Installation

Compatibility Note

Vue does **not** support IE8 and below, because it uses ECMAScript 5 features that are un-shimmable in IE8. However it supports all ECMAScript 5 compliant browsers.

Release Notes

Latest stable version: {{vue_version}}

Detailed release notes for each version are available on GitHub.

Vue Devtools

When using Vue, we recommend also installing the Vue Devtools in your browser, allowing you to inspect and debug your Vue applications in a more user-friendly interface.

Direct `<script>` Include

Simply download and include with a script tag. `Vue` will be registered as a global variable.

Don't use the minified version during development. You will miss out on all the nice warnings for common mistakes!

Development VersionWith full warnings and debug mode

Production VersionWarnings stripped, {{gz_size}}KB min+gzip

CDN

We recommend linking to a specific version number that you can update manually:

```
<script src="https://cdn.jsdelivr.net/npm/vue@2.5.16/dist/vue.js"></script>
```

You can browse the source of the NPM package at cdn.jsdelivr.net/npm/vue.

Vue is also available on unpkg and cdnjs (cdnjs takes some time to sync so the latest release may not be available yet).

Make sure to read about the different builds of Vue and use the **production version** in your published site, replacing `vue.js` with `vue.min.js`. This is a smaller build optimized for speed instead of development experience.

NPM

NPM is the recommended installation method when building large scale applications with Vue. It pairs nicely with module bundlers such as Webpack or Browserify. Vue also provides accompanying tools for authoring Single File Components.

```
# latest stable
$ npm install vue
```

CLI

Vue provides an official CLI for quickly scaffolding ambitious Single Page Applications. It provides batteries-included build setups for a modern frontend workflow. It takes only a few minutes to get up and running with hot-reload, lint-on-save, and production-ready builds. See the Vue CLI docs for more details.

The CLI assumes prior knowledge of Node.js and the associated build tools. If you are new to Vue or front-end build tools, we strongly suggest going through the guide without any build tools before using the CLI.

Explanation of Different Builds

In the `dist/` directory of the NPM package you will find many different builds of Vue.js. Here's an overview of the difference between them:

	UMD	CommonJS	ES Module
Full	vue.js	vue.common.js	vue.esm.js
Runtime-only	vue.runtime.js	vue.runtime.common.js	vue.runtime.esm.js
Full (production)	vue.min.js	-	-
Runtime-only (production)	vue.runtime.min.js	-	-

Terms

- **Full**: builds that contain both the compiler and the runtime.
- **Compiler**: code that is responsible for compiling template strings into JavaScript render functions.
- **Runtime**: code that is responsible for creating Vue instances, rendering and patching virtual DOM, etc. Basically everything minus the compiler.
- **UMD**: UMD builds can be used directly in the browser via a `<script>` tag. The default file from jsDelivr CDN at https://cdn.jsdelivr.net/npm/vue is the Runtime + Compiler UMD build (`vue.js`).
- **CommonJS**: CommonJS builds are intended for use with older bundlers like browserify or webpack 1. The default file for these bundlers (`pkg.main`) is the Runtime only CommonJS build (`vue.runtime.common.js`).
- **ES Module**: ES module builds are intended for use with modern bundlers like webpack 2 or rollup. The default file for these bundlers (`pkg.module`) is the Runtime only ES Module build (`vue.runtime.esm.js`).

Runtime + Compiler vs. Runtime-only

If you need to compile templates on the client (e.g. passing a string to the `template` option, or mounting to an element using its in-DOM HTML as the template), you will need the compiler and thus the full build:

```
// this requires the compiler
new Vue({
  template: '<div>{{ hi }}</div>'
})

// this does not
new Vue({
  render (h) {
    return h('div', this.hi)
  }
})
```

When using `vue-loader` or `vueify`, templates inside `*.vue` files are pre-compiled into JavaScript at build time. You don't really need the compiler in the final bundle, and can therefore use the runtime-only build.

Since the runtime-only builds are roughly 30% lighter-weight than their full-build counterparts, you should use it whenever you can. If you still wish to use the full build instead, you need to configure an alias in your bundler:

Webpack

```
module.exports = {
  // ...
  resolve: {
    alias: {
      'vue$': 'vue/dist/vue.esm.js' // 'vue/dist/vue.common.js' for webpack 1
    }
  }
}
```

Rollup

```
const alias = require('rollup-plugin-alias')

rollup({
  // ...
  plugins: [
    alias({
      'vue': 'vue/dist/vue.esm.js'
    })
  ]
})
```

Browserify

Add to your project's `package.json`:

```
{
  // ...
  "browser": {
    "vue": "vue/dist/vue.common.js"
  }
}
```

Parcel

Add to your project's `package.json`:

```
{
  // ...
  "alias": {
    "vue" : "./node_modules/vue/dist/vue.common.js"
  }
}
```

Development vs. Production Mode

Development/production modes are hard-coded for the UMD builds: the unminified files are for development, and the minified files are for production.

CommonJS and ES Module builds are intended for bundlers, therefore we don't provide minified versions for them. You will be responsible for minifying the final bundle yourself.

CommonJS and ES Module builds also preserve raw checks for `process.env.NODE_ENV` to determine the mode they should run in. You should use appropriate bundler configurations to replace these environment variables in order to control which mode Vue will run in. Replacing `process.env.NODE_ENV` with string literals also allows minifiers like UglifyJS to completely drop the development-only code blocks, reducing final file size.

Webpack

In Webpack 4+, you can use the `mode` option:

```
module.exports = {
  mode: 'production'
}
```

But in Webpack 3 and earlier, you'll need to use DefinePlugin:

```
var webpack = require('webpack')

module.exports = {
  // ...
  plugins: [
    // ...
    new webpack.DefinePlugin({
      'process.env': {
        NODE_ENV: JSON.stringify('production')
      }
    })
  ]
}
```

Rollup

Use rollup-plugin-replace:

```
const replace = require('rollup-plugin-replace')

rollup({
  // ...
```

```
  plugins: [
    replace({
      'process.env.NODE_ENV': JSON.stringify('production')
    })
  ]
}).then(...)
```

Browserify

Apply a global envify transform to your bundle.

```
NODE_ENV=production browserify -g envify -e main.js | uglifyjs -c -m > build.js
```

Also see Production Deployment Tips.

CSP environments

Some environments, such as Google Chrome Apps, enforce Content Security Policy (CSP), which prohibits the use of `new Function()` for evaluating expressions. The full build depends on this feature to compile templates, so is unusable in these environments.

On the other hand, the runtime-only build is fully CSP-compliant. When using the runtime-only build with Webpack + vue-loader or Browserify + vueify, your templates will be precompiled into `render` functions which work perfectly in CSP environments.

Dev Build

Important: the built files in GitHub's `/dist` folder are only checked-in during releases. To use Vue from the latest source code on GitHub, you will have to build it yourself!

```
git clone https://github.com/vuejs/vue.git node_modules/vue
cd node_modules/vue
npm install
npm run build
```

Bower

Only UMD builds are available from Bower.

```
# latest stable
$ bower install vue
```

AMD Module Loaders

All UMD builds can be used directly as an AMD module.

What is Vue.js?

Vue (pronounced /vju /, 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 and supporting libraries.

If you'd like to learn more about Vue before diving in, we created a video 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.

Getting Started

The official guide assumes intermediate level knowledge of HTML, CSS, and JavaScript. If you are totally new to frontend development, it might not be the best idea to jump right into a framework as your first step - grasp the basics then come back! Prior experience with other frameworks helps, but is not required.

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.

If you prefer something more interactive, you can also check out this tutorial series on Scrimba, which gives you a mix of screencast and code playground that you can pause and play around with anytime.

Declarative Rendering

Try this lesson on Scrimba

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!'
  }
})
```

{% raw %}

{{ message }}

{% endraw %}

We have already created our very first Vue app! This looks pretty similar to rendering a string template, but Vue has done a lot of work under the hood. The data and the DOM are now linked, and everything is now **reactive**. How do we know? Open your browser's JavaScript console (right now, on this page) and set `app.message` to a different value. You should see the rendered example above update accordingly.

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()
  }
})
```

{% raw %}

Hover your mouse over me for a few seconds to see my dynamically bound title!

{% endraw %}

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.

Conditionals and Loops

Try this lesson on Scrimba

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
  }
})
```

{% raw %}

Now you see me

{% endraw %}

Go ahead and enter `app3.seen = false` in the console. You should see the message disappear.

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' }
    ]
  }
})
```

{% raw %}

{{ todo.text }}

{% endraw %}

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

Handling User Input

Try this lesson on Scrimba

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('')
    }
  }
})
```

{% raw %}

{{ message }}

Reverse Message

{% endraw %}

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!'
  }
})
```

{% raw %}

{{ message }}

{% endraw %}

Composing with Components

Try this lesson on Scrimba

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>'
})
```

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

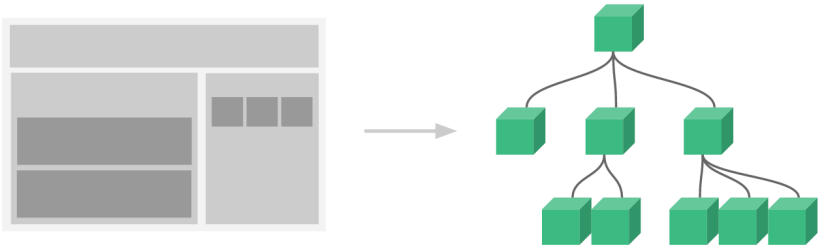

Figure 1: Component Tree

```
<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' }
    ]
  }
})
```

{% raw %}

<todo-item v-for=“item in groceryList” v-bind:todo=“item” :key=“item.id”>

{% endraw %}

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, 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

You may have noticed that Vue components are very similar to **Custom Elements**, which are part of the Web Components Spec. That's because Vue's component syntax is loosely modeled after the spec. For example, Vue components implement the Slot API and the `is` special attribute. However, there are

a few key differences:

1. The Web Components Spec is still in draft status, and is not natively implemented in every browser. 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.

Ready for More?

We've briefly introduced the most basic features of Vue.js core - the rest of this guide will cover them and other advanced features with much finer details, so make sure to read through it all!

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, 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.

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 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

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 for a full list of instance properties and methods.

Instance Lifecycle Hooks

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` 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`, `updated`, and `destroyed`. All lifecycle hooks are called with their `this` context pointing to the Vue instance invoking it.

Don't use arrow functions on an options property or callback, such as `created: () => console.log(this.a)` or `vm.$watch('a', newValue => this.myMethod())`. Since arrow functions are bound to the parent context, `this` will not be the Vue instance as you'd expect, often resulting in errors such as `Uncaught TypeError: Cannot read property of undefined` or `Uncaught TypeError: this.myMethod is not a function`.

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.

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.

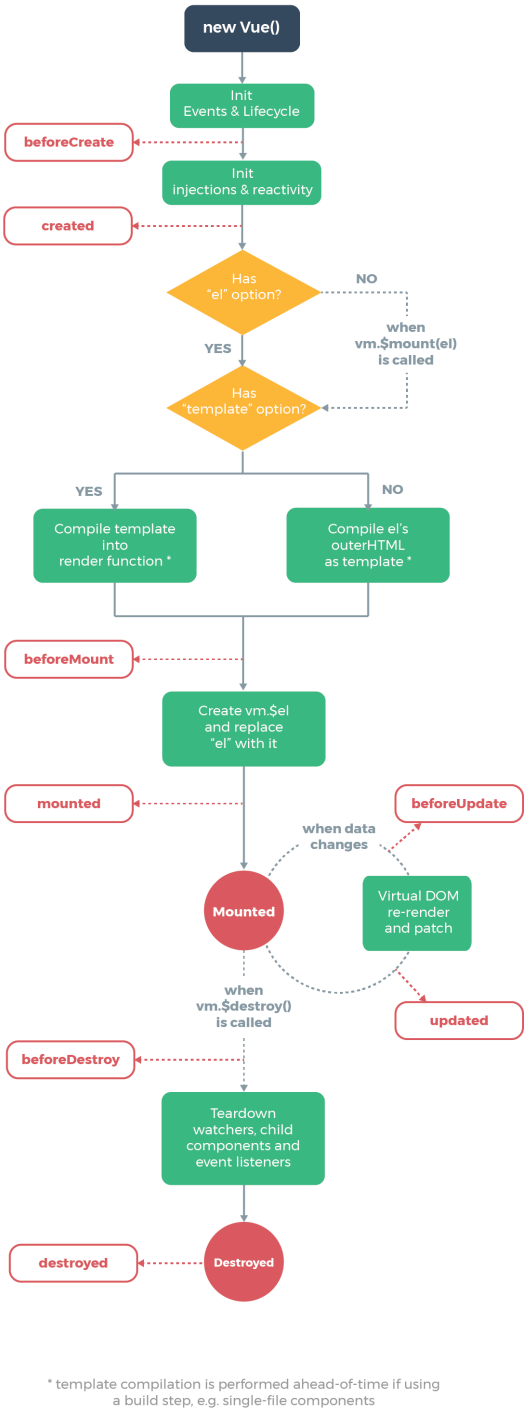

Figure 2: The Vue Instance Lifecycle

If you are familiar with Virtual DOM concepts and prefer the raw power of JavaScript, you can also directly write render functions instead of templates, with optional JSX support.

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>
```

{% raw %}

Using mustaches: {{ rawHtml }}

Using v-html directive:

{% endraw %}

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-html` to 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. Only use HTML interpolation on trusted content and **never** on user-provided content.

Attributes

Mustaches cannot be used inside HTML attributes. Instead, use a v-bind directive:

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

In the case of boolean attributes, where their mere existence implies `true`, `v-bind` works 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

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 such as `Math` and `Date`. You should not attempt to access user defined globals in template expressions.

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 `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.

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.

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

```
<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:

{% raw %}

Original message: "{{ message }}"

Computed reversed message: "{{ reversedMessage }}"

{% endraw %}

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

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 instead. For the end result, the two approaches are indeed exactly the same. However, the difference is that **computed properties are cached based on their dependencies.** A computed property will only re-evaluate when some of its 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

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

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

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>
```

Result:

{% raw %}

Ask a yes/no question:

{{ answer }}

{% endraw %}

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.

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.

Binding HTML Classes

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 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 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

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

> This section assumes knowledge of Vue Components. 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

Object Syntax

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

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

When you use a CSS property that requires vendor prefixes in `v-bind:style`, for example `transform`, Vue will automatically detect and add appropriate prefixes to the applied styles.

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.

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>
```

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`.

Check it out for yourself by entering some text in the input, then pressing the toggle button:

{% raw %}

```
<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>
```

<button @click="toggleLoginType">Toggle login type

{% endraw %}

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:

{% raw %}

```
<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>
```

<button @click="toggleLoginType">Toggle login type

{% endraw %}

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

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

`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`

When used together with `v-if`, `v-for` has a higher priority than `v-if`. See the list rendering guide for details.

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-for` directive requires a special syntax in the form of `item in items`, where `items` is the source data array and `item` is an **alias** for the array element being iterated on:

```
<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:

{% raw %}

{{item.message}}

{% endraw %}

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:

{% raw%}

{{ parentMessage }} - {{ index }} - {{ item.message }}

{% endraw %}

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:

{% raw %}

{{ value }}

{% endraw %}

You can also provide a second argument for the key:

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

{% raw %}

```
{{ key }}: {{ value }}
```

{% endraw %}

And another for the index:

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

{% raw %}

```
{{ index }}. {{ key }}: {{ value }}
```

{% endraw %}

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 `key` would be the unique id of each item. This special

attribute is a rough equivalent to `track-by` in 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:

{% raw %}

{{ n }}

{% endraw %}

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-if` on 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 = ''
    }
  }
})
```

{% raw %}

<form v-on:submit.prevent="addNewTodo"> Add a todo Add

{% endraw %}

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:

{% raw %}

Add 1

The button above has been clicked {{ counter }} times.

{% endraw %}

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:

{% raw %}

Greet

{% endraw %}

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: {% raw %}

Say hi Say what

{% endraw %}

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 e
<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 `passive` option.

```
<!-- 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

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 `keyCode`s 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 `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

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.

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>
```

{% raw %}

Message is: {{ message }}

{% endraw %}

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>
```

{% raw %}

Multiline message is:

{{ message }}

{% endraw %}

{% raw %}

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

{% endraw %}

Checkbox

Single checkbox, boolean value:

```
<input type="checkbox" id="checkbox" v-model="checked">
<label for="checkbox">{{ checked }}</label>
```

{% raw %}

{{ checked }}

{% endraw %}

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: []
  }
})
```

{% raw %}

Jack John Mike Checked names: {{ checkedNames }}

{% endraw %}

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>
```

{% raw %}

One Two Picked: {{ picked }}

{% endraw %}

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: ''
  }
})
```

{% raw %}

Please select one A B C Selected: {{ selected }}

{% endraw %}

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>
```

{% raw %}

A B C Selected: {{ selected }}

{% endraw %}

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' }
    ]
  }
})
```

{% raw %}

{{ option.text }} Selected: {{ selected }}

{% endraw %}

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 `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

```
<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 `number` modifier 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.

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' })
```

{% raw %}

{% endraw %}

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>
```

{% raw %}

{% endraw %}

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 `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:

{% raw %}

{% endraw %}

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.

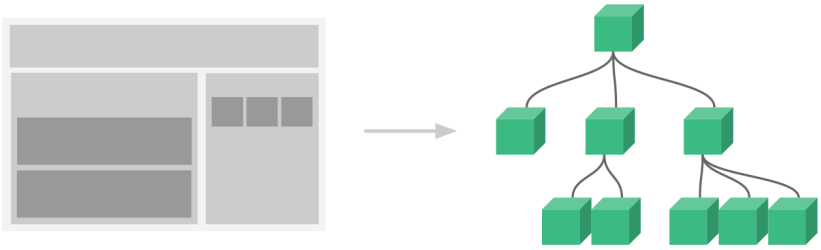

Figure 3: Component Tree

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 `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>
```

{% raw %}

{% endraw %}

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>
```

{% raw %}

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

{% endraw %}

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:

{% raw %}

Something bad happened.

{% endraw %}

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:

{% raw %}

{{ tab }}

{% endraw %}

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.

> 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.

