SAMPLE CHAPTER

Vue.js IN ACTION

Erik Hanchett
with Benjamin Listwon
Foreword by Chris Fritz

MANNING

Vue.js in Action

by Erik Hanchett with Ben Listwon

Sample Chapter 9

brief contents

9 Extending Vue

This chapter covers

- Learning about mixins
- Understanding custom directives
- Using the render function
- Implementing JSX

In the previous chapter we discussed transitions and animations. In this chapter we'll look at different ways we can reuse code in Vue.js. This is important because it allows us to extend the functionality of our Vue.js applications and make them more robust.

We'll begin by looking at *mixins*. Mixins are a way to share information between components; functionality is essentially "mixed" into the component. They're objects that have the same methods and properties that you'd see in any Vue.js component. Next, we'll look at *custom directives*. Custom directives allow us to register our own directives, which we can use to create whatever functionality we want. Then we'll look at the `render` *function*. With the `render` function we can go beyond using normal templates and create our own using JavaScript. Last, we'll look at using the `render` function with JSX, which is an XML-like syntax for JavaScript.

Don't worry, I haven't forgotten about the pet store application. In the next chapter we'll revisit it with Vuex.

9.1 Reusing functionality with mixins

Mixins are a great tool for many projects. They allow us to take small pieces of functionality and share them between one or many components. As you write your Vue.js applications, you'll notice that your components will start to look alike. One important aspect in software design is a concept known as DRY (don't repeat yourself). If you notice that you're repeating the same code in multiple components, it's time to refactor that code into a mixin.

Let's imagine you have an app that needs to collect a phone number or email address from your customer. We'll design our app to have two different components. Each component will contain a form with an input and a button. When the user clicks the button, it triggers an alert box that displays the text that was entered into the input box. It will look like figure 9.1 when we're done.

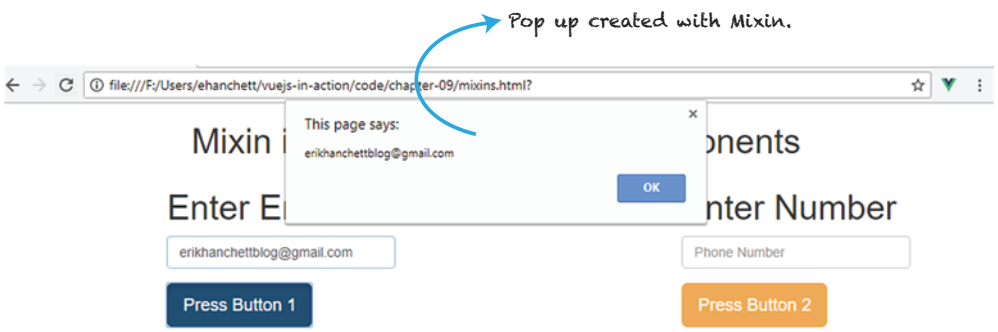

Figure 9.1 Mixin example with multiple components

This somewhat contrived example shows how we can extract logic as a mixin, in this case the logic that handles the button click and alert box. This keeps our code clean and avoids repeating code.

To get started with this example, create a file called mixins.html. To begin, we'll add our script tag for Vue.js and a link tag so we can add in Bootstrap for our styling. Then we'll add a basic HTML layout. The HTML will use Bootstrap's grid layout with one row and three columns. The first column will be set to `col-md-3` with an offset of `col-md-offset-2`. This column will display our first component. The next column will have a column size of `col-md-3`. The third column will have a column size of `col-md-3` and will show the last component.

Open your mixins.html file and enter the HTML code in the following listing. This is the first part of the code for this example. We'll add more code throughout

this section. If you'd like to see the completed code for this example, look for the mixins.html file included with the code for this book.

Listing 9.1 Adding our mixin HTML/CSS: chapter-09/mixin-html.html

```
<!DOCTYPE html>
    <script src="https://unpkg.com/vue"></script>
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css
➥ /bootstrap.min.css">
<html>
<head>
</head>
<body>
  <div id="app">
    <div id="container">
      <h1 class="text-center">{{title}}</h1>
      <div class="row">
        <div class="col-md-3 col-md-offset-2">
          <my-comp1 class="comp1"></my-comp1>
        </div>
        <div class="col-md-3">
            <h2 class="text-center">Or</h2>
        </div>
        <div class="col-md-3">
          <my-comp2 class="comp2"></my-comp2>
        </div> <!--end col-md-2-->
      </div><!-- end row -->
    </div> <!-- end container -->
  </div> <!-- end app -->
```

Adds Bootstrap CSS code to file

Lists the first component

Shows the Bootstrap grid system for columns

Lists the second component

Now that we've added HTML, we'll work on the Vue.js code. Open the mixins.html file and add an opening and closing script tag. It's worth mentioning for this example that we aren't using single-file components with Vue-CLI. If we were to do so, this would work the same way. The only difference is that each component and mixin would be in its own file.

