**Introduction**

[**What is Vue.js?**](https://vuejs.org/v2/guide/#What-is-Vue-js)

Vue (pronounced /vjuː/, like **view**) is a **progressive framework** for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. On the other hand, Vue is also perfectly capable of powering sophisticated Single-Page Applications when used in combination with [**modern tooling**](https://vuejs.org/v2/guide/single-file-components.html) and [**supporting libraries**](https://github.com/vuejs/awesome-vue#components--libraries).

If you’d like to learn more about Vue before diving in, we [**created a video**](https://vuejs.org/v2/guide/) walking through the core principles and a sample project.

If you are an experienced frontend developer and want to know how Vue compares to other libraries/frameworks, check out the [**Comparison with Other Frameworks**](https://vuejs.org/v2/guide/comparison.html).

## [Getting Started](https://vuejs.org/v2/guide/#Getting-Started)

The easiest way to try out Vue.js is using the [**Hello World example**](https://codesandbox.io/s/github/vuejs/vuejs.org/tree/master/src/v2/examples/vue-20-hello-world). Feel free to open it in another tab and follow along as we go through some basic examples. Or, you can [**create an index.html file**](https://github.com/vuejs/vuejs.org/blob/master/src/v2/examples/vue-20-hello-world/index.html) and include Vue with:

<!-- development version, includes helpful console warnings -->

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>

or:

<!-- production version, optimized for size and speed -->

<script src="https://cdn.jsdelivr.net/npm/vue"></script>

The [**Installation**](https://vuejs.org/v2/guide/installation.html) page provides more options of installing Vue. Note: We **do not** recommend that beginners start with vue-cli, especially if you are not yet familiar with Node.js-based build tools.

If you prefer something more interactive, you can also check out [**this tutorial series on Scrimba**](https://scrimba.com/g/gvuedocs), which gives you a mix of screencast and code playground that you can pause and play around with anytime.

## [Declarative Rendering](https://vuejs.org/v2/guide/" \l "Declarative-Rendering" \o "Declarative Rendering)

At the core of Vue.js is a system that enables us to declaratively render data to the DOM using straightforward template syntax:

<div id="app">

{{ message }}

</div>

var app = new Vue({

el: '#app',

data: {

message: 'Hello Vue!'

}

})

In addition to text interpolation, we can also bind element attributes like this:

<div id="app-2">

<span v-bind:title="message">

Hover your mouse over me for a few seconds

to see my dynamically bound title!

</span>

</div>

var app2 = new Vue({

el: '#app-2',

data: {

message: 'You loaded this page on ' + new Date().toLocaleString()

}

})

Here we are encountering something new. The v-bind attribute you are seeing is called a **directive**. Directives are prefixed with v- to indicate that they are special attributes provided by Vue, and as you may have guessed, they apply special reactive behavior to the rendered DOM. Here, it is basically saying “keep this element’s title attribute up-to-date with the message property on the Vue instance.”

If you open up your JavaScript console again and enter app2.message = 'some new message', you’ll once again see that the bound HTML - in this case the title attribute - has been updated.

It’s easy to toggle the presence of an element, too:

<div id="app-3">

<span v-if="seen">Now you see me</span>

</div>

var app3 = new Vue({

el: '#app-3',

data: {

seen: true

}

})

This example demonstrates that we can bind data to not only text and attributes, but also the **structure** of the DOM. Moreover, Vue also provides a powerful transition effect system that can automatically apply [**transition effects**](https://vuejs.org/v2/guide/transitions.html) when elements are inserted/updated/removed by Vue.

There are quite a few other directives, each with its own special functionality. For example, the v-for directive can be used for displaying a list of items using the data from an Array:

<div id="app-4">

<ol>

<li v-for="todo in todos">

{{ todo.text }}

</li>

</ol>

</div>

var app4 = new Vue({

el: '#app-4',

data: {

todos: [

{ text: 'Learn JavaScript' },

{ text: 'Learn Vue' },

{ text: 'Build something awesome' }

]

}

})

## [Conditionals and Loops](https://vuejs.org/v2/guide/#Conditionals-and-Loops)

It’s easy to toggle the presence of an element, too:

<div id="app-3">

<span v-if="seen">Now you see me</span>

</div>

var app3 = new Vue({

el: '#app-3',

data: {

seen: true

}

})

This example demonstrates that we can bind data to not only text and attributes, but also the **structure** of the DOM. Moreover, Vue also provides a powerful transition effect system that can automatically apply [**transition effects**](https://vuejs.org/v2/guide/transitions.html) when elements are inserted/updated/removed by Vue.

There are quite a few other directives, each with its own special functionality. For example, the v-for directive can be used for displaying a list of items using the data from an Array:

<div id="app-4">

<ol>

<li v-for="todo in todos">

{{ todo.text }}

</li>

</ol>

</div>

var app4 = new Vue({

el: '#app-4',

data: {

todos: [

{ text: 'Learn JavaScript' },

{ text: 'Learn Vue' },

{ text: 'Build something awesome' }

]

}

})

1. Learn JavaScript
2. Learn Vue
3. Build something awesome

In the console, enter app4.todos.push({ text: 'New item' }). You should see a new item appended to the list.

## [Handling User Input](https://vuejs.org/v2/guide/#Handling-User-Input)

To let users interact with your app, we can use the v-on directive to attach event listeners that invoke methods on our Vue instances:

<div id="app-5">

<p>{{ message }}</p>

<button v-on:click="reverseMessage">Reverse Message</button>

</div>

var app5 = new Vue({

el: '#app-5',

data: {

message: 'Hello Vue.js!'

},

methods: {

reverseMessage: function () {

this.message = this.message.split('').reverse().join('')

}

}

})

Note that in this method we update the state of our app without touching the DOM - all DOM manipulations are handled by Vue, and the code you write is focused on the underlying logic.

Vue also provides the v-model directive that makes two-way binding between form input and app state a breeze:

<div id="app-6">

<p>{{ message }}</p>

<input v-model="message">

</div>

var app6 = new Vue({

el: '#app-6',

data: {

message: 'Hello Vue!'

}

})

## [Composing with Components](https://vuejs.org/v2/guide/#Composing-with-Components)

The component system is another important concept in Vue, because it’s an abstraction that allows us to build large-scale applications composed of small, self-contained, and often reusable components. If we think about it, almost any type of application interface can be abstracted into a tree of components:



In Vue, a component is essentially a Vue instance with pre-defined options. Registering a component in Vue is straightforward:

// Define a new component called todo-item

Vue.component('todo-item', {

template: '<li>This is a todo</li>'

})

var app = new Vue(...)

Now you can compose it in another component’s template:

<ol>

<!-- Create an instance of the todo-item component -->

<todo-item></todo-item>

</ol>

But this would render the same text for every todo, which is not super interesting. We should be able to pass data from the parent scope into child components. Let’s modify the component definition to make it accept a [**prop**](https://vuejs.org/v2/guide/components.html#Props):

Vue.component('todo-item', {

// The todo-item component now accepts a

// "prop", which is like a custom attribute.

// This prop is called todo.

props: ['todo'],

template: '<li>{{ todo.text }}</li>'

})

Now we can pass the todo into each repeated component using v-bind:

<div id="app-7">

<ol>

<!--

Now we provide each todo-item with the todo object

it's representing, so that its content can be dynamic.

We also need to provide each component with a "key",

which will be explained later.

-->

<todo-item

v-for="item in groceryList"

v-bind:todo="item"

v-bind:key="item.id"

></todo-item>

</ol>

</div>

Vue.component('todo-item', {

props: ['todo'],

template: '<li>{{ todo.text }}</li>'

})

var app7 = new Vue({

el: '#app-7',

data: {

groceryList: [

{ id: 0, text: 'Vegetables' },

{ id: 1, text: 'Cheese' },

{ id: 2, text: 'Whatever else humans are supposed to eat' }

]

}

})

This is a contrived example, but we have managed to separate our app into two smaller units, and the child is reasonably well-decoupled from the parent via the props interface. We can now further improve our <todo-item> component with more complex template and logic without affecting the parent app.

