**Vue.js Introduction Page 1 to 7**

**Vue.js Instance Page 8 to 13**

**Vue.js Template Page 16 to 28**

**Vue.js Event Handling Page 29 to**

**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

The easiest way to try out Vue.js is using the **JSFiddle Hello World example**. 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** 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** 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.

**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()  } }) |

**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  } }) |

Condition and loop

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** 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

User Input or Events

## 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('')  }  } }) |

Component

he 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>' }) |

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**:

|  |
| --- |
| 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' }  ]  } }) |

# The Vue Instance

## Creating a Vue Instance

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

|  |
| --- |
| var vm = new Vue({  // options }) |

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 |

## 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 dataare 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 }) |

## 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

## Interpolations

### 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 msg property changes.

You can also perform one-time interpolations that do not update on data change by using the **v-once directive**, 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

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:

|  |
| --- |
| <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.

### 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 } }} |

## Directives

Directives are special attributes with the v- prefix. Directive attribute values are expected to be **a single JavaScript expression** (with the exception for 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

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 hrefattribute 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.

### 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-on directive 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** and **for v-model**, when we explore those features.

## Shorthands

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** where Vue.js manages every template. Therefore, Vue.js provides special shorthands for two of the most often used directives, v-bind and v-on:

### v-bind Shorthand

|  |
| --- |
| <!-- full syntax --> <a v-bind:href="url"> ... </a>  <!-- shorthand --> <a :href="url"> ... </a> |

### v-on Shorthand

|  |
| --- |
| <!-- full syntax --> <a v-on:click="doSomething"> ... </a>  <!-- shorthand --> <a @click="doSomething"> ... </a> |

They may look a bit different from normal HTML, but : and @ are valid chars for attribute names and all Vue.js 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.

# Conditional Rendering

## v-if

In string templates, for example Handlebars, we would write a conditional block like this:

|  |
| --- |
| <!-- Handlebars template --> {{#if ok}}  <h1>Yes</h1> {{/if}} |

In Vue, we use the v-if directive to achieve the same:

|  |
| --- |
| <h1 v-if="ok">Yes</h1> |

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

|  |
| --- |
| <h1 v-if="ok">Yes</h1> <h1 v-else>No</h1> |

### Conditional Groups with v-if on <template>

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

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

**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

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> |

# List Rendering

## 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

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

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

### 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

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

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** 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

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

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

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>

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

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>**). 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

**This section assumes knowledge of Components. 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 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** 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 = ''  }  } }) |

# Event Handling

## 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

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

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

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**. 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 passiveoption**.

|  |
| --- |
| <!-- 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, .passivecommunicates to the browser that you don’t want to prevent the event’s default behavior.

## Key Modifiers

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

|  |
| --- |
| <!-- only call `vm.submit()` when the `keyCode` is 13 --> <input v-on:keyup.13="submit"> |

Remembering all the keyCodes is a hassle, so Vue provides aliases for the most commonly used keys:

|  |
| --- |
| <!-- same as above --> <input v-on:keyup.enter="submit">  <!-- also works for shorthand --> <input @keyup.enter="submit"> |

Here’s the full list of key modifier aliases:

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

You can also **define custom key modifier aliases** via the global config.keyCodes object:

|  |
| --- |
| // enable `v-on:keyup.f1` Vue.config.keyCodes.f1 = 112 |

### Automatic Key Modifiers

**New in 2.5.0+**

You can also directly use any valid key names exposed via **KeyboardEvent.key** as modifiers by converting them to kebab-case:

|  |
| --- |
| <input @keyup.page-down="onPageDown"> |

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

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

## 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 @keyup.alt.67="clear">  <!-- Ctrl + Click --> <div @click.ctrl="doSomething">Do something</div> |

Note that modifier keys are different from regular keys and when used with keyupevents, 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

**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 @click.ctrl="onClick">A</button>  <!-- this will only fire when Ctrl and no other keys are pressed --> <button @click.ctrl.exact="onCtrlClick">A</button>  <!-- this will only fire when no system modifiers are pressed --> <button @click.exact="onClick">A</button> |

### 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?

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

You can use the v-model directive to create two-way data bindings on form input and textarea 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.