> 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
  }
}
```

2. **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>
```

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

Event Names

Unlike components and props, event names don't provide any automatic case transformation. Instead, the name of an emitted event must exactly match the

name used to listen to that event. For example, if emitting a camelCased event name:

```
this.$emit('myEvent')
```

Listening to the kebab-cased version will have no effect:

```
<my-component v-on:my-event="doSomething"></my-component>
```

Unlike components and props, event names will never be used as variable or property names in JavaScript, so there's no reason to use camelCase or PascalCase. Additionally, `v-on` event listeners inside DOM templates will be automatically transformed to lowercase (due to HTML's case-insensitivity), so `v-on:myEvent` would become `v-on:myevent` – making `myEvent` impossible to listen to.

For these reasons, we recommend you **always use kebab-case for event names**.

Customizing Component `v-model`

New in 2.2.0+

By default, `v-model` on a component uses `value` as the prop and `input` as the event, but some input types such as checkboxes and radio buttons may want to use the `value` attribute for a different purpose. Using the `model` option can avoid a conflict in such cases:

```
Vue.component('base-checkbox', {
  model: {
    prop: 'checked',
    event: 'change'
  },
  props: {
    checked: Boolean
  },
  template: `
    <input
      type="checkbox"
      v-bind:checked="checked"
      v-on:change="$emit('change', $event.target.checked)"
    >
  `
})
```

Now when using `v-model` on this component:

```
<base-checkbox v-model="lovingVue"></base-checkbox>
```

the value of `lovingVue` will be passed to the `checked` prop. The `lovingVue` property will then be updated when `<base-checkbox>` emits a `change` event with a new value.

Note that you still have to declare the checked prop in component's props option.

Binding Native Events to Components

There may be times when you want to listen directly to a native event on the root element of a component. In these cases, you can use the `.native` modifier for `v-on`:

```
<base-input v-on:focus.native="onFocus"></base-input>
```

This can be useful sometimes, but it's not a good idea when you're trying to listen on a very specific element, like an `<input>`. For example, the `<base-input>` component above might refactor so that the root element is actually a `<label>` element:

```
<label>
  {{ label }}
  <input
    v-bind="$attrs"
    v-bind:value="value"
    v-on:input="$emit('input', $event.target.value)"
  >
</label>
```

In that case, the `.native` listener in the parent would silently break. There would be no errors, but the `onFocus` handler wouldn't be called when we expected it to.

To solve this problem, Vue provides a `$listeners` property containing an object of listeners being used on the component. For example:

```
{
  focus: function (event) { /* ... */ }
  input: function (value) { /* ... */ },
}
```

Using the `$listeners` property, you can forward all event listeners on the component to a specific child element with `v-on="$listeners"`. For elements like `<input>`, that you also want to work with `v-model`, it's often useful to create a new computed property for listeners, like `inputListeners` below:

```
Vue.component('base-input', {
  inheritAttrs: false,
  props: ['label', 'value'],
  computed: {
```

```
    inputListeners: function () {
      var vm = this
      // `Object.assign` merges objects together to form a new object
      return Object.assign({},
        // We add all the listeners from the parent
        this.$listeners,
        // Then we can add custom listeners or override the
        // behavior of some listeners.
        {
          // This ensures that the component works with v-model
          input: function (event) {
            vm.$emit('input', event.target.value)
          }
        }
      )
    }
  },
  template: `
    <label>
      {{ label }}
      <input
        v-bind="$attrs"
        v-bind:value="value"
        v-on="inputListeners"
      >
    </label>
  `
})
```

Now the `<base-input>` component is a **fully transparent wrapper**, meaning it can be used exactly like a normal `<input>` element: all the same attributes and listeners will work, without the `.native` modifier.

`.sync` Modifier

> New in 2.3.0+

In some cases, we may need "two-way binding" for a prop. Unfortunately, true two-way binding can create maintenance issues, because child components can mutate the parent without the source of that mutation being obvious in both the parent and the child.

That's why instead, we recommend emitting events in the pattern of `update:my-prop-name`. For example, in a hypothetical component with a `title` prop, we could communicate the intent of assigning a new value with:

```
this.$emit('update:title', newTitle)
```

Then the parent can listen to that event and update a local data property, if it wants to. For example:

```
<text-document
  v-bind:title="doc.title"
  v-on:update:title="doc.title = $event"
></text-document>
```

For convenience, we offer a shorthand for this pattern with the `.sync` modifier:

```
<text-document v-bind:title.sync="doc.title"></text-document>
```

The `.sync` modifier can also be used with `v-bind` when using an object to set multiple props at once:

```
<text-document v-bind.sync="doc"></text-document>
```

This passes each property in the `doc` object (e.g. `title`) as an individual prop, then adds `v-on` update listeners for each one.

Using v-bind.sync with a literal object, such as in v-bind.sync="{ title: doc.title }", will not work, because there are too many edge cases to consider in parsing a complex expression like this.

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

Slot Content

Vue implements a content distribution API that's modeled after the current Web Components spec draft, using the `<slot>` element to serve as distribution outlets for content.

This allows you to compose components like this:

```
<navigation-link url="/profile">
  Your Profile
</navigation-link>
```

Then in the template for `<navigation-link>`, you might have:

```
<a
  v-bind:href="url"
  class="nav-link"
>
  <slot></slot>
</a>
```

When the component renders, the `<slot>` element will be replaced by "Your Profile". Slots can contain any template code, including HTML:

```
<navigation-link url="/profile">
  <!-- Add a Font Awesome icon -->
  <span class="fa fa-user"></span>
  Your Profile
</navigation-link>
```

Or even other components:

```
<navigation-link url="/profile">
  <!-- Use a component to add an icon -->
  <font-awesome-icon name="user"></font-awesome-icon>
  Your Profile
</navigation-link>
```

If `<navigation-link>` did **not** contain a `<slot>` element, any content passed to it would simply be discarded.

Named Slots

There are times when it's useful to have multiple slots. For example, in a hypothetical `base-layout` component with the following template:

```
<div class="container">
  <header>
    <!-- We want header content here -->
  </header>
  <main>
    <!-- We want main content here -->
  </main>
  <footer>
    <!-- We want footer content here -->
  </footer>
</div>
```

For these cases, the `<slot>` element has a special attribute, `name`, which can be used to define additional slots:

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

To provide content to named slots, we can use the `slot` attribute on a `<template>` element in the parent:

```
<base-layout>
  <template slot="header">
    <h1>Here might be a page title</h1>
  </template>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <template slot="footer">
    <p>Here's some contact info</p>
  </template>
</base-layout>
```

Or, the `slot` attribute can also be used directly on a normal element:

```
<base-layout>
  <h1 slot="header">Here might be a page title</h1>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <p slot="footer">Here's some contact info</p>
</base-layout>
```

There can still be one unnamed slot, which is the **default slot** that serves as a catch-all outlet for any unmatched content. In both examples above, the rendered HTML would be:

```
<div class="container">
  <header>
    <h1>Here might be a page title</h1>
  </header>
  <main>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </main>
  <footer>
    <p>Here's some contact info</p>
  </footer>
</div>
```

Default Slot Content

There are cases when it's useful to provide a slot with default content. For example, a `<submit-button>` component might want the content of the button to be "Submit" by default, but also allow users to override with "Save", "Upload", or anything else.

To achieve this, specify the default content in between the `<slot>` tags.

```
<button type="submit">
  <slot>Submit</slot>
</button>
```

If the slot is provided content by the parent, it will replace the default content.

Compilation Scope

When you want to use data inside a slot, such as in:

```
<navigation-link url="/profile">
  Logged in as {{ user.name }}
</navigation-link>
```

That slot has access to the same instance properties (i.e. the same "scope") as the rest of the template. The slot does **not** have access to `<navigation-link>`'s scope. For example, trying to access `url` would not work. As a rule, remember that:

> Everything in the parent template is compiled in parent scope; everything in the child template is compiled in the child scope.

Scoped Slots

> New in 2.1.0+

Sometimes you'll want to provide a component with a reusable slot that can access data from the child component. For example, a simple `<todo-list>` component may contain the following in its template:

```
<ul>
  <li
    v-for="todo in todos"
    v-bind:key="todo.id"
  >
    {{ todo.text }}
  </li>
</ul>
```

But in some parts of our app, we want the individual todo items to render something different than just the `todo.text`. This is where scoped slots come in.

To make the feature possible, all we have to do is wrap the todo item content in a `<slot>` element, then pass the slot any data relevant to its context: in this case, the `todo` object:

```
<ul>
  <li
    v-for="todo in todos"
    v-bind:key="todo.id"
  >
    <!-- We have a slot for each todo, passing it the -->
    <!-- `todo` object as a slot prop.                -->
    <slot v-bind:todo="todo">
      <!-- Fallback content -->
      {{ todo.text }}
    </slot>
  </li>
</ul>
```

Now when we use the `<todo-list>` component, we can optionally define an alternative `<template>` for todo items, but with access to data from the child via the `slot-scope` attribute:

```
<todo-list v-bind:todos="todos">
  <!-- Define `slotProps` as the name of our slot scope -->
  <template slot-scope="slotProps">
    <!-- Define a custom template for todo items, using -->
    <!-- `slotProps` to customize each todo.            -->
    <span v-if="slotProps.todo.isComplete"> </span>
    {{ slotProps.todo.text }}
  </template>
</todo-list>
```

> In 2.5.0+, `slot-scope` is no longer limited to the `<template>` element, but can instead be used on any element or component in the slot.

Destructuring `slot-scope`

The value of `slot-scope` can actually accept any valid JavaScript expression that can appear in the argument position of a function definition. This means in supported environments (single-file components or modern browsers) you can also use ES2015 destructuring in the expression, like so:

```
<todo-list v-bind:todos="todos">
  <template slot-scope="{ todo }">
    <span v-if="todo.isComplete"> </span>
    {{ todo.text }}
  </template>
</todo-list>
```

This is a great way to make scoped slots a little cleaner.

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

`keep-alive` with Dynamic Components

Earlier, we used the `is` attribute to switch between components in a tabbed interface:

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

When switching between these components though, you'll sometimes want to maintain their state or avoid re-rendering for performance reasons. For example, when expanding our tabbed interface a little:

{% raw %}

{{ tab }}

{% endraw %}

You'll notice that if you select a post, switch to the *Archive* tab, then switch back to *Posts*, it's no longer showing the post you selected. That's because each time you switch to a new tab, Vue creates a new instance of the `currentTabComponent`.

Recreating dynamic components is normally useful behavior, but in this case, we'd really like those tab component instances to be cached once they're created for the first time. To solve this problem, we can wrap our dynamic component with a `<keep-alive>` element:

```
<!-- Inactive components will be cached! -->
<keep-alive>
  <component v-bind:is="currentTabComponent"></component>
</keep-alive>
```

Check out the result below:

{% raw %}

{{ tab }}

{% endraw %}

Now the *Posts* tab maintains its state (the selected post) even when it's not rendered. See this fiddle for the complete code.

Note that `<keep-alive>` requires the components being switched between to all have names, either using the `name` option on a component, or through local/global registration.

Check out more details on `<keep-alive>` in the API reference.

Async Components

In large applications, we may need to divide the app into smaller chunks and only load a component from the server when it's needed. To make that easier, Vue allows you to define your component as a factory function that asynchronously resolves your component definition. Vue will only trigger the factory function when the component needs to be rendered and will cache the result for future re-renders. For example:

```
Vue.component('async-example', function (resolve, reject) {
  setTimeout(function () {
    // Pass the component definition to the resolve callback
    resolve({
      template: '<div>I am async!</div>'
    })
  }, 1000)
})
```

As you can see, the factory function receives a `resolve` callback, which should be called when you have retrieved your component definition from the server. You can also call `reject(reason)` to indicate the load has failed. The `setTimeout` here is for demonstration; how to retrieve the component is up to you. One recommended approach is to use async components together with Webpack's code-splitting feature:

```
Vue.component('async-webpack-example', function (resolve) {
  // This special require syntax will instruct Webpack to
  // automatically split your built code into bundles which
  // are loaded over Ajax requests.
  require(['./my-async-component'], resolve)
})
```

You can also return a `Promise` in the factory function, so with Webpack 2 and ES2015 syntax you can do:

```
Vue.component(
  'async-webpack-example',
  // The `import` function returns a Promise.
```

```
  () => import('./my-async-component')
)
```

When using local registration, you can also directly provide a function that returns a `Promise`:

```
new Vue({
  // ...
  components: {
    'my-component': () => import('./my-async-component')
  }
})
```

If you're a Browserify user that would like to use async components, its creator has unfortunately made it clear that async loading "is not something that Browserify will ever support." Officially, at least. The Browserify community has found some workarounds, which may be helpful for existing and complex applications. For all other scenarios, we recommend using Webpack for built-in, first-class async support.

Handling Loading State

> New in 2.3.0+

The async component factory can also return an object of the following format:

```
const AsyncComponent = () => ({
  // The component to load (should be a Promise)
  component: import('./MyComponent.vue'),
  // A component to use while the async component is loading
  loading: LoadingComponent,
  // A component to use if the load fails
  error: ErrorComponent,
  // Delay before showing the loading component. Default: 200ms.
  delay: 200,
  // The error component will be displayed if a timeout is
  // provided and exceeded. Default: Infinity.
  timeout: 3000
})
```

> Note that you must use Vue Router 2.4.0+ if you wish to use the above syntax for route components.

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

All the features on this page document the handling of edge cases, meaning unusual situations that sometimes require bending Vue's rules a little. Note however, that they all have disadvantages or situations where they could be

dangerous. These are noted in each case, so keep them in mind when deciding to use each feature.

Element & Component Access

In most cases, it's best to avoid reaching into other component instances or manually manipulating DOM elements. There are cases, however, when it can be appropriate.

Accessing the Root Instance

In every subcomponent of a `new Vue` instance, this root instance can be accessed with the `$root` property. For example, in this root instance:

```
// The root Vue instance
new Vue({
  data: {
    foo: 1
  },
  computed: {
    bar: function () { /* ... */ }
  },
  methods: {
    baz: function () { /* ... */ }
  }
})
```

All subcomponents will now be able to access this instance and use it as a global store:

```
// Get root data
this.$root.foo

// Set root data
this.$root.foo = 2

// Access root computed properties
this.$root.bar

// Call root methods
this.$root.baz()
```

This can be convenient for demos or very small apps with a handful of components. However, the pattern does not scale well to medium or large-scale applications, so we strongly recommend using Vuex to manage state in most cases.

Accessing the Parent Component Instance

Similar to `$root`, the `$parent` property can be used to access the parent instance from a child. This can be tempting to reach for as a lazy alternative to passing data with a prop.

In most cases, reaching into the parent makes your application more difficult to debug and understand, especially if you mutate data in the parent. When looking at that component later, it will be very difficult to figure out where that mutation came from.

There are cases however, particularly shared component libraries, when this *might* be appropriate. For example, in abstract components that interact with JavaScript APIs instead of rendering HTML, like these hypothetical Google Maps components:

```
<google-map>
  <google-map-markers v-bind:places="iceCreamShops"></google-map-markers>
</google-map>
```

The `<google-map>` component might define a `map` property that all subcomponents need access to. In this case `<google-map-markers>` might want to access that map with something like `this.$parent.getMap`, in order to add a set of markers to it. You can see this pattern in action here.

Keep in mind, however, that components built with this pattern are still inherently fragile. For example, imagine we add a new `<google-map-region>` component and when `<google-map-markers>` appears within that, it should only render markers that fall within that region:

```
<google-map>
  <google-map-region v-bind:shape="cityBoundaries">
    <google-map-markers v-bind:places="iceCreamShops"></google-map-markers>
  </google-map-region>
</google-map>
```

Then inside `<google-map-markers>` you might find yourself reaching for a hack like this:

```
var map = this.$parent.map || this.$parent.$parent.map
```

This has quickly gotten out of hand. That's why to provide context information to descendent components arbitrarily deep, we instead recommend dependency injection.

Accessing Child Component Instances & Child Elements

Despite the existence of props and events, sometimes you might still need to directly access a child component in JavaScript. To achieve this you can assign

a reference ID to the child component using the `ref` attribute. For example:

```
<base-input ref="usernameInput"></base-input>
```

Now in the component where you've defined this `ref`, you can use:

```
this.$refs.usernameInput
```

to access the `<base-input>` instance. This may be useful when you want to, for example, programmatically focus this input from a parent. In that case, the `<base-input>` component may similarly use a `ref` to provide access to specific elements inside it, such as:

```
<input ref="input">
```

And even define methods for use by the parent:

```
methods: {
  // Used to focus the input from the parent
  focus: function () {
    this.$refs.input.focus()
  }
}
```

Thus allowing the parent component to focus the input inside `<base-input>` with:

```
this.$refs.usernameInput.focus()
```

When `ref` is used together with `v-for`, the ref you get will be an array containing the child components mirroring the data source.

$refs < /code > areonlypopulatedafterthecomponenthasbeenrendered, andtheyarenotreactive.Itisonlymeant$
$youshouldavoidaccessing < code >$refs from within templates or computed properties.

Dependency Injection

Earlier, when we described Accessing the Parent Component Instance, we showed an example like this:

```
<google-map>
  <google-map-region v-bind:shape="cityBoundaries">
    <google-map-markers v-bind:places="iceCreamShops"></google-map-markers>
  </google-map-region>
</google-map>
```

In this component, all descendants of `<google-map>` needed access to a `getMap` method, in order to know which map to interact with. Unfortunately, using the `$parent` property didn't scale well to more deeply nested components.

That's where dependency injection can be useful, using two new instance options: `provide` and `inject`.

The `provide` options allows us to specify the data/methods we want to **provide** to descendent components. In this case, that's the `getMap` method inside `<google-map>`:

```
provide: function () {
  return {
    getMap: this.getMap
  }
}
```

Then in any descendants, we can use the `inject` option to receive specific properties we'd like to add to that instance:

```
inject: ['getMap']
```

You can see the full example here. The advantage over using `$parent` is that we can access `getMap` in *any* descendant component, without exposing the entire instance of `<google-map>`. This allows us to more safely keep developing that component, without fear that we might change/remove something that a child component is relying on. The interface between these components remains clearly defined, just as with `props`.