In a large application, it is necessary to divide the whole app into components to make development manageable. We will talk a lot more about components [**later in the guide**](https://vuejs.org/v2/guide/components.html), but here’s an (imaginary) example of what an app’s template might look like with components:

<div id="app">

<app-nav></app-nav>

<app-view>

<app-sidebar></app-sidebar>

<app-content></app-content>

</app-view>

</div>

### [Relation to Custom Elements](https://vuejs.org/v2/guide/#Relation-to-Custom-Elements)

You may have noticed that Vue components are very similar to **Custom Elements**, which are part of the [**Web Components Spec**](https://www.w3.org/wiki/WebComponents/). That’s because Vue’s component syntax is loosely modeled after the spec. For example, Vue components implement the [**Slot API**](https://github.com/w3c/webcomponents/blob/gh-pages/proposals/Slots-Proposal.md) and the is special attribute. However, there are a few key differences:

1. The Web Components Spec has been finalized, but is not natively implemented in every browser. Safari 10.1+, Chrome 54+ and Firefox 63+ natively support web components. In comparison, Vue components don’t require any polyfills and work consistently in all supported browsers (IE9 and above). When needed, Vue components can also be wrapped inside a native custom element.
2. Vue components provide important features that are not available in plain custom elements, most notably cross-component data flow, custom event communication and build tool integrations.

Although Vue doesn’t use custom elements internally, it has [**great interoperability**](https://custom-elements-everywhere.com/#vue) when it comes to consuming or distributing as custom elements. Vue CLI also supports building Vue components that register themselves as native custom elements.

## [Creating a Vue Instance](https://vuejs.org/v2/guide/instance.html#Creating-a-Vue-Instance)

Every Vue application starts by creating a new **Vue instance** with the Vue function:

var vm = new Vue({

// options

})

Although not strictly associated with the [**MVVM pattern**](https://en.wikipedia.org/wiki/Model_View_ViewModel), Vue’s design was partly inspired by it. As a convention, we often use the variable vm (short for ViewModel) to refer to our Vue instance.

When you create a Vue instance, you pass in an **options object**. The majority of this guide describes how you can use these options to create your desired behavior. For reference, you can also browse the full list of options in the [**API reference**](https://vuejs.org/v2/api/#Options-Data).

A Vue application consists of a **root Vue instance** created with new Vue, optionally organized into a tree of nested, reusable components. For example, a todo app’s component tree might look like this:

Root Instance

└─ TodoList

├─ TodoItem

│ ├─ DeleteTodoButton

│ └─ EditTodoButton

└─ TodoListFooter

├─ ClearTodosButton

└─ TodoListStatistics

We’ll talk about [**the component system**](https://vuejs.org/v2/guide/components.html) in detail later. For now, just know that all Vue components are also Vue instances, and so accept the same options object (except for a few root-specific options).

## [Data and Methods](https://vuejs.org/v2/guide/instance.html#Data-and-Methods)

When a Vue instance is created, it adds all the properties found in its data object to Vue’s **reactivity system**. When the values of those properties change, the view will “react”, updating to match the new values.

// Our data object

var data = { a: 1 }

// The object is added to a Vue instance

var vm = new Vue({

data: data

})

// Getting the property on the instance

// returns the one from the original data

vm.a == data.a // => true

// Setting the property on the instance

// also affects the original data

vm.a = 2

data.a // => 2

// ... and vice-versa

data.a = 3

vm.a // => 3

When this data changes, the view will re-render. It should be noted that properties in data are only **reactive** if they existed when the instance was created. That means if you add a new property, like:

vm.b = 'hi'

Then changes to b will not trigger any view updates. If you know you’ll need a property later, but it starts out empty or non-existent, you’ll need to set some initial value. For example:

data: {

newTodoText: '',

visitCount: 0,

hideCompletedTodos: false,

todos: [],

error: null

}

The only exception to this being the use of Object.freeze(), which prevents existing properties from being changed, which also means the reactivity system can’t track changes.

var obj = {

foo: 'bar'

}

Object.freeze(obj)

new Vue({

el: '#app',

data: obj

})

<div id="app">

<p>{{ foo }}</p>

<!-- this will no longer update `foo`! -->

<button v-on:click="foo = 'baz'">Change it</button>

</div>

In addition to data properties, Vue instances expose a number of useful instance properties and methods. These are prefixed with $ to differentiate them from user-defined properties. For example:

var data = { a: 1 }

var vm = new Vue({

el: '#example',

data: data

})

vm.$data === data // => true

vm.$el === document.getElementById('example') // => true

// $watch is an instance method

vm.$watch('a', function (newValue, oldValue) {

// This callback will be called when `vm.a` changes

})

In the future, you can consult the [**API reference**](https://vuejs.org/v2/api/#Instance-Properties) for a full list of instance properties and methods.

## [Instance Lifecycle Hooks](https://vuejs.org/v2/guide/instance.html#Instance-Lifecycle-Hooks)

[**Watch a free lesson on Vue School**](https://vueschool.io/lessons/understanding-the-vuejs-lifecycle-hooks?friend=vuejs)

Each Vue instance goes through a series of initialization steps when it’s created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes. Along the way, it also runs functions called **lifecycle hooks**, giving users the opportunity to add their own code at specific stages.

For example, the [**created**](https://vuejs.org/v2/api/#created) hook can be used to run code after an instance is created:

new Vue({

data: {

a: 1

},

created: function () {

// `this` points to the vm instance

console.log('a is: ' + this.a)

}

})

// => "a is: 1"

There are also other hooks which will be called at different stages of the instance’s lifecycle, such as [**mounted**](https://vuejs.org/v2/api/#mounted), [**updated**](https://vuejs.org/v2/api/#updated), and [**destroyed**](https://vuejs.org/v2/api/#destroyed). All lifecycle hooks are called with their this context pointing to the Vue instance invoking it.

Don’t use [**arrow functions**](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions) on an options property or callback, such as created: () => console.log(this.a) or vm.$watch('a', newValue => this.myMethod()). Since an arrow function doesn’t have a this, this will be treated as any other variable and lexically looked up through parent scopes until found, often resulting in errors such as Uncaught TypeError: Cannot read property of undefined or Uncaught TypeError: this.myMethod is not a function.

## [Lifecycle Diagram](https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram)

Below is a diagram for the instance lifecycle. You don’t need to fully understand everything going on right now, but as you learn and build more, it will be a useful reference.



# Template Syntax

Vue.js uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying Vue instance’s data. All Vue.js templates are valid HTML that can be parsed by spec-compliant browsers and HTML parsers.

Under the hood, Vue compiles the templates into Virtual DOM render functions. Combined with the reactivity system, Vue is able to intelligently figure out the minimal number of components to re-render and apply the minimal amount of DOM manipulations when the app state changes.

If you are familiar with Virtual DOM concepts and prefer the raw power of JavaScript, you can also [**directly write render functions**](https://vuejs.org/v2/guide/render-function.html) instead of templates, with optional JSX support.

## [Interpolations](https://vuejs.org/v2/guide/syntax.html#Interpolations)

### [Text](https://vuejs.org/v2/guide/syntax.html#Text)

The most basic form of data binding is text interpolation using the “Mustache” syntax (double curly braces):

<span>Message: {{ msg }}</span>

The mustache tag will be replaced with the value of the msg property on the corresponding data object. It will also be updated whenever the data object’s msgproperty changes.

You can also perform one-time interpolations that do not update on data change by using the [**v-once directive**](https://vuejs.org/v2/api/#v-once), but keep in mind this will also affect any other bindings on the same node:

<span v-once>This will never change: {{ msg }}</span>

### [Raw HTML](https://vuejs.org/v2/guide/syntax.html#Raw-HTML)

The double mustaches interprets the data as plain text, not HTML. In order to output real HTML, you will need to use the [**v-html directive**](https://vuejs.org/v2/api/#v-html):

<p>Using mustaches: {{ rawHtml }}</p>

<p>Using v-html directive: <span v-html="rawHtml"></span></p>

Using mustaches: <span style="color: red">This should be red.</span>

Using v-html directive: This should be red.

The contents of the span will be replaced with the value of the rawHtml property, interpreted as plain HTML - data bindings are ignored. Note that you cannot use v-htmlto compose template partials, because Vue is not a string-based templating engine. Instead, components are preferred as the fundamental unit for UI reuse and composition.

Dynamically rendering arbitrary HTML on your website can be very dangerous because it can easily lead to [**XSS vulnerabilities**](https://en.wikipedia.org/wiki/Cross-site_scripting). Only use HTML interpolation on trusted content and **never** on user-provided content.

