Introduction to Vue Render Functions (w/ Examples)

snipcart.com/blog/vue-render-functions

You may or may not have heard the term “render functions” before depending on which frameworks you’re used to tooling around with. Render functions are something specifically related to Vue.js. Though some other frameworks, like React, use the same wording, they're not technically using "Vue render functions".

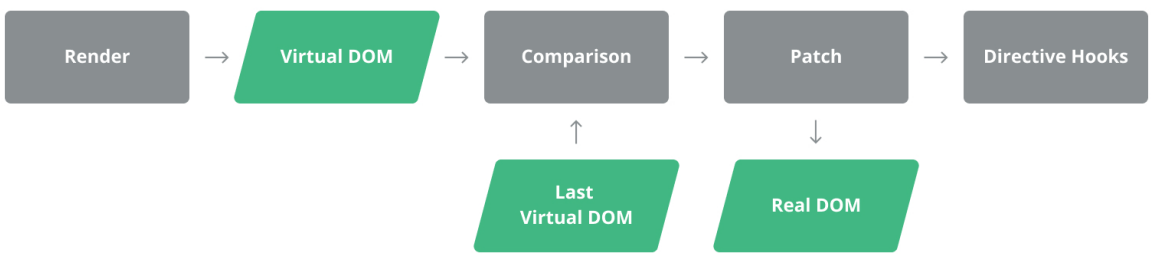
With that short disclaimer out of the way, let’s get down to business. What are they?

Each Vue component implements a render function. Most of the time, the function will be created by the Vue compiler. When you specify a `template` on your component, the content of this template will be processed by the Vue compiler that will return a render function. The render function essentially returns a virtual DOM node which will be rendered by Vue in your browser DOM.

Ok, so that's good and all, but now you may be wondering, "what in the hell is a 'virtual DOM?'"

A virtual Document Object Model (or “DOM”) allows Vue to render your component in its memory before updating your browser. This makes everything faster because, as you can already tell, there are fewer interactions with the browser.

When Vue updates your browser DOM, it compares the updated virtual DOM to the previous one and updates the real DOM *with only the parts that have been modified.* This means that fewer elements change, thereby improving performance. A render function returns a virtual DOM node, commonly named VNode in the Vue ecosystem, which is an interface that allows Vue to write these objects in your browser DOM. They contain all the information necessary to work with Vue. The next version of Vue will include a whole new virtual DOM implementation that will be even faster than it is at the moment. React, Riot, Inferno and many others also use the concept of Virtual DOM.



This image was found and modified from Michael Gallagher's article on Low Verbosity i18n

> React has a created a great explanation of virtual DOMs in their documentation if you're still not 100% clear on the topic.

You can implement a Vue render function in any Vue component. Also, given Vue reactivity, the render function will be called again whenever a reactive property of the component gets updated. Here's a quick example on how you could render a `h1` tag directly from a component render function:

```
new Vue({
  el: '#app',
  render(createElement) {
    return createElement('h1', 'Hello world');
  }
});
```

> View on Codepen

There are some built-in components that leverage the power of the render functions such as `transition` and `keep-alive`. These components manipulate the VNodes directly in the render function. Something that wouldn't be possible without this feature which Vue provides.

> You can also check out Vue's official (and awesome) documentation here.

How do Vue compilers fit in with render functions?

Most of the time, the Vue render function will be compiled by the Vue compiler during the build of your project (with Webpack, for instance). So, the compiler doesn't end up in your production code, saving you a couple of bytes. This is why when you use Single File Components, you don't really need to deal with render functions unless you really need/want to.

However, if you'd like to use the compiler in your code, you can use the full build of Vue which comes with the compiler. This can be quite handy. In fact, we're using it internally here at Snipcart in the new version of the cart (the v3.0) coming out very soon. In short, we're using the Vue compiler to compile custom templates that developers can inject into the cart, giving them more control over the checkout experience than they had before.

We wrote a component that implements a custom render function which fetches the template created by the customer and replaces our default template. And who knows? We might write something about this after the release! ;)

Ok, so stepping aside from our shameless plug, here's a quick example of how you can use the compiler to compile a template string into a render function:

```
const template = `
<ul>
  <li v-for="item in items">
   {{ item }}
  </li>
</ul>`;

const compiledTemplate = Vue.compile(template);

new Vue({
  el: '#app',
  data() {
    return {
      items: ['Item1', 'Item2']
    }
  },
  render(createElement) {
    return compiledTemplate.render.call(this, createElement);
  }
});
```

> View on Codepen

As you can see, the compiler returns an object which contains the renderfunction ready-to-use.