Add a new Vue instance in between the opening and closing script tag. Inside the Vue instance we'll add a data object that returns a title. Because we're using components, we'll also need to declare both components that we're using in this example. Add the code in this listing to the mixins.html file.

Listing 9.2 Adding the Vue.js instance: chapter-09/mixins-vue.html

```
...
<script>
  new Vue({
    el: '#app',
    data() {
      return {
        title: 'Mixin in example using two components'
      }
```

Shows the root Vue.js instance declaration

Lists the data object that returns the title property

```
    },
    components:{
        myComp1: comp1,
        myComp2: comp2
    }

  });
</script>
```

Shows the declaration of components for myComp1 and myComp2

We have a few other things left to do. We need to add both of our components and our mixin. Each component needs to display text, show an input, and show a button. The button needs to take whatever input was entered and display an alert box.

Each component is similar in a few ways. Both have a title, both have an input box, and both have a button. They also behave the same way after clicking a button. At first it might seem like a good idea to create one component but the visual look and feel of each component is different. For example, each button is styled differently and the input boxes themselves accept different values. For this example, we'll leave them as separate components.

With that said, we still have similar logic outside the template. We need to create a mixin that handles a method called `pressed` that displays an alert box. Open the mixin.html file and add a new `const` called `myButton` above the Vue.js instance. Make sure to add in the `pressed` function, an alert, and a data object that returns an item, as seen in this listing.

Listing 9.3 Adding the mixin: chapter-09/my-mixin.html

```
<script>
const myButton = {
  methods: {
    pressed(val) {
      alert(val);
    }
  },
  data() {
      return {
          item: ''
      }
  }
}
...
```

Notes the myButton object mixin

Shows methods for the mixin

Notes the pressed function that shows alert box

Now that we have the mixin in place, we can go ahead and add our components. After the `myButton` object, add two new components called `comp1` and `comp2`. Each one will contain an `h1` tag, a form, and a button.

In `comp1`, our input will use a `v-model` directive to bind the input to a property called `item`. In our button, we'll use the `v-on` directive shorthand `@` symbol to bind the click event to the `pressed` method. Then we'll pass the item property into the method. The last thing we need to add to our `comp1` is to declare the mixin we created. We add the mixins array at the bottom, as you can see in listing 9.4.

For `comp2`, we'll add an `h1` tag with a form, an input, and a button. For this component, we'll use the `v-model` directive to bind the item property. The button will use the `v-on` directive shorthand `@` to bind the click event to the `pressed` method, as the same way we did in `comp1`. We'll pass the `item` property into the method. As with the other component, we'll need to define which mixins we want with this component by using the `mixins` property array at that bottom.

INFO Mixins aren't shared between components. Each component receives its own copy of the mixin. Variables inside the mixin aren't shared.

I won't go into detail, but we've also added basic Bootstrap classes to the form elements to style them.

Listing 9.4 Adding in the components: chapter-09/comp1comp2.html

```
...
const comp1 = {
    template: `<div>
    <h1>Enter Email</h1>
    <form>
      <div class="form-group">
        <input v-model="item"
        type="email"
        class="form-control"
        placeholder="Email Address"/>
      </div>
      <div class="form-group">
        <button class="btn btn-primary btn-lg"
     @click.prevent="pressed(item)">Press Button 1</button>
      </div>
    </form>
    </div>`,
    mixins: [myButton]
  }
  const comp2 = {
    template: `<div>
    <h1>Enter Number</h1>
      <form>
        <div class="form-group">
            <input v-model="item"
            class="form-control"
            placeholder="Phone Number"/>
        </div>
        <div class="form-group">
          <button class="btn btn-warning btn-lg"
     @click.prevent="pressed(item)">Press Button 2</button>
        </div>
      </form>
    </div>`,
    mixins:[myButton]
  }
...
```

Shows the component 1 declaration

Inputs v-model directive binding item

Shows the v-on directive with alias @ binds click event to pressed

Notes the declaration of mixin for component

The v-model directive binds the input to the item.

The v-on directive with alias @ binds the click event to pressed.

Shows the mixin declaration at the bottom

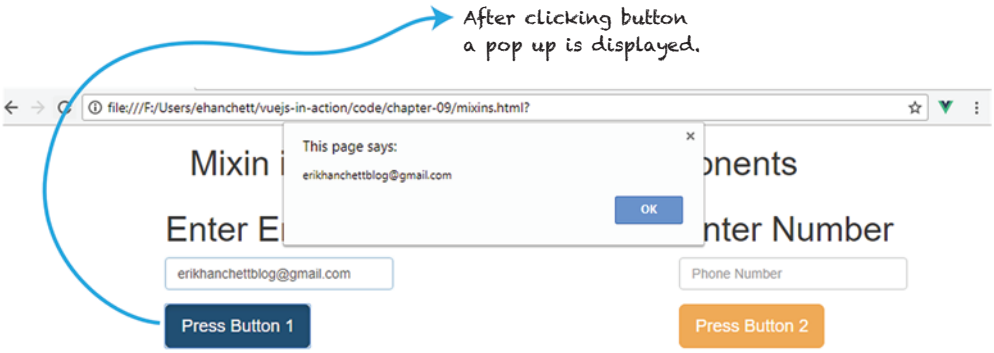

Figure 9.2 The image from figure 9.1 after clicking Press Button 1

Open a browser and load the mixins.html file we're working on. You should see figure 9.1. Go ahead and enter an email into the Enter Email box. Click the button and you should see a pop up, as seen in figure 9.2.

