Intro to Vue 3

Lessons

1. Intro to Vue 3

2:22

2. Creating the Vue App

6:56

3. Attribute Binding

3:53

4. Conditional Rendering

5:11

5. List Rendering

3:31

6. Event Handling

4:31

7. Class & Style Binding

6:37

8. Computed Properties

6:23

9. Components & Props

6:38

10. Communicating Events

3:57

11. Forms & v-model

8:27

Forms & v-model

In this lesson, we're going to look at the concept of attribute binding. If you're coding along with the repo, you can checkout the `L11-start` branch or the starting code on Codepen.

Our Goal

Create a form for users to add product reviews.

Introducing v-model

At the beginning of this course, we learned about `v-bind`, which creates a one-way binding, from the data to the template. When working with forms, however, this one way binding isn't enough. We also need to be binding from the template to the data.

For example, when a user inputs their name into an input field, we want to record and store that value in our data. The `v-model` directive helps us achieve this, creating two-way data binding.

To see this all in action, we're going to create a new `review-form` component.

The review-form component

We'll add a new **ReviewForm.js** file into our components folder, and scaffold out the component.

📄components/ReviewForm.js

```
app.component('review-form', {
  template:
  /*html*/
  `<form class="review-form">
    <h3>Leave a review</h3>
    <label for="name">Name:</label>
    <input id="name">

    <label for="review">Review:</label>
    <textarea id="review"></textarea>

    <label for="rating">Rating:</label>
    <select id="rating">
      <option>5</option>
      <option>4</option>
      <option>3</option>
      <option>2</option>
      <option>1</option>
    </select>

    <input class="button" type="submit" value="Submit">
  </form>`,
  data() {
    return {
      name: '',
      review: '',
      rating: null
    }
  }
})
```

Inside of our template, notice these elements:

- `<input id="name">`
- `<textarea id="review">`

- `<select id="rating">`

We want to bind these input fields to their respective data properties so that when the user fills out the form, we store their data locally.

```
  data() {
    return {
      name: '',
      review: '',
      rating: null
    }
  }
```

We'll achieve this by adding the `v-model` directive to each of those input elements.

📄components/ReviewForm.js

```
app.component('review-form', {
  template:
  /*html*/
  `<form class="review-form">
    <h3>Leave a review</h3>
    <label for="name">Name:</label>
    <input id="name" v-model="name">

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

    <label for="rating">Rating:</label>
    <select id="rating" v-model.number="rating">
      <option>5</option>
      <option>4</option>
      <option>3</option>
      <option>2</option>
      <option>1</option>
    </select>

    <input class="button" type="submit" value="Submit">
  </form>`,
  data() {
    return {
```

```
      name: '',
      review: '',
      rating: null
    }
})
```

Notice how on the `<select>` element, we used `v-model.number`, this is a modifier that typecasts the value as a number.

Submitting the Review Form

In order to submit this form, we'll add a listener at the top:

📄components/ReviewForm.js

```
app.component('review-form', {
  template:
  /*html*/
  `<form class="review-form"
@submit.prevent="onSubmit">
    ...
    <input class="button" type="submit" value="Submit">
  </form>`
  ...
})
```

We're using another modifier, `@submit.prevent="onSubmit"` to prevent the default behavior (a browser refresh). When this form is submitted, it will trigger the `onSubmit()` method, which we'll write now:

📄components/ReviewForm.js

```
...
data() {
  return {
    name: '',
    review: '',
    rating: null
```

```
    }
  },
  methods: {
    onSubmit() {
      let productReview = {
        name: this.name,
        review: this.review,
        rating: this.rating,
      }
      this.$emit('review-submitted', productReview)

      this.name = ''
      this.review = ''
      this.rating = null
    }
  }
...
```

That method will create a `productReview` object, containing the `name` , `review` and `rating` from our `data`. It will then `$emit` a `review-submitted` event, sending that `productReview` along as the payload.

Finally, we're clearing out the data fields.

Using the Review Form

Now that our review form is created, we can import it within **index.html**.

