Class and Style Bindings

A common need for data binding is manipulating an element's class list and inline styles. Since `class` and `style` are both attributes, we can use `v-bind` to assign them a string value dynamically, much like with other attributes. However, trying to generate those values using string concatenation can be annoying and error-prone. For this reason, Vue provides special enhancements when `v-bind` is used with `class` and `style`. In addition to strings, the expressions can also evaluate to objects or arrays.

Binding HTML Classes

Binding to Objects

We can pass an object to `:class` (short for `v-bind:class`) to dynamically toggle classes:

```template
<div :class="{ active: isActive }"></div>
```

The above syntax means the presence of the `active` class will be determined by the truthiness of the data property `isActive`.

You can have multiple classes toggled by having more fields in the object. In addition, the `:class` directive can also co-exist with the plain `class` attribute. So given the following state:

```js
const isActive = ref(true)
const hasError = ref(false)
```

And the following template:

```
  :class="{ active: isActive, 'text-danger': hasError }"
></div>
```

It will render:

```template
<div class="static active"></div>
```

When `isActive` or `hasError` changes, the class list will be updated accordingly. For example, if `hasError` becomes `true`, the class list will become `"static active text-danger"`.

The bound object doesn't have to be inline:

```js
const classObject = reactive({
  active: true,
  'text-danger': false
})
```

```template
<div :class="classObject"></div>
```

This will render:

```template
<div class="active"></div>
```

We can also bind to a computed property that returns an object. This is a common and powerful pattern:

```js
const isActive = ref(true)
const error = ref(null)

const classObject = computed(() => ({
  active: isActive.value && !error.value,
  'text-danger': error.value && error.value.type === 'fatal'
}))
```

```template
<div :class="classObject"></div>
```

Binding to Arrays

We can bind `:class` to an array to apply a list of classes:

```template
<div :class="[activeClass, errorClass]"></div>
```

Which will render:

```template
<div class="active text-danger"></div>
```

If you would like to also toggle a class in the list conditionally, you can do it with a ternary expression:

```template
<div :class="[isActive ? activeClass : '', errorClass]"></div>
```

This will always apply `errorClass`, but `activeClass` will only be applied when `isActive` is truthy.

However, this can be a bit verbose if you have multiple conditional classes. That's why it's also possible to use the object syntax inside the array syntax:

```template
<div :class="[{ [activeClass]: isActive }, errorClass]"></div>
```

With Components

> This section assumes knowledge of Components. Feel free to skip it and come back later.

When you use the `class` attribute on a component with a single root element, those classes will be added to the component's root element and merged with any existing class already on it.

For example, if we have a component named `MyComponent` with the following template:

```template
<!-- child component template -->
<p class="foo bar">Hi!</p>
```

Then add some classes when using it:

```template
<!-- when using the component -->
<MyComponent class="baz boo" />
```

The rendered HTML will be:

The same is true for class bindings:

```template
<MyComponent :class="{ active: isActive }" />
```

When `isActive` is truthy, the rendered HTML will be:

```template
<p class="foo bar active">Hi!</p>
```

If your component has multiple root elements, you would need to define which element will receive this class. You can do this using the `$attrs` component property:

```template
<!-- MyComponent template using $attrs -->
<p :class="$attrs.class">Hi!</p>
<span>This is a child component</span>
```

```template
<MyComponent class="baz" />
```

Will render:

```html
<p class="baz">Hi!</p>
<span>This is a child component</span>
```

You can learn more about component attribute inheritance in Fallthrough Attributes section.

Binding Inline Styles

Binding to Objects

`:style` supports binding to JavaScript object values - it corresponds to an HTML element's `style` property:

```js
const activeColor = ref('red')
const fontSize = ref(30)
```

```template
<div :style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
```

```template
<div :style="{ 'font-size': fontSize + 'px' }"></div>
```

It is often a good idea to bind to a style object directly so that the template is cleaner:

```js
const styleObject = reactive({
  color: 'red',
  fontSize: '30px'
})
```

```template
<div :style="styleObject"></div>
```

Again, object style binding is often used in conjunction with computed properties that return objects.

`:style` directives can also coexist with regular style attributes, just like `:class`.

Template:

```template
<h1 style="color: red" :style="'font-size: 1em'">hello</h1>
```

It will render:

```template
<h1 style="color: red; font-size: 1em;">hello</h1>
```

Binding to Arrays

We can bind `:style` to an array of multiple style objects. These objects will be merged and applied to the same element:

```template
<div :style="[baseStyles, overridingStyles]"></div>
```

Auto-prefixing

When you use a CSS property that requires a vendor prefix in `:style`, Vue will automatically add the appropriate prefix. Vue does this by checking at runtime to see which style properties are supported in the current browser. If the browser doesn't support a particular property then various prefixed variants will be tested to try to find one that is supported.

You can provide an array of multiple (prefixed) values to a style property, for example:

```template
<div :style="{ display: ['-webkit-box', '-ms-flexbox', 'flex'] }"></div>
```

This will only render the last value in the array which the browser supports. In this example, it will render `display: flex` for browsers that support the unprefixed version of flexbox.

Edit this page on GitHub

Previous
Computed Properties

Next
Conditional Rendering