This works the same if we enter values into the Phone Number box.

9.1.1 Global mixins

Until now, we've used named mixins that we've declared inside each of our components. Another type of mixin, a *global mixin,* doesn't require any type of declaration. Global mixins affect every Vue instance created in the app.

You'll need to be cautious when using global mixins. If you're using any special third-party tools, they'll also be affected. Global mixins are good to use when you're trying to add custom options that need to be added to every Vue.js component and instance. Let's say you need to add authentication to your app and you want the authenticated user to be available in every Vue component in the application. Instead of registering the mixin in every component, you can create a global mixin.

Let's look at our app from the last section. We'll go ahead and change it to a global mixin. First, take a copy of the mixins.html file from the previous example and copy it to mixins-global.html. We'll refactor our application in this file.

Look for the `const myButton` line inside the script tags. This is our mixin; to use global mixins, we need to change this from a `const` to a `Vue.mixin`. The `Vue.mixin` tells Vue.js that this is a global mixin and that it must be injected into every instance. Delete the `const` line and add the `Vue.mixin({` line to the top. Next, close the parenthesis at the bottom as you can see in the following listing.

Listing 9.5 Global mixin: chapter-09/global-mixin.html

```
...
  Vue.mixin({                          <-- Shows the declaration of the global mixin
    methods: {
      pressed(val) {
```

```
        alert(val);
      }
    },
    data() {
        return {
            item: ''
        }
    }
});
...
```

Notes the closing parenthesis

Now that we have the global mixin declared, we can remove the declarations for `myButton` inside the components. Delete the `mixins: [myButton]` line from each component. That should do it—now you're using global mixins! If you load the browser with the newly created mixins-global.html file, it should behave and look the exact same as you saw previously.

TROUBLESHOOTING If you run into any problems, it might be because you left the mixins declaration at the bottom of the component definitions. Make sure to delete any reference to `myButton` in your app or you'll get an error.

9.2 Learning custom directives with examples

In the last eight chapters, we've looked at all sorts of directives, including `v-on`, `v-model`, and `v-text`. But what if we needed to create our own special directive? That's where *custom directives* come in. Custom directives give us low-level DOM access to plain elements. We can take any element on the page, add a directive, and give it new functionality.

Keep in mind that custom directives are different than components and mixins. All three, mixins, custom directives, and components, help promote code reuse but there are differences. Components are great for taking a large piece of functionality and separating it into smaller parts and making it available as one tag. Usually this consists of more than one HTML element and includes a template. Mixins are great at separating logic into smaller reusable chunks of code that can be shared in multiple components and instances. Custom directives are geared toward adding low-level DOM access to elements. Before using any of these three, take a minute to understand which one will be best for the problem you're trying to solve.

Two types of directives exist, local and global. Global directives can be accessed throughout the app at any place on any element. Typically, when you're creating directives you want them to be global, so you can use them everywhere.

Local directives can be used only in the component that registered that directive. This is nice to use when you have a specific custom directive that only needs to be used in one component. For example, you might create a specific select drop-down that works with only one component.

Before we look at each one, let's create a simple local custom directive that sets the color and font size, and adds a Bootstrap class name to an element. It should look like figure 9.3 when we're done.

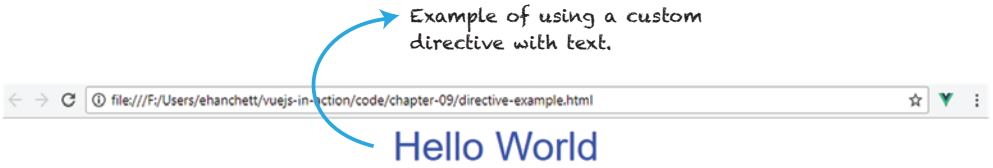

Figure 9.3 Hello World text added using a custom directive.

Open a new file and name it directive-example.html. Inside the new file, add simple HTML. The HTML should include the script tag for Vue and the stylesheet for the Bootstrap CDN. Inside our app, we'll create a new directive called `v-style-me`, as you can see in this listing. This directive will attach to the `p` tag.