In fact, you can think of dependency injection as sort of "long-range props", except:

- ancestor components don't need to know which descendants use the properties it provides
- descendant components don't need to know where injected properties are coming from

However, there are downsides to dependency injection. It couples components in your application to the way they're currently organized, making refactoring more difficult. Provided properties are also not reactive. This is by design, because using them to create a central data store scales just as poorly as using $root for the same purpose. If the properties you want to share are specific to your app, rather than generic, or if you ever want to update provided data inside ancestors, then that's a good sign that you probably need a real state management solution like Vuex instead.

Learn more about dependency injection in the API doc.

Programmatic Event Listeners

So far, you've seen uses of `$emit`, listened to with `v-on`, but Vue instances also offer other methods in its events interface. We can:

- Listen for an event with `$on(eventName, eventHandler)`

- Listen for an event only once with `$once(eventName, eventHandler)`
- Stop listening for an event with `$off(eventName, eventHandler)`

You normally won't have to use these, but they're available for cases when you need to manually listen for events on a component instance. They can also be useful as a code organization tool. For example, you may often see this pattern for integrating a 3rd-party library:

```
// Attach the datepicker to an input once
// it's mounted to the DOM.
mounted: function () {
  // Pikaday is a 3rd-party datepicker library
  this.picker = new Pikaday({
    field: this.$refs.input,
    format: 'YYYY-MM-DD'
  })
},
// Right before the component is destroyed,
// also destroy the datepicker.
beforeDestroy: function () {
  this.picker.destroy()
}
```

This has two potential issues:

- It requires saving the `picker` to the component instance, when it's possible that only lifecycle hooks need access to it. This isn't terrible, but it could be considered clutter.
- Our setup code is kept separate from our cleanup code, making it more difficult to programmatically clean up anything we set up.

You could resolve both issues with a programmatic listener:

```
mounted: function () {
  var picker = new Pikaday({
    field: this.$refs.input,
    format: 'YYYY-MM-DD'
  })

  this.$once('hook:beforeDestroy', function () {
    picker.destroy()
  })
}
```

Using this strategy, we could even use Pikaday with several input elements, with each new instance automatically cleaning up after itself:

```
mounted: function () {
  this.attachDatepicker('startDateInput')
  this.attachDatepicker('endDateInput')
```

```
},
methods: {
  attachDatepicker: function (refName) {
    var picker = new Pikaday({
      field: this.$refs[refName],
      format: 'YYYY-MM-DD'
    })

    this.$once('hook:beforeDestroy', function () {
      picker.destroy()
    })
  }
}
```

See this fiddle for the full code. Note, however, that if you find yourself having to do a lot of setup and cleanup within a single component, the best solution will usually be to create more modular components. In this case, we'd recommend creating a reusable `<input-datepicker>` component.

To learn more about programmatic listeners, check out the API for Events Instance Methods.

Note that Vue's event system is different from the browser's EventTarget API. Though they work similarly, *emit* < */code* >, < *code* >on, and $off are not aliases for dispatchEvent, addEventListener, and removeEventListener.

Circular References

Recursive Components

Components can recursively invoke themselves in their own template. However, they can only do so with the `name` option:

```
name: 'unique-name-of-my-component'
```

When you register a component globally using `Vue.component`, the global ID is automatically set as the component's `name` option.

```
Vue.component('unique-name-of-my-component', {
  // ...
})
```

If you're not careful, recursive components can also lead to infinite loops:

```
name: 'stack-overflow',
template: '<div><stack-overflow></stack-overflow></div>'
```

A component like the above will result in a "max stack size exceeded" error, so make sure recursive invocation is conditional (i.e. uses a `v-if` that will eventually

be `false`).

Circular References Between Components

Let's say you're building a file directory tree, like in Finder or File Explorer. You might have a `tree-folder` component with this template:

```
<p>
  <span>{{ folder.name }}</span>
  <tree-folder-contents :children="folder.children"/>
</p>
```

Then a `tree-folder-contents` component with this template:

```
<ul>
  <li v-for="child in children">
    <tree-folder v-if="child.children" :folder="child"/>
    <span v-else>{{ child.name }}</span>
  </li>
</ul>
```

When you look closely, you'll see that these components will actually be each other's descendent *and* ancestor in the render tree - a paradox! When registering components globally with `Vue.component`, this paradox is resolved for you automatically. If that's you, you can stop reading here.

However, if you're requiring/importing components using a **module system**, e.g. via Webpack or Browserify, you'll get an error:

```
Failed to mount component: template or render function not defined.
```

To explain what's happening, let's call our components A and B. The module system sees that it needs A, but first A needs B, but B needs A, but A needs B, etc. It's stuck in a loop, not knowing how to fully resolve either component without first resolving the other. To fix this, we need to give the module system a point at which it can say, "A needs B *eventually*, but there's no need to resolve B first."

In our case, let's make that point the `tree-folder` component. We know the child that creates the paradox is the `tree-folder-contents` component, so we'll wait until the `beforeCreate` lifecycle hook to register it:

```
beforeCreate: function () {
  this.$options.components.TreeFolderContents = require('./tree-folder-contents.vue').defaul
}
```

Or alternatively, you could use Webpack's asynchronous `import` when you register the component locally:

```
components: {
  TreeFolderContents: () => import('./tree-folder-contents.vue')
}
```

Problem solved!

Alternate Template Definitions

Inline Templates

When the `inline-template` special attribute is present on a child component, the component will use its inner content as its template, rather than treating it as distributed content. This allows more flexible template-authoring.

```
<my-component inline-template>
  <div>
    <p>These are compiled as the component's own template.</p>
    <p>Not parent's transclusion content.</p>
  </div>
</my-component>
```

However, inline-template makes the scope of your templates harder to reason about. As a best practice, prefer defining templates inside the component using the template option or in a <template> element in a .vue file.

X-Templates

Another way to define templates is inside of a script element with the type `text/x-template`, then referencing the template by an id. For example:

```
<script type="text/x-template" id="hello-world-template">
  <p>Hello hello hello</p>
</script>
```

```
Vue.component('hello-world', {
  template: '#hello-world-template'
})
```

These can be useful for demos with large templates or in extremely small applications, but should otherwise be avoided, because they separate templates from the rest of the component definition.

Controlling Updates

Thanks to Vue's Reactivity system, it always knows when to update (if you use it correctly). There are edge cases, however, when you might want to force

an update, despite the fact that no reactive data has changed. Then there are other cases when you might want to prevent unnecessary updates.

Forcing an Update

If you find yourself needing to force an update in Vue, in 99.99% of cases, you've made a mistake somewhere.

You may not have accounted for change detection caveats with arrays or objects, or you may be relying on state that isn't tracked by Vue's reactivity system, e.g. with `data`.

However, if you've ruled out the above and find yourself in this extremely rare situation of having to manually force an update, you can do so with `$forceUpdate`.

Cheap Static Components with `v-once`

Rendering plain HTML elements is very fast in Vue, but sometimes you might have a component that contains **a lot** of static content. In these cases, you can ensure that it's only evaluated once and then cached by adding the `v-once` directive to the root element, like this:

```
Vue.component('terms-of-service', {
  template: `
    <div v-once>
      <h1>Terms of Service</h1>
      ... a lot of static content ...
    </div>
  `
})
```

Once again, try not to overuse this pattern. While convenient in those rare cases when you have to render a lot of static content, it's simply not necessary unless you actually notice slow rendering – plus, it could cause a lot of confusion later. For example, imagine another developer who's not familiar with v-once or simply misses it in the template. They might spend hours trying to figure out why the template isn't updating correctly.

Overview

Vue provides a variety of ways to apply transition effects when items are inserted, updated, or removed from the DOM. This includes tools to:

- automatically apply classes for CSS transitions and animations
- integrate 3rd-party CSS animation libraries, such as Animate.css
- use JavaScript to directly manipulate the DOM during transition hooks

- integrate 3rd-party JavaScript animation libraries, such as Velocity.js

On this page, we'll only cover entering, leaving, and list transitions, but you can see the next section for managing state transitions.

Transitioning Single Elements/Components

Vue provides a `transition` wrapper component, allowing you to add entering/leaving transitions for any element or component in the following contexts:

- Conditional rendering (using `v-if`)
- Conditional display (using `v-show`)
- Dynamic components
- Component root nodes

This is what an example looks like in action:

```
<div id="demo">
  <button v-on:click="show = !show">
    Toggle
  </button>
  <transition name="fade">
    <p v-if="show">hello</p>
  </transition>
</div>
```

```
new Vue({
  el: '#demo',
  data: {
    show: true
  }
})
```

```
.fade-enter-active, .fade-leave-active {
  transition: opacity .5s;
}
.fade-enter, .fade-leave-to /* .fade-leave-active below version 2.1.8 */ {
  opacity: 0;
}
```

{% raw %}

Toggle

hello

{% endraw %}

When an element wrapped in a `transition` component is inserted or removed, this is what happens:

1. Vue will automatically sniff whether the target element has CSS transitions or animations applied. If it does, CSS transition classes will be added/removed at appropriate timings.
2. If the transition component provided JavaScript hooks, these hooks will be called at appropriate timings.
3. If no CSS transitions/animations are detected and no JavaScript hooks are provided, the DOM operations for insertion and/or removal will be executed immediately on next frame (Note: this is a browser animation frame, different from Vue's concept of `nextTick`).

Transition Classes

There are six classes applied for enter/leave transitions.

1. `v-enter`: Starting state for enter. Added before element is inserted, removed one frame after element is inserted.
2. `v-enter-active`: Active state for enter. Applied during the entire entering phase. Added before element is inserted, removed when transition/animation finishes. This class can be used to define the duration, delay and easing curve for the entering transition.
3. `v-enter-to`: **Only available in versions 2.1.8+.** Ending state for enter. Added one frame after element is inserted (at the same time `v-enter` is removed), removed when transition/animation finishes.
4. `v-leave`: Starting state for leave. Added immediately when a leaving transition is triggered, removed after one frame.
5. `v-leave-active`: Active state for leave. Applied during the entire leaving phase. Added immediately when leave transition is triggered, removed when the transition/animation finishes. This class can be used to define the duration, delay and easing curve for the leaving transition.
6. `v-leave-to`: **Only available in versions 2.1.8+.** Ending state for leave. Added one frame after a leaving transition is triggered (at the same time `v-leave` is removed), removed when the transition/animation finishes.

Each of these classes will be prefixed with the name of the transition. Here the `v-` prefix is the default when you use a `<transition>` element with no name. If you use `<transition name="my-transition">` for example, then the `v-enter` class would instead be `my-transition-enter`.

`v-enter-active` and `v-leave-active` give you the ability to specify different easing curves for enter/leave transitions, which you'll see an example of in the following section.

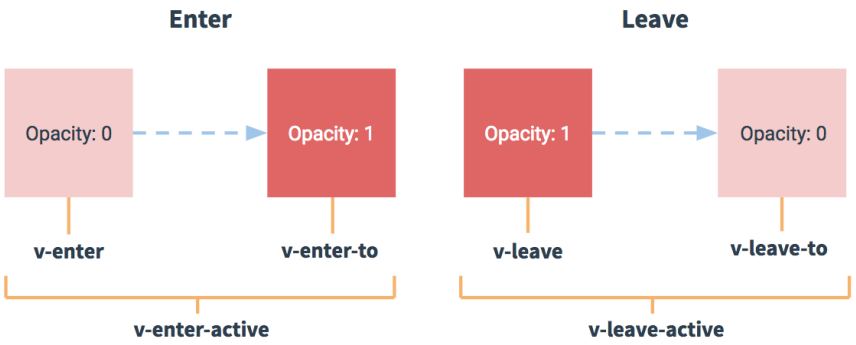

Figure 4: Transition Diagram

CSS Transitions

One of the most common transition types uses CSS transitions. Here's an example:

```
<div id="example-1">
  <button @click="show = !show">
    Toggle render
  </button>
  <transition name="slide-fade">
    <p v-if="show">hello</p>
  </transition>
</div>
```

```
new Vue({
  el: '#example-1',
  data: {
    show: true
  }
})
```

```
/* Enter and leave animations can use different */
/* durations and timing functions.              */
.slide-fade-enter-active {
  transition: all .3s ease;
}
.slide-fade-leave-active {
  transition: all .8s cubic-bezier(1.0, 0.5, 0.8, 1.0);
}
```

```
.slide-fade-enter, .slide-fade-leave-to
/* .slide-fade-leave-active below version 2.1.8 */ {
  transform: translateX(10px);
  opacity: 0;
}
```

{% raw %}

<button @click="show = !show"> Toggle render

hello

{% endraw %}

CSS Animations

CSS animations are applied in the same way as CSS transitions, the difference being that `v-enter` is not removed immediately after the element is inserted, but on an `animationend` event.

Here's an example, omitting prefixed CSS rules for the sake of brevity:

```
<div id="example-2">
  <button @click="show = !show">Toggle show</button>
  <transition name="bounce">
    <p v-if="show">Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris facilisis
  </transition>
</div>
```

```
new Vue({
  el: '#example-2',
  data: {
    show: true
  }
})
```

```
.bounce-enter-active {
  animation: bounce-in .5s;
}
.bounce-leave-active {
  animation: bounce-in .5s reverse;
}
@keyframes bounce-in {
  0% {
    transform: scale(0);
  }
  50% {
    transform: scale(1.5);
  }
```

```
  100% {
    transform: scale(1);
  }
}
```

{% raw %}

<button @click=“show = !show”>Toggle show

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris facilisis enim libero, at lacinia diam fermentum id. Pellentesque habitant morbi tristique senectus et netus.

{% endraw %}

Custom Transition Classes

You can also specify custom transition classes by providing the following attributes:

- `enter-class`
- `enter-active-class`
- `enter-to-class` (2.1.8+)
- `leave-class`
- `leave-active-class`
- `leave-to-class` (2.1.8+)

These will override the conventional class names. This is especially useful when you want to combine Vue’s transition system with an existing CSS animation library, such as Animate.css.

Here’s an example:

```
<link href="https://cdn.jsdelivr.net/npm/animate.css@3.5.1" rel="stylesheet" type="text/css"

<div id="example-3">
  <button @click="show = !show">
    Toggle render
  </button>
  <transition
    name="custom-classes-transition"
    enter-active-class="animated tada"
    leave-active-class="animated bounceOutRight"
  >
    <p v-if="show">hello</p>
  </transition>
</div>
```

```
new Vue({
  el: '#example-3',
  data: {
    show: true
  }
})
```

{% raw %}

<button @click=“show = !show”> Toggle render

hello

{% endraw %}

Using Transitions and Animations Together

Vue needs to attach event listeners in order to know when a transition has ended. It can either be `transitionend` or `animationend`, depending on the type of CSS rules applied. If you are only using one or the other, Vue can automatically detect the correct type.

However, in some cases you may want to have both on the same element, for example having a CSS animation triggered by Vue, along with a CSS transition effect on hover. In these cases, you will have to explicitly declare the type you want Vue to care about in a `type` attribute, with a value of either `animation` or `transition`.

Explicit Transition Durations

> New in 2.2.0+

In most cases, Vue can automatically figure out when the transition has finished. By default, Vue waits for the first `transitionend` or `animationend` event on the root transition element. However, this may not always be desired - for example, we may have a choreographed transition sequence where some nested inner elements have a delayed transition or a longer transition duration than the root transition element.

In such cases you can specify an explicit transition duration (in milliseconds) using the `duration` prop on the `<transition>` component:

```
<transition :duration="1000">...</transition>
```

You can also specify separate values for enter and leave durations:

```
<transition :duration="{ enter: 500, leave: 800 }">...</transition>
```

JavaScript Hooks

You can also define JavaScript hooks in attributes:

```
<transition
  v-on:before-enter="beforeEnter"
  v-on:enter="enter"
  v-on:after-enter="afterEnter"
  v-on:enter-cancelled="enterCancelled"

  v-on:before-leave="beforeLeave"
  v-on:leave="leave"
  v-on:after-leave="afterLeave"
  v-on:leave-cancelled="leaveCancelled"
>
  <!-- ... -->
</transition>
```

```
// ...
methods: {
  // --------
  // ENTERING
  // --------

  beforeEnter: function (el) {
    // ...
  },
  // the done callback is optional when
  // used in combination with CSS
  enter: function (el, done) {
    // ...
    done()
  },
  afterEnter: function (el) {
    // ...
  },
  enterCancelled: function (el) {
    // ...
  },

  // --------
  // LEAVING
  // --------

  beforeLeave: function (el) {
    // ...
```

```
  },
  // the done callback is optional when
  // used in combination with CSS
  leave: function (el, done) {
    // ...
    done()
  },
  afterLeave: function (el) {
    // ...
  },
  // leaveCancelled only available with v-show
  leaveCancelled: function (el) {
    // ...
  }
}
```

These hooks can be used in combination with CSS transitions/animations or on their own.

When using JavaScript-only transitions, **the `done` callbacks are required for the `enter` and `leave` hooks**. Otherwise, the hooks will be called synchronously and the transition will finish immediately.

It's also a good idea to explicitly add `v-bind:css="false"` for JavaScript-only transitions so that Vue can skip the CSS detection. This also prevents CSS rules from accidentally interfering with the transition.