### [Attributes](https://vuejs.org/v2/guide/syntax.html#Attributes)

Mustaches cannot be used inside HTML attributes. Instead, use a [**v-bind directive**](https://vuejs.org/v2/api/#v-bind):

<div v-bind:id="dynamicId"></div>

In the case of boolean attributes, where their mere existence implies true, v-bindworks a little differently. In this example:

<button v-bind:disabled="isButtonDisabled">Button</button>

If isButtonDisabled has the value of null, undefined, or false, the disabled attribute will not even be included in the rendered <button> element.

### [Using JavaScript Expressions](https://vuejs.org/v2/guide/syntax.html#Using-JavaScript-Expressions)

So far we’ve only been binding to simple property keys in our templates. But Vue.js actually supports the full power of JavaScript expressions inside all data bindings:

{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

<div v-bind:id="'list-' + id"></div>

These expressions will be evaluated as JavaScript in the data scope of the owner Vue instance. One restriction is that each binding can only contain **one single expression**, so the following will **NOT** work:

<!-- this is a statement, not an expression: -->

{{ var a = 1 }}

<!-- flow control won't work either, use ternary expressions -->

{{ if (ok) { return message } }}

Template expressions are sandboxed and only have access to a [**whitelist of globals**](https://github.com/vuejs/vue/blob/v2.6.10/src/core/instance/proxy.js#L9)such as Math and Date. You should not attempt to access user defined globals in template expressions.

## [Directives](https://vuejs.org/v2/guide/syntax.html#Directives)

Directives are special attributes with the v- prefix. Directive attribute values are expected to be **a single JavaScript expression** (with the exception of v-for, which will be discussed later). A directive’s job is to reactively apply side effects to the DOM when the value of its expression changes. Let’s review the example we saw in the introduction:

<p v-if="seen">Now you see me</p>

Here, the v-if directive would remove/insert the <p> element based on the truthiness of the value of the expression seen.

### [Arguments](https://vuejs.org/v2/guide/syntax.html#Arguments)

Some directives can take an “argument”, denoted by a colon after the directive name. For example, the v-bind directive is used to reactively update an HTML attribute:

<a v-bind:href="url"> ... </a>

Here href is the argument, which tells the v-bind directive to bind the element’s href attribute to the value of the expression url.

Another example is the v-on directive, which listens to DOM events:

<a v-on:click="doSomething"> ... </a>

Here the argument is the event name to listen to. We will talk about event handling in more detail too.

### [Dynamic Arguments](https://vuejs.org/v2/guide/syntax.html#Dynamic-Arguments)

**New in 2.6.0+**

Starting in version 2.6.0, it is also possible to use a JavaScript expression in a directive argument by wrapping it with square brackets:

<!--

Note that there are some constraints to the argument expression, as explained

in the "Dynamic Argument Expression Constraints" section below.

-->

<a v-bind:[attributeName]="url"> ... </a>

Here attributeName will be dynamically evaluated as a JavaScript expression, and its evaluated value will be used as the final value for the argument. For example, if your Vue instance has a data property, attributeName, whose value is "href", then this binding will be equivalent to v-bind:href.

Similarly, you can use dynamic arguments to bind a handler to a dynamic event name:

<a v-on:[eventName]="doSomething"> ... </a>

In this example, when eventName‘s value is "focus", v-on:[eventName] will be equivalent to v-on:focus.

#### Dynamic Argument Value Constraints

Dynamic arguments are expected to evaluate to a string, with the exception of null. The special value null can be used to explicitly remove the binding. Any other non-string value will trigger a warning.

#### Dynamic Argument Expression Constraints

Dynamic argument expressions have some syntax constraints because certain characters, such as spaces and quotes, are invalid inside HTML attribute names. For example, the following is invalid:

<!-- This will trigger a compiler warning. -->

<a v-bind:['foo' + bar]="value"> ... </a>

The workaround is to either use expressions without spaces or quotes, or replace the complex expression with a computed property.

When using in-DOM templates (templates directly written in an HTML file), you should also avoid naming keys with uppercase characters, as browsers will coerce attribute names into lowercase:

<!--

This will be converted to v-bind:[someattr] in in-DOM templates.

Unless you have a "someattr" property in your instance, your code won't work.

-->

<a v-bind:[someAttr]="value"> ... </a>

### [Modifiers](https://vuejs.org/v2/guide/syntax.html#Modifiers)

Modifiers are special postfixes denoted by a dot, which indicate that a directive should be bound in some special way. For example, the .prevent modifier tells the v-ondirective to call event.preventDefault() on the triggered event:

<form v-on:submit.prevent="onSubmit"> ... </form>

You’ll see other examples of modifiers later, [**for v-on**](https://vuejs.org/v2/guide/events.html#Event-Modifiers) and [**for v-model**](https://vuejs.org/v2/guide/forms.html#Modifiers), when we explore those features.

The v- prefix serves as a visual cue for identifying Vue-specific attributes in your templates. This is useful when you are using Vue.js to apply dynamic behavior to some existing markup, but can feel verbose for some frequently used directives. At the same time, the need for the v- prefix becomes less important when you are building a [**SPA**](https://en.wikipedia.org/wiki/Single-page_application), where Vue manages every template. Therefore, Vue provides special shorthands for two of the most often used directives, v-bind and v-on:

### [v-bind Shorthand](https://vuejs.org/v2/guide/syntax.html#v-bind-Shorthand)

<!-- full syntax -->

<a v-bind:href="url"> ... </a>

<!-- shorthand -->

<a :href="url"> ... </a>

<!-- shorthand with dynamic argument (2.6.0+) -->

<a :[key]="url"> ... </a>

### [v-on Shorthand](https://vuejs.org/v2/guide/syntax.html#v-on-Shorthand)

<!-- full syntax -->

<a v-on:click="doSomething"> ... </a>

<!-- shorthand -->

<a @click="doSomething"> ... </a>

<!-- shorthand with dynamic argument (2.6.0+) -->

<a @[event]="doSomething"> ... </a>

They may look a bit different from normal HTML, but : and @ are valid characters for attribute names and all Vue-supported browsers can parse it correctly. In addition, they do not appear in the final rendered markup. The shorthand syntax is totally optional, but you will likely appreciate it when you learn more about its usage later.

# Computed Properties and Watchers

## [Computed Properties](https://vuejs.org/v2/guide/computed.html#Computed-Properties)

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:

<div id="example">

{{ message.split('').reverse().join('') }}

</div>

At this point, the template is no longer simple and declarative. You have to look at it for a second before realizing that it displays message in reverse. The problem is made worse when you want to include the reversed message in your template more than once.

That’s why for any complex logic, you should use a **computed property**.

### [Basic Example](https://vuejs.org/v2/guide/computed.html#Basic-Example)

<div id="example">

<p>Original message: "{{ message }}"</p>

<p>Computed reversed message: "{{ reversedMessage }}"</p>

</div>

var vm = new Vue({

el: '#example',

data: {

message: 'Hello'

},

computed: {

// a computed getter

reversedMessage: function () {

// `this` points to the vm instance

return this.message.split('').reverse().join('')

}

}

})

Result:

Original message: "Hello"

Computed reversed message: "olleH"

Here we have declared a computed property reversedMessage. The function we provided will be used as the getter function for the property vm.reversedMessage:

console.log(vm.reversedMessage) // => 'olleH'

vm.message = 'Goodbye'

console.log(vm.reversedMessage) // => 'eybdooG'

You can open the console and play with the example vm yourself. The value of vm.reversedMessage is always dependent on the value of vm.message.

You can data-bind to computed properties in templates just like a normal property. Vue is aware that vm.reversedMessage depends on vm.message, so it will update any bindings that depend on vm.reversedMessage when vm.message changes. And the best part is that we’ve created this dependency relationship declaratively: the computed getter function has no side effects, which makes it easier to test and understand.

### [Computed Caching vs Methods](https://vuejs.org/v2/guide/computed.html#Computed-Caching-vs-Methods)

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

<p>Reversed message: "{{ reverseMessage() }}"</p>

// in component

methods: {

reverseMessage: function () {

return this.message.split('').reverse().join('')

}

}

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 message has not changed, multiple access to the reversedMessage computed property will immediately return the previously computed result without having to run the function again.

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

computed: {

now: function () {

return 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 **A**, 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 **A**. Without caching, we would be executing **A**’s getter many more times than necessary! In cases where you do not want caching, use a method instead.