For languages that require an **IME** (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

|  |
| --- |
| <input v-model="message" placeholder="edit me"> <p>Message is: {{ message }}</p> |



Message is:

### 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

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

|  |
| --- |
| <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

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: ''  } }) |

    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> |

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' }  ]  } }) |

 Selected: A



## 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

|  |
| --- |
| <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 valueattribute, 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

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

|  |
| --- |
| // when checked: vm.pick === vm.a |

### 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

### .lazy

By default, v-model syncs the input with the data after each input event (with the exception of IME composition as **stated above**). You can add the lazy modifier to instead sync after change events:

|  |
| --- |
| <!-- synced after "change" instead of "input" --> <input v-model.lazy="msg" > |

### .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.

### .trim

If you want 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

**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** in the Components guide.

# Components Basics

## 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

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

When we defined the <button-counter> component, you may have noticed that datawasn’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

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**.

## 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 propsoption:

|  |
| --- |
| 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**.

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**.

## 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** 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** instead.

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

## Sending Messages to Parents with 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. To emit an event to the parent, we can call the built-in **$emit method**, passing the name of the event:

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

Then on our blog post, we can listen for this event with v-on, just as we would with a native DOM event:

|  |
| --- |
| <blog-post  ...  v-on:enlarge-text="postFontSize += 0.1" ></blog-post> |

### 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

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

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**.

## 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**.

## 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 fiddle** to experiment with the full code, or **this version** for an example binding to a component’s options object, instead of its registered name.

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**.

## 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**
* **<script type="text/x-template">**

That’s all you need to know about dynamic components 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**, as well as the other pages in the Components In-Depth section of the sidebar.

# Component Registration

**This page assumes you’ve already read the Components Basics. Read that first if you are new to components.**

## Component Names

When registering a component, it will always be given a name. For example, in the global registration we’ve seen so far:

|  |
| --- |
| Vue.component('my-component-name', { /\* ... \*/ }) |

The component’s name is the first argument of Vue.component.

The name you give a component may depend on where you intend to use it. When using a component directly in the DOM (as opposed to in a string template or **single-file component**), we strongly recommend following the **W3C rules** for custom tag names (all-lowercase, must contain a hyphen). This helps you avoid conflicts with current and future HTML elements.

You can see other recommendations for component names in the **Style Guide**.

### Name Casing

You have two options when defining component names:

#### With kebab-case

|  |
| --- |
| Vue.component('my-component-name', { /\* ... \*/ }) |

When defining a component with kebab-case, you must also use kebab-case when referencing its custom element, such as in <my-component-name>.

#### With PascalCase

|  |
| --- |
| Vue.component('MyComponentName', { /\* ... \*/ }) |

When defining a component with PascalCase, you can use either case when referencing its custom element. That means both <my-component-name> and <MyComponentName> are acceptable. Note, however, that only kebab-case names are valid directly in the DOM (i.e. non-string templates).

## Global Registration

So far, we’ve only created components using Vue.component:

|  |
| --- |
| Vue.component('my-component-name', {  // ... options ... }) |

These components are **globally registered**. That means they can be used in the template of any root Vue instance (new Vue) created after registration. For example:

|  |
| --- |
| Vue.component('component-a', { /\* ... \*/ }) Vue.component('component-b', { /\* ... \*/ }) Vue.component('component-c', { /\* ... \*/ })  new Vue({ el: '#app' }) |

|  |
| --- |
| <div id="app">  <component-a></component-a>  <component-b></component-b>  <component-c></component-c> </div> |

This even applies to all subcomponents, meaning all three of these components will also be available inside each other.

## Local Registration

Global registration often isn’t ideal. For example, if you’re using a build system like Webpack, globally registering all components means that even if you stop using a component, it could still be included in your final build. This unnecessarily increases the amount of JavaScript your users have to download.

