**List Rendering**

[**Mapping an Array to Elements with v-for**](https://vuejs.org/v2/guide/list.html#Mapping-an-Array-to-Elements-with-v-for)

We can use the v-for directive to render a list of items based on an array. The v-fordirective 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:

|  |
| --- |
| <ul id="example-1">  <li v-for="item in items">  {{ item.message }}  </li> </ul> |

|  |
| --- |
| var example1 = new Vue({  el: '#example-1',  data: {  items: [  { message: 'Foo' },  { message: 'Bar' }  ]  } }) |

data: {  
 items: [ 'Foo','Bar' ]  
 }

Result:

* Foo
* Bar

Inside v-for blocks we have full access to parent scope properties. v-for also supports an optional second argument for the index of the current item.

|  |
| --- |
| <ul id="example-2">  <li v-for="(item, index) in items">  {{ parentMessage }} - {{ index }} - {{ item.message }}  </li> </ul> |

|  |
| --- |
| var example2 = new Vue({  el: '#example-2',  data: {  parentMessage: 'Parent',  items: [  { message: 'Foo' },  { message: 'Bar' }  ]  } }) |

Result:

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

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

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

[**v-for with an Object**](https://vuejs.org/v2/guide/list.html#v-for-with-an-Object)

You can also use v-for to iterate through the properties of an object.

|  |
| --- |
| <ul id="v-for-object" class="demo">  <li v-for="value in object">  {{ value }}  </li> </ul> |

|  |
| --- |
| new Vue({  el: '#v-for-object',  data: {  object: {  firstName: 'John',  lastName: 'Doe',  age: 30  }  } }) |

Result:

* John
* Doe
* 30

You can also provide a second argument for the key:

|  |
| --- |
| <div v-for="(value, key) in object">  {{ key }}: {{ value }} </div> |

firstName: John

lastName: Doe

age: 30

And another for the index:

|  |
| --- |
| <div v-for="(value, key, index) in object">  {{ index }}. {{ key }}: {{ value }} </div> |

0. firstName: John

1. lastName: Doe

2. age: 30

When iterating over an object, the order is based on the key enumeration order of Object.keys(), which is **not** guaranteed to be consistent across JavaScript engine implementations.

[**key**](https://vuejs.org/v2/guide/list.html#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 is similar to the behavior of track-by="$index" in Vue 1.x.

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. An ideal value for keywould be the unique id of each item. This special attribute is a rough equivalent to track-byin 1.x, but it works like an attribute, so you need to use v-bind to bind it to dynamic values (using shorthand here):

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

It is recommended to provide a key with v-for whenever possible, unless the iterated DOM content is simple, or you are intentionally relying on the default behavior for performance gains.

Since it’s a generic mechanism for Vue to identify nodes, the key also has other uses that are not specifically tied to v-for, as we will see later in the guide.

[**Array Change Detection**](https://vuejs.org/v2/guide/list.html#Array-Change-Detection)

[**Mutation Methods**](https://vuejs.org/v2/guide/list.html#Mutation-Methods)

Vue wraps an observed array’s mutation methods so they will also trigger view updates. The wrapped methods are:

* push() : add new row to object/array
* pop() : read last data from object/array
* shift(): position wise
* unshift():
* splice(index): delete
* sort()
* reverse()

a = [22,33,444,3]

a.sort() // ascending

a.sort().reverse() // descending

You can open the console and play with the previous examples’ items array by calling their mutation methods. For example: example1.items.push({ message: 'Baz' }).

[**Replacing an Array**](https://vuejs.org/v2/guide/list.html#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 slice(), which do not mutate the original array but **always return a new array**. When working with non-mutating methods, you can replace the old array with the new one:

|  |
| --- |
| example1.items = example1.items.filter(function (item) {  return 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.

[**Caveats**](https://vuejs.org/v2/guide/list.html#Caveats)

Due to limitations in JavaScript, Vue **cannot** detect the following changes to an array:

1. When you directly set an item with the index, e.g. vm.items[indexOfItem] = newValue
2. When you modify the length of the array, e.g. vm.items.length = newLength

For example:

|  |
| --- |
| var vm = new Vue({  data: {  items: ['a', 'b', 'c']  } }) vm.items[1] = 'x' // is NOT reactive vm.items.length = 2 // is NOT reactive |

To overcome caveat 1, both of the following will accomplish the same as vm.items[indexOfItem] = newValue, but will also trigger state updates in the reactivity system:

|  |
| --- |
| // Vue.set Vue.set(vm.items, indexOfItem, newValue) |

|  |
| --- |
| // Array.prototype.splice vm.items.splice(indexOfItem, 1, newValue) |

You can also use the **[vm.$set](https://vuejs.org/v2/api/" \l "vm-set)** instance method, which is an alias for the global Vue.set:

|  |
| --- |
| vm.$set(vm.items, indexOfItem, newValue) |

To deal with caveat 2, you can use splice:

|  |
| --- |
| vm.items.splice(newLength) |

[**Object Change Detection Caveats**](https://vuejs.org/v2/guide/list.html#Object-Change-Detection-Caveats)

Again due to limitations of modern JavaScript, **Vue cannot detect property addition or deletion**. For example:

|  |
| --- |
| var vm = new Vue({  data: {  a: 1  } }) // `vm.a` is now reactive  vm.b = 2 // `vm.b` is NOT reactive |