### [Computed vs Watched Property](https://vuejs.org/v2/guide/computed.html#Computed-vs-Watched-Property)

Vue does provide a more generic way to observe and react to data changes on a Vue instance: **watch properties**. When you have some data that needs to change based on some other data, it is tempting to overuse watch - especially if you are coming from an AngularJS background. However, it is often a better idea to use a computed property rather than an imperative watch callback. Consider this example:

<div id="demo">{{ fullName }}</div>

var vm = new Vue({

el: '#demo',

data: {

firstName: 'Foo',

lastName: 'Bar',

fullName: 'Foo Bar'

},

watch: {

firstName: function (val) {

this.fullName = val + ' ' + this.lastName

},

lastName: function (val) {

this.fullName = this.firstName + ' ' + val

}

}

})

The above code is imperative and repetitive. Compare it with a computed property version:

var vm = new Vue({

el: '#demo',

data: {

firstName: 'Foo',

lastName: 'Bar'

},

computed: {

fullName: function () {

return this.firstName + ' ' + this.lastName

}

}

})

Much better, isn’t it?

### [Computed Setter](https://vuejs.org/v2/guide/computed.html#Computed-Setter)

Computed properties are by default getter-only, but you can also provide a setter when you need it:

// ...

computed: {

fullName: {

// getter

get: function () {

return this.firstName + ' ' + this.lastName

},

// setter

set: function (newValue) {

var names = newValue.split(' ')

this.firstName = names[0]

this.lastName = names[names.length - 1]

}

}

}

// ...

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

## [Watchers](https://vuejs.org/v2/guide/computed.html#Watchers)

While computed properties are more appropriate in most cases, there are times when a custom watcher is necessary. That’s why Vue provides a more generic way to react to data changes through the watch option. This is most useful when you want to perform asynchronous or expensive operations in response to changing data.

For example:

<div id="watch-example">

<p>

Ask a yes/no question:

<input v-model="question">

</p>

<p>{{ answer }}</p>

</div>

<!-- Since there is already a rich ecosystem of ajax libraries -->

<!-- and collections of general-purpose utility methods, Vue core -->

<!-- is able to remain small by not reinventing them. This also -->

<!-- gives you the freedom to use what you're familiar with. -->

<script src="https://cdn.jsdelivr.net/npm/axios@0.12.0/dist/axios.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/lodash@4.13.1/lodash.min.js"></script>

<script>

var watchExampleVM = new Vue({

el: '#watch-example',

data: {

question: '',

answer: 'I cannot give you an answer until you ask a question!'

},

watch: {

// whenever question changes, this function will run

question: function (newQuestion, oldQuestion) {

this.answer = 'Waiting for you to stop typing...'

this.debouncedGetAnswer()

}

},

created: function () {

// \_.debounce is a function provided by lodash to limit how

// often a particularly expensive operation can be run.

// In this case, we want to limit how often we access

// yesno.wtf/api, waiting until the user has completely

// finished typing before making the ajax request. To learn

// more about the \_.debounce function (and its cousin

// \_.throttle), visit: https://lodash.com/docs#debounce

this.debouncedGetAnswer = \_.debounce(this.getAnswer, 500)

},

methods: {

getAnswer: function () {

if (this.question.indexOf('?') === -1) {

this.answer = 'Questions usually contain a question mark. ;-)'

return

}

this.answer = 'Thinking...'

var vm = this

axios.get('https://yesno.wtf/api')

.then(function (response) {

vm.answer = \_.capitalize(response.data.answer)

})

.catch(function (error) {

vm.answer = 'Error! Could not reach the API. ' + error

})

}

}

})

</script>

In this case, using the watch option allows us to perform an asynchronous operation (accessing an API), limit how often we perform that operation, and set intermediary states until we get a final answer. None of that would be possible with a computed property.

In addition to the watch option, you can also use the imperative **[vm.$watch API](https://vuejs.org/v2/api/" \l "vm-watch)**.

**Class and Style Bindings**

A common need for data binding is manipulating an element’s class list and its inline styles. Since they are both attributes, we can use v-bind to handle them: we only need to calculate a final string with our expressions. However, meddling with string concatenation is 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.

### [Object Syntax](https://vuejs.org/v2/guide/class-and-style.html#Object-Syntax)

We can pass an object to v-bind:class to dynamically toggle classes:

<div v-bind:class="{ active: isActive }"></div>

The above syntax means the presence of the active class will be determined by the [**truthiness**](https://developer.mozilla.org/en-US/docs/Glossary/Truthy) of the data property isActive.

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

<div

class="static"

v-bind:class="{ active: isActive, 'text-danger': hasError }"

></div>

And the following data:

data: {

isActive: true,

hasError: false

}

It will render:

<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:

<div v-bind:class="classObject"></div>

data: {

classObject: {

active: true,

'text-danger': false

}

}

This will render the same result. We can also bind to a [**computed property**](https://vuejs.org/v2/guide/computed.html) that returns an object. This is a common and powerful pattern:

<div v-bind:class="classObject"></div>

data: {

isActive: true,

error: null

},

computed: {

classObject: function () {

return {

active: this.isActive && !this.error,

'text-danger': this.error && this.error.type === 'fatal'

}

}

}

### [Array Syntax](https://vuejs.org/v2/guide/class-and-style.html#Array-Syntax)

We can pass an array to v-bind:class to apply a list of classes:

<div v-bind:class="[activeClass, errorClass]"></div>

data: {

activeClass: 'active',

errorClass: 'text-danger'

}

Which will render:

<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:

<div v-bind:class="[isActive ? activeClass : '', errorClass]"></div>

This will always apply errorClass, but will only apply activeClass 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 array syntax:

<div v-bind:class="[{ active: isActive }, errorClass]"></div>

### [With Components](https://vuejs.org/v2/guide/class-and-style.html#With-Components)

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

When you use the class attribute on a custom component, those classes will be added to the component’s root element. Existing classes on this element will not be overwritten.

For example, if you declare this component:

Vue.component('my-component', {

template: '<p class="foo bar">Hi</p>'

})

Then add some classes when using it:

<my-component class="baz boo"></my-component>

The rendered HTML will be:

<p class="foo bar baz boo">Hi</p>

The same is true for class bindings:

<my-component v-bind:class="{ active: isActive }"></my-component>

When isActive is truthy, the rendered HTML will be:

<p class="foo bar active">Hi</p>

## [Binding Inline Styles](https://vuejs.org/v2/guide/class-and-style.html#Binding-Inline-Styles)

### [Object Syntax](https://vuejs.org/v2/guide/class-and-style.html#Object-Syntax-1)

The object syntax for v-bind:style is pretty straightforward - it looks almost like CSS, except it’s a JavaScript object. You can use either camelCase or kebab-case (use quotes with kebab-case) for the CSS property names:

<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>

data: {

activeColor: 'red',

fontSize: 30

}

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

<div v-bind:style="styleObject"></div>

data: {

styleObject: {

color: 'red',

fontSize: '13px'

}

}

Again, the object syntax is often used in conjunction with computed properties that return objects.

### [Array Syntax](https://vuejs.org/v2/guide/class-and-style.html#Array-Syntax-1)

The array syntax for v-bind:style allows you to apply multiple style objects to the same element:

<div v-bind:style="[baseStyles, overridingStyles]"></div>

### [Auto-prefixing](https://vuejs.org/v2/guide/class-and-style.html#Auto-prefixing)

When you use a CSS property that requires [**vendor prefixes**](https://developer.mozilla.org/en-US/docs/Glossary/Vendor_Prefix) in v-bind:style, for example transform, Vue will automatically detect and add appropriate prefixes to the applied styles.

### [Multiple Values](https://vuejs.org/v2/guide/class-and-style.html#Multiple-Values)

**2.3.0+**

Starting in 2.3.0+ you can provide an array of multiple (prefixed) values to a style property, for example:

<div v-bind: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.

**Conditional Rendering**

## [v-if](https://vuejs.org/v2/guide/conditional.html#v-if)

The directive v-if is used to conditionally render a block. The block will only be rendered if the directive’s expression returns a truthy value.

<h1 v-if="awesome">Vue is awesome!</h1>

It is also possible to add an “else block” with v-else:

<h1 v-if="awesome">Vue is awesome!</h1>

<h1 v-else>Oh no 😢</h1>

### [Conditional Groups with v-if on <template>](https://vuejs.org/v2/guide/conditional.html#Conditional-Groups-with-v-if-on-lt-template-gt)

Because v-if is a directive, it has to be attached to a single element. But what if we want to toggle more than one element? In this case we can use v-if on a <template>element, which serves as an invisible wrapper. The final rendered result will not include the <template> element.

