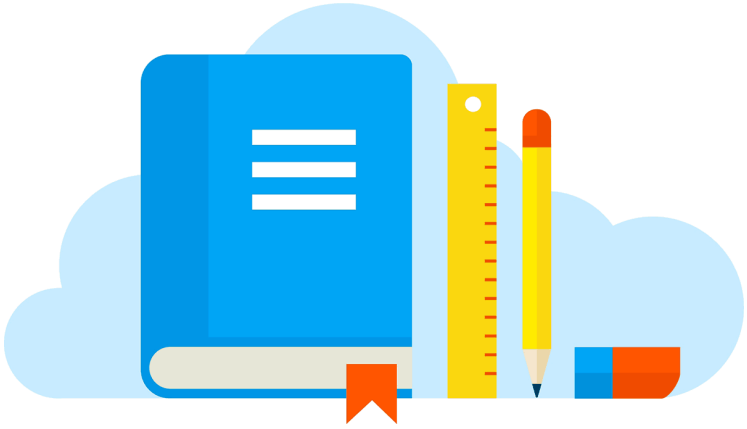


COMP1842 Lab Task

Week 3 - 26/09/2024

COS1104 Nguyen Dinh Hoang Son GCS220616

Introduction

Task 1

Lesson 1: The Vue Instance

Overview

In this lesson, I learned the fundamental concepts of Vue.js by building a simple product page. The key takeaway is how Vue allows us to display data dynamically on a webpage and how it provides reactivity to ensure the page updates when the data changes.

Challenge Solution

javascript

```
var app = new Vue({
  el: '#app',
  data: {
    product: 'Socks',
    image: './assets/images/socks.jpg',
    altText: 'A pair of socks',
```

```
    link: 'https://en.wikipedia.org/wiki/Sock'
  }
});
```

Code Line 9. A `link` property is added to the Vue instance's `data`, containing a URL.

html

```
<a v-bind:href="link">More details</a>
```

Code Line 10. Using `v-bind:href`, the `href` attribute of the anchor (`a`) tag is bound to the `link` data property. This makes the anchor's URL dynamic based on the data.

Conclusion

In this lesson, I learned how to:

- Use `v-bind` to dynamically bind data to HTML attributes like `src`, `alt`, and `href`.
- Apply the shorthand (`:`) for cleaner code.
- Make attributes responsive to data changes within the Vue instance.

By completing the challenge, I bound a dynamic link to an anchor tag, further demonstrating my understanding of attribute binding in Vue.js.

Lesson 3: Conditional Rendering

Overview

In this lesson, I learned how to conditionally display elements in Vue using directives like `v-if`, `v-else`, and `v-show`. This allows for dynamic changes in the UI based on the state of the application.

Challenge Solution

To complete the challenge, I added an `onSale` property to conditionally render a message when the product is on sale.

javascript

```
data: {
  product: 'Socks',
  inStock: true,
  onSale: true
}
```

Code Line 16. Adding the `onSale` property to the data object.

html

```
<span v-if="onSale">On Sale!</span>
```

Code Line 17. Using `v-if` to conditionally render a span that displays "On Sale!" if the `onSale` property is `true`.

Lesson 4: List Rendering

Overview

In this lesson, I learned how to use Vue's `v-for` directive to display lists of data on a webpage. This allows dynamic rendering of lists by looping through arrays or objects and displaying each element as HTML.

Challenge Solution

To meet the challenge, we can add an array of sizes and use `v-for` to render a list of available sizes.

javascript

```
var app = new Vue({
  el: '#app',
  data: {
    product: 'Socks',
    details: ['80% cotton', '20% polyester', 'Gender-neutral'],
    variants: [
      { variantId: 2234, color: 'green' },
      { variantId: 2235, color: 'blue' }
    ],
    sizes: ['S', 'M', 'L']
```

```
  }
});
```

Code Line 20: The array `sizes` is added to the data object, which contains the available sizes of the product.

```
html
<ul>
  <li v-for="size in sizes">{{ size }}</li>
</ul>
```

Code Line 21: This loop iterates over the `sizes` array and renders each size in a list item (`li`).

Conclusion

In this lesson, I learned how to use the `v-for` directive to render arrays and object arrays in Vue.js. This directive allows for looping through data, displaying each item dynamically on the page, and using dot notation to access specific properties in objects. I also learned the importance of using a unique key for each element when rendering lists.