📄index.html

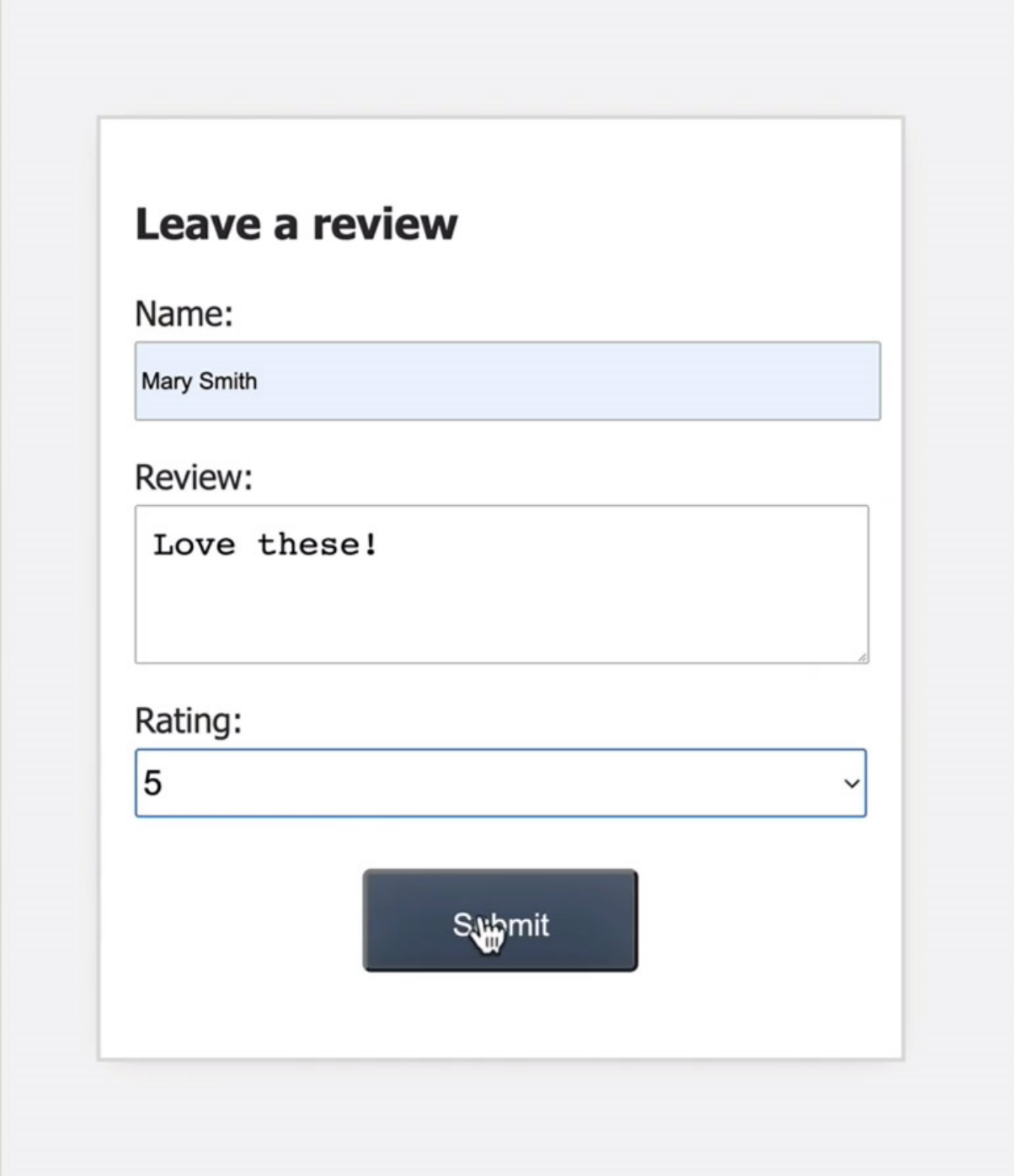
```
<!-- Import Components -->
...
<script src="./components/ReviewForm.js"></script>
...
```

Then we'll head into `product-display`, and actually use the component within its template, below the "product-container".

📄components/ProductDisplay.js

```
template:
  /*html*/
  `<div class="product-display">
    <div class="product-container">
      ...
    </div>
    <review-form></review-form>
  </div>`
})
```

Now in the browser, we can see the review form.



It looks like it's working… except when we click the submit button, we're emitting the event, but we haven't listened for it anywhere. Like we learned in the previous lesson, we need to listen for the `review-submitted` event in the parent scope (in `product-display`).

When the event is “heard”, we’ll add the `productReview` payload to the `product-display` component’s data.