<template v-if="ok">

<h1>Title</h1>

<p>Paragraph 1</p>

<p>Paragraph 2</p>

</template>

### [v-else](https://vuejs.org/v2/guide/conditional.html#v-else)

You can use the v-else directive to indicate an “else block” for v-if:

<div v-if="Math.random() > 0.5">

Now you see me

</div>

<div v-else>

Now you don't

</div>

A v-else element must immediately follow a v-if or a v-else-if element - otherwise it will not be recognized.

### [v-else-if](https://vuejs.org/v2/guide/conditional.html#v-else-if)

**New in 2.1.0+**

The v-else-if, as the name suggests, serves as an “else if block” for v-if. It can also be chained multiple times:

<div v-if="type === 'A'">

A

</div>

<div v-else-if="type === 'B'">

B

</div>

<div v-else-if="type === 'C'">

C

</div>

<div v-else>

Not A/B/C

</div>

Similar to v-else, a v-else-if element must immediately follow a v-if or a v-else-if element.

### [Controlling Reusable Elements with key](https://vuejs.org/v2/guide/conditional.html#Controlling-Reusable-Elements-with-key)

Vue tries to render elements as efficiently as possible, often re-using them instead of rendering from scratch. Beyond helping make Vue very fast, this can have some useful advantages. For example, if you allow users to toggle between multiple login types:

<template v-if="loginType === 'username'">

<label>Username</label>

<input placeholder="Enter your username">

</template>

<template v-else>

<label>Email</label>

<input placeholder="Enter your email address">

</template>

Then switching the loginType in the code above will not erase what the user has already entered. Since both templates use the same elements, the <input> is not replaced - just its placeholder.

This isn’t always desirable though, so Vue offers a way for you to say, “These two elements are completely separate - don’t re-use them.” Add a key attribute with unique values:

<template v-if="loginType === 'username'">

<label>Username</label>

<input placeholder="Enter your username" key="username-input">

</template>

<template v-else>

<label>Email</label>

<input placeholder="Enter your email address" key="email-input">

</template>

Now those inputs will be rendered from scratch each time you toggle. See for yourself:

Username

Toggle login type

Note that the <label> elements are still efficiently re-used, because they don’t have key attributes.

## [v-show](https://vuejs.org/v2/guide/conditional.html#v-show)

Another option for conditionally displaying an element is the v-show directive. The usage is largely the same:

<h1 v-show="ok">Hello!</h1>

The difference is that an element with v-show will always be rendered and remain in the DOM; v-show only toggles the display CSS property of the element.

Note that v-show doesn’t support the <template> element, nor does it work with v-else.

## [v-if vs v-show](https://vuejs.org/v2/guide/conditional.html#v-if-vs-v-show)

v-if is “real” conditional rendering because it ensures that event listeners and child components inside the conditional block are properly destroyed and re-created during toggles.

v-if is also **lazy**: if the condition is false on initial render, it will not do anything - the conditional block won’t be rendered until the condition becomes true for the first time.

In comparison, v-show is much simpler - the element is always rendered regardless of initial condition, with CSS-based toggling.

Generally speaking, v-if has higher toggle costs while v-show has higher initial render costs. So prefer v-show if you need to toggle something very often, and prefer v-if if the condition is unlikely to change at runtime.

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

Using v-if and v-for together is **not recommended**. See the [**style guide**](https://vuejs.org/v2/style-guide/#Avoid-v-if-with-v-for-essential) for further information.

When used together with v-if, v-for has a higher priority than v-if. See the [**list rendering guide**](https://vuejs.org/v2/guide/list.html#v-for-with-v-if) for details.

# 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' }

]

}

})

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: {

title: 'How to do lists in Vue',

author: 'Jane Doe',

publishedAt: '2016-04-10'

}

}

})

Result:

* How to do lists in Vue
* Jane Doe
* 2016-04-10

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

<div v-for="(value, name) in object">

{{ name }}: {{ value }}

</div>

title: How to do lists in Vue

author: Jane Doe

publishedAt: 2016-04-10

And another for the index:

<div v-for="(value, name, index) in object">

{{ index }}. {{ name }}: {{ value }}

</div>

0. title: How to do lists in Vue

1. author: Jane Doe

2. publishedAt: 2016-04-10

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

## [Maintaining State](https://vuejs.org/v2/guide/list.html#Maintaining-State)

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:

<div v-for="item in items" v-bind:key="item.id">

<!-- content -->

</div>

It is recommended to provide a key attribute 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.

Don’t use non-primitive values like objects and arrays as v-for keys. Use string or numeric values instead.

For detailed usage of the key attribute, please see the [**key API documentation**](https://vuejs.org/v2/api/#key).

## [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()
* pop()
* shift()
* unshift()
* splice()
* sort()
* reverse()

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, propertyName, 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:

<ul v-for="set in sets">

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

</ul>

data: {

sets: [[ 1, 2, 3, 4, 5 ], [6, 7, 8, 9, 10]]

},

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)

Note that it’s **not** recommended to use v-if and v-for together. Refer to [**style guide**](https://vuejs.org/v2/style-guide/#Avoid-v-if-with-v-for-essential) for details.

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-if on 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 = ''

}

}

})

Top of Form

# Event Handling

## [Listening to Events](https://vuejs.org/v2/guide/events.html#Listening-to-Events)

We can use the v-on directive to listen to DOM events and run some JavaScript when they’re triggered.

For example:

<div id="example-1">

<button v-on:click="counter += 1">Add 1</button>

<p>The button above has been clicked {{ counter }} times.</p>

</div>

var example1 = new Vue({

el: '#example-1',

data: {

counter: 0

}

})

Result:

Add 1

The button above has been clicked 0 times.

## [Method Event Handlers](https://vuejs.org/v2/guide/events.html#Method-Event-Handlers)

The logic for many event handlers will be more complex though, so keeping your JavaScript in the value of the v-on attribute isn’t feasible. That’s why v-on can also accept the name of a method you’d like to call.

For example:

<div id="example-2">

<!-- `greet` is the name of a method defined below -->

<button v-on:click="greet">Greet</button>

</div>

var example2 = new Vue({

el: '#example-2',

data: {

name: 'Vue.js'

},

// define methods under the `methods` object

methods: {

greet: function (event) {

// `this` inside methods points to the Vue instance

alert('Hello ' + this.name + '!')

// `event` is the native DOM event

if (event) {

alert(event.target.tagName)

}

}

}

})

// you can invoke methods in JavaScript too

example2.greet() // => 'Hello Vue.js!'

Result:

Greet

## [Methods in Inline Handlers](https://vuejs.org/v2/guide/events.html#Methods-in-Inline-Handlers)

Instead of binding directly to a method name, we can also use methods in an inline JavaScript statement:

<div id="example-3">

<button v-on:click="say('hi')">Say hi</button>

<button v-on:click="say('what')">Say what</button>

</div>

new Vue({

el: '#example-3',

methods: {

say: function (message) {

alert(message)

}

}

})

Result:

Say hi Say what

Sometimes we also need to access the original DOM event in an inline statement handler. You can pass it into a method using the special $event variable:

<button v-on:click="warn('Form cannot be submitted yet.', $event)">

Submit

</button>

// ...

methods: {

warn: function (message, event) {

// now we have access to the native event

if (event) {

event.preventDefault()

}

alert(message)

}

}

[**Event Modifiers**](https://vuejs.org/v2/guide/events.html#Event-Modifiers)

It is a very common need to call event.preventDefault() or event.stopPropagation() inside event handlers. Although we can do this easily inside methods, it would be better if the methods can be purely about data logic rather than having to deal with DOM event details.

To address this problem, Vue provides **event modifiers** for v-on. Recall that modifiers are directive postfixes denoted by a dot.

* .stop
* .prevent
* .capture
* .self
* .once
* .passive

<!-- the click event's propagation will be stopped -->

<a v-on:click.stop="doThis"></a>

<!-- the submit event will no longer reload the page -->

<form v-on:submit.prevent="onSubmit"></form>

<!-- modifiers can be chained -->

<a v-on:click.stop.prevent="doThat"></a>

<!-- just the modifier -->

<form v-on:submit.prevent></form>

<!-- use capture mode when adding the event listener -->

<!-- i.e. an event targeting an inner element is handled here before being handled by that element -->

<div v-on:click.capture="doThis">...</div>

<!-- only trigger handler if event.target is the element itself -->

<!-- i.e. not from a child element -->

<div v-on:click.self="doThat">...</div>

Order matters when using modifiers because the relevant code is generated in the same order. Therefore using v-on:click.prevent.self will prevent **all clicks** while v-on:click.self.prevent will only prevent clicks on the element itself.