Task 2

Lesson 5: Event Handling

Overview

In this lesson, I learned how to handle DOM events in Vue.js using the `v-on` directive. This directive allows us to listen for various user interactions and trigger corresponding methods, enhancing the interactivity of our web application.

Challenge Solution

To solve the challenge, I created a new button to decrement the cart value, along with a method to handle the event.

html

```
<button @click="removeFromCart">Remove from Cart</button>
```

Code Line 30: A button is added to trigger the `removeFromCart` method when clicked.

javascript

```
methods: {
  removeFromCart() {
    if (this.cart > 0) {
      this.cart -= 1;
    }
  }
}
```

Code Line 31: The `removeFromCart` method decreases the cart value by 1, ensuring that it doesn't go below 0.

Conclusion

In this lesson, I learned how to handle events in Vue.js using the `v-on` directive and its shorthand `@`. Events like `click` and `mouseover` can trigger methods, which can take arguments to perform more complex tasks. Additionally, I explored how `this` refers to the current Vue instance's data and how it can be used to dynamically update content.

Lesson 6: Class & Style Binding

Overview

In this lesson, I learned how to dynamically style HTML elements by binding Vue.js data to their class and style attributes. This allows for more interactive and responsive UI elements that reflect changes in the underlying data.

Challenge Solution: Binding a Class to the "Out of Stock" Text

To meet the challenge, we can dynamically bind a class to the "Out of Stock" text that adds a `line-through` style when the product is out of stock.

html

```
<p :class="{ outOfStockText: !inStock }">Out of Stock</p>
```

Code Line 36: The `outOfStockText` class is applied to the `<p>` tag when the product is out of stock, adding a `line-through` effect.

css

```
.outOfStockText {
  text-decoration: line-through;
}
```

Code Line 37: The `outOfStockText` class uses CSS to apply the `line-through` effect.

Conclusion

In this lesson, I learned how to use Vue.js for both style and class binding. This allows us to dynamically update the style and appearance of elements based on reactive data. We explored binding the `background-color` of elements using `v-bind:style` and conditionally applying CSS classes using `v-bind:class`. Additionally, I learned how to manage disabled states for buttons using both attribute and class bindings.

Lesson 7: Computed Properties

Overview

In this lesson, I learned how to use **computed properties** in Vue.js. These properties calculate values based on existing data and update automatically when the underlying data changes. This makes them ideal for combining or processing reactive data without duplicating logic.

Challenge Solution: `onSale` Computed Property

To meet the challenge, we add a boolean data property `onSale` and create a computed property that shows a message when the product is on sale.

javascript

```
computed: {
        title() {
            return this.brand + ' ' + this.product
```

```
        },
        image(){
            return this.variants[this.selectedVariant].variantImage
        },
        inStock(){
            return
this.variants[this.selectedVariant].variantQuantity
        },
        sale_message() {
          if (this.onSale) {
            return this.brand + ' ' + this.product + ' are on sale!'
          }
            return  this.brand + ' ' + this.product + ' are not on
sale'
        }
    }
```

Code Line 41: The computed property `sale` checks if `onSale` is true. If the product is on sale, it appends "is on Sale!.

html

```
<h1>{{ sale_message }}</h1>
```

Code Line 42: The `<h1>` tag now displays the `sale` computed property, which dynamically reflects whether the product is on sale.

Conclusion

In this lesson, I learned that computed properties in Vue.js calculate values based on reactive data and are automatically updated when their dependencies change. They are cached for efficiency and should be used for pure functions that don't mutate data. By leveraging computed properties, we can make the application cleaner and more scalable.

Lesson 8: Components

Overview

In this lesson, I learned how to create and use components in Vue.js. Components are reusable blocks of code with their own structure and functionality, making applications

more modular and easier to maintain. By breaking code into smaller, reusable parts, it becomes more flexible and manageable.

Challenge Solution: Product Component with Props

To meet the challenge, we created a `product-details` component and passed data to it via props. Here's how it's done:

javascript

Copy code

```
Vue.component('product-details', {
  props: {
    details: {
      type: Array,
      required: true
    }
  },
  template: `
    <ul>
      <li v-for="detail in details">{{ detail }}</li>
    </ul>
  `
})
```

In the parent component, we passed the data to the `product-details` component:

javascript

Copy code

```
<product-details :details="['80% cotton', '20% polyester',
'Gender-neutral']"></product-details>
```

Code Explanation

- **Line 1**: We define a new component, `product-details`, using `Vue.component()`.
- **Line 2-5**: The `props` object specifies that the `details` prop should be an array and is required.
- **Line 6-9**: We define the template for the component, which uses `v-for` to iterate over the `details` array and display each item in a `<li>` element.

