List Rendering

Watch a free video lesson on Vue School

`v-for`

We can use the `v-for` directive to render a list of items based on an array. The `v-for` directive requires a special syntax in the form of `item in items`, where `items` is the source data array and `item` is an **alias** for the array element being iterated on:

```js
const items = ref([{ message: 'Foo' }, { message: 'Bar' }])
```

```template
<li v-for="item in items">
  {{ item.message }}
</li>
```

Inside the `v-for` scope, template expressions have access to all parent scope properties. In addition, `v-for` also supports an optional second alias for the index of the current item:

```js
const parentMessage = ref('Parent')
const items = ref([{ message: 'Foo' }, { message: 'Bar' }])
```

```template
<li v-for="(item, index) in items">
  {{ parentMessage }} - {{ index }} - {{ item.message }}
</li>
```

- Parent - 0 - Foo
- Parent - 1 - Bar

Try it in the Playground

```js
const parentMessage = 'Parent'
const items = [
  /* ... */
]

items.forEach((item, index) => {
  // has access to outer scope `parentMessage`
  // but `item` and `index` are only available in here
  console.log(parentMessage, item.message, index)
})
```

Notice how the `v-for` value matches the function signature of the `forEach` callback. In fact, you can use destructuring on the `v-for` item alias similar to destructuring function arguments:

```template
<li v-for="{ message } in items">
  {{ message }}
</li>

<!-- with index alias -->
<li v-for="({ message }, index) in items">
  {{ message }} {{ index }}
</li>
```

For nested `v-for`, scoping also works similar to nested functions. Each `v-for` scope has access to parent scopes:

```template
<li v-for="item in items">
  <span v-for="childItem in item.children">
    {{ item.message }} {{ childItem }}
  </span>
</li>
```

You can also use `of` as the delimiter instead of `in`, so that it is closer to JavaScript's syntax for iterators:

```template
<div v-for="item of items"></div>
```

`v-for` with an Object

You can also use `v-for` to iterate through the properties of an object. The iteration order will be based on the result of calling `Object.values()` on the object:

```
  author: 'Jane Doe',
  publishedAt: '2016-04-10'
})
```

```template
<ul>
  <li v-for="value in myObject">
    {{ value }}
  </li>
</ul>
```

You can also provide a second alias for the property's name (a.k.a. key):

```template
<li v-for="(value, key) in myObject">
  {{ key }}: {{ value }}
</li>
```

And another for the index:

```template
<li v-for="(value, key, index) in myObject">
  {{ index }}. {{ key }}: {{ value }}
</li>
```

Try it in the Playground

`v-for` with a Range

`v-for` can also take an integer. In this case it will repeat the template that many times, based on a range of `1...n`.

```template
<span v-for="n in 10">{{ n }}</span>
```

Note here `n` starts with an initial value of `1` instead of `0`.

`v-for` on `<template>`

Similar to template `v-if`, you can also use a `<template>` tag with `v-for` to render a block of multiple elements. For example:

```
    <li>{{ item.msg }}</li>
    <li class="divider" role="presentation"></li>
  </template>
</ul>
```

`v-for` with `v-if`

When they exist on the same node, `v-if` has a higher priority than `v-for`. That means the `v-if` condition will not have access to variables from the scope of the `v-for`:

template

```
<!--
This will throw an error because property "todo"
is not defined on instance.
-->
<li v-for="todo in todos" v-if="!todo.isComplete">
  {{ todo.name }}
</li>
```

This can be fixed by moving `v-for` to a wrapping `<template>` tag (which is also more explicit):

template

```
<template v-for="todo in todos">
  <li v-if="!todo.isComplete">
    {{ todo.name }}
  </li>
</template>
```

> ⚠ Note
>
> It's not recommended to use `v-if` and `v-for` on the same element due to implicit precedence.
>
> There are two common cases where this can be tempting:
>
> - To filter items in a list (e.g. `v-for="user in users" v-if="user.isActive"`). In these cases, replace `users` with a new computed property that returns your filtered list (e.g. `activeUsers`).
> - To avoid rendering a list if it should be hidden (e.g. `v-for="user in users" v-if="shouldShowUsers"`). In these cases, move the `v-if` to a container element (e.g. `ul`, `ol`).

