Vue mastery

# Intro to Vue 3

## Lessons

# Computed Properties

In this lesson, we're going to look at the concept of computed properties. If you're coding along with the repo, you can checkout the `L8-start` branch or the starting code on Codepen.

---

## Our Goal

Update both the variant image AND whether it's in stock or not, using computed properties.

---

## A Simple Computed Property

In the starting code, you'll notice we have a new data property:

**main.js**

```
data() {
  return {
    product: 'Socks',
    brand: 'Vue Mastery'
}
```

What if we wanted to combine the `brand` and the `product`, in our template? We could do that within an expression like so:

**index.html**

```
<h1>{{ brand + ' ' + product }}</h1>
```

If we checked this out in the browser, we'd see "Vue mastery Socks" displayed. But wouldn't it be neat if, instead of handling this logic in the inner HTML, our app had the ability to compute that value for us? For example, taking the `brand` and the `product`, adding them together, and returning that new value.

Computed properties are exactly like they sound: properties we can add to a Vue app that compute values for us. They help us keep computational logic out of the template and give us performance improvements that we'll cover soon. For now, let's turn this simple example into a computed property. We'll alter the `h1`'s expression like so:

**index.html**

```
<h1>{{ title }}</h1>
```

Now, `title` is the name of a computed property that we'll create now. First, we'll add the `computed` option to the app, just below our `methods`, then create the `title` property.

**main.js**

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

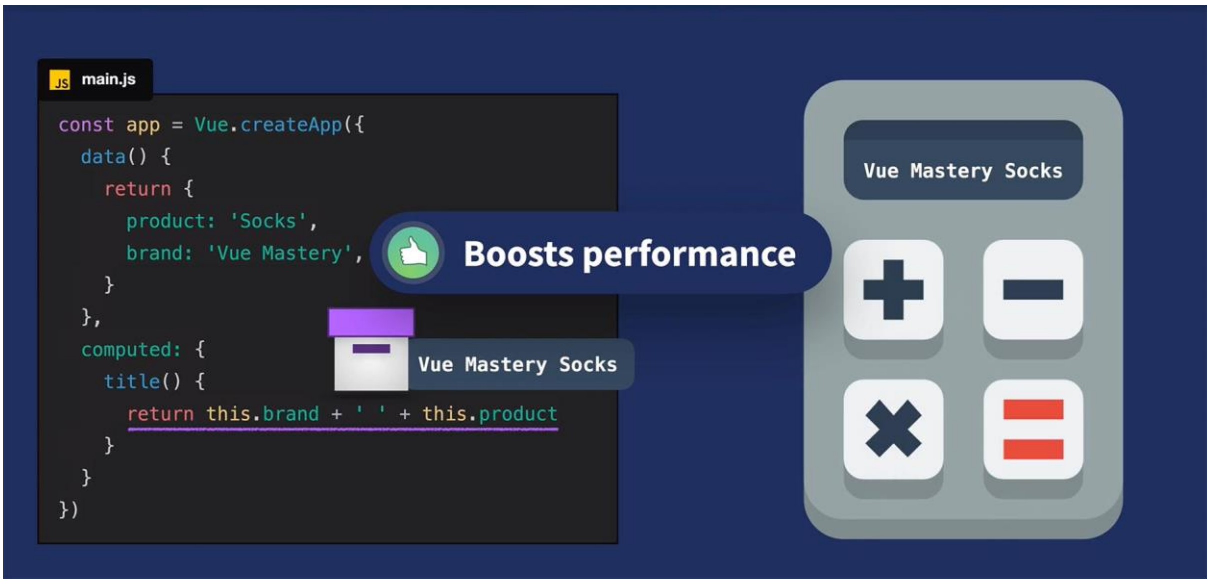
If we check out the browser, we'll still see "Vue Mastery Socks" displayed, except now we've abstracted that computational logic out of the template and contained it neatly on the options object.

But how exactly are computed properties working? Let's take a deeper look.