Conclusion

In this lesson, I learned that Vue.js components allow us to break our application into manageable, reusable blocks of code. Props enable data sharing between parent and child components, making it possible to pass dynamic content between them. Components make our codebase more organized, scalable, and maintainable.

Lesson 9: Communicating Events

Overview

In this lesson, I learned how to communicate between components by emitting events in Vue.js. This allows child components to notify their parent components that something has happened, passing data along with the notification.

Challenge Solution: Emitting and Handling Events

To meet the challenge, we created a communication system where a child component emits an event when a button is clicked, and the parent listens for this event and updates its data accordingly. Here's how it was done:

In the Product Component

We added the `$emit` method inside the `addToCart` method to send an event named `add-to-cart` along with the product's `variantId`:

javascript

Copy code

```
methods: {
  addToCart() {
    this.$emit('add-to-cart',
this.variants[this.selectedVariant].variantId)
  }
```

```
}
```

In the Parent Component

The root instance listens for the `add-to-cart` event using the `@add-to-cart` directive and calls the `updateCart` method:

html

Copy code

```
<product :premium="premium" @add-to-cart="updateCart"></product>
```

The `updateCart` method in the parent then adds the product's `variantId` to the `cart` array:

javascript

Copy code

```
methods: {
  updateCart(id) {
    this.cart.push(id)
  }
}
```

Code Explanation

- **Line 3**: In the `addToCart` method, we emit the event `'add-to-cart'` and pass the `variantId` as an argument.
- **Line 9**: In the parent template, `@add-to-cart` listens for the event emitted by the product component and triggers the `updateCart` method.
- **Line 13**: The `updateCart` method updates the parent's `cart` array with the product's `variantId`.

Conclusion

In this lesson, I learned how to use the `$emit` method to send events from child components to their parent components in Vue.js. By handling these events, we can update the parent's state and create a dynamic, interactive application. This process is crucial for maintaining clean and modular code, where components can communicate without tightly coupling their logic.

Lesson 10: Forms & v-model

Overview

In this lesson, I learned how to work with forms in Vue.js using the `v-model` directive for two-way data binding, and how to perform custom form validation. The goal was to create a form that allows users to submit product reviews, with validation to ensure that all required fields are completed.

Challenge Solution: Product Review Form

To meet the challenge, I created a form that collects the user's name, review, and rating. Here's how the `product-review` component was structured:

javascript

Copy code

```
Vue.component('product-review', {
  template: `
    <form class="review-form" @submit.prevent="onSubmit">
      <p>
        <label for="name">Name:</label>
        <input id="name" v-model="name" placeholder="name" required>
      </p>

      <p>
        <label for="review">Review:</label>
        <textarea id="review" v-model="review" required></textarea>
      </p>

      <p>
        <label for="rating">Rating:</label>
```

```
      <select id="rating" v-model.number="rating" required>
        <option>5</option>
        <option>4</option>
        <option>3</option>
        <option>2</option>
        <option>1</option>
      </select>
    </p>

    <p>
      <input type="submit" value="Submit">
    </p>

    <p v-if="errors.length">
      <b>Please correct the following error(s):</b>
      <ul>
        <li v-for="error in errors">{{ error }}</li>
      </ul>
    </p>
  </form>
`,
data() {
  return {
    name: null,
    review: null,
    rating: null,
    errors: []
  }
},
methods: {
  onSubmit() {
    if (this.name && this.review && this.rating) {
      let productReview = {
        name: this.name,
        review: this.review,
        rating: this.rating
      }
      this.$emit('review-submitted', productReview)
      this.name = null
```

```
        this.review = null
        this.rating = null
      } else {
        if (!this.name) this.errors.push("Name required.")
        if (!this.review) this.errors.push("Review required.")
        if (!this.rating) this.errors.push("Rating required.")
      }
    }
  }
})
```

Code Explanation

- **v-model**: Used to create two-way binding between form inputs and the component's data.
- **.number modifier**: Ensures the rating is converted to a number.
- **@submit.prevent**: Prevents the form from reloading the page when submitted.
- **onSubmit method**: Checks if all fields are filled before submitting, and adds any errors to the `errors` array if fields are missing.