Listing 9.6 Vue.js local custom directive: chapter-09/directive-html.html

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://unpkg.com/vue"></script>
    <link rel="stylesheet"
href=https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap
➥ .min.css>
</head>
<body>
  <div id='app'>
      <p v-style-me>
          {{welcome}}
      </p>
  </div>
```

Shows the Bootstrap added to the app

Lists the custom directive for style-me

All custom directives start with a `v-*`. Now that we have our custom directive in place on our `p` tag, we can add the Vue logic to the app.

Create a Vue instance and a data function. This function will return a welcome message. Next, we'll need to add a directives object. This will register a local custom directive. Inside that directives object we can create our directives.

Create a directive called `styleMe`. Each directive has access to a number of arguments that it can use:

- *el*—The element the directive is bound to.
- *binding*—An object containing several properties, including `name`, `value`, `oldValue`, and `expression`. (See the custom-directive guide for the full list at http://mng.bz/4NNI.)
- *vnode*—The virtual node produced by Vue's compiler.
- *oldVnode*—The previous virtual node.

For our example we'll use only `el`, for the element. This is always the first argument in the list. Keep in mind that our `styleMe` element is in camelCase. Because it was declared in camelCase, it must be in kebab case (`v-style-me`) in the template.

All custom directives must specify a hook. Much like the lifecycle and animation hooks we looked at in earlier chapters, the custom directive also has many similar hooks. These hooks are called at various times of the custom directive's life cycle:

- *bind*—This hook is called only once, when the directive is bound to the element. This is a good place to do setup work.
- *inserted*—This is called when the bound element has been inserted into the parent node.
- *update*—This is called after the containing component VNode has updated.
- *componentUpdate*—This is called after the containing component's VNode and the children of the VNodes have updated.
- *unbind*—This is called when the directive is unbound from the element.

You may be wondering what a VNode is. In Vue.js the *VNode* is part of the virtual DOM that Vue creates when the application is started. VNode is short for virtual node and is used in the virtual tree that's created when Vue.js interacts with the DOM.

For our simple example, we'll use the bind hook. This is fired as soon as the directive is bound to the element. The bind hook is a good place to do setup work to style the element. Using JavaScript, we'll use the `style` and `className` methods of the element. First add the color blue, then the `fontSize 42px`, and finally the `className` text-center inside the bind hook.

Go ahead and update the directive-example.html file. Your code should match the following listing.

Listing 9.7 Local directive Vue instance: chapter-09/directive-vue.html

```
<script>
  new Vue({
  el: '#app',
  data() {
    return {
      welcome: 'Hello World'
    }
  },
  directives: {
    styleMe(el, binding, vnode, oldVnode) {
      bind: {
        el.style.color = "blue";
        el.style.fontSize= "42px";
        el.className="text-center";
      }
    }
  }
  });
</script>
```

The data function returns the welcome property.

This is where directives are registered.

Shows the name of the local custom directive, with arguments

Notes the bind hook

```
</body>
</html>
```

Load up the browser and you should see the Hello World message. Now that we have this custom directive, we can use it on any element. Create a new `div` element and add the `v-style-me` directive. You'll notice that after you refresh the browser the text is centered, the font size is changed, and the color is blue.

9.2.1 Global custom directives with modifiers, values, and args

Now that we have a local directive, let's see what it looks like using a global directive. We'll convert our simple example, then we'll look at the binding argument. With the binding argument, we'll add a couple of new features to our custom directive. Let's give to the directive the ability to pass in the color of the text. In addition, we'll add a modifier so we can choose the size of our text, and we'll pass an arg for the class name. When it's all done it will look like figure 9.4.

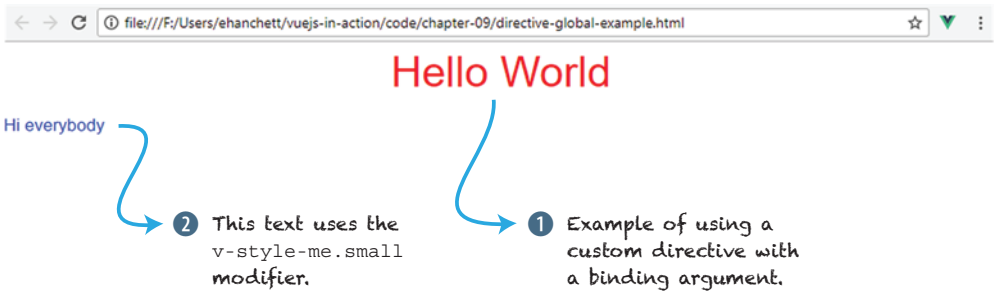

Figure 9.4 Using global directives with the binding argument

Copy the last example directive-example.html to directive-global-example.html. The first thing we need to do is to remove the directives object from the Vue.js instance. Go into our newly created directive-global-example.html file and remove the directives object below the data object.

Next, we'll need to create a new `Vue.directive`. This will tell Vue.js that we're creating a global directive. The first argument is the name of the directive. Go ahead and name it `style-me`. Then we'll assign the name of the hook. We'll use the bind hook the same way we did in the last example.

Inside the bind hook we'll have two arguments, `el` and `binding`. The first argument is the element itself. As we did in the previous example, we can use the `el` argument to manipulate the element the directive is attached to by changing its `fontSize`, `className`, and `color`. The second argument is called `binding`. This object has several properties; let's take a look at `binding.modifiers`, `binding.value`, and `binding.arg`.

The easiest binding property to work with is `binding.value`. When we add our new custom directive to an element, we can specify a value with it. For example, we could bind `'red'` to `binding.value`, as follows:

```
v-style-me="'red'"
```

We can also use object literals to pass in multiple values:

```
v-style-me="{ color: 'orange', text: 'Hi there' }"
```

We could then access each value using `binding.value.color` and `binding.value.text`. In listing 9.8, you can see that we set the element `el.style.color` to `binding.value`. If `binding.value` does not exist, it defaults to blue.

The `binding.modifiers` are accessed by adding a dot to the end of the custom directive:

```
v-style-me.small
v-style-me.large
```

When we access `binding.modifers.large`, it will return `true` or `false`, depending if the custom directive was declared when attached to the element. In listing 9.8, you can see that we check if `binding.modifiers.large` is `true`. If so, we set the font size to `42px`. Else if `binding.modifiers.small` is `true`, the font size is set to `17px`. If neither one of these modifiers is present, the font size isn't changed.

The last binding property we'll look at is `binding.arg`, declared in the custom directive with a colon and then the name. In this example, `text-center` is the argument:

```
v-style-me:text-center
```

With all that said, you can chain `modifiers`, `args`, and `values` together. We can combine all three. The `binding.arg` is `'red'`, the `binding.modifier` is set to `large`, and the `binding.value` is `text-center`.

```
v-style-me:text-center.large="'red'".
```

After adding the global custom directive make sure to go back into the HTML and add the second custom directive with text that displays `Hi everybody`. In this text, we'll use the binding modifier small on it, as shown in the following listing.

Listing 9.8 Completed Vue global directive: chapter-09/directive-global-example.html

```
<!DOCTYPE html>
    <script src="https://unpkg.com/vue"></script>
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap
➥ .min.css">                                          <-- Shows the Bootstrap CSS added to the app
<html>
<head>
```

```
</head>
<body>
  <div id='app'>
      <p v-style-me:text-center.large="'red'">
          {{welcome}}
      </p>
      <div v-style-me.small>Hi everybody</div>
  </div>
<script>
  Vue.directive('style-me', {
    bind(el, binding) {
        el.style.color = binding.value || "blue";

        if(binding.modifiers.large)
          el.style.fontSize= "42px";
        else if(binding.modifiers.small)
          el.style.fontSize="17px"

        el.className=binding.arg;
      }
  });

new Vue({
  el: '#app',
  data() {
    return {
      welcome: 'Hello World'
    }
  }
});
</script>
</body>
</html>
```

Notes the custom directive declaration with values, arg, and modifier

Lists the second custom directive with only a modifier

The global custom directive declaration uses the bind hook.

The element el.style.color is set to the binding.value or blue.

binding.modifiers are checked here to change the font size.

binding.arg is set to the class name on the element.

After loading up the browser, you should see the Hello World message. Notice how the second text, Hi Everybody, isn't centered and is smaller to the left of the screen. We got this because we used the `v-style-me` directive with only the small modifier. In that case, it changed the font size; however, it left the default color blue.

If you look closely at the source, you'll notice that a class of "undefined" was added to the second text `div`, because we assigned the `binding.arg` to `el.className` in the custom directive. However, because we didn't declare it, it's undefined by default. Beware this could happen: it would probably be a good idea to do a check on `binding.arg` before we set it to `el.className`. I'll leave that for you to do.

9.3 Render functions and JSX

Until now we've written all Vue.js applications using templates. This will work most of the time; however, there may be instances where you need to have the full power of JavaScript. For those cases, we can define our own render function instead of using a template. The render function will operate similarly to templates. It will output HTML, but you must write it in JavaScript.

JSX is an XML-like syntax that can be converted to JavaScript using a plugin. It's a way we can define our HTML inside JavaScript, like the `render` function. It's more commonly used with React, another frontend framework. We can use the full power of JSX with Vue.js, with the help of a Babel plugin.

Using JSX isn't the same as using the `render` function. To use JSX, you need to install a special plugin. But the `render` function works without any special setup inside your Vue.js instances.

In my experience, using the `render` function in Vue.js to create complex HTML is difficult. Common directives such as `v-for`, `v-if`, and `v-model` aren't available. You have alternatives for these directives, but you'll have to write extra JavaScript. But JSX is a strong suitable alternative. The JSX community is large and the syntax is much closer to templates, and you still get the benefit and power of JavaScript. The Babel plugin for Vue.js and JSX is well-supported and maintained, which is also nice. For those reasons, I'll only give a simple overview of the `render` function before we move on to JSX.

TIP If you'd like to learn about the `render` function in more detail, check out the official guides at https://vuejs.org/v2/guide/render-function.html.

9.3.1 Render function example

Let's create a simple example using `render`. Let's imagine we have a global Vue.js component that has a property named welcome. We want it to display the welcome message with an HTML header. We'll use a prop called header to pass in which header to use, `h1`, `h2`, `h3`, `h4`, or `h5`. In addition, we'll add a click event to the message so it shows an alert box. To make this good looking, we'll use Bootstrap's classes. Let's make sure our `h1` tag also has a class of `text-center` on it. When it's all done it will look like figure 9.5.

Figure 9.5 An example of using the render function

Create a file called render-basic.html. In this file, we'll create our small app. Before we create our component, let's create the HTML. Inside the HTML, add your script for Vue and your link to the Bootstrap CDN.

Inside the body, include a `div` with ID of `app` and a new component called `my-comp`. The `div` is optional; we could assign the ID of app directly to the `my-comp`

component. For the sake of clarity, we'll leave the `div` in as is. Inside that component, it will have a prop called `header` that will be assigned to 1. Inside the opening and closing brackets, we'll put in a name: Erik. Remember from a previous chapter anything in between the brackets of a component can be referenced with a slot. We can access slots using the `render` function, as we'll see next. Copy and paste the code from the following listing into the render-basic.html file.

Listing 9.9 Render basics html: chapter-09/render-html.html

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://unpkg.com/vue"></script>
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css
">
</head>
<body>
    <div id="app">
        <my-comp header="1">Erik</my-comp>
    </div>
<script>
```