Now let's dive into an example. Here's a JavaScript transition using Velocity.js:

```
<!--
Velocity works very much like jQuery.animate and is
a great option for JavaScript animations
-->
<script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script

<div id="example-4">
  <button @click="show = !show">
    Toggle
  </button>
  <transition
    v-on:before-enter="beforeEnter"
    v-on:enter="enter"
    v-on:leave="leave"
    v-bind:css="false"
  >
    <p v-if="show">
      Demo
    </p>
```

```
  </transition>
</div>

new Vue({
  el: '#example-4',
  data: {
    show: false
  },
  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
    },
    enter: function (el, done) {
      Velocity(el, { opacity: 1, fontSize: '1.4em' }, { duration: 300 })
      Velocity(el, { fontSize: '1em' }, { complete: done })
    },
    leave: function (el, done) {
      Velocity(el, { translateX: '15px', rotateZ: '50deg' }, { duration: 600 })
      Velocity(el, { rotateZ: '100deg' }, { loop: 2 })
      Velocity(el, {
        rotateZ: '45deg',
        translateY: '30px',
        translateX: '30px',
        opacity: 0
      }, { complete: done })
    }
  }
})
```

{% raw %}

<button @click="show = !show"> Toggle

Demo

{% endraw %}

Transitions on Initial Render

If you also want to apply a transition on the initial render of a node, you can add the `appear` attribute:

```
<transition appear>
  <!-- ... -->
</transition>
```

By default, this will use the transitions specified for entering and leaving. If you'd like however, you can also specify custom CSS classes:

```
<transition
  appear
  appear-class="custom-appear-class"
  appear-to-class="custom-appear-to-class" (2.1.8+)
  appear-active-class="custom-appear-active-class"
>
  <!-- ... -->
</transition>
```

and custom JavaScript hooks:

```
<transition
  appear
  v-on:before-appear="customBeforeAppearHook"
  v-on:appear="customAppearHook"
  v-on:after-appear="customAfterAppearHook"
  v-on:appear-cancelled="customAppearCancelledHook"
>
  <!-- ... -->
</transition>
```

Transitioning Between Elements

We discuss transitioning between components later, but you can also transition between raw elements using `v-if`/`v-else`. One of the most common two-element transitions is between a list container and a message describing an empty list:

```
<transition>
  <table v-if="items.length > 0">
    <!-- ... -->
  </table>
  <p v-else>Sorry, no items found.</p>
</transition>
```

This works well, but there's one caveat to be aware of:

When toggling between elements that have **the same tag name**, you must tell Vue that they are distinct elements by giving them unique `key` attributes. Otherwise, Vue's compiler will only replace the content of the element for efficiency. Even when technically unnecessary though, **it's considered good practice to always key multiple items within a `<transition>` component.**

For example:

```
<transition>
  <button v-if="isEditing" key="save">
    Save
```

```
  </button>
  <button v-else key="edit">
    Edit
  </button>
</transition>
```

In these cases, you can also use the `key` attribute to transition between different states of the same element. Instead of using `v-if` and `v-else`, the above example could be rewritten as:

```
<transition>
  <button v-bind:key="isEditing">
    {{ isEditing ? 'Save' : 'Edit' }}
  </button>
</transition>
```

It's actually possible to transition between any number of elements, either by using multiple `v-if`s or binding a single element to a dynamic property. For example:

```
<transition>
  <button v-if="docState === 'saved'" key="saved">
    Edit
  </button>
  <button v-if="docState === 'edited'" key="edited">
    Save
  </button>
  <button v-if="docState === 'editing'" key="editing">
    Cancel
  </button>
</transition>
```

Which could also be written as:

```
<transition>
  <button v-bind:key="docState">
    {{ buttonMessage }}
  </button>
</transition>

// ...
computed: {
  buttonMessage: function () {
    switch (this.docState) {
      case 'saved': return 'Edit'
      case 'edited': return 'Save'
      case 'editing': return 'Cancel'
    }
  }
```

```
}
```

Transition Modes

There's still one problem though. Try clicking the button below:

{% raw %}

<button v-if="on" key="on" @click="on = false"> on <button v-else key="off" @click="on = true"> off

{% endraw %}

As it's transitioning between the "on" button and the "off" button, both buttons are rendered - one transitioning out while the other transitions in. This is the default behavior of `<transition>` - entering and leaving happens simultaneously.

Sometimes this works great, like when transitioning items are absolutely positioned on top of each other:

{% raw %}

```
<transition name="no-mode-absolute-fade">
  <button v-if="on" key="on" @click="on = false">
    on
  </button>
  <button v-else key="off" @click="on = true">
    off
  </button>
</transition>
```

{% endraw %}

And then maybe also translated so that they look like slide transitions:

{% raw %}

```
<transition name="no-mode-translate-fade">
  <button v-if="on" key="on" @click="on = false">
    on
  </button>
  <button v-else key="off" @click="on = true">
    off
  </button>
</transition>
```

{% endraw %}

Simultaneous entering and leaving transitions aren't always desirable though, so Vue offers some alternative **transition modes**:

- `in-out`: New element transitions in first, then when complete, the current element transitions out.
- `out-in`: Current element transitions out first, then when complete, the new element transitions in.

Now let's update the transition for our on/off buttons with `out-in`:

```
<transition name="fade" mode="out-in">
  <!-- ... the buttons ... -->
</transition>
```

{% raw %}

<button v-if="on" key="on" @click="on = false"> on <button v-else key="off" @click="on = true"> off

{% endraw %}

With one attribute addition, we've fixed that original transition without having to add any special styling.

The `in-out` mode isn't used as often, but can sometimes be useful for a slightly different transition effect. Let's try combining it with the slide-fade transition we worked on earlier:

{% raw %}

```
<transition name="in-out-translate-fade" mode="in-out">
  <button v-if="on" key="on" @click="on = false">
    on
  </button>
  <button v-else key="off" @click="on = true">
    off
  </button>
</transition>
```

{% endraw %}

Pretty cool, right?

Transitioning Between Components

Transitioning between components is even simpler - we don't even need the `key` attribute. Instead, we wrap a dynamic component:

```
<transition name="component-fade" mode="out-in">
  <component v-bind:is="view"></component>
</transition>
```

```
new Vue({
  el: '#transition-components-demo',
```

```
  data: {
    view: 'v-a'
  },
  components: {
    'v-a': {
      template: '<div>Component A</div>'
    },
    'v-b': {
      template: '<div>Component B</div>'
    }
  }
})
```

```
.component-fade-enter-active, .component-fade-leave-active {
  transition: opacity .3s ease;
}
.component-fade-enter, .component-fade-leave-to
/* .component-fade-leave-active below version 2.1.8 */ {
  opacity: 0;
}
```

{% raw %}

A B

{% endraw %}

List Transitions

So far, we've managed transitions for:

- Individual nodes
- Multiple nodes where only 1 is rendered at a time

So what about for when we have a whole list of items we want to render simultaneously, for example with `v-for`? In this case, we'll use the `<transition-group>` component. Before we dive into an example though, there are a few things that are important to know about this component:

- Unlike `<transition>`, it renders an actual element: a `<span>` by default. You can change the element that's rendered with the `tag` attribute.
- Transition modes are not available, because we are no longer alternating between mutually exclusive elements.
- Elements inside are **always required** to have a unique `key` attribute.

List Entering/Leaving Transitions

Now let's dive into an example, transitioning entering and leaving using the same CSS classes we've used previously:

```
<div id="list-demo">
  <button v-on:click="add">Add</button>
  <button v-on:click="remove">Remove</button>
  <transition-group name="list" tag="p">
    <span v-for="item in items" v-bind:key="item" class="list-item">
      {{ item }}
    </span>
  </transition-group>
</div>
```

```
new Vue({
  el: '#list-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9],
    nextNum: 10
  },
  methods: {
    randomIndex: function () {
      return Math.floor(Math.random() * this.items.length)
    },
    add: function () {
      this.items.splice(this.randomIndex(), 0, this.nextNum++)
    },
    remove: function () {
      this.items.splice(this.randomIndex(), 1)
    },
  }
})
```

```
.list-item {
  display: inline-block;
  margin-right: 10px;
}
.list-enter-active, .list-leave-active {
  transition: all 1s;
}
.list-enter, .list-leave-to /* .list-leave-active below version 2.1.8 */ {
  opacity: 0;
  transform: translateY(30px);
}
```

{% raw %}

Add Remove <span v-for=“item in items” :key=“item” class=“list-item”> {{ item }}

{% endraw %}

There's one problem with this example. When you add or remove an item, the ones around it instantly snap into their new place instead of smoothly transitioning. We'll fix that later.

List Move Transitions

The `<transition-group>` component has another trick up its sleeve. It can not only animate entering and leaving, but also changes in position. The only new concept you need to know to use this feature is the addition of **the `v-move` class**, which is added when items are changing positions. Like the other classes, its prefix will match the value of a provided `name` attribute and you can also manually specify a class with the `move-class` attribute.

This class is mostly useful for specifying the transition timing and easing curve, as you'll see below:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"></script

<div id="flip-list-demo" class="demo">
  <button v-on:click="shuffle">Shuffle</button>
  <transition-group name="flip-list" tag="ul">
    <li v-for="item in items" v-bind:key="item">
      {{ item }}
    </li>
  </transition-group>
</div>

new Vue({
  el: '#flip-list-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9]
  },
  methods: {
    shuffle: function () {
      this.items = _.shuffle(this.items)
    }
  }
})

.flip-list-move {
  transition: transform 1s;
}
```

{% raw %}

Shuffle <li v-for=“item in items” :key=“item”> {{ item }}

{% endraw %}

This might seem like magic, but under the hood, Vue is using an animation technique called FLIP to smoothly transition elements from their old position to their new position using transforms.

We can combine this technique with our previous implementation to animate every possible change to our list!

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"></script

<div id="list-complete-demo" class="demo">
  <button v-on:click="shuffle">Shuffle</button>
  <button v-on:click="add">Add</button>
  <button v-on:click="remove">Remove</button>
  <transition-group name="list-complete" tag="p">
    <span
      v-for="item in items"
      v-bind:key="item"
      class="list-complete-item"
    >
      {{ item }}
    </span>
  </transition-group>
</div>
```

```
new Vue({
  el: '#list-complete-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9],
    nextNum: 10
  },
  methods: {
    randomIndex: function () {
      return Math.floor(Math.random() * this.items.length)
    },
    add: function () {
      this.items.splice(this.randomIndex(), 0, this.nextNum++)
    },
    remove: function () {
      this.items.splice(this.randomIndex(), 1)
    },
    shuffle: function () {
      this.items = _.shuffle(this.items)
    }
```

```
  }
})

.list-complete-item {
  transition: all 1s;
  display: inline-block;
  margin-right: 10px;
}
.list-complete-enter, .list-complete-leave-to
/* .list-complete-leave-active below version 2.1.8 */ {
  opacity: 0;
  transform: translateY(30px);
}
.list-complete-leave-active {
  position: absolute;
}
```

{% raw %}

Shuffle Add Remove <span v-for="item in items" :key="item" class="list-complete-item"> {{ item }}

{% endraw %}

One important note is that these FLIP transitions do not work with elements set to `display: inline`. As an alternative, you can use `display: inline-block` or place elements in a flex context.

These FLIP animations are also not limited to a single axis. Items in a multidimensional grid can be transitioned too:

{% raw %}

Lazy Sudoku

Keep hitting the shuffle button until you win.

<button @click="shuffle"> Shuffle <div v-for="cell in cells" :key="cell.id" class="cell"> {{ cell.number }}

{% endraw %}

Staggering List Transitions

By communicating with JavaScript transitions through data attributes, it's also possible to stagger transitions in a list:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script

<div id="staggered-list-demo">
  <input v-model="query">
```

```
  <transition-group
    name="staggered-fade"
    tag="ul"
    v-bind:css="false"
    v-on:before-enter="beforeEnter"
    v-on:enter="enter"
    v-on:leave="leave"
  >
    <li
      v-for="(item, index) in computedList"
      v-bind:key="item.msg"
      v-bind:data-index="index"
    >{{ item.msg }}</li>
  </transition-group>
</div>

new Vue({
  el: '#staggered-list-demo',
  data: {
    query: '',
    list: [
      { msg: 'Bruce Lee' },
      { msg: 'Jackie Chan' },
      { msg: 'Chuck Norris' },
      { msg: 'Jet Li' },
      { msg: 'Kung Fury' }
    ]
  },
  computed: {
    computedList: function () {
      var vm = this
      return this.list.filter(function (item) {
        return item.msg.toLowerCase().indexOf(vm.query.toLowerCase()) !== -1
      })
    }
  },
  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
      el.style.height = 0
    },
    enter: function (el, done) {
      var delay = el.dataset.index * 150
      setTimeout(function () {
        Velocity(
          el,
```

```
          { opacity: 1, height: '1.6em' },
          { complete: done }
        )
      }, delay)
    },
    leave: function (el, done) {
      var delay = el.dataset.index * 150
      setTimeout(function () {
        Velocity(
          el,
          { opacity: 0, height: 0 },
          { complete: done }
        )
      }, delay)
    }
  }
})
```

{% raw %}

{{ item.msg }}

{% endraw %}

Reusable Transitions

Transitions can be reused through Vue's component system. To create a reusable transition, all you have to do is place a `<transition>` or `<transition-group>` component at the root, then pass any children into the transition component.

Here's an example using a template component:

```
Vue.component('my-special-transition', {
  template: '\
    <transition\
      name="very-special-transition"\
      mode="out-in"\
      v-on:before-enter="beforeEnter"\
      v-on:after-enter="afterEnter"\
    >\
      <slot></slot>\
    </transition>\
  ',
  methods: {
    beforeEnter: function (el) {
      // ...
    },
```

```
    afterEnter: function (el) {
      // ...
    }
  }
})
```

And functional components are especially well-suited to this task:

```
Vue.component('my-special-transition', {
  functional: true,
  render: function (createElement, context) {
    var data = {
      props: {
        name: 'very-special-transition',
        mode: 'out-in'
      },
      on: {
        beforeEnter: function (el) {
          // ...
        },
        afterEnter: function (el) {
          // ...
        }
      }
    }
    return createElement('transition', data, context.children)
  }
})
```

Dynamic Transitions

Yes, even transitions in Vue are data-driven! The most basic example of a dynamic transition binds the `name` attribute to a dynamic property.

```
<transition v-bind:name="transitionName">
  <!-- ... -->
</transition>
```

This can be useful when you've defined CSS transitions/animations using Vue's transition class conventions and want to switch between them.

Really though, any transition attribute can be dynamically bound. And it's not only attributes. Since event hooks are methods, they have access to any data in the context. That means depending on the state of your component, your JavaScript transitions can behave differently.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script
```

```
<div id="dynamic-fade-demo" class="demo">
  Fade In: <input type="range" v-model="fadeInDuration" min="0" v-bind:max="maxFadeDuration"
  Fade Out: <input type="range" v-model="fadeOutDuration" min="0" v-bind:max="maxFadeDuratio
  <transition
    v-bind:css="false"
    v-on:before-enter="beforeEnter"
    v-on:enter="enter"
    v-on:leave="leave"
  >
    <p v-if="show">hello</p>
  </transition>
  <button
    v-if="stop"
    v-on:click="stop = false; show = false"
  >Start animating</button>
  <button
    v-else
    v-on:click="stop = true"
  >Stop it!</button>
</div>
```

```
new Vue({
  el: '#dynamic-fade-demo',
  data: {
    show: true,
    fadeInDuration: 1000,
    fadeOutDuration: 1000,
    maxFadeDuration: 1500,
    stop: true
  },
  mounted: function () {
    this.show = false
  },
  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
    },
    enter: function (el, done) {
      var vm = this
      Velocity(el,
        { opacity: 1 },
        {
          duration: this.fadeInDuration,
          complete: function () {
            done()
            if (!vm.stop) vm.show = false
```

```
          }
        }
      )
    },
    leave: function (el, done) {
      var vm = this
      Velocity(el,
        { opacity: 0 },
        {
          duration: this.fadeOutDuration,
          complete: function () {
            done()
            vm.show = true
          }
        }
      )
    }
  }
})
```

{% raw %}

Fade In: Fade Out:

hello

Start animating Stop it!

{% endraw %}

Finally, the ultimate way of creating dynamic transitions is through components that accept props to change the nature of the transition(s) to be used. It may sound cheesy, but the only limit really is your imagination.

Vue's transition system offers many simple ways to animate entering, leaving, and lists, but what about animating your data itself? For example:

- numbers and calculations
- colors displayed
- the positions of SVG nodes
- the sizes and other properties of elements

All of these are either already stored as raw numbers or can be converted into numbers. Once we do that, we can animate these state changes using 3rd-party libraries to tween state, in combination with Vue's reactivity and component systems.

