Computed Properties

Watch a free video lesson on Vue School

Basic Example

In-template expressions are very convenient, but they are meant for simple operations. Putting too much logic in your templates can make them bloated and hard to maintain. For example, if we have an object with a nested array:

```js
const author = reactive({
  name: 'John Doe',
  books: [
    'Vue 2 - Advanced Guide',
    'Vue 3 - Basic Guide',
    'Vue 4 - The Mystery'
  ]
})
```

And we want to display different messages depending on if `author` already has some books or not:

```template
<p>Has published books:</p>
<span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
```

At this point, the template is getting a bit cluttered. We have to look at it for a second before realizing that it performs a calculation depending on `author.books`. More importantly, we probably don't want to repeat ourselves if we need to include this calculation in the template more than once.

That's why for complex logic that includes reactive data, it is recommended to use a **computed property**. Here's the same example, refactored:

```
const author = reactive({
  name: 'John Doe',
  books: [
    'Vue 2 - Advanced Guide',
    'Vue 3 - Basic Guide',
    'Vue 4 - The Mystery'
  ]
})

// a computed ref
const publishedBooksMessage = computed(() => {
  return author.books.length > 0 ? 'Yes' : 'No'
})
</script>

<template>
  <p>Has published books:</p>
  <span>{{ publishedBooksMessage }}</span>
</template>
```

Try it in the Playground

Here we have declared a computed property `publishedBooksMessage`. The `computed()` function expects to be passed a getter function, and the returned value is a **computed ref**. Similar to normal refs, you can access the computed result as `publishedBooksMessage.value`. Computed refs are also auto-unwrapped in templates so you can reference them without `.value` in template expressions.

A computed property automatically tracks its reactive dependencies. Vue is aware that the computation of `publishedBooksMessage` depends on `author.books`, so it will update any bindings that depend on `publishedBooksMessage` when `author.books` changes.

See also: Typing Computed TS

Computed Caching vs. Methods

You may have noticed we can achieve the same result by invoking a method in the expression:

template

```
<p>{{ calculateBooksMessage() }}</p>
```

js

```
// in component
function calculateBooksMessage() {
```

Instead of a computed property, we can define the same function as a method. For the end result, the two approaches are indeed exactly the same. However, the difference is that **computed properties are cached based on their reactive dependencies.** A computed property will only re-evaluate when some of its reactive dependencies have changed. This means as long as `author.books` has not changed, multiple access to `publishedBooksMessage` will immediately return the previously computed result without having to run the getter function again.

This also means the following computed property will never update, because `Date.now()` is not a reactive dependency:

```js
const now = computed(() => Date.now())
```

In comparison, a method invocation will **always** run the function whenever a re-render happens.

Why do we need caching? Imagine we have an expensive computed property `list`, which requires looping through a huge array and doing a lot of computations. Then we may have other computed properties that in turn depend on `list`. Without caching, we would be executing `list`'s getter many more times than necessary! In cases where you do not want caching, use a method call instead.

Writable Computed

Computed properties are by default getter-only. If you attempt to assign a new value to a computed property, you will receive a runtime warning. In the rare cases where you need a "writable" computed property, you can create one by providing both a getter and a setter:

```vue
<script setup>
import { ref, computed } from 'vue'

const firstName = ref('John')
const lastName = ref('Doe')

const fullName = computed({
  // getter
  get() {
    return firstName.value + ' ' + lastName.value
  },
  // setter
  set(newValue) {
    // Note: we are using destructuring assignment syntax here.
    [firstName.value, lastName.value] = newValue.split(' ')
```

```
</script>
```

Now when you run `fullName.value = 'John Doe'`, the setter will be invoked and `firstName` and `lastName` will be updated accordingly.

Getting the Previous Value

- Only supported in 3.4+

In case you need it, you can get the previous value returned by the computed property accessing the first argument of the getter:

```vue
<script setup>
import { ref, computed } from 'vue'

const count = ref(2)

// This computed will return the value of count when it's less or equal to 3.
// When count is >=4, the last value that fulfilled our condition will be ret
// instead until count is less or equal to 3
const alwaysSmall = computed((previous) => {
  if (count.value <= 3) {
    return count.value
  }

  return previous
})
</script>
```

In case you're using a writable computed:

```vue
<script setup>
import { ref, computed } from 'vue'

const count = ref(2)

const alwaysSmall = computed({
  get(previous) {
    if (count.value <= 3) {
      return count.value
    }

    return previous
  },
  set(newValue) {
    count.value = newValue * 2
  }
```

Best Practices

Getters should be side-effect free

It is important to remember that computed getter functions should only perform pure computation and be free of side effects. For example, **don't mutate other state, make async requests, or mutate the DOM inside a computed getter!** Think of a computed property as declaratively describing how to derive a value based on other values - its only responsibility should be computing and returning that value. Later in the guide we will discuss how we can perform side effects in reaction to state changes with watchers.

Avoid mutating computed value

The returned value from a computed property is derived state. Think of it as a temporary snapshot - every time the source state changes, a new snapshot is created. It does not make sense to mutate a snapshot, so a computed return value should be treated as read-only and never be mutated - instead, update the source state it depends on to trigger new computations.

Edit this page on GitHub

Previous
Reactivity Fundamentals

Next
Class and Style Bindings