7 Ways To Define A Component Template in Vue.js

vuejsdevelopers.com/2017/03/24/vue-js-component-templates

There's plenty of choice when it comes to defining component templates in Vue. By my count there are at least seven different ways:

- String
- Template literal
- X-Templates
- Inline
- Render functions
- JSX
- Single page components

And maybe more!

In this article we'll go through examples of each and address the pros and cons so you know which one is the best to use in any particular situation.

1. Strings

By default a template will be defined as a string in your JS file. I think we can all agree that templates in a string are quite incomprehensible. This method doesn't have much going for it other than the wide browser support.

```
Vue.component('my-checkbox', {
 template: `<div class="checkbox-wrapper" @click="check"><div :class="{
checkbox: true, checked: checked }"></div><div class="title">{{ title }}</div>
</div>`,
 data() {
  return { checked: false, title: 'Check me' }
 },
 methods: {
  check() { this.checked = !this.checked; }
 }
});
```

2. Template literals

ES6 template literals ("backticks") allow you to define your template across multiple lines, something you can't do in a regular Javascript string. These are much easier to read and are supported now in many new browsers, though you should probably still transpile down to ES5 to be safe.

This method isn't perfect, though; I find that most IDEs still give you grief with syntax highlighting, and formatting the tabs, newlines etc can still be a pain.

```
Vue.component('my-checkbox', {
 template: `<div class="checkbox-wrapper" @click="check">
       <div :class="{ checkbox: true, checked: checked }"></div>
       <div class="title">{{ title }}</div>
      </div>`,
 data() {
  return { checked: false, title: 'Check me' }
 },
 methods: {
  check() { this.checked = !this.checked; }
 }
});
```

3. X-Templates

With this method your template is defined inside a script tag in the *index.html* file. Th script tag is marked with `text/x-template` and referenced by an id in your component definition.

I like that this method allows you to write your HTML in proper HTML markup, but the downside is that it separate the template from the rest of the component definition.

```
Vue.component('my-checkbox', {
 template: '#checkbox-template',
 data() {
  return { checked: false, title: 'Check me' }
 },
 methods: {
  check() { this.checked = !this.checked; }
 }
});
```

```
<script type="text/x-template" id="checkbox-template">
 <div class="checkbox-wrapper" @click="check">
  <div :class="{ checkbox: true, checked: checked }"></div>
  <div class="title">{{ title }}</div>
 </div>
</script>
```

4. Inline Templates

By adding the `inline-template` attribute to a component, you indicate to Vue that the inner content is its template, rather than treating it as distributed content (see slots).

It suffers the same downside as x-templates, but one advantage is that content is in the correct place within the HTML template and so can be rendered on page load rather than waiting until Javascript is run.

```
Vue.component('my-checkbox', {
 data() {
  return { checked: false, title: 'Check me' }
 },
 methods: {
  check() { this.checked = !this.checked; }
 }
});
```

```
<my-checkbox inline-template>
 <div class="checkbox-wrapper" @click="check">
  <div :class="{ checkbox: true, checked: checked }"></div>
  <div class="title">{{ title }}</div>
 </div>
</my-checkbox>
```

5. Render functions

Render functions require you to define your template as a Javascript object. They are clearly the most verbose and abstract of the template options.

However, the advantages are that your template is closer to the compiler and gives you access to full Javascript functionality rather than the subset offered by directives.

```
Vue.component('my-checkbox', {
 data() {
  return { checked: false, title: 'Check me' }
 },
 methods: {
  check() { this.checked = !this.checked; }
 },
 render(createElement) {
  return createElement(
   'div',
     {
     attrs: {
      'class': 'checkbox-wrapper'
     },
     on: {
      click: this.check
     }
     },
     [
       createElement(
         'div',
         {
           'class': {
            checkbox: true,
            checked: this.checked
           }
         }
       ),
       createElement(
         'div',
         {
           attrs: {
             'class': 'title'
           }
         },
         [ this.title ]
       )
     ]
  );
 }
});
```

6. JSX

The most controversial template option in Vue is JSX. Some developers see JSX as ugly, unintuitive and as a betrayal to the Vue ethos.

JSX requires you to transpile first, as it is not readable by browsers. But, if you need to use render functions, JSX is surely a less abstract way of defining a template.

```
Vue.component('my-checkbox', {
 data() {
  return { checked: false, title: 'Check me' }
 },
 methods: {
  check() { this.checked = !this.checked; }
 },
 render() {
  return <div class="checkbox-wrapper" onClick={ this.check }>
          <div class={{ checkbox: true, checked: this.checked }}></div>
          <div class="title">{ this.title }</div>
        </div>
 }
});
```

7. Single File Components

So long as you are comfortable with using a build tool in your setup, Single File Components are the king of template options. They bring the best of both worlds: allowing you to write markup while keeping all your component defintion in one file.

They require transpiling and some IDEs don't support syntax highlighting for this file type, but are otherwise hard to beat.

```
<template>
  <div class="checkbox-wrapper" @click="check">
    <div :class="{ checkbox: true, checked: checked }"></div>
    <div class="title">{{ title }}</div>
  </div>
</template>
<script>
  export default {
    data() {
      return { checked: false, title: 'Check me' }
    },
    methods: {
      check() { this.checked = !this.checked; }
    }
  }
</script>
```

You might argue there are even more template definition possibilities since you can use template pre-processors like Pug with SFCs!

Which is the best?

Of course there's no one perfect way, and each should be judged on the *use case* you have. I think the best developers will be aware of all the possiblities and have each as a tool in their Vue.js toolbelt!