Animating State with Watchers

Watchers allow us to animate changes of any numerical property into another property. That may sound complicated in the abstract, so let's dive into an example using GreenSock:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/1.20.3/TweenMax.min.js"></script>

<div id="animated-number-demo">
  <input v-model.number="number" type="number" step="20">
  <p>{{ animatedNumber }}</p>
</div>
```

```
new Vue({
  el: '#animated-number-demo',
  data: {
    number: 0,
    tweenedNumber: 0
  },
  computed: {
    animatedNumber: function() {
      return this.tweenedNumber.toFixed(0);
    }
  },
  watch: {
    number: function(newValue) {
      TweenLite.to(this.$data, 0.5, { tweenedNumber: newValue });
    }
  }
})
```

{% raw %}

<input v-model.number="number" type="number" step="20">

{{ animatedNumber }}

{% endraw %}

When you update the number, the change is animated below the input. This makes for a nice demo, but what about something that isn't directly stored as a number, like any valid CSS color for example? Here's how we could accomplish this with Tween.js and Color.js:

```
<script src="https://cdn.jsdelivr.net/npm/tween.js@16.3.4"></script>
<script src="https://cdn.jsdelivr.net/npm/color-js@1.0.3"></script>

<div id="example-7">
  <input
```

```
    v-model="colorQuery"
    v-on:keyup.enter="updateColor"
    placeholder="Enter a color"
  >
  <button v-on:click="updateColor">Update</button>
  <p>Preview:</p>
  <span
    v-bind:style="{ backgroundColor: tweenedCSSColor }"
    class="example-7-color-preview"
  ></span>
  <p>{{ tweenedCSSColor }}</p>
</div>
```

```
var Color = net.brehaut.Color

new Vue({
  el: '#example-7',
  data: {
    colorQuery: '',
    color: {
      red: 0,
      green: 0,
      blue: 0,
      alpha: 1
    },
    tweenedColor: {}
  },
  created: function () {
    this.tweenedColor = Object.assign({}, this.color)
  },
  watch: {
    color: function () {
      function animate () {
        if (TWEEN.update()) {
          requestAnimationFrame(animate)
        }
      }

      new TWEEN.Tween(this.tweenedColor)
        .to(this.color, 750)
        .start()

      animate()
    }
  },
  computed: {
```

```
    tweenedCSSColor: function () {
      return new Color({
        red: this.tweenedColor.red,
        green: this.tweenedColor.green,
        blue: this.tweenedColor.blue,
        alpha: this.tweenedColor.alpha
      }).toCSS()
    }
  },
  methods: {
    updateColor: function () {
      this.color = new Color(this.colorQuery).toRGB()
      this.colorQuery = ''
    }
  }
})

.example-7-color-preview {
  display: inline-block;
  width: 50px;
  height: 50px;
}
```

{% raw %}

<input v-model=“colorQuery” v-on:keyup.enter=“updateColor” placeholder=“Enter a color” > Update

Preview:

{{ tweenedCSSColor }}

{% endraw %}

Dynamic State Transitions

As with Vue’s transition components, the data backing state transitions can be updated in real time, which is especially useful for prototyping! Even using a simple SVG polygon, you can achieve many effects that would be difficult to conceive of until you’ve played with the variables a little.

{% raw %}

<polygon :points=“points” class=“demo-polygon”> Sides: {{ sides }} <input class=“demo-range-input” type=“range” min=“3” max=“500” v-model.number=“sides” > Minimum Radius: {{ minRadius }}% <input class=“demo-range-input” type=“range” min=“0” max=“90” v-model.number=“minRadius” > Update Interval: {{ updateInterval }}

milliseconds <input class="demo-range-input" type="range" min="10" max="2000" v-model.number="updateInterval" >

{% endraw %}

See this fiddle for the complete code behind the above demo.

Organizing Transitions into Components

Managing many state transitions can quickly increase the complexity of a Vue instance or component. Fortunately, many animations can be extracted out into dedicated child components. Let's do this with the animated integer from our earlier example:

```
<script src="https://cdn.jsdelivr.net/npm/tween.js@16.3.4"></script>

<div id="example-8">
  <input v-model.number="firstNumber" type="number" step="20"> +
  <input v-model.number="secondNumber" type="number" step="20"> =
  {{ result }}
  <p>
    <animated-integer v-bind:value="firstNumber"></animated-integer> +
    <animated-integer v-bind:value="secondNumber"></animated-integer> =
    <animated-integer v-bind:value="result"></animated-integer>
  </p>
</div>
```

```
// This complex tweening logic can now be reused between
// any integers we may wish to animate in our application.
// Components also offer a clean interface for configuring
// more dynamic transitions and complex transition
// strategies.
Vue.component('animated-integer', {
  template: '<span>{{ tweeningValue }}</span>',
  props: {
    value: {
      type: Number,
      required: true
    }
  },
  data: function () {
    return {
      tweeningValue: 0
    }
  },
  watch: {
    value: function (newValue, oldValue) {
```

```
      this.tween(oldValue, newValue)
    }
  },
  mounted: function () {
    this.tween(0, this.value)
  },
  methods: {
    tween: function (startValue, endValue) {
      var vm = this
      function animate () {
        if (TWEEN.update()) {
          requestAnimationFrame(animate)
        }
      }

      new TWEEN.Tween({ tweeningValue: startValue })
        .to({ tweeningValue: endValue }, 500)
        .onUpdate(function (object) {
          vm.tweeningValue = object.tweeningValue.toFixed(0)
        })
        .start()

      animate()
    }
  }
})

// All complexity has now been removed from the main Vue instance!
new Vue({
  el: '#example-8',
  data: {
    firstNumber: 20,
    secondNumber: 40
  },
  computed: {
    result: function () {
      return this.firstNumber + this.secondNumber
    }
  }
})
```

{% raw %}

<input v-model.number="firstNumber" type="number" step="20"> + <input v-model.number="secondNumber" type="number" step="20"> = {{ result }}

{% endraw %}

{% endraw %}

Within child components, we can use any combination of transition strategies that have been covered on this page, along with those offered by Vue's built-in transition system. Together, there are very few limits to what can be accomplished.

Bringing Designs to Life

To animate, by one definition, means to bring to life. Unfortunately, when designers create icons, logos, and mascots, they're usually delivered as images or static SVGs. So although GitHub's octocat, Twitter's bird, and many other logos resemble living creatures, they don't really seem alive.

Vue can help. Since SVGs are just data, we only need examples of what these creatures look like when excited, thinking, or alarmed. Then Vue can help transition between these states, making your welcome pages, loading indicators, and notifications more emotionally compelling.

Sarah Drasner demonstrates this in the demo below, using a combination of timed and interactivity-driven state changes:

See the Pen Vue-controlled Wall-E by Sarah Drasner (@sdras) on CodePen.

Basics

Mixins are a flexible way to distribute reusable functionalities for Vue components. A mixin object can contain any component options. When a component uses a mixin, all options in the mixin will be "mixed" into the component's own options.

Example:

```
// define a mixin object
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}

// define a component that uses this mixin
var Component = Vue.extend({
```

```
  mixins: [myMixin]
})

var component = new Component() // => "hello from mixin!"
```

Option Merging

When a mixin and the component itself contain overlapping options, they will be "merged" using appropriate strategies.

For example, data objects undergo a shallow merge (one property deep), with the component's data taking priority in cases of conflicts.

```
var mixin = {
  data: function () {
    return {
      message: 'hello',
      foo: 'abc'
    }
  }
}

new Vue({
  mixins: [mixin],
  data: function () {
    return {
      message: 'goodbye',
      bar: 'def'
    }
  },
  created: function () {
    console.log(this.$data)
    // => { message: "goodbye", foo: "abc", bar: "def" }
  }
})
```

Hook functions with the same name are merged into an array so that all of them will be called. Mixin hooks will be called **before** the component's own hooks.

```
var mixin = {
  created: function () {
    console.log('mixin hook called')
  }
}

new Vue({
  mixins: [mixin],
```

```
  created: function () {
    console.log('component hook called')
  }
})

// => "mixin hook called"
// => "component hook called"
```

Options that expect object values, for example `methods`, `components` and `directives`, will be merged into the same object. The component's options will take priority when there are conflicting keys in these objects:

```
var mixin = {
  methods: {
    foo: function () {
      console.log('foo')
    },
    conflicting: function () {
      console.log('from mixin')
    }
  }
}

var vm = new Vue({
  mixins: [mixin],
  methods: {
    bar: function () {
      console.log('bar')
    },
    conflicting: function () {
      console.log('from self')
    }
  }
})

vm.foo() // => "foo"
vm.bar() // => "bar"
vm.conflicting() // => "from self"
```

Note that the same merge strategies are used in `Vue.extend()`.

Global Mixin

You can also apply a mixin globally. Use with caution! Once you apply a mixin globally, it will affect **every** Vue instance created afterwards. When used properly, this can be used to inject processing logic for custom options:

```
// inject a handler for `myOption` custom option
Vue.mixin({
  created: function () {
    var myOption = this.$options.myOption
    if (myOption) {
      console.log(myOption)
    }
  }
})

new Vue({
  myOption: 'hello!'
})
// => "hello!"
```

Use global mixins sparsely and carefully, because it affects every single Vue instance created, including third party components. In most cases, you should only use it for custom option handling like demonstrated in the example above. It's also a good idea to ship them as Plugins to avoid duplicate application.

Custom Option Merge Strategies

When custom options are merged, they use the default strategy which overwrites the existing value. If you want a custom option to be merged using custom logic, you need to attach a function to `Vue.config.optionMergeStrategies`:

```
Vue.config.optionMergeStrategies.myOption = function (toVal, fromVal) {
  // return mergedVal
}
```

For most object-based options, you can use the same strategy used by `methods`:

```
var strategies = Vue.config.optionMergeStrategies
strategies.myOption = strategies.methods
```

A more advanced example can be found on Vuex's 1.x merging strategy:

```
const merge = Vue.config.optionMergeStrategies.computed
Vue.config.optionMergeStrategies.vuex = function (toVal, fromVal) {
  if (!toVal) return fromVal
  if (!fromVal) return toVal
  return {
    getters: merge(toVal.getters, fromVal.getters),
    state: merge(toVal.state, fromVal.state),
    actions: merge(toVal.actions, fromVal.actions)
  }
}
```

Intro

In addition to the default set of directives shipped in core (`v-model` and `v-show`), Vue also allows you to register your own custom directives. Note that in Vue 2.0, the primary form of code reuse and abstraction is components - however there may be cases where you need some low-level DOM access on plain elements, and this is where custom directives would still be useful. An example would be focusing on an input element, like this one:

{% raw %}

{% endraw %}

When the page loads, that element gains focus (note: `autofocus` doesn't work on mobile Safari). In fact, if you haven't clicked on anything else since visiting this page, the input above should be focused now. Now let's build the directive that accomplishes this:

```
// Register a global custom directive called `v-focus`
Vue.directive('focus', {
  // When the bound element is inserted into the DOM...
  inserted: function (el) {
    // Focus the element
    el.focus()
  }
})
```

If you want to register a directive locally instead, components also accept a `directives` option:

```
directives: {
  focus: {
    // directive definition
    inserted: function (el) {
      el.focus()
    }
  }
}
```

Then in a template, you can use the new `v-focus` attribute on any element, like this:

```
<input v-focus>
```

Hook Functions

A directive definition object can provide several hook functions (all optional):

- `bind`: called only once, when the directive is first bound to the element. This is where you can do one-time setup work.
- `inserted`: called when the bound element has been inserted into its parent node (this only guarantees parent node presence, not necessarily in-document).
- `update`: called after the containing component's VNode has updated, **but possibly before its children have updated**. The directive's value may or may not have changed, but you can skip unnecessary updates by comparing the binding's current and old values (see below on hook arguments).
- `componentUpdated`: called after the containing component's VNode **and the VNodes of its children** have updated.
- `unbind`: called only once, when the directive is unbound from the element.

We'll explore the arguments passed into these hooks (i.e. `el`, `binding`, `vnode`, and `oldVnode`) in the next section.

Directive Hook Arguments

Directive hooks are passed these arguments:

- `el`: The element the directive is bound to. This can be used to directly manipulate the DOM.
- `binding`: An object containing the following properties.
  - `name`: The name of the directive, without the `v-` prefix.
  - `value`: The value passed to the directive. For example in `v-my-directive="1 + 1"`, the value would be `2`.
  - `oldValue`: The previous value, only available in `update` and `componentUpdated`. It is available whether or not the value has changed.
  - `expression`: The expression of the binding as a string. For example in `v-my-directive="1 + 1"`, the expression would be `"1 + 1"`.
  - `arg`: The argument passed to the directive, if any. For example in `v-my-directive:foo`, the arg would be `"foo"`.
  - `modifiers`: An object containing modifiers, if any. For example in `v-my-directive.foo.bar`, the modifiers object would be `{ foo: true, bar: true }`.
- `vnode`: The virtual node produced by Vue's compiler. See the VNode API for full details.
- `oldVnode`: The previous virtual node, only available in the `update` and `componentUpdated` hooks.

Apart from `el`, you should treat these arguments as read-only and never modify them. If you need to share information across hooks, it is recommended to do

so through element's dataset.

An example of a custom directive using some of these properties:

```
<div id="hook-arguments-example" v-demo:foo.a.b="message"></div>
```

```
Vue.directive('demo', {
  bind: function (el, binding, vnode) {
    var s = JSON.stringify
    el.innerHTML =
      'name: '       + s(binding.name) + '<br>' +
      'value: '      + s(binding.value) + '<br>' +
      'expression: ' + s(binding.expression) + '<br>' +
      'argument: '   + s(binding.arg) + '<br>' +
      'modifiers: '  + s(binding.modifiers) + '<br>' +
      'vnode keys: ' + Object.keys(vnode).join(', ')
  }
})

new Vue({
  el: '#hook-arguments-example',
  data: {
    message: 'hello!'
  }
})
```

{% raw %} <div id="hook-arguments-example" v-demo:foo.a.b="message" class="demo">

{% endraw %}

Function Shorthand

In many cases, you may want the same behavior on `bind` and `update`, but don't care about the other hooks. For example:

```
Vue.directive('color-swatch', function (el, binding) {
  el.style.backgroundColor = binding.value
})
```

Object Literals

If your directive needs multiple values, you can also pass in a JavaScript object literal. Remember, directives can take any valid JavaScript expression.

```
<div v-demo="{ color: 'white', text: 'hello!' }"></div>
```

```
Vue.directive('demo', function (el, binding) {
  console.log(binding.value.color) // => "white"
  console.log(binding.value.text)  // => "hello!"
})
```

Basics

Vue recommends using templates to build your HTML in the vast majority of cases. There are situations however, where you really need the full programmatic power of JavaScript. That's where you can use the **render function**, a closer-to-the-compiler alternative to templates.

Let's dive into a simple example where a `render` function would be practical. Say you want to generate anchored headings:

```
<h1>
  <a name="hello-world" href="#hello-world">
    Hello world!
  </a>
</h1>
```

For the HTML above, you decide you want this component interface:

```
<anchored-heading :level="1">Hello world!</anchored-heading>
```

When you get started with a component that only generates a heading based on the `level` prop, you quickly arrive at this:

```
<script type="text/x-template" id="anchored-heading-template">
  <h1 v-if="level === 1">
    <slot></slot>
  </h1>
  <h2 v-else-if="level === 2">
    <slot></slot>
  </h2>
  <h3 v-else-if="level === 3">
    <slot></slot>
  </h3>
  <h4 v-else-if="level === 4">
    <slot></slot>
  </h4>
  <h5 v-else-if="level === 5">
    <slot></slot>
  </h5>
  <h6 v-else-if="level === 6">
    <slot></slot>
  </h6>
</script>
```

```
Vue.component('anchored-heading', {
  template: '#anchored-heading-template',
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})
```

That template doesn't feel great. It's not only verbose, but we're duplicating `<slot></slot>` for every heading level and will have to do the same when we add the anchor element.

While templates work great for most components, it's clear that this isn't one of them. So let's try rewriting it with a `render` function:

```
Vue.component('anchored-heading', {
  render: function (createElement) {
    return createElement(
      'h' + this.level,   // tag name
      this.$slots.default // array of children
    )
  },
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})
```

Much simpler! Sort of. The code is shorter, but also requires greater familiarity with Vue instance properties. In this case, you have to know that when you pass children without a `slot` attribute into a component, like the `Hello world!` inside of `anchored-heading`, those children are stored on the component instance at `$slots.default`. If you haven't already, **it's recommended to read through the instance properties API before diving into render functions.**

Nodes, Trees, and the Virtual DOM

Before we dive into render functions, it's important to know a little about how browsers work. Take this HTML for example:

```
<div>
  <h1>My title</h1>
```

```
  Some text content
  <!-- TODO: Add tagline  -->
</div>
```

When a browser reads this code, it builds a tree of "DOM nodes" to help it keep track of everything, just as you might build a family tree to keep track of your extended family.

The tree of DOM nodes for the HTML above looks like this:

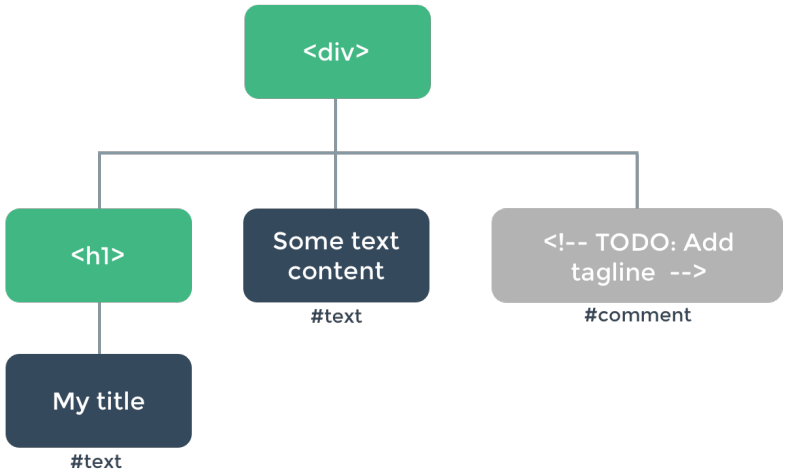

Figure 5: DOM Tree Visualization

Every element is a node. Every piece of text is a node. Even comments are nodes! A node is just a piece of the page. And as in a family tree, each node can have children (i.e. each piece can contain other pieces).

Updating all these nodes efficiently can be difficult, but thankfully, you never have to do it manually. Instead, you tell Vue what HTML you want on the page, in a template:

```
<h1>{{ blogTitle }}</h1>
```

Or a render function:

```
render: function (createElement) {
  return createElement('h1', this.blogTitle)
}
```

And in both cases, Vue automatically keeps the page updated, even when `blogTitle` changes.

The Virtual DOM

Vue accomplishes this by building a **virtual DOM** to keep track of the changes it needs to make to the real DOM. Taking a closer look at this line:

```
return createElement('h1', this.blogTitle)
```

What is `createElement` actually returning? It's not *exactly* a real DOM element. It could perhaps more accurately be named `createNodeDescription`, as it contains information describing to Vue what kind of node it should render on the page, including descriptions of any child nodes. We call this node description a "virtual node", usually abbreviated to **VNode**. "Virtual DOM" is what we call the entire tree of VNodes, built by a tree of Vue components.

`createElement` Arguments

The next thing you'll have to become familiar with is how to use template features in the `createElement` function. Here are the arguments that `createElement` accepts:

```
// @returns {VNode}
createElement(
  // {String | Object | Function}
  // An HTML tag name, component options, or async
  // function resolving to one of these. Required.
  'div',

  // {Object}
  // A data object corresponding to the attributes
  // you would use in a template. Optional.
  {
    // (see details in the next section below)
  },

  // {String | Array}
  // Children VNodes, built using `createElement()`,
  // or using strings to get 'text VNodes'. Optional.
  [
    'Some text comes first.',
    createElement('h1', 'A headline'),
    createElement(MyComponent, {
      props: {
        someProp: 'foobar'
      }
    })
```

```
  ]
)
```

The Data Object In-Depth

One thing to note: similar to how `v-bind:class` and `v-bind:style` have special treatment in templates, they have their own top-level fields in VNode data objects. This object also allows you to bind normal HTML attributes as well as DOM properties such as `innerHTML` (this would replace the `v-html` directive):