Vue.js script is added.

Bootstrap stylesheet is added.

Component is added with header prop.

Now that we have the HTML in place, we must add a root Vue.js instance before we can add a global component. At the bottom of the page inside the script tags, add `new Vue({el: '#app'})`.

After we add our root Vue.js instance, let's create our global component. Make sure to have the root Vue.js instance after the global component or you'll get an error. The first thing we'll need inside our component is a data function that returns welcome. We'll make it say, "Hello World." We also need to declare our prop header.

Instead of declaring a template property on the component, we'll declare a `render` function. Inside that function it will have one argument called `createElement`. (You may sometimes see this as `h`; it's the same thing.) Inside the `render` function, you must return `createElement` with the proper arguments.

When you return `createElement`, you're building a virtual DOM by describing the elements and child elements you'd like to define in your HTML. This is also known as a virtual node (VNode). You are essentially creating a tree of VNodes that represents the virtual DOM.

As you can see from listing 9.10, the `createElement` accepts three arguments. The first is a string, object, or function. This is usually where you put in the DOM element, such as a `div`. The second argument is an object. This represents the attributes you want included on the element. The third is an array or string. If it's a string, it represents the text that will be included inside the tag. But if it's an array, it usually represents the child VNodes. Each VNode is like adding another `createElement`. It has the same arguments. For this example, we'll use only a string.

Inside our `Vue.component` add the `render(createElement)` function. Inside the function, return `createElement`, as you see in listing 9.10. The first argument we want is the `header` tag. We need to use the prop passed in to form the header value. In this case we sent in 1, which we need to create an `h1`. We can concatenate the letter `h` with our prop `this.header`.

The next argument is the attribute object. Because we're using Bootstrap, we want to use the `text-center` class to align the text in the middle of the screen. To do that, we create an object and have the first property be `class`, with the value being `text-center`. In JavaScript, `class` is a defined keyword, so it must be in quotes. Our next attribute is our event. Event handlers are represented by the keyword `on` when using the `render` function. In this example, we use the click event and display an alert box that shows `clicked`.

The last argument is the array or string. This will define what we see inside our `header` tag. To make things more interesting, we'll combine the welcome message that we defined in the data function and the text inside the opening and closing brackets of the component, `Erik`. To get this text, we can use `this.$slots.default[0].text`. This will get the text in the brackets as if it was the default text in a slot. We can now concatenate that with the welcome message. Copy this code into the render-basic.html. It should be all you need.

Listing 9.10 Adding the `render` function: chapter-09/render-js.html

```
Vue.component('my-comp',{
  render(createElement) {
      return createElement('h'+this.header,
      {'class':'text-center',
        on: {
          click(e){
            alert('Clicked');
          }
        }
      },
      this.welcome + this.$slots.default[0].text  )
  },
  data() {
         return {
      welcome: 'Hello World '
  }
},
  props: ['header']
});
new Vue({el: '#app'})
</script>
</body>
</html>
```

Notes the global component called my-comp

The render function is paired with the createElement argument.