Custom Form Validation

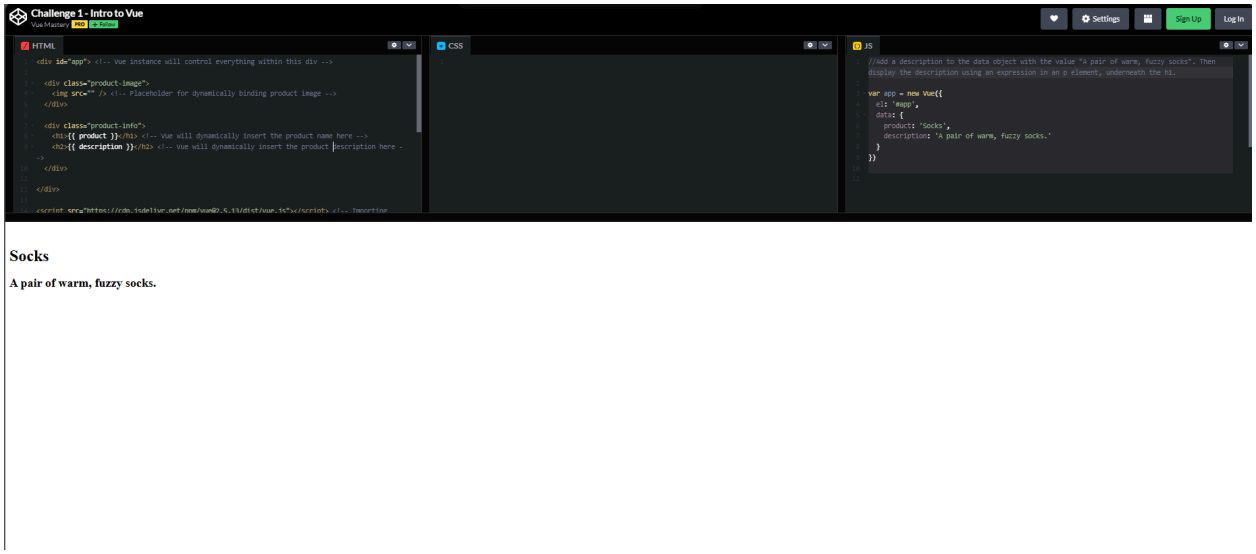
Custom validation was implemented to check if the name, review, and rating fields were filled out. If not, error messages were displayed using `v-if` and `v-for` to loop through and display any errors.

Conclusion

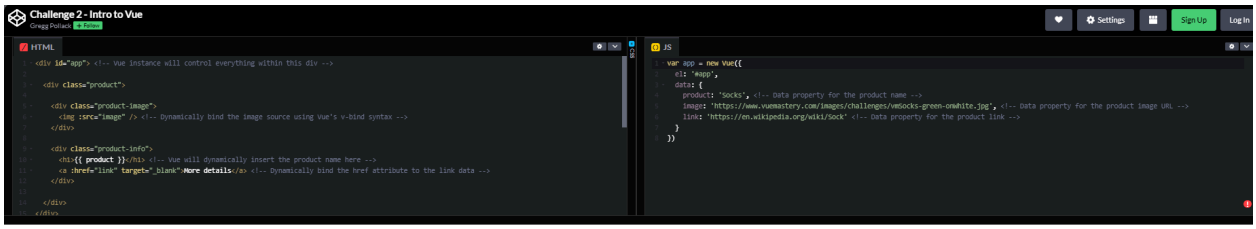
In this lesson, I learned how to use the `v-model` directive for two-way data binding in forms, create custom form validation, and emit events from child components to parent components. This makes form handling in Vue.js more dynamic and interactive.

Appendices

Lesson 1: Introduction to Vue.js



Lesson 2: Event Handling



Lesson 3: Conditional Rendering

Socks

In Stock

On Sale!