Vue does not allow dynamically adding new root-level reactive properties to an already created instance. However, it’s possible to add reactive properties to a nested object using the Vue.set(object, key, value) method. For example, given:

|  |
| --- |
| var vm = new Vue({  data: {  userProfile: {  name: 'Anika'  }  } }) |

You could add a new age property to the nested userProfile object with:

|  |
| --- |
| Vue.set(vm.userProfile, 'age', 27) |

You can also use the vm.$set instance method, which is an alias for the global Vue.set:

|  |
| --- |
| vm.$set(vm.userProfile, 'age', 27) |

Sometimes you may want to assign a number of new properties to an existing object, for example using Object.assign() or \_.extend(). In such cases, you should create a fresh object with properties from both objects. So instead of:

|  |
| --- |
| Object.assign(vm.userProfile, {  age: 27,  favoriteColor: 'Vue Green' }) |

You would add new, reactive properties with:

|  |
| --- |
| vm.userProfile = Object.assign({}, vm.userProfile, {  age: 27,  favoriteColor: 'Vue Green' }) |

[**Displaying Filtered/Sorted Results**](https://vuejs.org/v2/guide/list.html#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:

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

|  |
| --- |
| data: {  numbers: [ 1, 2, 3, 4, 5 ] }, computed: {  evenNumbers: function () {  return this.numbers.filter(function (number) {  return number % 2 === 0  })  } } |

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

|  |
| --- |
| <li v-for="n in even(numbers)">{{ n }}</li> |

|  |
| --- |
| data: {  numbers: [ 1, 2, 3, 4, 5 ] }, methods: {  even: function (numbers) {  return numbers.filter(function (number) {  return number % 2 === 0  })  } } |

[**v-for with a Range**](https://vuejs.org/v2/guide/list.html#v-for-with-a-Range)

v-for can also take an integer. In this case it will repeat the template that many times.

|  |
| --- |
| <div>  <span v-for="n in 10">{{ n }} </span> </div> |

Result:

1 2 3 4 5 6 7 8 9 10

[**v-for on a <template>**](https://vuejs.org/v2/guide/list.html#v-for-on-a-lt-template-gt)

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

|  |
| --- |
| <ul>  <template v-for="item in items">  <li>{{ item.msg }}</li>  <li class="divider" role="presentation"></li>  </template> </ul> |

[**v-for with v-if**](https://vuejs.org/v2/guide/list.html#v-for-with-v-if)

When they exist on the same node, v-for has a higher priority than v-if. That means the v-if will be run on each iteration of the loop separately. This can be useful when you want to render nodes for only *some* items, like below:

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

The above only renders the todos that are not complete.

If instead, your intent is to conditionally skip execution of the loop, you can place the v-ifon a wrapper element (or [**<template>**](https://vuejs.org/v2/guide/conditional.html#Conditional-Groups-with-v-if-on-lt-template-gt)). For example:

|  |
| --- |
| <ul v-if="todos.length">  <li v-for="todo in todos">  {{ todo }}  </li> </ul> <p v-else>No todos left!</p> |

[**v-for with a Component**](https://vuejs.org/v2/guide/list.html#v-for-with-a-Component)

**This section assumes knowledge of**[**Components**](https://vuejs.org/v2/guide/components.html)**. Feel free to skip it and come back later.**

You can directly use v-for on a custom component, like any normal element:

|  |
| --- |
| <my-component v-for="item in items" :key="item.id"></my-component> |

**In 2.2.0+, when using v-for with a component, a**[**key**](https://vuejs.org/v2/guide/list.html#key)**is now required.**

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:

|  |
| --- |
| <my-component  v-for="(item, index) in items"  v-bind:item="item"  v-bind:index="index"  v-bind:key="item.id" ></my-component> |

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.

Here’s a complete example of a simple todo list:

|  |
| --- |
| <div id="todo-list-example">  <form v-on:submit.prevent="addNewTodo">  <label for="new-todo">Add a todo</label>  <input  v-model="newTodoText"  id="new-todo"  placeholder="E.g. Feed the cat"  >  <button>Add</button>  </form>  <ul>  <li  is="todo-item"  v-for="(todo, index) in todos"  v-bind:key="todo.id"  v-bind:title="todo.title"  v-on:remove="todos.splice(index, 1)"  ></li>  </ul> </div> |

Note the is="todo-item" attribute. This is necessary in DOM templates, because only an <li> element is valid inside a <ul>. It does the same thing as <todo-item>, but works around a potential browser parsing error. See [**DOM Template Parsing Caveats**](https://vuejs.org/v2/guide/components.html#DOM-Template-Parsing-Caveats) to learn more.

|  |
| --- |
| Vue.component('todo-item', {  template: '\  <li>\  {{ title }}\  <button v-on:click="$emit(\'remove\')">Remove</button>\  </li>\  ',  props: ['title'] })  new Vue({  el: '#todo-list-example',  data: {  newTodoText: '',  todos: [  {  id: 1,  title: 'Do the dishes',  },  {  id: 2,  title: 'Take out the trash',  },  {  id: 3,  title: 'Mow the lawn'  }  ],  nextTodoId: 4  },  methods: {  addNewTodo: function () {  this.todos.push({  id: this.nextTodoId++,  title: this.newTodoText  })  this.newTodoText = ''  }  } }) |