**New in 2.1.4+**

<!-- the click event will be triggered at most once -->

<a v-on:click.once="doThis"></a>

Unlike the other modifiers, which are exclusive to native DOM events, the .once modifier can also be used on [**component events**](https://vuejs.org/v2/guide/components-custom-events.html). If you haven’t read about components yet, don’t worry about this for now.

**New in 2.3.0+**

Vue also offers the .passive modifier, corresponding to **[addEventListener‘s passive option](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener" \l "Parameters" \t "_blank)**.

<!-- the scroll event's default behavior (scrolling) will happen -->

<!-- immediately, instead of waiting for `onScroll` to complete -->

<!-- in case it contains `event.preventDefault()` -->

<div v-on:scroll.passive="onScroll">...</div>

The .passive modifier is especially useful for improving performance on mobile devices.

Don’t use .passive and .prevent together, because .prevent will be ignored and your browser will probably show you a warning. Remember, .passive communicates to the browser that you *don’t* want to prevent the event’s default behavior.

## [Key Modifiers](https://vuejs.org/v2/guide/events.html#Key-Modifiers)

When listening for keyboard events, we often need to check for specific keys. Vue allows adding key modifiers for v-on when listening for key events:

<!-- only call `vm.submit()` when the `key` is `Enter` -->

<input v-on:keyup.enter="submit">

You can directly use any valid key names exposed via **[KeyboardEvent.key](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key/Key_Values" \t "_blank)** as modifiers by converting them to kebab-case.

<input v-on:keyup.page-down="onPageDown">

In the above example, the handler will only be called if $event.key is equal to 'PageDown'.

### [Key Codes](https://vuejs.org/v2/guide/events.html#Key-Codes)

The use of keyCode events [**is deprecated**](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/keyCode) and may not be supported in new browsers.

Using keyCode attributes is also permitted:

<input v-on:keyup.13="submit">

Vue provides aliases for the most commonly used key codes when necessary for legacy browser support:

* .enter
* .tab
* .delete (captures both “Delete” and “Backspace” keys)
* .esc
* .space
* .up
* .down
* .left
* .right

A few keys (.esc and all arrow keys) have inconsistent key values in IE9, so these built-in aliases should be preferred if you need to support IE9.

A few keys (.esc and all arrow keys) have inconsistent key values in IE9, so these built-in aliases should be preferred if you need to support IE9.

You can also [**define custom key modifier aliases**](https://vuejs.org/v2/api/#keyCodes) via the global config.keyCodesobject:

// enable `v-on:keyup.f1`

Vue.config.keyCodes.f1 = 112

[**System Modifier Keys**](https://vuejs.org/v2/guide/events.html#System-Modifier-Keys)

**New in 2.1.0+**

You can use the following modifiers to trigger mouse or keyboard event listeners only when the corresponding modifier key is pressed:

* .ctrl
* .alt
* .shift
* .meta

**Note: On Macintosh keyboards, meta is the command key (⌘). On Windows keyboards, meta is the Windows key (⊞). On Sun Microsystems keyboards, meta is marked as a solid diamond (◆). On certain keyboards, specifically MIT and Lisp machine keyboards and successors, such as the Knight keyboard, space-cadet keyboard, meta is labeled “META”. On Symbolics keyboards, meta is labeled “META” or “Meta”.**

For example:

<!-- Alt + C -->

<input v-on:keyup.alt.67="clear">

<!-- Ctrl + Click -->

<div v-on:click.ctrl="doSomething">Do something</div>

Note that modifier keys are different from regular keys and when used with keyup events, they have to be pressed when the event is emitted. In other words, keyup.ctrl will only trigger if you release a key while holding down ctrl. It won’t trigger if you release the ctrl key alone. If you do want such behaviour, use the keyCode for ctrl instead: keyup.17.

### [.exact Modifier](https://vuejs.org/v2/guide/events.html#exact-Modifier)

**New in 2.5.0+**

The .exact modifier allows control of the exact combination of system modifiers needed to trigger an event.

<!-- this will fire even if Alt or Shift is also pressed -->

<button v-on:click.ctrl="onClick">A</button>

<!-- this will only fire when Ctrl and no other keys are pressed -->

<button v-on:click.ctrl.exact="onCtrlClick">A</button>

<!-- this will only fire when no system modifiers are pressed -->

<button v-on:click.exact="onClick">A</button>

### [Mouse Button Modifiers](https://vuejs.org/v2/guide/events.html#Mouse-Button-Modifiers)

**New in 2.2.0+**

* .left
* .right
* .middle

These modifiers restrict the handler to events triggered by a specific mouse button.

## [Why Listeners in HTML?](https://vuejs.org/v2/guide/events.html#Why-Listeners-in-HTML)

You might be concerned that this whole event listening approach violates the good old rules about “separation of concerns”. Rest assured - since all Vue handler functions and expressions are strictly bound to the ViewModel that’s handling the current view, it won’t cause any maintenance difficulty. In fact, there are several benefits in using v-on:

1. It’s easier to locate the handler function implementations within your JS code by skimming the HTML template.
2. Since you don’t have to manually attach event listeners in JS, your ViewModel code can be pure logic and DOM-free. This makes it easier to test.
3. When a ViewModel is destroyed, all event listeners are automatically removed. You don’t need to worry about cleaning it up yourself.

# Form Input Bindings

## [Basic Usage](https://vuejs.org/v2/guide/forms.html#Basic-Usage)

You can use the v-model directive to create two-way data bindings on form input, textarea, and select elements. It automatically picks the correct way to update the element based on the input type. Although a bit magical, v-model is essentially syntax sugar for updating data on user input events, plus special care for some edge cases.

v-model will ignore the initial value, checked or selected attributes found on any form elements. It will always treat the Vue instance data as the source of truth. You should declare the initial value on the JavaScript side, inside the data option of your component.

v-model internally uses different properties and emits different events for different input elements:

* text and textarea elements use value property and input event;
* checkboxes and radiobuttons use checked property and change event;
* select fields use value as a prop and change as an event.

For languages that require an [**IME**](https://en.wikipedia.org/wiki/Input_method) (Chinese, Japanese, Korean etc.), you’ll notice that v-model doesn’t get updated during IME composition. If you want to cater for these updates as well, use input event instead.

### [Text](https://vuejs.org/v2/guide/forms.html#Text)

<input v-model="message" placeholder="edit me">

<p>Message is: {{ message }}</p>

Message is:

### [Multiline text](https://vuejs.org/v2/guide/forms.html#Multiline-text)

<span>Multiline message is:</span>

<p style="white-space: pre-line;">{{ message }}</p>

<br>

<textarea v-model="message" placeholder="add multiple lines"></textarea>

Multiline message is:

Interpolation on textareas (<textarea>{{text}}</textarea>) won't work. Use v-model instead.

### [Checkbox](https://vuejs.org/v2/guide/forms.html#Checkbox)

Single checkbox, boolean value:

<input type="checkbox" id="checkbox" v-model="checked">

<label for="checkbox">{{ checked }}</label>

 false

Multiple checkboxes, bound to the same Array:

<div id='example-3'>

<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">

<label for="jack">Jack</label>

<input type="checkbox" id="john" value="John" v-model="checkedNames">

<label for="john">John</label>

<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">

<label for="mike">Mike</label>

<br>

<span>Checked names: {{ checkedNames }}</span>

</div>

new Vue({

el: '#example-3',

data: {

checkedNames: []

}

})

 Jack  John  Mike   
Checked names: []

### [Radio](https://vuejs.org/v2/guide/forms.html#Radio)

<input type="radio" id="one" value="One" v-model="picked">

<label for="one">One</label>

<br>

<input type="radio" id="two" value="Two" v-model="picked">

<label for="two">Two</label>

<br>

<span>Picked: {{ picked }}</span>

 One   
 Two   
Picked:

### [Select](https://vuejs.org/v2/guide/forms.html#Select)

Single select:

<select v-model="selected">

<option disabled value="">Please select one</option>

<option>A</option>

<option>B</option>

<option>C</option>

</select>

<span>Selected: {{ selected }}</span>

new Vue({

el: '...',

data: {

selected: ''

}

})

Please select one A B C Selected:

If the initial value of your v-model expression does not match any of the options, the <select> element will render in an “unselected” state. On iOS this will cause the user not being able to select the first item because iOS does not fire a change event in this case. It is therefore recommended to provide a disabled option with an empty value, as demonstrated in the example above.