Maintaining State with `key`

When Vue is updating a list of elements rendered with `v-for`, by default it uses an "in-place patch" strategy. If the order of the data items has changed, instead of moving the DOM elements to match the order of the items, Vue will patch each element in-place and make sure it reflects what should be rendered at that particular index.

This default mode is efficient, but **only suitable when your list render output does not rely on child component state or temporary DOM state (e.g. form input values)**.

To give Vue a hint so that it can track each node's identity, and thus reuse and reorder existing elements, you need to provide a unique `key` attribute for each item:

```template
<div v-for="item in items" :key="item.id">
  <!-- content -->
</div>
```

When using `<template v-for>`, the `key` should be placed on the `<template>` container:

```template
<template v-for="todo in todos" :key="todo.name">
  <li>{{ todo.name }}</li>
</template>
```

> ⓘ Note
>
> `key` here is a special attribute being bound with `v-bind`. It should not be confused with the property key variable when using `v-for` with an object.

It is recommended to provide a `key` attribute with `v-for` whenever possible, unless the iterated DOM content is simple (i.e. contains no components or stateful DOM elements), or you are intentionally relying on the default behavior for performance gains.

The `key` binding expects primitive values - i.e. strings and numbers. Do not use objects as `v-for` keys. For detailed usage of the `key` attribute, please see the `key` API documentation.

`v-for` with a Component

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

```template
<MyComponent v-for="item in items" :key="item.id" />
```

However, this won't automatically pass any data to the component, because components have isolated scopes of their own. In order to pass the iterated data into the component, we should also use props:

```template
<MyComponent
  v-for="(item, index) in items"
  :item="item"
  :index="index"
  :key="item.id"
/>
```

The reason for not automatically injecting `item` into the component is because that makes the component tightly coupled to how `v-for` works. Being explicit about where its data comes from makes the component reusable in other situations.

Check out this example of a simple todo list to see how to render a list of components using `v-for`, passing different data to each instance.

Array Change Detection

Mutation Methods

Vue is able to detect when a reactive array's mutation methods are called and trigger necessary updates. These mutation methods are:

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

Replacing an Array

Mutation methods, as the name suggests, mutate the original array they are called on. In comparison, there are also non-mutating methods, e.g. `filter()`, `concat()` and

```js
// `items` is a ref with array value
items.value = items.value.filter((item) => item.message.match(/Foo/))
```

You might think this will cause Vue to throw away the existing DOM and re-render the entire list - luckily, that is not the case. Vue implements some smart heuristics to maximize DOM element reuse, so replacing an array with another array containing overlapping objects is a very efficient operation.

Displaying Filtered/Sorted Results

Sometimes we want to display a filtered or sorted version of an array without actually mutating or resetting the original data. In this case, you can create a computed property that returns the filtered or sorted array.

For example:

```js
const numbers = ref([1, 2, 3, 4, 5])

const evenNumbers = computed(() => {
  return numbers.value.filter((n) => n % 2 === 0)
})
```

```template
<li v-for="n in evenNumbers">{{ n }}</li>
```

In situations where computed properties are not feasible (e.g. inside nested `v-for` loops), you can use a method:

```js
const sets = ref([
  [1, 2, 3, 4, 5],
  [6, 7, 8, 9, 10]
])

function even(numbers) {
  return numbers.filter((number) => number % 2 === 0)
}
```

```template
<ul v-for="numbers in sets">
  <li v-for="n in even(numbers)">{{ n }}</li>
</ul>
```

original array before calling these methods:

```diff
- return numbers.reverse()
+ return [...numbers].reverse()
```

Edit this page on GitHub

Previous
Conditional Rendering

Next
Event Handling