---

## Think of them like a Calculator

I like to think of computed properties kind of like a calculator, because they calculate or *compute* values for us. This computational calculator takes our values, `brand` and `product`, adds them together, and gives us the result.



main.js

```
const app = Vue.createApp({
  data() {
    return {
      product: 'Socks',
      brand: 'Vue Mastery',
    }
  },
  computed: {
    title() {
      return this.brand + ' ' + this.product
    }
  }
})
```

Like I mentioned earlier, computed properties provide us a performance improvement. This is because they cache the calculated value. The value (`'Vue Mastery Socks'`) gets stored away and only updates when it needs to, when one of its dependencies change. For example, if the `brand` were to change from `'Vue Mastery'` to `'Node Mastery'`, our computed property would receive that new `brand` dependency, then recalculate and return the new value: `'Node Mastery Socks'`

Now that we're starting to comprehend computed properties, let's implement a more practical example in our Vue app.

---

## Computing Image & Quantity

Heading back into our code, let's add a new `quantity` property to our variant objects.

📄**main.js**

```
data() {
  return {
    ...
    variants: [
      { id: 2234, color: 'green', image:
'./assets/images/socks_green.jpg', quantity: 50 },
      { id: 2235, color: 'blue', image:
'./assets/images/socks_blue.jpg', quantity: 0 },
    ]
  }
}
```

Notice how the green socks have a `quantity` of `50` while the blue socks have `0`. In other words, the green socks are in stock and the blue socks are out of stock. However, we're currently displaying "In stock" or "Out of stock" based on the `inStock` data value, which no longer reflects the truth about our product and its variant quantities. So we'll want to create a computed property we can use to display "In stock" or "Out of stock" based on these new quantities.

To get started, remember how we updated the variant image, based on which variant color is moused over? Instead of that mouseover event triggering the `updateImage()` method, we're going to have it trigger a new method called `updateVariant()`.

### 📄index.html

```
<div
  v-for="(variant, index) in variants"
  :key="variant.id"
  @mouseover="updateVariant(index)" <! -- new method --
>
  class="color-circle"
  :style="{ backgroundColor: variant.color }">
</div>
```

Notice how we're passing in the `index` of the currently hovered-on variant: `updateVariant(index)`. We got access to that `index` by adding it as a second parameter in our `v-for` directive:

`v-for="(variant, index) in variants"`

Why are we passing in the `index`? We're going to use it to tell our app which variant is currently hovered on, so it can use that information to trigger the updating of both the image AND whether that variant is in stock or not.

We'll add a new data property to our app, which will be updated to equal that `index`

### 📄main.js

```
data() {
  return {
    ...
    selectedVariant: 0,
    ...
  }
}
```

Our `updateVariant()` method will set the `selectedVariant`'s value equal to the `index` of the currently hovered-on variant.

**main.js**

```
updateVariant(index) {
  this.selectedVariant = index
}
```

Now, we've implemented a way for our app to know which product variant is being engaged with, and we're able to use that information to trigger the computing of which image to show and whether to show "In stock" or "Out of stock", based on which variant the user is moused over.

---

We're now ready to delete `image` and `inStock` from our data, and replace those with computed properties of the same names.

**main.js**

```
computed: {
  image() {
    return ??
  },
  inStock() {
    return ??
  }
}
```

So how do we grab the variant image and quantity? That will look like this:

📄**main.js**

```
image() {
  return this.variants[this.selectedVariant].image
}
```