In these cases, you can define your components as plain JavaScript objects:

|  |
| --- |
| var ComponentA = { /\* ... \*/ } var ComponentB = { /\* ... \*/ } var ComponentC = { /\* ... \*/ } |

Then define the components you’d like to use in a components option:

|  |
| --- |
| new Vue({  el: '#app'  components: {  'component-a': ComponentA,  'component-b': ComponentB  } }) |

For each property in the components object, the key will be the name of the custom element, while the value will contain the options object for the component.

Note that **locally registered components are not also available in subcomponents**. For example, if you wanted ComponentA to be available in ComponentB, you’d have to use:

|  |
| --- |
| var ComponentA = { /\* ... \*/ }  var ComponentB = {  components: {  'component-a': ComponentA  },  // ... } |

Or if you’re using ES2015 modules, such as through Babel and Webpack, that might look more like:

|  |
| --- |
| import ComponentA from './ComponentA.vue'  export default {  components: {  ComponentA  },  // ... } |

Note that in ES2015+, placing a variable name like ComponentA inside an object is shorthand for ComponentA: ComponentA, meaning the name of the variable is both:

* the custom element name to use in the template, and
* the name of the variable containing the component options

## Module Systems

If you’re not using a module system with import/require, you can probably skip this section for now. If you are, we have some special instructions and tips just for you.

### Local Registration in a Module System

If you’re still here, then it’s likely you’re using a module system, such as with Babel and Webpack. In these cases, we recommend creating a components directory, with each component in its own file.

Then you’ll need to import each component you’d like to use, before you locally register it. For example, in a hypothetical ComponentB.js or ComponentB.vue file:

|  |
| --- |
| import ComponentA from './ComponentA' import ComponentC from './ComponentC'  export default {  components: {  ComponentA,  ComponentC  },  // ... } |

Now both ComponentA and ComponentC can be used inside ComponentB‘s template.

### Automatic Global Registration of Base Components

Many of your components will be relatively generic, possibly only wrapping an element like an input or a button. We sometimes refer to these as **base components** and they tend to be used very frequently across your components.

The result is that many components may include long lists of base components:

|  |
| --- |
| import BaseButton from './BaseButton.vue' import BaseIcon from './BaseIcon.vue' import BaseInput from './BaseInput.vue'  export default {  components: {  BaseButton,  BaseIcon,  BaseInput  } } |

Just to support relatively little markup in a template:

|  |
| --- |
| <BaseInput  v-model="searchText"  @keydown.enter="search" /> <BaseButton @click="search">  <BaseIcon name="search"/> </BaseButton> |

Fortunately, if you’re using Webpack (or **Vue CLI 3+**, which uses Webpack internally), you can use require.context to globally register only these very common base components. Here’s an example of the code you might use to globally import base components in your app’s entry file (e.g. src/main.js):

|  |
| --- |
| import Vue from 'vue' import upperFirst from 'lodash/upperFirst' import camelCase from 'lodash/camelCase'  const requireComponent = require.context(  // The relative path of the components folder  './components',  // Whether or not to look in subfolders  false,  // The regular expression used to match base component filenames  /Base[A-Z]\w+\.(vue|js)$/ )  requireComponent.keys().forEach(fileName => {  // Get component config  const componentConfig = requireComponent(fileName)   // Get PascalCase name of component  const componentName = upperFirst(  camelCase(  // Strip the leading `'./` and extension from the filename  fileName.replace(/^\.\/(.\*)\.\w+$/, '$1')  )  )   // Register component globally  Vue.component(  componentName,  // Look for the component options on `.default`, which will  // exist if the component was exported with `export default`,  // otherwise fall back to module's root.  componentConfig.default || componentConfig  ) }) |

Remember that **global registration must take place before the root Vue instance is created (with new Vue)**. **Here’s an example** of this pattern in a real project context.