The importance of event binding in Vue render functions

That brings us to event binding. The `createElement` function can receive a parameter called the data object. This object can have multiple properties that are the equivalent to the directives like `v-bind:on` which you use in your standard templates. Here's an example of a simple counter component with a button that increases the click count.

```
new Vue({
  el: '#app',
  data() {
    return {
      clickCount: 0,
    }
  },
  methods: {
    onClick() {
      this.clickCount += 1;
    }
  },
  render(createElement) {
    const button = createElement('button', {
      on: {
        click: this.onClick
      }
    }, 'Click me');

    const counter = createElement('span', [
      'Number of clicks:',
      this.clickCount
    ]);

    return createElement('div', [
      button, counter
    ])
  }
});
```

> View on Codepen

But the data object isn't limited to event binding! You can also apply classes to the element as you'd do with the `v-bind:class` directive.

```
new Vue({
  el: '#app',
  data() {
    return {
      clickCount: 0,
    }
  },
  computed: {
    backgroundColor() {
      return {
        'pink': this.clickCount%2 === 0,
        'green': this.clickCount%2 !== 0,
      };
    }
  },
  methods: {
    onClick() {
      this.clickCount += 1;
    }
  },
  render(createElement) {
    const button = createElement('button', {
      on: {
        click: this.onClick
      }
    }, 'Click me');

    const counter = createElement('span', {
      class: this.backgroundColor,
    }, [
      'Number of clicks:',
      this.clickCount
    ]);

    return createElement('div', [
      button, counter
    ])
  }
});
```

> View on Codepen

> You'll find more information about the data object in this section of Vue's documentation.

Real-life use case of template overriding

I think it's very interesting to understand how Vue works under the hood. Like I said earlier, the only way to know if you're using a tool the most efficiently is to know exactly how it works.

That's not to say that you should start converting all your templates to render functions (you shouldn't) but, sometimes they can come in handy so you should at least know how to use them.

With the example above, I'll show you how we used a custom render function in a component which allows some of our components to be overridable. This is something that we had to implement in the next version of our cart. Just as a quick heads up, I tried to streamline what we did as much as possible for this example.

First, let's create our initial markup.

```
<div id="app">
  <heading>
    <span>Heading title is: {{ title }}</span>
  </heading>
</div>
```

Inside our `div` where the Vue app will mount, we'll define a custom template. Basically, we want the template to override the default version of the `heading` component we'll create.

The first thing we'll do is scan through the custom templates and precompile them with the Vue compiler:

```
const templates = [];
const templatesContainer = document.getElementById('app');


for (var i = 0; i < templatesContainer.children.length; i++) {
  const template = templatesContainer.children.item(i);
  templates.push({
    name: template.nodeName.toLowerCase(),
    renderFunction: Vue.compile(template.innerHTML),
  });
}
```

Then, let's create the `heading` component:

```
Vue.component('heading', {
  props: ['title'],
  template: `
  <overridable name="heading">
    <h1>
      {{ title }}
    </h1>
  </overridable>`
});
```

Now, it's just a simple component that has one `props` named title. The default template will render a `h1` tag with the `title` in it. We'll wrap this component with the `overridable` component that we'll then create.

This is where we'll use a custom render function.

```
Vue.component('overridable', {
  props: ['name'],
  render(createElement) {

    const template = templates.find(x => x.name === this.name);


    if (!template) {
      return this.$slots.default[0];
    }


    return template.renderFunction.render.call(this.$parent, createElement);
  }
});
```

Then, let's mount our Vue app:

```
new Vue({
  el: '#app',
  template: `<heading title="Hello world"></heading>`
});
```

We can see in this example, that the default template is used, so it's a standard `h1` tag.

If we add the custom template inside the `div#app` then we'll see that the `heading` component will now render the custom template we specified.

Conclusion

Overall, it's been fun playing with render functions and has come in handy with our v3.0. Vue render functions are a fundamental piece of Vue itself, so I really think it's valuable to take some time and thoroughly understand the concept (especially if your regularly using the framework). And as Vue.js evolves and becomes more efficient, the knowledge you'll build by knowing what's under the hood will help you evolve right along with it.

In other words, understanding Vue render functions is one small—but important—step to your digital evolution. :)