Lesson 4: Lists and Rendering with `v-for`

Socks

In Stock

- 80% cotton
- 20% polyester
- Gender-neutral

green

blue

- S
- M
- L

Lesson 5: Computed Properties

Socks

In Stock

- 80% cotton
- 20% polyester
- Gender-neutral

green

blue

Add to cart | Remove from cart

Cart(0)

Lesson 6: Class and Style Binding

Socks

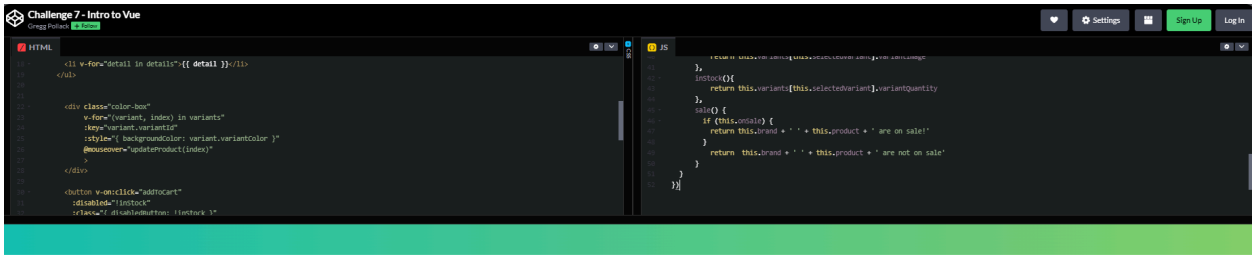
~~Out of Stock~~

- 80% cotton
- 20% polyester
- Gender-neutral

Add to cart

Cart(0)

Lesson 7: Computed Properties



Vue Mastery Socks

In Stock

Vue Mastery Socks are on sale!

- 80% cotton
- 20% polyester
- Gender-neutral

Add to cart

Cart(0)

Lesson 8: Components

Socks

In Stock

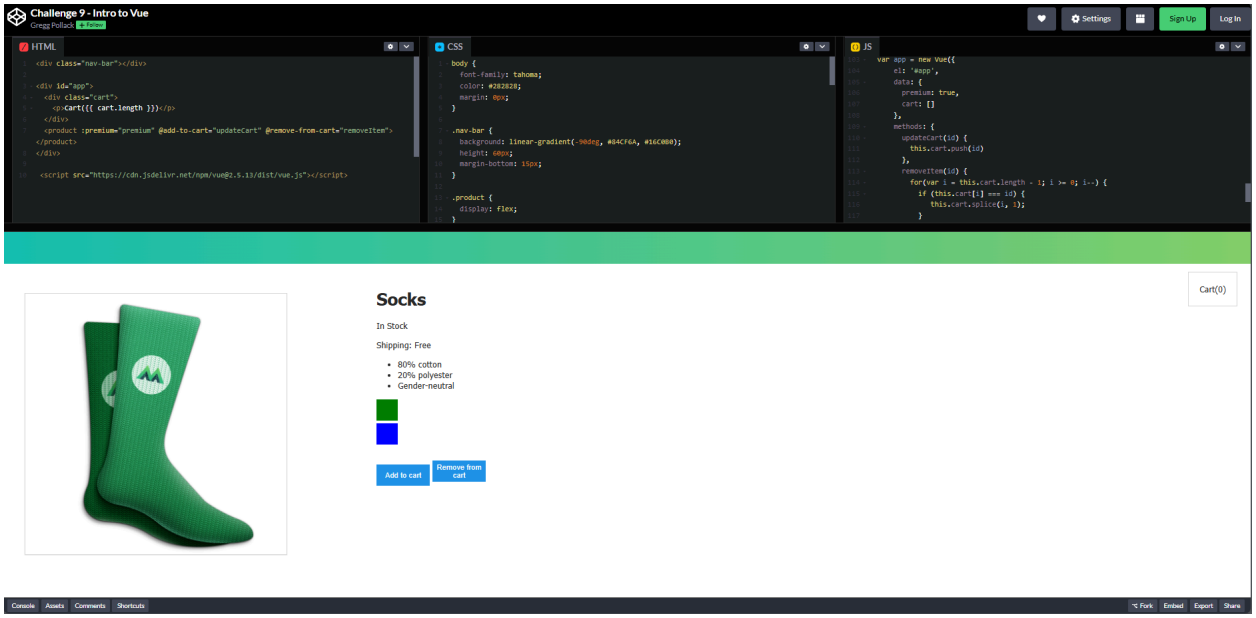
Shipping: Free

- 80% cotton
- 20% polyester
- Gender-neutral

Add to cart

Cart(0)

Lesson 9: Communicating Events



Lesson 10: Forms & v-model