# Props

**This page assumes you’ve already read the Components Basics. Read that first if you are new to components.**

## Prop Casing (camelCase vs kebab-case)

HTML attribute names are case-insensitive, so browsers will interpret any uppercase characters as lowercase. That means when you’re using in-DOM templates, camelCased prop names need to use their kebab-cased (hyphen-delimited) equivalents:

|  |
| --- |
| Vue.component('blog-post', {  // camelCase in JavaScript  props: ['postTitle'],  template: '<h3>{{ postTitle }}</h3>' }) |

|  |
| --- |
| <!-- kebab-case in HTML --> <blog-post post-title="hello!"></blog-post> |

Again, if you’re using string templates, this limitation does not apply.

## Prop Types

So far, we’ve only seen props listed as an array of strings:

|  |
| --- |
| props: ['title', 'likes', 'isPublished', 'commentIds', 'author'] |

Usually though, you’ll want every prop to be a specific type of value. In these cases, you can list props as an object, where the properties’ names and values contain the prop names and types, respectively:

|  |
| --- |
| props: {  title: String,  likes: Number,  isPublished: Boolean,  commentIds: Array,  author: Object } |

This not only documents your component, but will also warn users in the browser’s JavaScript console if they pass the wrong type. You’ll learn much more about **type checks and other prop validations** further down this page.

## Passing Static or Dynamic Props

So far, you’ve seen props passed a static value, like in:

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

You’ve also seen props assigned dynamically with v-bind, such as in:

|  |
| --- |
| <!-- Dynamically assign the value of a variable --> <blog-post v-bind:title="post.title"></blog-post>  <!-- Dynamically assign the value of a complex expression --> <blog-post v-bind:title="post.title + ' by ' + post.author.name"></blog-post> |

In the two examples above, we happen to pass string values, but any type of value can actually be passed to a prop.

### Passing a Number

|  |
| --- |
| <!-- Even though `42` is static, we need v-bind to tell Vue that --> <!-- this is a JavaScript expression rather than a string. --> <blog-post v-bind:likes="42"></blog-post>  <!-- Dynamically assign to the value of a variable. --> <blog-post v-bind:likes="post.likes"></blog-post> |

### Passing a Boolean

|  |
| --- |
| <!-- Including the prop with no value will imply `true`. --> <blog-post is-published></blog-post>  <!-- Even though `false` is static, we need v-bind to tell Vue that --> <!-- this is a JavaScript expression rather than a string. --> <blog-post v-bind:is-published="false"></blog-post>  <!-- Dynamically assign to the value of a variable. --> <blog-post v-bind:is-published="post.isPublished"></blog-post> |

### Passing an Array

|  |
| --- |
| <!-- Even though the array is static, we need v-bind to tell Vue that --> <!-- this is a JavaScript expression rather than a string. --> <blog-post v-bind:comment-ids="[234, 266, 273]"></blog-post>  <!-- Dynamically assign to the value of a variable. --> <blog-post v-bind:comment-ids="post.commentIds"></blog-post> |

### Passing an Object

|  |
| --- |
| <!-- Even though the object is static, we need v-bind to tell Vue that --> <!-- this is a JavaScript expression rather than a string. --> <blog-post v-bind:author="{ name: 'Veronica', company: 'Veridian Dynamics' }"></blog-post>  <!-- Dynamically assign to the value of a variable. --> <blog-post v-bind:author="post.author"></blog-post> |

### Passing the Properties of an Object

If you want to pass all the properties of an object as props, you can use v-bind without an argument (v-bind instead of v-bind:prop-name). For example, given a post object:

|  |
| --- |
| post: {  id: 1,  title: 'My Journey with Vue' } |

The following template:

|  |
| --- |
| <blog-post v-bind="post"></blog-post> |

Will be equivalent to:

|  |
| --- |
| <blog-post  v-bind:id="post.id"  v-bind:title="post.title" ></blog-post> |

## One-Way Data Flow

All props form a **one-way-down binding** between the child property and the parent one: when the parent property updates, it will flow down to the child, but not the other way around. This prevents child components from accidentally mutating the parent’s state, which can make your app’s data flow harder to understand.

In addition, every time the parent component is updated, all props in the child component will be refreshed with the latest value. This means you should **not** attempt to mutate a prop inside a child component. If you do, Vue will warn you in the console.