Returns the createElement argument

The attribute object with the class and click event defined

Notes the string that will show in the header element

Lists the root Vue.js instance that must be added

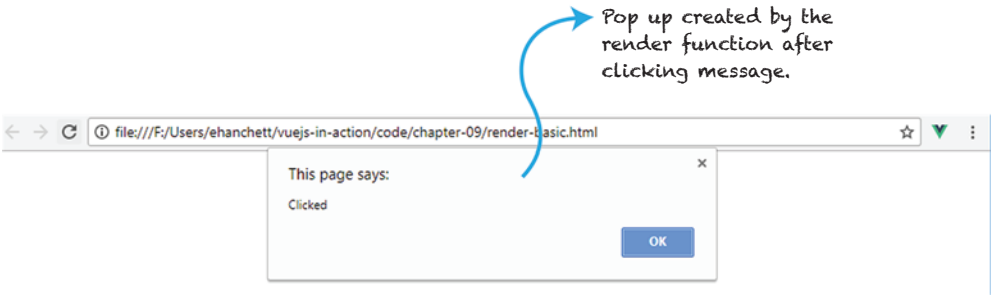

Figure 9.6 The alert box created using the render function after clicking the message.

Load the render-basic.html file and look at the output, it should show the Hello World Erik message. Change the prop header to a different value. You should see the text get smaller as you increase the number. Click the message and you should see figure 9.6.

9.3.2 JSX example

With JSX, you can create HTML similarly to templates and still have the full power of JavaScript. To do this, we'll create an app using Vue-CLI. Our goal is to re-create the last example but use JSX instead. This example should accept a property, a class, and show an alert box when the message is clicked.

Before we start, make sure you have Vue-CLI installed: the instructions are in appendix A. Open a terminal window and create an app called jsx-example. You'll be asked to respond to a few prompts: enter no for all of them.

Listing 9.11 Terminal commands to create jsx project

```
$ vue init webpack jsx-example

? Project name jsx-example
? Project description A Vue.js project
? Author Erik Hanchett <erikhanchettblog@gmail.com>
? Vue build standalone
? Install vue-router? No
? Use ESLint to lint your code? No
? Setup unit tests No
? Setup e2e tests with Nightwatch? No

   vue-cli · Generated "jsx-example".

   To get started:

     cd jsx-example
     npm install
```

Next, change to the jsx-example directory and run `npm install`. This will install all the dependencies. Now we'll need to install the Babel plugin for JSX. If you have any

problems with the installation of the plugin, please check the official GitHub page at https://github.com/vuejs/babel-plugin-transform-vue-jsx. The example I'm about to introduce to you only touches on the basics of JSX. I highly recommend reading the official documentation on the GitHub website on all the options available. Run `npm install` for all the recommended libraries, as shown here.

Listing 9.12 Terminal commands to install plugin

```
$ npm install\
  babel-plugin-syntax-jsx\
  babel-plugin-transform-vue-jsx\
  babel-helper-vue-jsx-merge-props\
  babel-preset-env\
  --save-dev
```

After you have the plugin installed, you'll need to update the .babelrc file. This is in the root folder. Inside it, you'll see many presets and plugins. Add `"env"` to the list of presets and `"transform-vue-jsx"` to the list of plugins, as in this listing.

Listing 9.13 Update .babelrc: chapter-09/jsx-example/.babelrc

```
{
  "presets": [
    ["env", {
      "modules": false
    }],
    "stage-2",
    "env"
  ],
  "plugins": ["transform-runtime", "transform-vue-jsx"],
  "env": {
    "test": {
      "presets": ["env", "stage-2"]     }
  }
}
```

Adds env to the list of presets

Adds transform-vue-jsx to the list of plugins

Now that we've set up JSX, we can start coding. By default, Vue-CLI creates a HelloWorld.vue file. We'll use that file, but we'll need to make several modifications. Go into the src/App.vue file and update the template. Remove the `img` node and add the `HelloWorld` component with two props. The first prop is our header, which will determine the header tag level. The second prop will be called name. In listing 9.10, we used slots instead of a named prop. In the following listing, we'll make a slight modification and pass the name in as a property instead. The result will be the same. Update your src/App.vue file so it matches this.

Listing 9.14 Update App.vue: chapter-09/jsx-example/src/App.vue

```
<template>
  <div id="app">
```

```
    <HelloWorld header="1" name="Erik"></HelloWorld>
  </div>
</template>

<script>
import HelloWorld from './components/HelloWorld'

export default {
  name: 'app',
  components: {
    HelloWorld
  }
}
</script>

<style>
</style>
```

The Hello World component with two props, name and header

Because we're using Bootstrap, we need to include it somewhere. Find the index.html file in the root folder. Add the link to the Bootstrap CDN, as shown here.