Adding product reviews

We’ll add the event listener here on the `review-form`, where it’s being used:

📄components/ProductDisplay.js

```
template:
  /*html*/
  `<div class="product-display">
    <div class="product-container">
      ...
    </div>
    <review-form @review
submitted="addReview"></review-form>
  </div>`
})
```

When the event happens, we’ll trigger a new `addReview()` method. This will add product reviews to our `product-display` component, which means that component needs a new `reviews` array in its data.

📄components/ProductDisplay.js

```
...
data() {
  return {
    ...
    reviews: []
  }
}
...
```

Now let’s flesh out the `addReview()` method:

components/ProductDisplay.js

```
...
data() {
  return {
    ...
    reviews: []
   }
 },
methods: {
  ...
  addReview(review) {
    this.reviews.push(review)
  }
},
...
```

As you can see, it which takes in the `review` that we got from the `review-submitted` event payload, and pushes it into the `reviews` array.

Displaying the reviews

Now that we've implemented the ability to add reviews, we need to be displaying those reviews. Let's create a new component to do that. That component will be called `review-list`, which we'll scaffold out like this:

components/ReviewList.js

```
app.component('review-list', {
  props: {
    reviews: {
      type: Array,
      required: true
    }
  },
  template:
```

```
  /*html*/
  `
  <div class="review-container">
  <h3>Reviews:</h3>
    <ul>
      <li v-for="(review, index) in reviews"
:key="index">
        {{ review.name }} gave this {{ review.rating }}
stars
        <br/>
        "{{ review.review }}"
        <br/>
      </li>
    </ul>
  </div>
`
})
```

It will have a prop so that it can receive the `reviews` and print them out in the template using `v-for`, including the `index`, so that we can bind the `:key` attribute to it.

Now we can import this component inside **index.html**:

📄index.html

```
<!-- Import Components -->
...
<script src="./components/ReviewList.js"></script>
...
```

Then add it within `product-display`, right above the `review-form`:

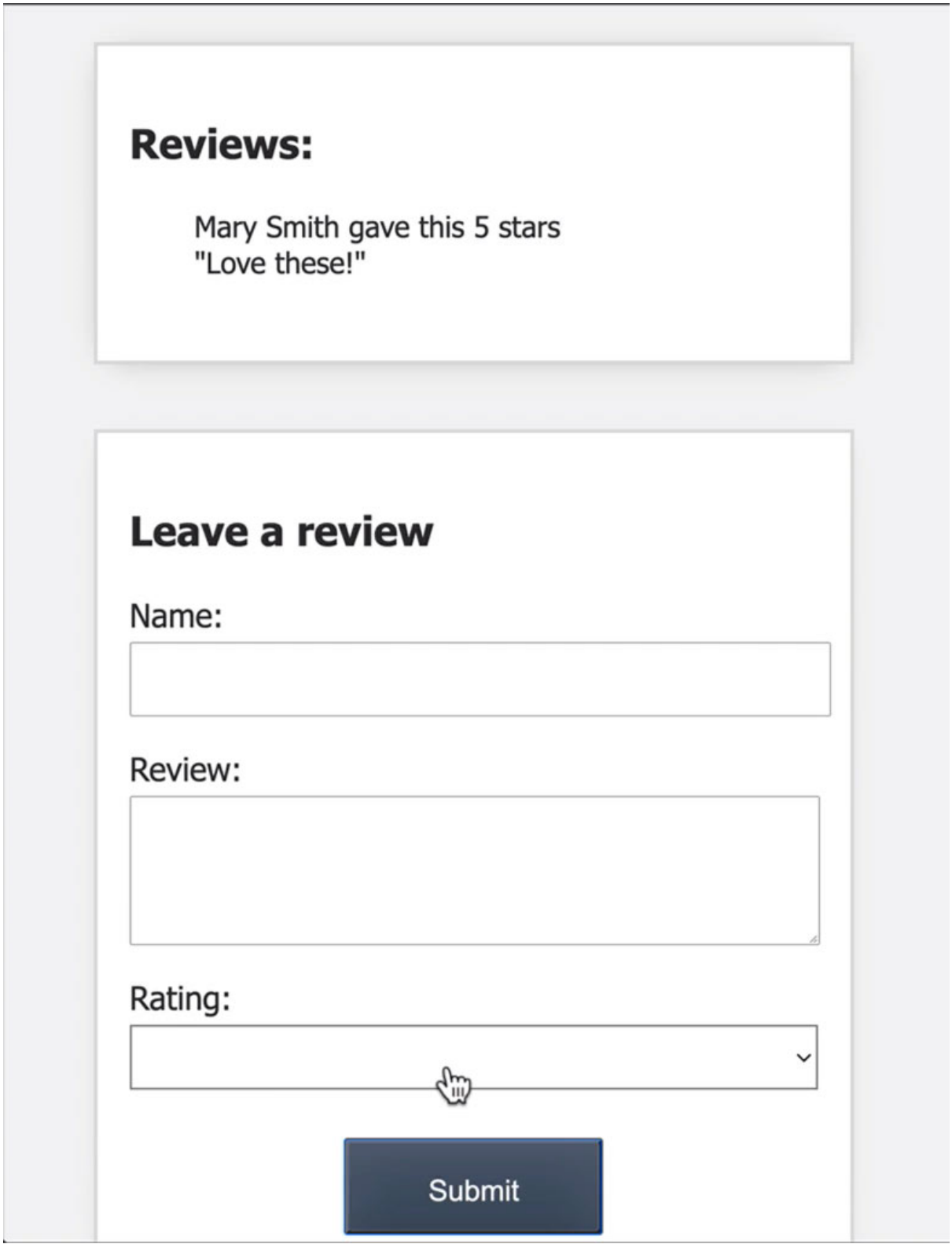
📄components/ProductDisplay.js

```
template:
  /*html*/
  `<div class="product-display">
    <div class="product-container">
      ...
    </div>
    <review-list :reviews="reviews"></review-list>
```

```
    <review-form @review
submitted="addReview"></review-form>
  </div>`
})
```