There are usually two cases where it’s tempting to mutate a prop:

1. **The prop is used to pass in an initial value; the child component wants to use it as a local data property afterwards.** In this case, it’s best to define a local data property that uses the prop as its initial value:

|  |
| --- |
| props: ['initialCounter'], data: function () {  return {  counter: this.initialCounter  } } |

1. **The prop is passed in as a raw value that needs to be transformed.** In this case, it’s best to define a computed property using the prop’s value:

|  |
| --- |
| props: ['size'], computed: {  normalizedSize: function () {  return this.size.trim().toLowerCase()  } } |

Note that objects and arrays in JavaScript are passed by reference, so if the prop is an array or object, mutating the object or array itself inside the child component **will**affect parent state.

## Prop Validation

Components can specify requirements for its props, such as the types you’ve already seen. If a requirement isn’t met, Vue will warn you in the browser’s JavaScript console. This is especially useful when developing a component that’s intended to be used by others.

To specify prop validations, you can provide an object with validation requirements to the value of props, instead of an array of strings. For example:

|  |
| --- |
| Vue.component('my-component', {  props: {  // Basic type check (`null` matches any type)  propA: Number,  // Multiple possible types  propB: [String, Number],  // Required string  propC: {  type: String,  required: true  },  // Number with a default value  propD: {  type: Number,  default: 100  },  // Object with a default value  propE: {  type: Object,  // Object or array defaults must be returned from  // a factory function  default: function () {  return { message: 'hello' }  }  },  // Custom validator function  propF: {  validator: function (value) {  // The value must match one of these strings  return ['success', 'warning', 'danger'].indexOf(value) !== -1  }  }  } }) |

When prop validation fails, Vue will produce a console warning (if using the development build).

Note that props are validated **before** a component instance is created, so instance properties (e.g. data, computed, etc) will not be available inside default or validator functions.

### Type Checks

The type can be one of the following native constructors:

* String
* Number
* Boolean
* Array
* Object
* Date
* Function
* Symbol

In addition, type can also be a custom constructor function and the assertion will be made with an instanceof check. For example, given the following constructor function exists:

|  |
| --- |
| function Person (firstName, lastName) {  this.firstName = firstName  this.lastName = lastName } |

You could use:

|  |
| --- |
| Vue.component('blog-post', {  props: {  author: Person  } }) |

to validate that the value of the author prop was created with new Person.

## Non-Prop Attributes

A non-prop attribute is an attribute that is passed to a component, but does not have a corresponding prop defined.

While explicitly defined props are preferred for passing information to a child component, authors of component libraries can’t always foresee the contexts in which their components might be used. That’s why components can accept arbitrary attributes, which are added to the component’s root element.

For example, imagine we’re using a 3rd-party bootstrap-date-input component with a Bootstrap plugin that requires a data-date-picker attribute on the input. We can add this attribute to our component instance:

|  |
| --- |
| <bootstrap-date-input data-date-picker="activated"></bootstrap-date-input> |

And the data-date-picker="activated" attribute will automatically be added to the root element of bootstrap-date-input.

### Replacing/Merging with Existing Attributes

Imagine this is the template for bootstrap-date-input:

|  |
| --- |
| <input type="date" class="form-control"> |

To specify a theme for our date picker plugin, we might need to add a specific class, like this:

|  |
| --- |
| <bootstrap-date-input  data-date-picker="activated"  class="date-picker-theme-dark" ></bootstrap-date-input> |

In this case, two different values for class are defined:

* form-control, which is set by the component in its template
* date-picker-theme-dark, which is passed to the component by its parent

For most attributes, the value provided to the component will replace the value set by the component. So for example, passing type="text" will replace type="date" and probably break it! Fortunately, the class and style attributes are a little smarter, so both values are merged, making the final value: form-control date-picker-theme-dark.