Listing 9.15 Update the index.html file: chapter-09/jsx-example/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>jsx-example</title>
    <link rel="stylesheet"
 href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap
 ➥ .min.css">
  </head>
  <body>
    <div id="app"></div>
  </body>
</html>
```

Adds Bootstrap CDN

Open the src/components/HelloWorld.vue file. Delete the top template—because we'll be using JSX, we no longer need it. Inside the `export default`, let's set up the data object first. The data object will return welcome, as well as a message. The `msg` property will construct the HTML for the header and it will concatenate the message we want to show on screen using ES6 template literals.

Add a `methods` object as well. This will have a `pressed` method that will trigger an alert box that shows `clicked`. Last, add a props array at the bottom, for header and name.

The `render` function for JSX is similar to the `render` function we used in our first example. By convention, instead of using `createElement`, we use the letter `h`. Then we return the JSX we need.

As you can see in listing 9.16, the `render` function returns several tags. The first is a `div` tag that surrounds all the JSX. Then we have a `div` with a class that equals

`text-center`. We then add an on-click event handler that is assigned to `this.pressed`. In normal Vue.js, data binding is done with text interpolation using the Mustache syntax (double braces). In JSX, we use only a single brace.

The last thing we add is a special property called `domPropsInnerHTML`. This is a special option added by the `babel-plugin-transform-vue` plugin. If you're familiar with React, it's similar to the `dangerouslySetInnerHTML` option. It takes `this.msg` and interprets it as HTML. Be aware that taking user input and converting it to HTML may lead to cross-site-scripting attacks, so be cautious whenever you use `domPropsInnerHTML`. Copy the text from the following listing if you haven't already and save it in your project.

Listing 9.16 Update HelloWorld.vue: chapter-09/jsx-example/HelloWorld.vue

```
<script>
export default {
  name: 'HelloWorld',
  render(h) {
    return (
      <div>
        <div class="text-center"
        on-click={this.pressed}
        domPropsInnerHTML={this.msg}></div>
      </div>
    )
  },
  data () {
    return {
      welcome: 'Hello World ',
      msg: `<h${this.header}>Hello World ${this.name}
            </h${this.header}>`,
    }
  },
  methods: {
    pressed() {
      alert('Clicked')
    }

  },
  props:['header', 'name']
}
</script>

<style scoped>
</style>
```

A JSX render function

Notes a div tag surrounding the JSX

Notes a div tag with the class attribute

The on click event handler that creates pop up box

The domPropsInnerHTML adds this.msg to div.

The Hello World with name added message

This is the method that shows the alert box.

Shows the two props for header and name

Save the file and start your server. Run `npm run dev` (or `npm run serve` if you're on vue-cli 3.0) on the command line, and you can open the webpage at http://localhost:8080. It should show the `Hello World Erik` message. Try to change several of the values passed into the `HelloWorld` component inside App.vue. By changing the header, you can change the level of header tag that's displayed.

Exercise

Use your knowledge from this chapter to answer this question:

What is a mixin and when should you use it?

See the solution in appendix B.

Summary

- You can share code snippets between multiple components using mixins.
- You can use custom directives to change the behavior of individual elements.
- You can use modifiers, values, and args to pass information into custom directives to create dynamic elements on the page.
- The render function gives you the full power of JavaScript inside your HTML.
- JSX can be used in your Vue.js application as an alternative to using the `render` function and still allow you to use the full power of JavaScript inside your HTML.

JAVASCRIPT/WEB DEVELOPMENT

Vue.js IN ACTION

Hanchett • Listwon

Vue.js is a lightweight frontend framework, offering easy two-way data binding, a reactive UI, and a common-sense project structure. It uses UI patterns and modern HTML to deliver impossibly fast page loads and silky smooth transitions—all from a tiny code footprint. It's a delight to develop in Vue using ordinary JavaScript and its integrated Vuex state management tool.

Vue.js in Action is your guide to building modern web apps. You'll start by exploring the reactive UI model while you get comfortable with Vue's unique features. Then, you'll go deeper as you build a shopping cart with an admin interface and the ability to manage stock! Finally, you'll extend your app, adding transitions, tests, and other key features until it's production ready.

What's Inside

- Clearly annotated code and illustrations
- Modeling data and consuming APIs
- Easy state management with Vuex
- Creating custom directives

Written for web developers with some experience in JavaScript, HTML, and CSS.

Erik Hanchett and **Benjamin Listwon** are experienced web engineers and fearless explorers of new ideas.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/vue-js-in-action

"Carefully explains the foundational concepts for understanding what Vue is doing and why."

—From the Foreword by Chris Fritz, Vue Core Team Member

"An excellent hands-on introduction to Vue.js and its ecosystem."

—Alex Miller, Slalom

"Practical examples make learning easy and offer a solid foundation for your own projects."

—Doug Warren, Java Web Services

"Provides a strong understanding of the intrinsic mechanisms of Vues.js. Priceless."

—Philippe Charrière
Clever Cloud

ISBN-13: 978-1-61729-462-4
ISBN-10: 1-61729-462-4
54499
9 781617 294624

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]