Multiple select (bound to Array):

<select v-model="selected" multiple>

<option>A</option>

<option>B</option>

<option>C</option>

</select>

<br>

<span>Selected: {{ selected }}</span>

A B C   
Selected: []

Dynamic options rendered with v-for:

<select v-model="selected">

<option v-for="option in options" v-bind:value="option.value">

{{ option.text }}

</option>

</select>

<span>Selected: {{ selected }}</span>

new Vue({

el: '...',

data: {

selected: 'A',

options: [

{ text: 'One', value: 'A' },

{ text: 'Two', value: 'B' },

{ text: 'Three', value: 'C' }

]

}

})

       One            Two            Three      Selected: A

## [Value Bindings](https://vuejs.org/v2/guide/forms.html#Value-Bindings)

For radio, checkbox and select options, the v-model binding values are usually static strings (or booleans for checkbox):

<!-- `picked` is a string "a" when checked -->

<input type="radio" v-model="picked" value="a">

<!-- `toggle` is either true or false -->

<input type="checkbox" v-model="toggle">

<!-- `selected` is a string "abc" when the first option is selected -->

<select v-model="selected">

<option value="abc">ABC</option>

</select>

But sometimes we may want to bind the value to a dynamic property on the Vue instance. We can use v-bind to achieve that. In addition, using v-bind allows us to bind the input value to non-string values.

### [Checkbox](https://vuejs.org/v2/guide/forms.html#Checkbox-1)

<input

type="checkbox"

v-model="toggle"

true-value="yes"

false-value="no"

>

// when checked:

vm.toggle === 'yes'

// when unchecked:

vm.toggle === 'no'

The true-value and false-value attributes don’t affect the input’s value attribute, because browsers don’t include unchecked boxes in form submissions. To guarantee that one of two values is submitted in a form (e.g. “yes” or “no”), use radio inputs instead.

### [Radio](https://vuejs.org/v2/guide/forms.html#Radio-1)

<input type="radio" v-model="pick" v-bind:value="a">

// when checked:

vm.pick === vm.a

### [Select Options](https://vuejs.org/v2/guide/forms.html#Select-Options)

<select v-model="selected">

<!-- inline object literal -->

<option v-bind:value="{ number: 123 }">123</option>

</select>

// when selected:

typeof vm.selected // => 'object'

vm.selected.number // => 123

## [Modifiers](https://vuejs.org/v2/guide/forms.html#Modifiers)

### [.lazy](https://vuejs.org/v2/guide/forms.html#lazy)

By default, v-model syncs the input with the data after each input event (with the exception of IME composition as [**stated above**](https://vuejs.org/v2/guide/forms.html#vmodel-ime-tip)). You can add the lazy modifier to instead sync after change events:

<!-- synced after "change" instead of "input" -->

<input v-model.lazy="msg" >

### [.number](https://vuejs.org/v2/guide/forms.html#number)

If you want user input to be automatically typecast as a number, you can add the numbermodifier to your v-model managed inputs:

<input v-model.number="age" type="number">

This is often useful, because even with type="number", the value of HTML input elements always returns a string. If the value cannot be parsed with parseFloat(), then the original value is returned.

### [.trim](https://vuejs.org/v2/guide/forms.html#trim)

If you want whitespace from user input to be trimmed automatically, you can add the trim modifier to your v-model-managed inputs:

<input v-model.trim="msg">

## [v-model with Components](https://vuejs.org/v2/guide/forms.html#v-model-with-Components)

**If you’re not yet familiar with Vue’s components, you can skip this for now.**

HTML’s built-in input types won’t always meet your needs. Fortunately, Vue components allow you to build reusable inputs with completely customized behavior. These inputs even work with v-model! To learn more, read about [**custom inputs**](https://vuejs.org/v2/guide/components.html#Using-v-model-on-Components) in the Components guide.

# Components Basics

## [Base Example](https://vuejs.org/v2/guide/components.html#Base-Example)

Here’s an example of a Vue component:

// Define a new component called button-counter

Vue.component('button-counter', {

data: function () {

return {

count: 0

}

},

template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'

})

Components are reusable Vue instances with a name: in this case, <button-counter>. We can use this component as a custom element inside a root Vue instance created with new Vue:

<div id="components-demo">

<button-counter></button-counter>

</div>

new Vue({ el: '#components-demo' })

You clicked me 0 times.

Since components are reusable Vue instances, they accept the same options as new Vue, such as data, computed, watch, methods, and lifecycle hooks. The only exceptions are a few root-specific options like el.

## [Reusing Components](https://vuejs.org/v2/guide/components.html#Reusing-Components)

Components can be reused as many times as you want:

<div id="components-demo">

<button-counter></button-counter>

<button-counter></button-counter>

<button-counter></button-counter>

</div>

You clicked me 0 times. You clicked me 0 times. You clicked me 0 times.

Notice that when clicking on the buttons, each one maintains its own, separate count. That’s because each time you use a component, a new **instance** of it is created.

### [data Must Be a Function](https://vuejs.org/v2/guide/components.html#data-Must-Be-a-Function)

When we defined the <button-counter> component, you may have noticed that data wasn’t directly provided an object, like this:

data: {

count: 0

}

Instead, **a component’s data option must be a function**, so that each instance can maintain an independent copy of the returned data object:

data: function () {

return {

count: 0

}

}

If Vue didn’t have this rule, clicking on one button would affect the data of *all other instances*, like below:

You clicked me 0 times. You clicked me 0 times. You clicked me 0 times.

## [Organizing Components](https://vuejs.org/v2/guide/components.html#Organizing-Components)

It’s common for an app to be organized into a tree of nested components:



For example, you might have components for a header, sidebar, and content area, each typically containing other components for navigation links, blog posts, etc.

To use these components in templates, they must be registered so that Vue knows about them. There are two types of component registration: **global** and **local**. So far, we’ve only registered components globally, using Vue.component:

Vue.component('my-component-name', {

// ... options ...

})

Globally registered components can be used in the template of any root Vue instance (new Vue) created afterwards – and even inside all subcomponents of that Vue instance’s component tree.

That’s all you need to know about registration for now, but once you’ve finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on [**Component Registration**](https://vuejs.org/v2/guide/components-registration.html).

## [Passing Data to Child Components with Props](https://vuejs.org/v2/guide/components.html#Passing-Data-to-Child-Components-with-Props)

Earlier, we mentioned creating a component for blog posts. The problem is, that component won’t be useful unless you can pass data to it, such as the title and content of the specific post we want to display. That’s where props come in.

Props are custom attributes you can register on a component. When a value is passed to a prop attribute, it becomes a property on that component instance. To pass a title to our blog post component, we can include it in the list of props this component accepts, using a props option:

Vue.component('blog-post', {

props: ['title'],

template: '<h3>{{ title }}</h3>'

})

A component can have as many props as you’d like and by default, any value can be passed to any prop. In the template above, you’ll see that we can access this value on the component instance, just like with data.

Once a prop is registered, you can pass data to it as a custom attribute, like this:

<blog-post title="My journey with Vue"></blog-post>

<blog-post title="Blogging with Vue"></blog-post>

<blog-post title="Why Vue is so fun"></blog-post>

### My journey with Vue

### Blogging with Vue

### Why Vue is so fun

In a typical app, however, you’ll likely have an array of posts in data:

new Vue({

el: '#blog-post-demo',

data: {

posts: [

{ id: 1, title: 'My journey with Vue' },

{ id: 2, title: 'Blogging with Vue' },

{ id: 3, title: 'Why Vue is so fun' }

]

}

})

Then want to render a component for each one:

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:title="post.title"

></blog-post>

Above, you’ll see that we can use v-bind to dynamically pass props. This is especially useful when you don’t know the exact content you’re going to render ahead of time, like when [**fetching posts from an API**](https://codesandbox.io/s/github/vuejs/vuejs.org/tree/master/src/v2/examples/vue-20-component-blog-post-example).

That’s all you need to know about props for now, but once you’ve finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on [**Props**](https://vuejs.org/v2/guide/components-props.html).