### Disabling Attribute Inheritance

If you do **not** want the root element of a component to inherit attributes, you can set inheritAttrs: false in the component’s options. For example:

|  |
| --- |
| Vue.component('my-component', {  inheritAttrs: false,  // ... }) |

This can be especially useful in combination with the $attrs instance property, which contains the attribute names and values passed to a component, such as:

|  |
| --- |
| {  class: 'username-input',  placeholder: 'Enter your username' } |

With inheritAttrs: false and $attrs, you can manually decide which element you want to forward attributes to, which is often desirable for **base components**:

|  |
| --- |
| Vue.component('base-input', {  inheritAttrs: false,  props: ['label', 'value'],  template: `  <label>  {{ label }}  <input  v-bind="$attrs"  v-bind:value="value"  v-on:input="$emit('input', $event.target.value)"  >  </label>  ` }) |

This pattern allows you to use base components more like raw HTML elements, without having to care about which element is actually at its root:

|  |
| --- |
| <base-input  v-model="username"  class="username-input"  placeholder="Enter your username" ></base-input> |

# Routing

## Official Router

For most Single Page Applications, it’s recommended to use the officially-supported **vue-router library**. For more details, see vue-router’s **documentation**.

## Simple Routing From Scratch

If you only need very simple routing and do not wish to involve a full-featured router library, you can do so by dynamically rendering a page-level component like this:

|  |
| --- |
| const NotFound = { template: '<p>Page not found</p>' } const Home = { template: '<p>home page</p>' } const About = { template: '<p>about page</p>' }  const routes = {  '/': Home,  '/about': About }  new Vue({  el: '#app',  data: {  currentRoute: window.location.pathname  },  computed: {  ViewComponent () {  return routes[this.currentRoute] || NotFound  }  },  render (h) { return h(this.ViewComponent) } }) |

Combined with the HTML5 History API, you can build a very basic but fully-functional client-side router. To see that in practice, check out **this example app**.

# State Management

## Official Flux-Like Implementation

Large applications can often grow in complexity, due to multiple pieces of state scattered across many components and the interactions between them. To solve this problem, Vue offers **vuex**: our own Elm-inspired state management library. It even integrates into **vue-devtools**, providing zero-setup access to **time travel debugging**.

### Information for React Developers

If you’re coming from React, you may be wondering how vuex compares to **redux**, the most popular Flux implementation in that ecosystem. Redux is actually view-layer agnostic, so it can easily be used with Vue via **simple bindings**. Vuex is different in that it knows it’s in a Vue app. This allows it to better integrate with Vue, offering a more intuitive API and improved development experience.

## Simple State Management from Scratch

It is often overlooked that the source of truth in Vue applications is the raw data object - a Vue instance only proxies access to it. Therefore, if you have a piece of state that should be shared by multiple instances, you can share it by identity:

|  |
| --- |
| const sourceOfTruth = {}  const vmA = new Vue({  data: sourceOfTruth })  const vmB = new Vue({  data: sourceOfTruth }) |

Now whenever sourceOfTruth is mutated, both vmA and vmB will update their views automatically. Subcomponents within each of these instances would also have access via this.$root.$data. We have a single source of truth now, but debugging would be a nightmare. Any piece of data could be changed by any part of our app at any time, without leaving a trace.

