaded to Canvas

Resources:

- **Checkout.vue (https://canvas.vt.edu/courses/136000/files/19368745/download)**

Project 8: Client-Side Validation

Overview

In this project you will construct a checkout form and learn how to validate form data in the browser (client). You will be introduced to the art of form validation using the **Vuelidate (https://vuelidate.js.org/)** validation framework.

1. Project Setup

Establish a new project `[Name]BookstoreValidate` according to our standard setup procedure. Copy the relevant files from your Session project and make the relevant name changes. Run your project. It should behave the same as your last project.

In IntelliJ, open a new Terminal window using the `View | Tool Windows | Terminal` menu option (or just click the + on the current terminal window). Change into the client folder (cd client).

Submit Assignment

- **Install the validator javascript library**: `npm install validator --save`
  - We are using this library for validations for phone and credit card.
- **Install the vue-json-tree-view plugin** for Vue: `npm install vue-json-tree-view --save`
  - We are going to use this tree view to visualize the validation object to make debugging easier.
- **Use Vuelidate and the Tree View plugins** in `main.js`

```
import Vuelidate from 'vuelidate'
import TreeView from 'vue-json-tree-view'

Vue.use(Vuelidate)
Vue.use(TreeView)
```

2. Create a Checkout Page

Install the provided `Checkout.vue` resource into your client/src/views folder, overwriting the (mostly) empty component you made previously.

In the script section, some of the lines are commented out so that you do not get "unused variable" errors. For example:

```
// email

// import isCreditCard from "validator/lib/isCreditCard";
// import isMobilePhone from "validator/lib/isMobilePhone";

// const phone = value => isMobilePhone(value, "en-US");
// const creditCard = value => isCreditCard(value);
```

As you have a need for these variables, you can uncomment these lines.

This `Checkout.vue` contains the form elements required on the page, together with some support for validation out-of-the-box. **Launch your bookstore, add an item to the cart and navigate to the Checkout page**. You should see a blank form, with expiry dates set for the current month and a missing expiry year. Also notice a 'tree view' in the box to the right. This 'tree view' shows the current validation state of the form. Type a name with only 3 characters into the "Name" field so that your can see what an error message looks like. You should see something like the following:

Submit Assignment

Create a (mostly blank) Confirmation Page

Create a new route and view called `Confirmation` for a confirmation page.

Please have a short message on the page so that there is some space between the header and footer.

3. Checkout Page Requirements

The checkout page must have the following elements:

- If your cart is empty, the checkout page should have a message indicating you need to add an item to your cart to checkout, and a button to "Continue Shopping" taking the customer to the selected category page.
- A form to capture customer details: name, address, phone, email, credit card number and credit card expiry date (month and year).
- A checkout button to complete the purchase of the cart.
- The total amount to be charged along with subtotal and surcharge values should also be

Submit Assignment

4. Checkout Validation Requirements

The form fields should be validated as follows using Vuelidate plugin and the validator library.

- All fields including name and address are required
- Name and address fields should be between 4 and 45 characters in length
- Email: use Vuelidate's `email` validator to check this
- Phone, ccNumber: use the provided functions `phone` and `creditCard` to validate these.
- Expiration date: we will NOT be validating the expiration date on the client side (we will validate it on the server side)

Error messages should be displayed somewhere sensible (like underneath the field that has an error).

Do NOT use HTML5 form validation. In particular:

- Input tags should not have a "requires" attribute
- Input tags should have type="text" (not, for example, type="email")
- Input tags should not have a min length

If you want to ensure you are not using HTML5 form validation, just keep the input tags the way they are, except where you need to add a validation attribute.