Notice how we've added `:reviews="reviews"` so we can pass the `reviews` that live on `product-display` into the `review-list`.

Checking this out in the browser, we'll add a new review, click submit, and see the review is being displayed.



So far so good, but when we refresh (and there are no reviews) we still see an empty box because the `review-list` component is still being rendered with no reviews to print out. Let's fix that,

and only render that component when we have `reviews` to display.

📄components/ProductDisplay.js

```
template:
  /*html*/
  `<div class="product-display">
    ...
    <review-list v-if="reviews.length"
:reviews="reviews"></review-list>
    ...
  </div>`
})
```

In other words, if the `reviews` array is empty, we will not show the `review-list` component.

With a refresh, it looks like it's working, and the component only shows up after we've added a review.

Basic Form Validation

To finish out this lesson, we're going to add some very basic validation to our `review-form`.

📄components/ReviewForm.js

```
methods: {
  onSubmit() {
    if (this.name === '' || this.review === '' ||
this.rating === null) {
      alert('Review is incomplete. Please fill out
every field.')
      return
    }
  ...
  }
```

```
}
```

Before we create a `productReview`, we'll check if `this.name` or `this.review` or `this.rating` are empty. If any of these things are the case, we will show an `alert`, which says: 'Review is incomplete. Please fill out every field.' And then `return` out of the method.

This is an overly simplified method of form validation. If you're interested in learning more production-level form validation practices, check out our lessons on Vuelidate in our Next-Level Vue course.

Congratulations!

You've made it to the end of the Intro to Vue 3 course and completed your first step on your path to Vue Mastery. To continue learning, I invite you to check out our library of Vue.js courses, which cover all the topics you'll need to succeed as a Vue developer.

If you're seeking a more guided and hands-on learning experience, we recommend checking out this Vue Training by our partner Jeffrey Biles, who will serve as your mentor through his 8-week training.

Coding Challenge

For your final coding challenge:

Add a question to the `review-form`: 'Would you recommend this product? '

Record and emit the response, and display it within `review-list`

As a reminder, if you're coding along with our repo, you can check out `L11-end` branch, and you can view the solution code on Codepen.

Download Video

Share Lesson

Lesson Resources

- Starting Code
- Ending Code

Discuss in our Facebook Group Send us Feedback

Previous LessonNext Lesson

Vue mastery

As the ultimate resource for Vue.js developers, Vue Mastery produces weekly lessons so you can learn what you need to succeed as a Vue.js Developer.

VUE MASTERY

-
-
-
-
-
-
-
-

ABOUT US

-
-
-
-