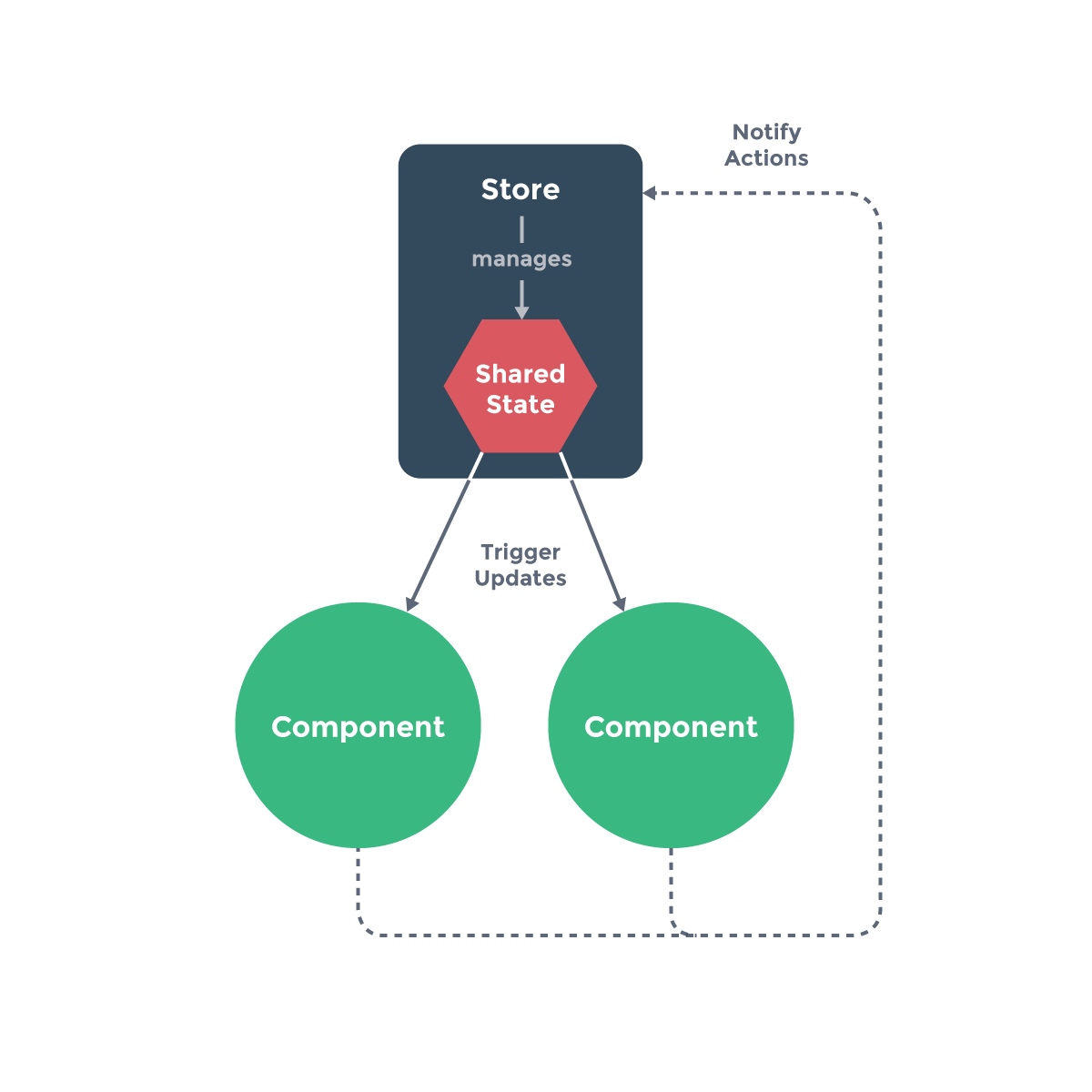
To help solve this problem, we can adopt a **store pattern**:

|  |
| --- |
| var store = {  debug: true,  state: {  message: 'Hello!'  },  setMessageAction (newValue) {  if (this.debug) console.log('setMessageAction triggered with', newValue)  this.state.message = newValue  },  clearMessageAction () {  if (this.debug) console.log('clearMessageAction triggered')  this.state.message = ''  } } |

Notice all actions that mutate the store’s state are put inside the store itself. This type of centralized state management makes it easier to understand what type of mutations could happen and how are they triggered. Now when something goes wrong, we’ll also have a log of what happened leading up to the bug.

In addition, each instance/component can still own and manage its own private state:

|  |
| --- |
| var vmA = new Vue({  data: {  privateState: {},  sharedState: store.state  } })  var vmB = new Vue({  data: {  privateState: {},  sharedState: store.state  } }) |



It’s important to note that you should never replace the original state object in your actions - the components and the store need to share reference to the same object in order for mutations to be observed.

As we continue developing the convention where components are never allowed to directly mutate state that belongs to a store, but should instead dispatch events that notify the store to perform actions, we eventually arrive at the **Flux** architecture. The benefit of this convention is we can record all state mutations happening to the store and implement advanced debugging helpers such as mutation logs, snapshots, and history re-rolls / time travel.