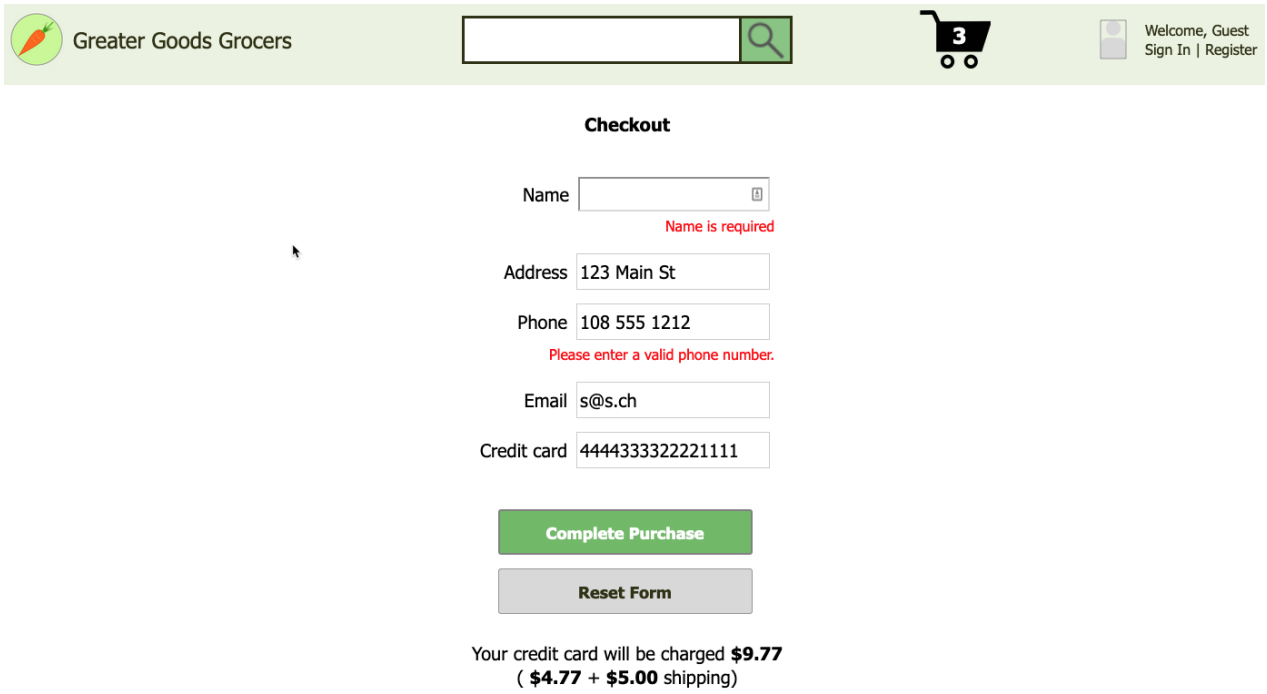
5. Completing the Checkout Template

The form provided has validations for name implemented as an example. It also provides a useful tree view showing validation status for every field in the validations property of the component. Use this tree to see how the field validation properties change as you type in your form fields.

Once you have your validations working on each field, don't forget to remove the tree view from your Checkout component - it is obviously not desirable for use by your customers.

Here is an example of a similar checkout page from our sample site.

Submit Assignment



An example showing different erroneous and successful validation states of input fields

To complete the checkout form you should complete the TODO tasks in the provided Checkout.vue file.

This includes:

- Add the address field to the form
- Complete styling for all elements on the page
- Implement validation messages for all fields (excluding the expiry date fields)
- Finish implementing the expiry year field element to display all years from the current year for the next 15 years as possible expiry years.
- Display the cart total, subtotal and surcharge near the checkout button.

6. Submitting the Form

Now that are elements are validating, we need to submit the form.

If any elements are in error, the form submission should fail and the customer should remain on the checkout page with an error message. If all elements pass validation, we should process the form and proceed to the confirmation page.

Let's now add a submit event listener to your checkout <form> to process the form. The submit event should call the method named 'submitOrder' that is provided. Don't forget to prevent the browser from submitting the form. (`@submit.prevent="submitOrder"`)

Submit Assignment

```
    submitOrder() {
        console.log("Submit order");
        this.$v.$touch(); // Ensures validators always run
        if (this.$v.$invalid) {
            this.checkoutStatus = "ERROR";
        } else {
            this.checkoutStatus = "PENDING";
            setTimeout(() => {
              this.checkoutStatus = "OK";
        setTimeout(() => {this.$router.push({ name: 'confirmation' })}, 1000);

            }, 1000);
        }
      }
```

The code first ensures that all validators have been executed using this.$v.touch(). This ensures validators are executed if someone simply clicks the checkout button with no form element interaction.

Then the code updates the `checkoutStatus` data field to track the form processing state. Note that the template provided includes a `checkoutStatusTextHolder` section that displays messages on the view, driven by a `checkoutStatus` data field. You are able to re-style those messages as you see fit for your site.

Then for this project, successful form submission means moving through the `checkoutStatus == 'OK'` state and landing on the (mostly blank) confirmation page. The code to perform these action uses the standard `setTimeout` function in Javascript that runs code after a time delay, and is temporary. Our intent is to replace this with the real order being sent and saved on the server in the next project.

7. Style Requirements

The styling of the form in the startup code is fairly close to what is already acceptable. The input and selector fields are aligned to the right and there is space between the labels and the fields. There are a few stylistic requirements stated explicitly as TODOs in the starter code:

- Fix the proximity problem with the error messages. In the starter code, the error messages are closer to the following fields than the are to the fields they represent. Fix this so that error message are closer to the fields they represent than they are to the next following fields.
- One way to fix the above problem is to wrap the label-input and the error elements in another div and tweak the styles using flexbox.
- Error messages should be in a differently styled font than labels and inputs (usually smaller, sometimes italics, usually red or some other vibrant color).

Submit Assignment

- Make sure you get rid of the JSON tree view box in the final submission!

In addition to these TODO's, the basic styling guidelines mentioned in previous assignments remain. Pay attention to alignment, proximity, contrast, and consistency. Style your buttons appropriately, put space around all elements, etc.

Also, look at the CSS in the style section of the starter code. Note the use of ">" in the selectors and the use of "em" instead of "px". Make sure you understand what these mean. Also, note that only two classes are used in the CSS due to the fact that we are leveraging scoped styles.

Enter Web URL

http://

Submit Assignment