```
{
  // Same API as `v-bind:class`, accepting either
  // a string, object, or array of strings and objects.
  class: {
    foo: true,
    bar: false
  },
  // Same API as `v-bind:style`, accepting either
  // a string, object, or array of objects.
  style: {
    color: 'red',
    fontSize: '14px'
  },
  // Normal HTML attributes
  attrs: {
    id: 'foo'
  },
  // Component props
  props: {
    myProp: 'bar'
  },
  // DOM properties
  domProps: {
    innerHTML: 'baz'
  },
  // Event handlers are nested under `on`, though
  // modifiers such as in `v-on:keyup.enter` are not
  // supported. You'll have to manually check the
  // keyCode in the handler instead.
  on: {
    click: this.clickHandler
  },
  // For components only. Allows you to listen to
  // native events, rather than events emitted from
  // the component using `vm.$emit`.
```

```
  nativeOn: {
    click: this.nativeClickHandler
  },
  // Custom directives. Note that the `binding`'s
  // `oldValue` cannot be set, as Vue keeps track
  // of it for you.
  directives: [
    {
      name: 'my-custom-directive',
      value: '2',
      expression: '1 + 1',
      arg: 'foo',
      modifiers: {
        bar: true
      }
    }
  ],
  // Scoped slots in the form of
  // { name: props => VNode | Array<VNode> }
  scopedSlots: {
    default: props => createElement('span', props.text)
  },
  // The name of the slot, if this component is the
  // child of another component
  slot: 'name-of-slot',
  // Other special top-level properties
  key: 'myKey',
  ref: 'myRef'
}
```

Complete Example

With this knowledge, we can now finish the component we started:

```
var getChildrenTextContent = function (children) {
  return children.map(function (node) {
    return node.children
      ? getChildrenTextContent(node.children)
      : node.text
  }).join('')
}

Vue.component('anchored-heading', {
  render: function (createElement) {
    // create kebabCase id
```

```
    var headingId = getChildrenTextContent(this.$slots.default)
      .toLowerCase()
      .replace(/\W+/g, '-')
      .replace(/(^\-|\-$)/g, '')

    return createElement(
      'h' + this.level,
      [
        createElement('a', {
          attrs: {
            name: headingId,
            href: '#' + headingId
          }
        }, this.$slots.default)
      ]
    )
  },
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})
```

Constraints

VNodes Must Be Unique

All VNodes in the component tree must be unique. That means the following render function is invalid:

```
render: function (createElement) {
  var myParagraphVNode = createElement('p', 'hi')
  return createElement('div', [
    // Yikes - duplicate VNodes!
    myParagraphVNode, myParagraphVNode
  ])
}
```

If you really want to duplicate the same element/component many times, you can do so with a factory function. For example, the following render function is a perfectly valid way of rendering 20 identical paragraphs:

```
render: function (createElement) {
  return createElement('div',
    Array.apply(null, { length: 20 }).map(function () {
```

```
      return createElement('p', 'hi')
    })
  )
}
```

Replacing Template Features with Plain JavaScript

v-if and v-for

Wherever something can be easily accomplished in plain JavaScript, Vue render functions do not provide a proprietary alternative. For example, in a template using `v-if` and `v-for`:

```
<ul v-if="items.length">
  <li v-for="item in items">{{ item.name }}</li>
</ul>
<p v-else>No items found.</p>
```

This could be rewritten with JavaScript's `if`/`else` and `map` in a render function:

```
props: ['items'],
render: function (createElement) {
  if (this.items.length) {
    return createElement('ul', this.items.map(function (item) {
      return createElement('li', item.name)
    }))
  } else {
    return createElement('p', 'No items found.')
  }
}
```

v-model

There is no direct `v-model` counterpart in render functions - you will have to implement the logic yourself:

```
props: ['value'],
render: function (createElement) {
  var self = this
  return createElement('input', {
    domProps: {
      value: self.value
    },
    on: {
      input: function (event) {
        self.$emit('input', event.target.value)
```

```
      }
    }
  })
}
```

This is the cost of going lower-level, but it also gives you much more control over the interaction details compared to `v-model`.

Event & Key Modifiers

For the `.passive`, `.capture` and `.once` event modifiers, Vue offers prefixes that can be used with `on`:

Modifier(s)	Prefix
`.passive`	`&`
`.capture`	`!`
`.once`	`~`
`.capture.once` or `.once.capture`	`~!`

For example:

```
on: {
  '!click': this.doThisInCapturingMode,
  '~keyup': this.doThisOnce,
  '~!mouseover': this.doThisOnceInCapturingMode
}
```

For all other event and key modifiers, no proprietary prefix is necessary, because you can use event methods in the handler:

Modifier(s)	Equivalent in Handler
`.stop`	`event.stopPropagation()`
`.prevent`	`event.preventDefault()`
`.self`	`if (event.target !== event.currentTarget) return`
Keys:`.enter`, `.13`	`if (event.keyCode !== 13) return` (change `13` to another key code for other key modifiers)
Modifiers Keys:`.ctrl`, `.alt`, `.shift`, `.meta`	`if (!event.ctrlKey) return` (change `ctrlKey` to `altKey`, `shiftKey`, or `metaKey`, respectively)

Here's an example with all of these modifiers used together:

```
on: {
  keyup: function (event) {
    // Abort if the element emitting the event is not
    // the element the event is bound to
    if (event.target !== event.currentTarget) return
    // Abort if the key that went up is not the enter
    // key (13) and the shift key was not held down
    // at the same time
    if (!event.shiftKey || event.keyCode !== 13) return
    // Stop event propagation
    event.stopPropagation()
    // Prevent the default keyup handler for this element
    event.preventDefault()
    // ...
  }
}
```

Slots

You can access static slot contents as Arrays of VNodes from `this.$slots`:

```
render: function (createElement) {
  // `<div><slot></slot></div>`
  return createElement('div', this.$slots.default)
}
```

And access scoped slots as functions that return VNodes from `this.$scopedSlots`:

```
props: ['message'],
render: function (createElement) {
  // `<div><slot :text="message"></slot></div>`
  return createElement('div', [
    this.$scopedSlots.default({
      text: this.message
    })
  ])
}
```

To pass scoped slots to a child component using render functions, use the `scopedSlots` field in VNode data:

```
render: function (createElement) {
  return createElement('div', [
    createElement('child', {
      // pass `scopedSlots` in the data object
      // in the form of { name: props => VNode | Array<VNode> }
      scopedSlots: {
```

```
        default: function (props) {
          return createElement('span', props.text)
        }
      }
    })
  ])
}
```

JSX

If you're writing a lot of `render` functions, it might feel painful to write something like this:

```
createElement(
  'anchored-heading', {
    props: {
      level: 1
    }
  }, [
    createElement('span', 'Hello'),
    ' world!'
  ]
)
```

Especially when the template version is so simple in comparison:

```
<anchored-heading :level="1">
  <span>Hello</span> world!
</anchored-heading>
```

That's why there's a Babel plugin to use JSX with Vue, getting us back to a syntax that's closer to templates:

```
import AnchoredHeading from './AnchoredHeading.vue'

new Vue({
  el: '#demo',
  render: function (h) {
    return (
      <AnchoredHeading level={1}>
        <span>Hello</span> world!
      </AnchoredHeading>
    )
  }
})
```

Aliasing `createElement` to `h` is a common convention you'll see in the Vue ecosystem and is actually required for JSX. If `h` is not available in the scope,

your app will throw an error.

For more on how JSX maps to JavaScript, see the usage docs.

Functional Components

The anchored heading component we created earlier is relatively simple. It doesn't manage any state, watch any state passed to it, and it has no lifecycle methods. Really, it's only a function with some props.

In cases like this, we can mark components as `functional`, which means that they're stateless (no reactive data) and instanceless (no `this` context). A **functional component** looks like this:

```
Vue.component('my-component', {
  functional: true,
  // Props are optional
  props: {
    // ...
  },
  // To compensate for the lack of an instance,
  // we are now provided a 2nd context argument.
  render: function (createElement, context) {
    // ...
  }
})
```

> Note: in versions before 2.3.0, the `props` option is required if you wish to accept props in a functional component. In 2.3.0+ you can omit the `props` option and all attributes found on the component node will be implicitly extracted as props.

In 2.5.0+, if you are using single-file components, template-based functional components can be declared with:

```
<template functional>
</template>
```

Everything the component needs is passed through `context`, which is an object containing:

- `props`: An object of the provided props
- `children`: An array of the VNode children
- `slots`: A function returning a slots object
- `data`: The entire data object, passed to the component as the 2nd argument of `createElement`
- `parent`: A reference to the parent component
- `listeners`: (2.3.0+) An object containing parent-registered event listeners. This is an alias to `data.on`

- `injections`: (2.3.0+) if using the `inject` option, this will contain resolved injections.

After adding `functional: true`, updating the render function of our anchored heading component would require adding the `context` argument, updating `this.$slots.default` to `context.children`, then updating `this.level` to `context.props.level`.

Since functional components are just functions, they're much cheaper to render. However, the lack of a persistent instance means they won't show up in the Vue devtools component tree.

They're also very useful as wrapper components. For example, when you need to:

- Programmatically choose one of several other components to delegate to
- Manipulate children, props, or data before passing them on to a child component

Here's an example of a `smart-list` component that delegates to more specific components, depending on the props passed to it:

```
var EmptyList = { /* ... */ }
var TableList = { /* ... */ }
var OrderedList = { /* ... */ }
var UnorderedList = { /* ... */ }

Vue.component('smart-list', {
  functional: true,
  props: {
    items: {
      type: Array,
      required: true
    },
    isOrdered: Boolean
  },
  render: function (createElement, context) {
    function appropriateListComponent () {
      var items = context.props.items

      if (items.length === 0)           return EmptyList
      if (typeof items[0] === 'object') return TableList
      if (context.props.isOrdered)      return OrderedList

      return UnorderedList
    }

    return createElement(
      appropriateListComponent(),
```

```
      context.data,
      context.children
    )
  }
})
```

Passing Attributes and Events to Child Elements/Components

On normal components, attributes not defined as props are automatically added to the root element of the component, replacing or intelligently merging with any existing attributes of the same name.

Functional components, however, require you to explicitly define this behavior:

```
Vue.component('my-functional-button', {
  functional: true,
  render: function (createElement, context) {
    // Transparently pass any attributes, event listeners, children, etc.
    return createElement('button', context.data, context.children)
  }
})
```

By passing `context.data` as the second argument to `createElement`, we are passing down any attributes or event listeners used on `my-functional-button`. It's so transparent, in fact, that events don't even require the `.native` modifier.

If you are using template-based functional components, you will also have to manually add attributes and listeners. Since we have access to the individual context contents, we can use `data.attrs` to pass along any HTML attributes and `listeners` *(the alias for* ***`data.on`****)* to pass along any event listeners.

```
<template functional>
  <button
    class="btn btn-primary"
    v-bind="data.attrs"
    v-on="listeners"
  >
    <slot/>
  </button>
</template>
```

`slots()` vs `children`

You may wonder why we need both `slots()` and `children`. Wouldn't `slots().default` be the same as `children`? In some cases, yes - but what if you have a functional component with the following children?

```
<my-functional-component>
  <p slot="foo">
    first
  </p>
  <p>second</p>
</my-functional-component>
```

For this component, `children` will give you both paragraphs, `slots().default` will give you only the second, and `slots().foo` will give you only the first. Having both `children` and `slots()` therefore allows you to choose whether this component knows about a slot system or perhaps delegates that responsibility to another component by passing along `children`.

Template Compilation

You may be interested to know that Vue's templates actually compile to render functions. This is an implementation detail you usually don't need to know about, but if you'd like to see how specific template features are compiled, you may find it interesting. Below is a little demo using `Vue.compile` to live-compile a template string:

{% raw %}

```
<label>render:</label>
<pre><code>{{ result.render }}</code></pre>
<label>staticRenderFns:</label>
<pre v-for="(fn, index) in result.staticRenderFns"><code>_m({{ index }}): {{ fn }}</code></p
<pre v-if="!result.staticRenderFns.length"><code>{{ result.staticRenderFns }}</code></pre>

<label>Compilation Error:</label>
<pre><code>{{ result }}</code></pre>
```

{% endraw %}

Writing a Plugin

Plugins usually add global-level functionality to Vue. There is no strictly defined scope for a plugin - there are typically several types of plugins you can write:

1. Add some global methods or properties. e.g. vue-custom-element
2. Add one or more global assets: directives/filters/transitions etc. e.g. vue-touch
3. Add some component options by global mixin. e.g. vue-router
4. Add some Vue instance methods by attaching them to Vue.prototype.

5. A library that provides an API of its own, while at the same time injecting some combination of the above. e.g. vue-router

A Vue.js plugin should expose an `install` method. The method will be called with the `Vue` constructor as the first argument, along with possible options:

```
MyPlugin.install = function (Vue, options) {
  // 1. add global method or property
  Vue.myGlobalMethod = function () {
    // something logic ...
  }

  // 2. add a global asset
  Vue.directive('my-directive', {
    bind (el, binding, vnode, oldVnode) {
      // something logic ...
    }
    ...
  })

  // 3. inject some component options
  Vue.mixin({
    created: function () {
      // something logic ...
    }
    ...
  })

  // 4. add an instance method
  Vue.prototype.$myMethod = function (methodOptions) {
    // something logic ...
  }
}
```

Using a Plugin

Use plugins by calling the `Vue.use()` global method:

```
// calls `MyPlugin.install(Vue)`
Vue.use(MyPlugin)
```

You can optionally pass in some options:

```
Vue.use(MyPlugin, { someOption: true })
```

`Vue.use` automatically prevents you from using the same plugin more than once, so calling it multiple times on the same plugin will install the plugin only once.

Some plugins provided by Vue.js official plugins such as `vue-router` automatically calls `Vue.use()` if `Vue` is available as a global variable. However in a module environment such as CommonJS, you always need to call `Vue.use()` explicitly:

```
// When using CommonJS via Browserify or Webpack
var Vue = require('vue')
var VueRouter = require('vue-router')

// Don't forget to call this
Vue.use(VueRouter)
```

Checkout awesome-vue for a huge collection of community-contributed plugins and libraries.

Vue.js allows you to define filters that can be used to apply common text formatting. Filters are usable in two places: **mustache interpolations and `v-bind` expressions** (the latter supported in 2.1.0+). Filters should be appended to the end of the JavaScript expression, denoted by the "pipe" symbol:

```
<!-- in mustaches -->
{{ message | capitalize }}

<!-- in v-bind -->
<div v-bind:id="rawId | formatId"></div>
```

You can define local filters in a component's options:

```
filters: {
  capitalize: function (value) {
    if (!value) return ''
    value = value.toString()
    return value.charAt(0).toUpperCase() + value.slice(1)
  }
}
```

or define a filter globally before creating the Vue instance:

```
Vue.filter('capitalize', function (value) {
  if (!value) return ''
  value = value.toString()
  return value.charAt(0).toUpperCase() + value.slice(1)
})

new Vue({
  // ...
})
```

Below is an example of our `capitalize` filter being used:

{% raw %}

{{ message | capitalize }}

{% endraw %}

The filter's function always receives the expression's value (the result of the former chain) as its first argument. In the above example, the `capitalize` filter function will receive the value of `message` as its argument.

Filters can be chained:

```
{{ message | filterA | filterB }}
```

In this case, `filterA`, defined with a single argument, will receive the value of `message`, and then the `filterB` function will be called with the result of `filterA` passed into `filterB`'s single argument.

Filters are JavaScript functions, therefore they can take arguments:

```
{{ message | filterA('arg1', arg2) }}
```

Here `filterA` is defined as a function taking three arguments. The value of `message` will be passed into the first argument. The plain string `'arg1'` will be passed into the `filterA` as its second argument, and the value of expression `arg2` will be evaluated and passed in as the third argument.

Turn on Production Mode

During development, Vue provides a lot of warnings to help you with common errors and pitfalls. However, these warning strings become useless in production and bloat your app's payload size. In addition, some of these warning checks have small runtime costs that can be avoided in production mode.

Without Build Tools

If you are using the full build, i.e. directly including Vue via a script tag without a build tool, make sure to use the minified version (`vue.min.js`) for production. Both versions can be found in the Installation guide.

With Build Tools

When using a build tool like Webpack or Browserify, the production mode will be determined by `process.env.NODE_ENV` inside Vue's source code, and it will be in development mode by default. Both build tools provide ways to overwrite this variable to enable Vue's production mode, and warnings will be stripped by minifiers during the build. All `vue-cli` templates have these pre-configured for you, but it would be beneficial to know how it is done:

Webpack

In Webpack 4+, you can use the `mode` option:

```
module.exports = {
  mode: 'production'
}
```

But in Webpack 3 and earlier, you'll need to use DefinePlugin:

```
var webpack = require('webpack')

module.exports = {
  // ...
  plugins: [
    // ...
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': JSON.stringify('production')
    })
  ]
}
```

Browserify

- Run your bundling command with the actual `NODE_ENV` environment variable set to `"production"`. This tells `vueify` to avoid including hot-reload and development related code.
- Apply a global envify transform to your bundle. This allows the minifier to strip out all the warnings in Vue's source code wrapped in env variable conditional blocks. For example:

  ```
  NODE_ENV=production browserify -g envify -e main.js | uglifyjs -c -m > build.js
  ```

- Or, using envify with Gulp:

  ```
  // Use the envify custom module to specify environment variables
  var envify = require('envify/custom')

  browserify(browserifyOptions)
    .transform(vueify)
    .transform(
      // Required in order to process node_modules files
      { global: true },
      envify({ NODE_ENV: 'production' })
    )
    .bundle()
  ```

- Or, using envify with Grunt and grunt-browserify:

```
// Use the envify custom module to specify environment variables
var envify = require('envify/custom')

browserify: {
  dist: {
    options: {
      // Function to deviate from grunt-browserify's default order
      configure: b => b
        .transform('vueify')
        .transform(
          // Required in order to process node_modules files
          { global: true },
          envify({ NODE_ENV: 'production' })
        )
        .bundle()
    }
  }
}
```

Rollup

Use rollup-plugin-replace:

```
const replace = require('rollup-plugin-replace')

rollup({
  // ...
  plugins: [
    replace({
      'process.env.NODE_ENV': JSON.stringify( 'production' )
    })
  ]
}).then(...)
```

Pre-Compiling Templates

When using in-DOM templates or in-JavaScript template strings, the template-to-render-function compilation is performed on the fly. This is usually fast enough in most cases, but is best avoided if your application is performance-sensitive.

The easiest way to pre-compile templates is using Single-File Components - the associated build setups automatically performs pre-compilation for you, so the built code contains the already compiled render functions instead of raw template strings.

If you are using Webpack, and prefer separating JavaScript and template files, you can use vue-template-loader, which also transforms the template files into JavaScript render functions during the build step.

Extracting Component CSS

When using Single-File Components, the CSS inside components are injected dynamically as `<style>` tags via JavaScript. This has a small runtime cost, and if you are using server-side rendering it will cause a "flash of unstyled content". Extracting the CSS across all components into the same file will avoid these issues, and also result in better CSS minification and caching.

Refer to the respective build tool documentations to see how it's done:

- Webpack + vue-loader (the `vue-cli` webpack template has this pre-configured)
- Browserify + vueify
- Rollup + rollup-plugin-vue

Tracking Runtime Errors

If a runtime error occurs during a component's render, it will be passed to the global `Vue.config.errorHandler` config function if it has been set. It might be a good idea to leverage this hook together with an error-tracking service like Sentry, which provides an official integration for Vue.

Introduction

In many Vue projects, global components will be defined using `Vue.component`, followed by `new Vue({ el: '#container' })` to target a container element in the body of every page.

This can work very well for small to medium-sized projects, where JavaScript is only used to enhance certain views. In more complex projects however, or when your frontend is entirely driven by JavaScript, these disadvantages become apparent:

- **Global definitions** force unique names for every component
- **String templates** lack syntax highlighting and require ugly slashes for multiline HTML
- **No CSS support** means that while HTML and JavaScript are modularized into components, CSS is conspicuously left out
- **No build step** restricts us to HTML and ES5 JavaScript, rather than preprocessors like Pug (formerly Jade) and Babel

All of these are solved by **single-file components** with a `.vue` extension, made possible with build tools such as Webpack or Browserify.

Here's an example of a file we'll call `Hello.vue`:

Now we get:

- Complete syntax highlighting
- CommonJS modules
- Component-scoped CSS

As promised, we can also use preprocessors such as Pug, Babel (with ES2015 modules), and Stylus for cleaner and more feature-rich components.

These specific languages are only examples. You could as easily use Bublé, TypeScript, SCSS, PostCSS - or whatever other preprocessors that help you be productive. If using Webpack with `vue-loader`, it also has first-class support for CSS Modules.

What About Separation of Concerns?

One important thing to note is that **separation of concerns is not equal to separation of file types.** In modern UI development, we have found that instead of dividing the codebase into three huge layers that interweave with one another, it makes much more sense to divide them into loosely-coupled components and compose them. Inside a component, its template, logic and styles are inherently coupled, and collocating them actually makes the component more cohesive and maintainable.

Even if you don't like the idea of Single-File Components, you can still leverage its hot-reloading and pre-compilation features by separating your JavaScript and CSS into separate files:

```
<!-- my-component.vue -->
<template>
  <div>This will be pre-compiled</div>
</template>
<script src="./my-component.js"></script>
<style src="./my-component.css"></style>
```

Getting Started

Example Sandbox

If you want to dive right in and start playing with single-file components, check out this simple todo app on CodeSandbox.

For Users New to Module Build Systems in JavaScript

With `.vue` components, we're entering the realm of advanced JavaScript applications. That means learning to use a few additional tools if you haven't already:

- **Node Package Manager (NPM)**: Read the Getting Started guide through section *10: Uninstalling global packages*.
- **Modern JavaScript with ES2015/16**: Read through Babel's Learn ES2015 guide. You don't have to memorize every feature right now, but keep this page as a reference you can come back to.

After you've taken a day to dive into these resources, we recommend checking out the webpack template. Follow the instructions and you should have a Vue project with `.vue` components, ES2015, and hot-reloading in no time!

To learn more about Webpack itself, check out their official docs and Webpack Academy. In Webpack, each file can be transformed by a "loader" before being included in the bundle, and Vue offers the vue-loader plugin to translate single-file (`.vue`) components.

For Advanced Users

Whether you prefer Webpack or Browserify, we have documented templates for both simple and more complex projects. We recommend browsing github.com/vuejs-templates, picking a template that's right for you, then following the instructions in the README to generate a new project with vue-cli.

Setup and Tooling

Anything compatible with a module-based build system will work, but if you're looking for a specific recommendation try the Karma test runner. It has a lot of community plugins, including support for Webpack and Browserify. For detailed setup please refer to each project's respective documentation. These example Karma configurations for Webpack and Browserify can help you get started.

Simple Assertions

You don't have to do anything special in your components to make them testable. Export the raw options:

```
<template>
  <span>{{ message }}</span>
</template>

<script>
  export default {
    data () {
      return {
        message: 'hello!'
      }
    },
    created () {
      this.message = 'bye!'
    }
  }
</script>
```

Then import the component options along with Vue, and you can make many common assertions:

```
// Import Vue and the component being tested
import Vue from 'vue'
import MyComponent from 'path/to/MyComponent.vue'

// Here are some Jasmine 2.0 tests, though you can
// use any test runner / assertion library combo you prefer
describe('MyComponent', () => {
  // Inspect the raw component options
  it('has a created hook', () => {
    expect(typeof MyComponent.created).toBe('function')
  })

  // Evaluate the results of functions in
  // the raw component options
  it('sets the correct default data', () => {
    expect(typeof MyComponent.data).toBe('function')
    const defaultData = MyComponent.data()
    expect(defaultData.message).toBe('hello!')
  })

  // Inspect the component instance on mount
  it('correctly sets the message when created', () => {
    const vm = new Vue(MyComponent).$mount()
    expect(vm.message).toBe('bye!')
  })

  // Mount an instance and inspect the render output
```

```
  it('renders the correct message', () => {
    const Constructor = Vue.extend(MyComponent)
    const vm = new Constructor().$mount()
    expect(vm.$el.textContent).toBe('bye!')
  })
})
```

Writing Testable Components

A component's render output is primarily determined by the props they receive. If a component's render output solely depends on its props it becomes straightforward to test, similar to asserting the return value of a pure function with different arguments. Take a simplified example:

```
<template>
  <p>{{ msg }}</p>
</template>

<script>
  export default {
    props: ['msg']
  }
</script>
```

You can assert its render output with different props using the `propsData` option:

```
import Vue from 'vue'
import MyComponent from './MyComponent.vue'

// helper function that mounts and returns the rendered text
function getRenderedText (Component, propsData) {
  const Constructor = Vue.extend(Component)
  const vm = new Constructor({ propsData: propsData }).$mount()
  return vm.$el.textContent
}

describe('MyComponent', () => {
  it('renders correctly with different props', () => {
    expect(getRenderedText(MyComponent, {
      msg: 'Hello'
    })).toBe('Hello')

    expect(getRenderedText(MyComponent, {
      msg: 'Bye'
    })).toBe('Bye')
```

```
  })
})
```

Asserting Asynchronous Updates

Since Vue performs DOM updates asynchronously, assertions on DOM updates resulting from state change will have to be made in a `Vue.nextTick` callback:

```
// Inspect the generated HTML after a state update
it('updates the rendered message when vm.message updates', done => {
  const vm = new Vue(MyComponent).$mount()
  vm.message = 'foo'

  // wait a "tick" after state change before asserting DOM updates
  Vue.nextTick(() => {
    expect(vm.$el.textContent).toBe('foo')
    done()
  })
})
```

We are planning to work on a collection of common test helpers to make it easier to render components with different constraints (e.g. shallow rendering that ignores child components) and assert their output.

For more in-depth information on unit testing in Vue, check out vue-test-utils and our cookbook entry about unit testing vue components.

> In Vue 2.5.0+ we have greatly improved our type declarations to work with the default object-based API. At the same time it introduces a few changes that require upgrade actions. Read this blog post for more details.

Official Declaration in NPM Packages

A static type system can help prevent many potential runtime errors, especially as applications grow. That's why Vue ships with official type declarations for TypeScript - not only in Vue core, but also for vue-router and vuex as well.

Since these are published on NPM, and the latest TypeScript knows how to resolve type declarations in NPM packages, this means when installed via NPM, you don't need any additional tooling to use TypeScript with Vue.

Recommended Configuration

```
// tsconfig.json
{
```

```
  "compilerOptions": {
    // this aligns with Vue's browser support
    "target": "es5",
    // this enables stricter inference for data properties on `this`
    "strict": true,
    // if using webpack 2+ or rollup, to leverage tree shaking:
    "module": "es2015",
    "moduleResolution": "node"
  }
}
```

Note that you have to include `strict: true` (or at least `noImplicitThis: true` which is a part of `strict` flag) to leverage type checking of `this` in component methods otherwise it is always treated as `any` type.

See TypeScript compiler options docs for more details.

Development Tooling

Project Creation

Vue CLI 3 can generate new projects that use TypeScript. To get started:

```
# 1. Install Vue CLI, if it's not already installed
npm install --global @vue/cli

# 2. Create a new project, then choose the "Manually select features" option
vue create my-project-name
```

Editor Support

For developing Vue applications with TypeScript, we strongly recommend using Visual Studio Code, which provides great out-of-the-box support for TypeScript. If you are using single-file components (SFCs), get the awesome Vetur extension, which provides TypeScript inference inside SFCs and many other great features.

WebStorm also provides out-of-the-box support for both TypeScript and Vue.

Basic Usage

To let TypeScript properly infer types inside Vue component options, you need to define components with `Vue.component` or `Vue.extend`:

```
import Vue from 'vue'

const Component = Vue.extend({
```

```
  // type inference enabled
})

const Component = {
  // this will NOT have type inference,
  // because TypeScript can't tell this is options for a Vue component.
}
```

Class-Style Vue Components

If you prefer a class-based API when declaring components, you can use the officially maintained vue-class-component decorator:

```
import Vue from 'vue'
import Component from 'vue-class-component'

// The @Component decorator indicates the class is a Vue component
@Component({
  // All component options are allowed in here
  template: '<button @click="onClick">Click!</button>'
})
export default class MyComponent extends Vue {
  // Initial data can be declared as instance properties
  message: string = 'Hello!'

  // Component methods can be declared as instance methods
  onClick (): void {
    window.alert(this.message)
  }
}
```

Augmenting Types for Use with Plugins

Plugins may add to Vue's global/instance properties and component options. In these cases, type declarations are needed to make plugins compile in TypeScript. Fortunately, there's a TypeScript feature to augment existing types called module augmentation.

For example, to declare an instance property `$myProperty` with type `string`:

```
// 1. Make sure to import 'vue' before declaring augmented types
import Vue from 'vue'

// 2. Specify a file with the types you want to augment
//    Vue has the constructor type in types/vue.d.ts
```

```
declare module 'vue/types/vue' {
  // 3. Declare augmentation for Vue
  interface Vue {
    $myProperty: string
  }
}
```

After including the above code as a declaration file (like `my-property.d.ts`) in your project, you can use `$myProperty` on a Vue instance.

```
var vm = new Vue()
console.log(vm.$myProperty) // This should compile successfully
```

You can also declare additional global properties and component options:

```
import Vue from 'vue'

declare module 'vue/types/vue' {
  // Global properties can be declared
  // on the `VueConstructor` interface
  interface VueConstructor {
    $myGlobal: string
  }
}

// ComponentOptions is declared in types/options.d.ts
declare module 'vue/types/options' {
  interface ComponentOptions<V extends Vue> {
    myOption?: string
  }
}
```

The above declarations allow the following code to be compiled:

```
// Global property
console.log(Vue.$myGlobal)

// Additional component option
var vm = new Vue({
  myOption: 'Hello'
})
```

Annotating Return Types

Because of the circular nature of Vue's declaration files, TypeScript may have difficulties inferring the types of certain methods. For this reason, you may need to annotate the return type on methods like `render` and those in `computed`.

```
import Vue, { VNode } from 'vue'

const Component = Vue.extend({
  data () {
    return {
      msg: 'Hello'
    }
  },
  methods: {
    // need annotation due to `this` in return type
    greet (): string {
      return this.msg + ' world'
    }
  },
  computed: {
    // need annotation
    greeting(): string {
      return this.greet() + '!'
    }
  },
  // `createElement` is inferred, but `render` needs return type
  render (createElement): VNode {
    return createElement('div', this.greeting)
  }
})
```

If you find type inference or member completion isn't working, annotating certain methods may help address these problems. Using the `--noImplicitAny` option will help find many of these unannotated methods.

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.

Integrating 3rd-Party Routers

If there's a 3rd-party router you prefer to use, such as Page.js or Director, integration is similarly easy. Here's a complete example using Page.js.

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
  }
})
```

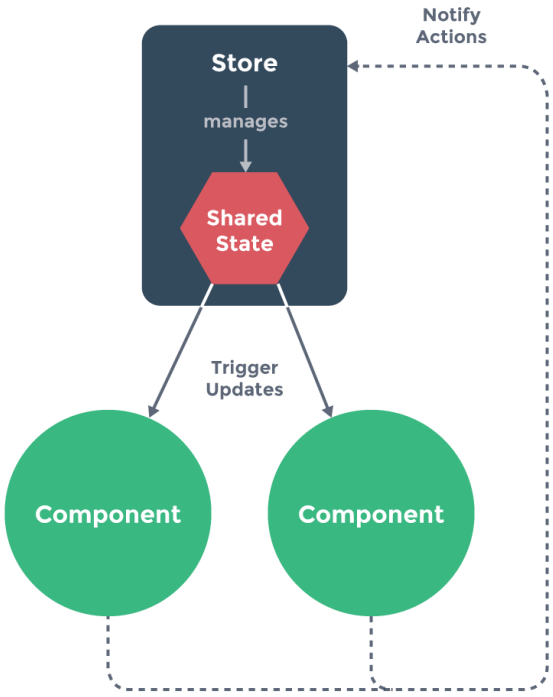

Figure 6: State Management

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.

This brings us full circle back to vuex, so if you've read this far it's probably time to try it out!

The Complete SSR Guide

We have created a standalone guide for creating server-rendered Vue applications. This is a very in-depth guide for those who are already familiar with client-side Vue development, server-side Node.js development and webpack. Check it out at ssr.vuejs.org.

Nuxt.js

Properly configuring all the discussed aspects of a production-ready server-rendered app can be a daunting task. Luckily, there is an excellent community project that aims to make all of this easier: Nuxt.js. Nuxt.js is a higher-level framework built on top of the Vue ecosystem which provides an extremely streamlined development experience for writing universal Vue applications. Better yet, you can even use it as a static site generator (with pages authored as single-file Vue components)! We highly recommend giving it a try.

Now it's time to take a deep dive! One of Vue's most distinct features is the unobtrusive reactivity system. Models are just plain JavaScript objects. When you modify them, the view updates. It makes state management simple and intuitive, but it's also important to understand how it works to avoid some common gotchas. In this section, we are going to dig into some of the lower-level details of Vue's reactivity system.

How Changes Are Tracked

When you pass a plain JavaScript object to a Vue instance as its `data` option, Vue will walk through all of its properties and convert them to getter/setters using Object.defineProperty. This is an ES5-only and un-shimmable feature, which is why Vue doesn't support IE8 and below.

The getter/setters are invisible to the user, but under the hood they enable Vue to perform dependency-tracking and change-notification when properties are accessed or modified. One caveat is that browser consoles format getter/setters differently when converted data objects are logged, so you may want to install vue-devtools for a more inspection-friendly interface.

Every component instance has a corresponding **watcher** instance, which records any properties "touched" during the component's render as dependencies. Later on when a dependency's setter is triggered, it notifies the watcher, which in turn causes the component to re-render.

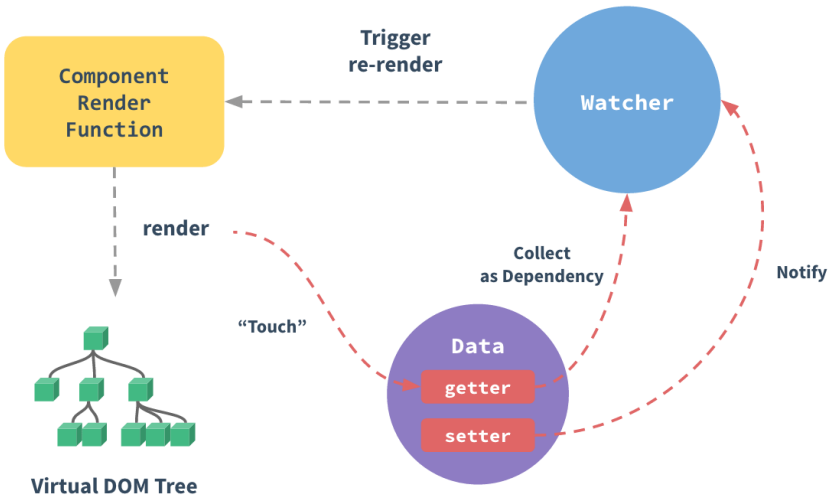

Figure 7: Reactivity Cycle

Change Detection Caveats

Due to the limitations of modern JavaScript (and the abandonment of `Object.observe`), Vue **cannot detect property addition or deletion**. Since Vue performs the getter/setter conversion process during instance initialization, a property must be present in the `data` object in order for Vue to convert it and make it reactive. 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:

```
Vue.set(vm.someObject, 'b', 2)
```

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

```
this.$set(this.someObject, 'b', 2)
```

Sometimes you may want to assign a number of properties to an existing object, for example using `Object.assign()` or `_.extend()`. However, new properties added to the object will not trigger changes. In such cases, create a fresh object with properties from both the original object and the mixin object:

```
// instead of `Object.assign(this.someObject, { a: 1, b: 2 })`
this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 })
```

There are also a few array-related caveats, which were discussed earlier in the list rendering section.

Declaring Reactive Properties

Since Vue doesn't allow dynamically adding root-level reactive properties, you have to initialize Vue instances by declaring all root-level reactive data properties upfront, even with an empty value:

```
var vm = new Vue({
  data: {
    // declare message with an empty value
    message: ''
  },
  template: '<div>{{ message }}</div>'
})
// set `message` later
vm.message = 'Hello!'
```

If you don't declare `message` in the data option, Vue will warn you that the render function is trying to access a property that doesn't exist.