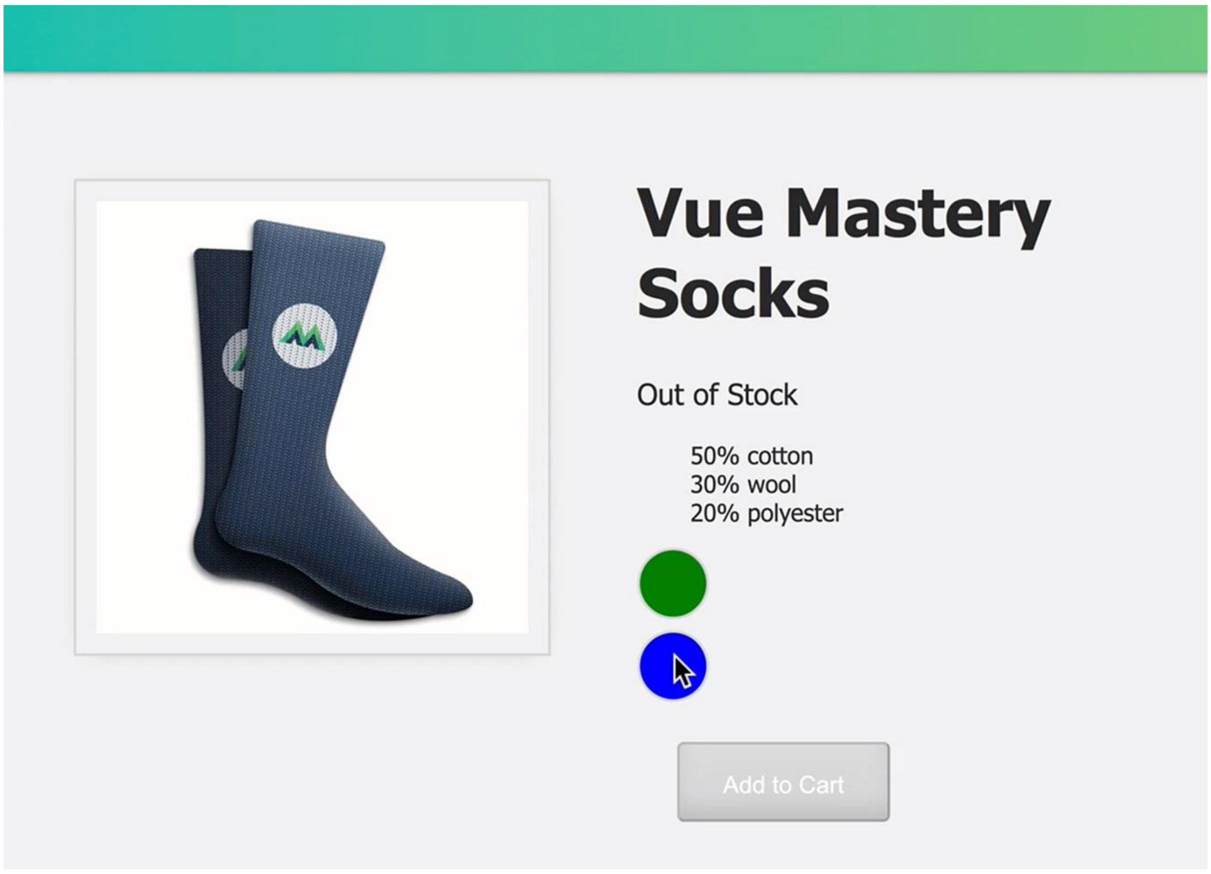
We're targeting the first or second element of our `variants` array based off the `selectedVariant`, which is either `0` or `1`, depending on which variant color circle is hovered on. Then we just use dot notation to grab the `image` off that variant.

The logic for computing `inStock` is nearly identical:

📄**main.js**

```
inStock() {
  return this.variants[this.selectedVariant].inStock
}
```

Checking this out in the browser, when we hover over the color circles, not only are we updating the variant image, but we're also displaying whether that variant is in stock or out of stock, using its quantity.



Notice how the button is still automatically updating for us, enabling and disabling. That's because, in our template, we're still using `inStock`.

**index.html**

```
<button
  class="button"
  :class="{ disabledButton: !inStock }"
  :disabled="!inStock"
  v-on:click="addToCart">
  Add to Cart
</button>
```

Now `inStock` is no longer a data property; it's the new computed property.

## Coding Challenge

We’ve reached the end of the lesson and we’re onto our challenge:

**Add an `onSale` boolean to the data.**

**Use a computed property to display the string: ’`brand` + ’ ’ `product` + ’ ’ is on sale’, whenever `onSale` is `true`.**

As a reminder, if you’re coding along with our repo, you can check out `L8-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 GroupSend 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

- 
- 
- 
-