## [A Single Root Element](https://vuejs.org/v2/guide/components.html#A-Single-Root-Element)

When building out a <blog-post> component, your template will eventually contain more than just the title:

<h3>{{ title }}</h3>

At the very least, you’ll want to include the post’s content:

<h3>{{ title }}</h3>

<div v-html="content"></div>

If you try this in your template however, Vue will show an error, explaining that **every component must have a single root element**. You can fix this error by wrapping the template in a parent element, such as:

<div class="blog-post">

<h3>{{ title }}</h3>

<div v-html="content"></div>

</div>

As our component grows, it’s likely we’ll not only need the title and content of a post, but also the published date, comments, and more. Defining a prop for each related piece of information could become very annoying:

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:title="post.title"

v-bind:content="post.content"

v-bind:publishedAt="post.publishedAt"

v-bind:comments="post.comments"

></blog-post>

So this might be a good time to refactor the <blog-post> component to accept a single post prop instead:

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:post="post"

></blog-post>

Vue.component('blog-post', {

props: ['post'],

template: `

<div class="blog-post">

<h3>{{ post.title }}</h3>

<div v-html="post.content"></div>

</div>

`

})

The above example and some future ones use JavaScript’s [**template literal**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals) to make multi-line templates more readable. These are not supported by Internet Explorer (IE), so if you must support IE and are not transpiling (e.g. with Babel or TypeScript), use [**newline escapes**](https://css-tricks.com/snippets/javascript/multiline-string-variables-in-javascript/) instead.

Now, whenever a new property is added to post objects, it will automatically be available inside <blog-post>.

## [Listening to Child Components Events](https://vuejs.org/v2/guide/components.html#Listening-to-Child-Components-Events)

As we develop our <blog-post> component, some features may require communicating back up to the parent. For example, we may decide to include an accessibility feature to enlarge the text of blog posts, while leaving the rest of the page its default size:

In the parent, we can support this feature by adding a postFontSize data property:

new Vue({

el: '#blog-posts-events-demo',

data: {

posts: [/\* ... \*/],

postFontSize: 1

}

})

Which can be used in the template to control the font size of all blog posts:

<div id="blog-posts-events-demo">

<div :style="{ fontSize: postFontSize + 'em' }">

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:post="post"

></blog-post>

</div>

</div>

Now let’s add a button to enlarge the text right before the content of every post:

Vue.component('blog-post', {

props: ['post'],

template: `

<div class="blog-post">

<h3>{{ post.title }}</h3>

<button>

Enlarge text

</button>

<div v-html="post.content"></div>

</div>

`

})

The problem is, this button doesn’t do anything:

<button>

Enlarge text

</button>

When we click on the button, we need to communicate to the parent that it should enlarge the text of all posts. Fortunately, Vue instances provide a custom events system to solve this problem. The parent can choose to listen to any event on the child component instance with v-on, just as we would with a native DOM event:

<blog-post

...

v-on:enlarge-text="postFontSize += 0.1"

></blog-post>

Then the child component can emit an event on itself by calling the built-in [**$emitmethod**](https://vuejs.org/v2/api/#vm-emit), passing the name of the event:

<button v-on:click="$emit('enlarge-text')">

Enlarge text

</button>

Thanks to the v-on:enlarge-text="postFontSize += 0.1" listener, the parent will receive the event and update postFontSize value.

### My journey with Vue

Enlarge text

...content...

### Blogging with Vue

Enlarge text

...content...

### Why Vue is so fun

Enlarge text

...content...

### [Emitting a Value With an Event](https://vuejs.org/v2/guide/components.html#Emitting-a-Value-With-an-Event)

It’s sometimes useful to emit a specific value with an event. For example, we may want the <blog-post> component to be in charge of how much to enlarge the text by. In those cases, we can use $emit‘s 2nd parameter to provide this value:

<button v-on:click="$emit('enlarge-text', 0.1)">

Enlarge text

</button>

Then when we listen to the event in the parent, we can access the emitted event’s value with $event:

<blog-post

...

v-on:enlarge-text="postFontSize += $event"

></blog-post>

Or, if the event handler is a method:

<blog-post

...

v-on:enlarge-text="onEnlargeText"

></blog-post>

Then the value will be passed as the first parameter of that method:

methods: {

onEnlargeText: function (enlargeAmount) {

this.postFontSize += enlargeAmount

}

}

### [Using v-model on Components](https://vuejs.org/v2/guide/components.html#Using-v-model-on-Components)

Custom events can also be used to create custom inputs that work with v-model. Remember that:

<input v-model="searchText">

does the same thing as:

<input

v-bind:value="searchText"

v-on:input="searchText = $event.target.value"

>

When used on a component, v-model instead does this:

<custom-input

v-bind:value="searchText"

v-on:input="searchText = $event"

></custom-input>

For this to actually work though, the <input> inside the component must:

* Bind the value attribute to a value prop
* On input, emit its own custom input event with the new value

Here’s that in action:

Vue.component('custom-input', {

props: ['value'],

template: `

<input

v-bind:value="value"

v-on:input="$emit('input', $event.target.value)"

>

`

})

Now v-model should work perfectly with this component:

<custom-input v-model="searchText"></custom-input>

That’s all you need to know about custom component events for now, but once you’ve finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on [**Custom Events**](https://vuejs.org/v2/guide/components-custom-events.html).

## [Content Distribution with Slots](https://vuejs.org/v2/guide/components.html#Content-Distribution-with-Slots)

Just like with HTML elements, it’s often useful to be able to pass content to a component, like this:

<alert-box>

Something bad happened.

</alert-box>

Which might render something like:

**Error!** Something bad happened.

Fortunately, this task is made very simple by Vue’s custom <slot> element:

Vue.component('alert-box', {

template: `

<div class="demo-alert-box">

<strong>Error!</strong>

<slot></slot>

</div>

`

})

As you’ll see above, we just add the slot where we want it to go – and that’s it. We’re done!

That’s all you need to know about slots for now, but once you’ve finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on [**Slots**](https://vuejs.org/v2/guide/components-slots.html).

## [Dynamic Components](https://vuejs.org/v2/guide/components.html#Dynamic-Components)

Sometimes, it’s useful to dynamically switch between components, like in a tabbed interface:

HomePostsArchive

Home component

The above is made possible by Vue’s <component> element with the is special attribute:

<!-- Component changes when currentTabComponent changes -->

<component v-bind:is="currentTabComponent"></component>

In the example above, currentTabComponent can contain either:

* the name of a registered component, or
* a component’s options object

See [**this example**](https://codesandbox.io/s/github/vuejs/vuejs.org/tree/master/src/v2/examples/vue-20-dynamic-components) to experiment with the full code, or [**this version**](https://codesandbox.io/s/github/vuejs/vuejs.org/tree/master/src/v2/examples/vue-20-dynamic-components-with-binding) for an example binding to a component’s options object, instead of its registered name.

Keep in mind that this attribute can be used with regular HTML elements, however they will be treated as components, which means all attributes **will be bound as DOM attributes**. For some properties such as value to work as you would expect, you will need to bind them using the [**.prop modifier**](https://vuejs.org/v2/api/#v-bind).

That’s all you need to know about dynamic components for now, but once you’ve finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on [**Dynamic & Async Components**](https://vuejs.org/v2/guide/components-dynamic-async.html).

## [DOM Template Parsing Caveats](https://vuejs.org/v2/guide/components.html#DOM-Template-Parsing-Caveats)

Some HTML elements, such as <ul>, <ol>, <table> and <select> have restrictions on what elements can appear inside them, and some elements such as <li>, <tr>, and <option> can only appear inside certain other elements.

This will lead to issues when using components with elements that have such restrictions. For example:

<table>

<blog-post-row></blog-post-row>

</table>

The custom component <blog-post-row> will be hoisted out as invalid content, causing errors in the eventual rendered output. Fortunately, the is special attribute offers a workaround:

<table>

<tr is="blog-post-row"></tr>

</table>

It should be noted that **this limitation does not apply if you are using string templates from one of the following sources**:

* String templates (e.g. template: '...')
* [**Single-file (.vue) components**](https://vuejs.org/v2/guide/single-file-components.html)
* [**<script type="text/x-template">**](https://vuejs.org/v2/guide/components-edge-cases.html#X-Templates)

That’s all you need to know about DOM template parsing caveats for now – and actually, the end of Vue’s Essentials. Congratulations! There’s still more to learn, but first, we recommend taking a break to play with Vue yourself and build something fun.

Once you feel comfortable with the knowledge you’ve just digested, we recommend coming back to read the full guide on [**Dynamic & Async Components**](https://vuejs.org/v2/guide/components-dynamic-async.html), as well as the other pages in the Components In-Depth section of the sidebar.