There are technical reasons behind this restriction - it eliminates a class of edge cases in the dependency tracking system, and also makes Vue instances play nicer with type checking systems. But there is also an important consideration

in terms of code maintainability: the `data` object is like the schema for your component's state. Declaring all reactive properties upfront makes the component code easier to understand when revisited later or read by another developer.

Async Update Queue

In case you haven't noticed yet, Vue performs DOM updates **asynchronously**. Whenever a data change is observed, it will open a queue and buffer all the data changes that happen in the same event loop. If the same watcher is triggered multiple times, it will be pushed into the queue only once. This buffered de-duplication is important in avoiding unnecessary calculations and DOM manipulations. Then, in the next event loop "tick", Vue flushes the queue and performs the actual (already de-duped) work. Internally Vue tries native `Promise.then` and `MessageChannel` for the asynchronous queuing and falls back to `setTimeout(fn, 0)`.

For example, when you set `vm.someData = 'new value'`, the component will not re-render immediately. It will update in the next "tick", when the queue is flushed. Most of the time we don't need to care about this, but it can be tricky when you want to do something that depends on the post-update DOM state. Although Vue.js generally encourages developers to think in a "data-driven" fashion and avoid touching the DOM directly, sometimes it might be necessary to get your hands dirty. In order to wait until Vue.js has finished updating the DOM after a data change, you can use `Vue.nextTick(callback)` immediately after the data is changed. The callback will be called after the DOM has been updated. For example:

```
<div id="example">{{ message }}</div>
```

```
var vm = new Vue({
  el: '#example',
  data: {
    message: '123'
  }
})
vm.message = 'new message' // change data
vm.$el.textContent === 'new message' // false
Vue.nextTick(function () {
  vm.$el.textContent === 'new message' // true
})
```

There is also the `vm.$nextTick()` instance method, which is especially handy inside components, because it doesn't need global `Vue` and its callback's `this` context will be automatically bound to the current Vue instance:

```
Vue.component('example', {
  template: '<span>{{ message }}</span>',
```

```
  data: function () {
    return {
      message: 'not updated'
    }
  },
  methods: {
    updateMessage: function () {
      this.message = 'updated'
      console.log(this.$el.textContent) // => 'not updated'
      this.$nextTick(function () {
        console.log(this.$el.textContent) // => 'updated'
      })
    }
  }
})
```

This is definitely the most difficult page in the guide to write, but we do feel it's important. Odds are, you've had problems you tried to solve and you've used another library to solve them. You're here because you want to know if Vue can solve your specific problems better. That's what we hope to answer for you.

We also try very hard to avoid bias. As the core team, we obviously like Vue a lot. There are some problems we think it solves better than anything else out there. If we didn't believe that, we wouldn't be working on it. We do want to be fair and accurate though. Where other libraries offer significant advantages, such as React's vast ecosystem of alternative renderers or Knockout's browser support back to IE6, we try to list these as well.

We'd also like **your** help keeping this document up-to-date because the JavaScript world moves fast! If you notice an inaccuracy or something that doesn't seem quite right, please let us know by opening an issue.

React

React and Vue share many similarities. They both:

- utilize a virtual DOM
- provide reactive and composable view components
- maintain focus in the core library, with concerns such as routing and global state management handled by companion libraries

Being so similar in scope, we've put more time into fine-tuning this comparison than any other. We want to ensure not only technical accuracy, but also balance. We point out where React outshines Vue, for example in the richness of their ecosystem and abundance of their custom renderers.

With that said, it's inevitable that the comparison would appear biased towards Vue to some React users, as many of the subjects explored are to some extent

subjective. We acknowledge the existence of varying technical taste, and this comparison primarily aims to outline the reasons why Vue could potentially be a better fit if your preferences happen to coincide with ours.

Some of the sections below may also be slightly outdated due to recent updates in React 16+, and we are planning to work with the React community to revamp this section in the near future.

Runtime Performance

Both React and Vue are exceptionally and similarly fast, so speed is unlikely to be a deciding factor in choosing between them. For specific metrics though, check out this 3rd party benchmark, which focuses on raw render/update performance with very simple component trees.

Optimization Efforts

In React, when a component's state changes, it triggers the re-render of the entire component sub-tree, starting at that component as root. To avoid unnecessary re-renders of child components, you need to either use `PureComponent` or implement `shouldComponentUpdate` whenever you can. You may also need to use immutable data structures to make your state changes more optimization-friendly. However, in certain cases you may not be able to rely on such optimizations because `PureComponent/shouldComponentUpdate` assumes the entire sub tree's render output is determined by the props of the current component. If that is not the case, then such optimizations may lead to inconsistent DOM state.

In Vue, a component's dependencies are automatically tracked during its render, so the system knows precisely which components actually need to re-render when state changes. Each component can be considered to have `shouldComponentUpdate` automatically implemented for you, without the nested component caveats.

Overall this removes the need for a whole class of performance optimizations from the developer's plate, and allows them to focus more on building the app itself as it scales.

HTML & CSS

In React, everything is just JavaScript. Not only are HTML structures expressed via JSX, the recent trends also tend to put CSS management inside JavaScript as well. This approach has its own benefits, but also comes with various trade-offs that may not seem worthwhile for every developer.

Vue embraces classic web technologies and builds on top of them. To show you what that means, we'll dive into some examples.

JSX vs Templates

In React, all components express their UI within render functions using JSX, a declarative XML-like syntax that works within JavaScript.

Render functions with JSX have a few advantages:

- You can leverage the power of a full programming language (JavaScript) to build your view. This includes temporary variables, flow controls, and directly referencing JavaScript values in scope.
- The tooling support (e.g. linting, type checking, editor autocompletion) for JSX is in some ways more advanced than what's currently available for Vue templates.

In Vue, we also have render functions and even support JSX, because sometimes you do need that power. However, as the default experience we offer templates as a simpler alternative. Any valid HTML is also a valid Vue template, and this leads to a few advantages of its own:

- For many developers who have been working with HTML, templates feel more natural to read and write. The preference itself can be somewhat subjective, but if it makes the developer more productive then the benefit is objective.
- HTML-based templates make it much easier to progressively migrate existing applications to take advantage of Vue's reactivity features.
- It also makes it much easier for designers and less experienced developers to parse and contribute to the codebase.
- You can even use pre-processors such as Pug (formerly known as Jade) to author your Vue templates.

Some argue that you'd need to learn an extra DSL (Domain-Specific Language) to be able to write templates - we believe this difference is superficial at best. First, JSX doesn't mean the user doesn't need to learn anything - it's additional syntax on top of plain JavaScript, so it can be easy for someone familiar with JavaScript to learn, but saying it's essentially free is misleading. Similarly, a template is just additional syntax on top of plain HTML and thus has very low learning cost for those who are already familiar with HTML. With the DSL we are also able to help the user get more done with less code (e.g. `v-on` modifiers). The same task can involve a lot more code when using plain JSX or render functions.

On a higher level, we can divide components into two categories: presentational ones and logical ones. We recommend using templates for presentational com-

ponents and render function / JSX for logical ones. The percentage of these components depends on the type of app you are building, but in general we find presentational ones to be much more common.

Component-Scoped CSS

Unless you spread components out over multiple files (for example with CSS Modules), scoping CSS in React is often done via CSS-in-JS solutions (e.g. styled-components, glamorous, and emotion). This introduces a new component-oriented styling paradigm that is different from the normal CSS authoring process. Additionally, although there is support for extracting CSS into a single stylesheet at build time, it is still common that a runtime will need to be included in the bundle for styling to work properly. While you gain access to the dynamism of JavaScript while constructing your styles, the tradeoff is often increased bundle size and runtime cost.

If you are a fan of CSS-in-JS, many of the popular CSS-in-JS libraries support Vue (e.g. styled-components-vue and vue-emotion). The main difference between React and Vue here is that the default method of styling in Vue is through more familiar `style` tags in single-file components.

Single-file components give you full access to CSS in the same file as the rest of your component code.

```
<style scoped>
  @media (min-width: 250px) {
    .list-container:hover {
      background: orange;
    }
  }
</style>
```

The optional `scoped` attribute automatically scopes this CSS to your component by adding a unique attribute (such as `data-v-21e5b78`) to elements and compiling `.list-container:hover` to something like `.list-container[data-v-21e5b78]:hover`.

Lastly, the styling in Vue's single-file component's is very flexible. Through vue-loader, you can use any preprocessor, post-processor, and even deep integration with CSS Modules – all within the `<style>` element.

Scale

Scaling Up

For large applications, both Vue and React offer robust routing solutions. The React community has also been very innovative in terms of state management solutions (e.g. Flux/Redux). These state management patterns and even Redux

itself can be easily integrated into Vue applications. In fact, Vue has even taken this model a step further with Vuex, an Elm-inspired state management solution that integrates deeply into Vue that we think offers a superior development experience.

Another important difference between these offerings is that Vue's companion libraries for state management and routing (among other concerns) are all officially supported and kept up-to-date with the core library. React instead chooses to leave these concerns to the community, creating a more fragmented ecosystem. Being more popular though, React's ecosystem is considerably richer than Vue's.

Finally, Vue offers a CLI project generator that makes it trivially easy to start a new project using your choice of build system, including webpack, Browserify, or even no build system. React is also making strides in this area with create-react-app, but it currently has a few limitations:

- It does not allow any configuration during project generation, while Vue's project templates allow Yeoman-like customization.
- It only offers a single template that assumes you're building a single-page application, while Vue offers a wide variety of templates for various purposes and build systems.
- It cannot generate projects from user-built templates, which can be especially useful for enterprise environments with pre-established conventions.

It's important to note that many of these limitations are intentional design decisions made by the create-react-app team and they do have their advantages. For example, as long as your project's needs are very simple and you never need to "eject" to customize your build process, you'll be able to update it as a dependency. You can read more about the differing philosophy here.

Scaling Down

React is renowned for its steep learning curve. Before you can really get started, you need to know about JSX and probably ES2015+, since many examples use React's class syntax. You also have to learn about build systems, because although you could technically use Babel Standalone to live-compile your code in the browser, it's absolutely not suitable for production.

While Vue scales up just as well as React, it also scales down just as well as jQuery. That's right - to get started, all you have to do is drop a single script tag into the page:

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

Then you can start writing Vue code and even ship the minified version to production without feeling guilty or having to worry about performance problems.

Since you don't need to know about JSX, ES2015, or build systems to get started

with Vue, it also typically takes developers less than a day reading the guide to learn enough to build non-trivial applications.

Native Rendering

React Native enables you to write native-rendered apps for iOS and Android using the same React component model. This is great in that as a developer, you can apply your knowledge of a framework across multiple platforms. On this front, Vue has an official collaboration with Weex, a cross-platform UI framework created by Alibaba Group and being incubated by the Apache Software Foundation (ASF). Weex allows you to use the same Vue component syntax to author components that can not only be rendered in the browser, but also natively on iOS and Android!

At this moment, Weex is still in active development and is not as mature and battle-tested as React Native, but its development is driven by the production needs of the largest e-commerce business in the world, and the Vue team will also actively collaborate with the Weex team to ensure a smooth experience for Vue developers.

Another option Vue developers will soon have is NativeScript, via a community-driven plugin.

With MobX

MobX has become quite popular in the React community and it actually uses a nearly identical reactivity system to Vue. To a limited extent, the React + MobX workflow can be thought of as a more verbose Vue, so if you're using that combination and are enjoying it, jumping into Vue is probably the next logical step.

Preact and Other React-Like Libraries

React-like libraries usually try to share as much of their API and ecosystem with React as is feasible. For that reason, the vast majority of comparisons above will also apply to them. The main difference will typically be a reduced ecosystem, often significantly, compared to React. Since these libraries cannot be 100% compatible with everything in the React ecosystem, some tooling and companion libraries may not be usable. Or, even if they appear to work, they could break at any time unless your specific React-like library is officially supported on par with React.

AngularJS (Angular 1)

Some of Vue's syntax will look very similar to AngularJS (e.g. `v-if` vs `ng-if`). This is because there were a lot of things that AngularJS got right and these were an inspiration for Vue very early in its development. There are also many pains that come with AngularJS however, where Vue has attempted to offer a significant improvement.

Complexity

Vue is much simpler than AngularJS, both in terms of API and design. Learning enough to build non-trivial applications typically takes less than a day, which is not true for AngularJS.

Flexibility and Modularity

AngularJS has strong opinions about how your applications should be structured, while Vue is a more flexible, modular solution. While this makes Vue more adaptable to a wide variety of projects, we also recognize that sometimes it's useful to have some decisions made for you, so that you can just start coding.

That's why we offer a webpack template that can set you up within minutes, while also granting you access to advanced features such as hot module reloading, linting, CSS extraction, and much more.

Data binding

AngularJS uses two-way binding between scopes, while Vue enforces a one-way data flow between components. This makes the flow of data easier to reason about in non-trivial applications.

Directives vs Components

Vue has a clearer separation between directives and components. Directives are meant to encapsulate DOM manipulations only, while components are self-contained units that have their own view and data logic. In AngularJS, directives do everything and components are just a specific kind of directive.

Runtime Performance

Vue has better performance and is much, much easier to optimize because it doesn't use dirty checking. AngularJS becomes slow when there are a lot of watchers, because every time anything in the scope changes, all these watchers

need to be re-evaluated again. Also, the digest cycle may have to run multiple times to "stabilize" if some watcher triggers another update. AngularJS users often have to resort to esoteric techniques to get around the digest cycle, and in some situations, there's no way to optimize a scope with many watchers.

Vue doesn't suffer from this at all because it uses a transparent dependency-tracking observation system with async queueing - all changes trigger independently unless they have explicit dependency relationships.

Interestingly, there are quite a few similarities in how Angular and Vue are addressing these AngularJS issues.

Angular (Formerly known as Angular 2)

We have a separate section for the new Angular because it really is a completely different framework from AngularJS. For example, it features a first-class component system, many implementation details have been completely rewritten, and the API has also changed quite drastically.

TypeScript

Angular essentially requires using TypeScript, given that almost all its documentation and learning resources are TypeScript-based. TypeScript has its benefits - static type checking can be very useful for large-scale applications, and can be a big productivity boost for developers with backgrounds in Java and C#.

However, not everyone wants to use TypeScript. In many smaller-scale use cases, introducing a type system may result in more overhead than productivity gain. In those cases you'd be better off going with Vue instead, since using Angular without TypeScript can be challenging.

Finally, although not as deeply integrated with TypeScript as Angular is, Vue also offers official typings and official decorator for those who wish to use TypeScript with Vue. We are also actively collaborating with the TypeScript and VSCode teams at Microsoft to improve the TS/IDE experience for Vue + TS users.

Runtime Performance

Both frameworks are exceptionally fast, with very similar metrics on benchmarks. You can browse specific metrics for a more granular comparison, but speed is unlikely to be a deciding factor.

Size

Recent versions of Angular, with AOT compilation and tree-shaking, have been able to get its size down considerably. However, a full-featured Vue 2 project with Vuex + Vue Router included (~30KB gzipped) is still significantly lighter than an out-of-the-box, AOT-compiled application generated by `angular-cli` (~65KB gzipped).

Flexibility

Vue is much less opinionated than Angular, offering official support for a variety of build systems, with no restrictions on how you structure your application. Many developers enjoy this freedom, while some prefer having only one Right Way to build any application.

Learning Curve

To get started with Vue, all you need is familiarity with HTML and ES5 JavaScript (i.e. plain JavaScript). With these basic skills, you can start building non-trivial applications within less than a day of reading the guide.

Angular's learning curve is much steeper. The API surface of the framework is huge and as a user you will need to familiarize yourself with a lot more concepts before getting productive. The complexity of Angular is largely due to its design goal of targeting only large, complex applications - but that does make the framework a lot more difficult for less-experienced developers to pick up.

Ember

Ember is a full-featured framework that is designed to be highly opinionated. It provides a lot of established conventions and once you are familiar enough with them, it can make you very productive. However, it also means the learning curve is high and flexibility suffers. It's a trade-off when you try to pick between an opinionated framework and a library with a loosely coupled set of tools that work together. The latter gives you more freedom but also requires you to make more architectural decisions.

That said, it would probably make a better comparison between Vue core and Ember's templating and object model layers:

- Vue provides unobtrusive reactivity on plain JavaScript objects and fully automatic computed properties. In Ember, you need to wrap everything in Ember Objects and manually declare dependencies for computed properties.

- Vue's template syntax harnesses the full power of JavaScript expressions, while Handlebars' expression and helper syntax is intentionally quite limited in comparison.
- Performance-wise, Vue outperforms Ember by a fair margin, even after the latest Glimmer engine update in Ember 2.x. Vue automatically batches updates, while in Ember you need to manually manage run loops in performance-critical situations.

Knockout

Knockout was a pioneer in the MVVM and dependency tracking spaces and its reactivity system is very similar to Vue's. Its browser support is also very impressive considering everything it does, with support back to IE6! Vue on the other hand only supports IE9+.

Over time though, Knockout development has slowed and it's begun to show its age a little. For example, its component system lacks a full set of lifecycle hooks and although it's a very common use case, the interface for passing children to a component feels a little clunky compared to Vue's.

There also seem to be philosophical differences in the API design which if you're curious, can be demonstrated by how each handles the creation of a simple todo list. It's definitely somewhat subjective, but many consider Vue's API to be less complex and better structured.

Polymer

Polymer is another Google-sponsored project and in fact was a source of inspiration for Vue as well. Vue's components can be loosely compared to Polymer's custom elements and both provide a very similar development style. The biggest difference is that Polymer is built upon the latest Web Components features and requires non-trivial polyfills to work (with degraded performance) in browsers that don't support those features natively. In contrast, Vue works without any dependencies or polyfills down to IE9.

In Polymer, the team has also made its data-binding system very limited in order to compensate for the performance. For example, the only expressions supported in Polymer templates are boolean negation and single method calls. Its computed property implementation is also not very flexible.

Riot

Riot 3.0 provides a similar component-based development model (which is called a "tag" in Riot), with a minimal and beautifully designed API. Riot and Vue

probably share a lot in design philosophies. However, despite being a bit heavier than Riot, Vue does offer some significant advantages:

- Better performance. Riot traverses a DOM tree rather than using a virtual DOM, so suffers from the same performance issues as AngularJS.
- More mature tooling support. Vue provides official support for webpack and Browserify, while Riot relies on community support for build system integration.