### Essentials 基础

# Installation 安装

### Compatibility Note 兼容性

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](https://caniuse.com/#feat=es5).

Vue **不支持** IE8 及以下版本，因为 Vue 使用了 IE8 无法模拟的 ECMAScript 5 特性。但它支持所有[**兼容 ECMAScript 5 的浏览器**](https://caniuse.com/#feat=es5)。

### Release Notes 更新日志

Latest stable version: 2.6.10

最新稳定版本：2.6.10

Detailed release notes for each version are available on [GitHub](https://github.com/vuejs/vue/releases).

每个版本的更新日志见 [**GitHub**](https://github.com/vuejs/vue/releases)。

## [Vue Devtools](https://vuejs.org/v2/guide/installation.html#Vue-Devtools)

When using Vue, we recommend also installing the[***Vue Devtools***](https://github.com/vuejs/vue-devtools#vue-devtools) in your browser, allowing you to inspect and debug your Vue applications in a more user-friendly interface.

在使用 Vue 时，我们推荐在你的浏览器上安装 [**Vue Devtools**](https://github.com/vuejs/vue-devtools#vue-devtools)。它允许你在一个更友好的界面中审查和调试 Vue 应用。

## [Direct <script> Include](https://vuejs.org/v2/guide/installation.html#Direct-lt-script-gt-Include)

## [直接用 <script> 引入](https://cn.vuejs.org/v2/guide/installation.html#%E7%9B%B4%E6%8E%A5%E7%94%A8-lt-script-gt-%E5%BC%95%E5%85%A5)

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

直接下载并用 <script> 标签引入，Vue 会被注册为一个全局变量。

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

在开发环境下不要使用压缩版本，不然你就失去了所有常见错误相关的警告!

[Development Version](https://vuejs.org/js/vue.js)：With full warnings and debug mode

开发版本：包含完整的警告和调试模式  
  
[Production Version](https://vuejs.org/js/vue.min.js)：Warnings stripped, 33.30KB min+gzip

生产版本：删除了警告，33.30KB min+gzip

For prototyping or learning purposes, you can use the latest version with:

对于制作原型或学习，你可以这样使用最新版本：

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

For production, we recommend linking to a specific version number and build to avoid unexpected breakage from newer versions:

对于生产环境，我们推荐链接到一个明确的版本号和构建文件，以避免新版本造成的不可预期的破坏：

<script src="https://cdn.jsdelivr.net/npm/vue@2.6.10/dist/vue.js">

</script>

If you are using native ES Modules, there is also an ES Modules compatible build:

如果你使用原生 ES Modules，这里也有一个兼容 ES Module 的构建文件：

<script type="module">

import Vue from 'https://cdn.jsdelivr.net/npm/vue@2.6.10/dist/vue.esm.browser.js'

</script>

You can browse the source of the NPM package at [**cdn.jsdelivr.net/npm/vue**](https://cdn.jsdelivr.net/npm/vue/).

你可以在 [**cdn.jsdelivr.net/npm/vue**](https://cdn.jsdelivr.net/npm/vue/) 浏览 NPM 包的源代码。

Vue is also available on [**unpkg**](https://unpkg.com/vue@2.6.10/dist/vue.js) and [**cdnjs**](https://cdnjs.cloudflare.com/ajax/libs/vue/2.6.10/vue.js) (cdnjs takes some time to sync so the latest release may not be available yet).

Vue 也可以在 [**unpkg**](https://unpkg.com/vue@2.6.10/dist/vue.js) 和 [**cdnjs**](https://cdnjs.cloudflare.com/ajax/libs/vue/2.6.10/vue.js) 上获取 (cdnjs 的版本更新可能略滞后)。

Make sure to read about [**the different builds of Vue**](https://vuejs.org/v2/guide/installation.html#Explanation-of-Different-Builds) 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.

请确认了解[**不同构建版本**](https://cn.vuejs.org/v2/guide/installation.html#%E5%AF%B9%E4%B8%8D%E5%90%8C%E6%9E%84%E5%BB%BA%E7%89%88%E6%9C%AC%E7%9A%84%E8%A7%A3%E9%87%8A)并在你发布的站点中使用**生产环境版本**，把 vue.js 换成 vue.min.js。这是一个更小的构建，可以带来比开发环境下更快的速度体验。

## [NPM](https://vuejs.org/v2/guide/installation.html#NPM)

NPM is the recommended installation method when building large scale applications with Vue. It pairs nicely with module bundlers such as [**Webpack**](https://webpack.js.org/) or [**Browserify**](http://browserify.org/). Vue also provides accompanying tools for authoring [**Single File Components**](https://vuejs.org/v2/guide/single-file-components.html).

在用 Vue 构建大型应用时推荐使用 NPM 安装[**[1]**](https://cn.vuejs.org/v2/guide/installation.html#footnote-1)。NPM 能很好地和诸如 [**webpack**](https://webpack.js.org/) 或 [**Browserify**](http://browserify.org/) 模块打包器配合使用。同时 Vue 也提供配套工具来开发[**单文件组件**](https://cn.vuejs.org/v2/guide/single-file-components.html)。

## [CLI](https://vuejs.org/v2/guide/installation.html#CLI) [命令行工具 (CLI)](https://cn.vuejs.org/v2/guide/installation.html#%E5%91%BD%E4%BB%A4%E8%A1%8C%E5%B7%A5%E5%85%B7-CLI)

Vue provides an [**official CLI**](https://github.com/vuejs/vue-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**](https://cli.vuejs.org/) for more details.

Vue 提供了一个[**官方的 CLI**](https://github.com/vuejs/vue-cli)，为单页面应用 (SPA) 快速搭建繁杂的脚手架。它为现代前端工作流提供了 batteries-included 的构建设置。只需要几分钟的时间就可以运行起来并带有热重载、保存时 lint 校验，以及生产环境可用的构建版本。更多详情可查阅 [**Vue CLI 的文档**](https://cli.vuejs.org/)。

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**](https://vuejs.org/v2/guide/) without any build tools before using the CLI.

CLI 工具假定用户对 Node.js 和相关构建工具有一定程度的了解。如果你是新手，我们强烈建议先在不用构建工具的情况下通读[**指南**](https://cn.vuejs.org/v2/guide/)，在熟悉 Vue 本身之后再使用 CLI。

## [Explanation of Different Builds](https://vuejs.org/v2/guide/installation.html#Explanation-of-Different-Builds) [对不同构建版本的解释](https://cn.vuejs.org/v2/guide/installation.html#%E5%AF%B9%E4%B8%8D%E5%90%8C%E6%9E%84%E5%BB%BA%E7%89%88%E6%9C%AC%E7%9A%84%E8%A7%A3%E9%87%8A)

In the [**dist/ directory of the NPM package**](https://cdn.jsdelivr.net/npm/vue/dist/) you will find many different builds of Vue.js. Here’s an overview of the difference between them:

在 [**NPM 包的 dist/ 目录**](https://cdn.jsdelivr.net/npm/vue/dist/)你将会找到很多不同的 Vue.js 构建版本。这里列出了它们之间的差别：

### [Terms](https://vuejs.org/v2/guide/installation.html#Terms) 术[语](https://cn.vuejs.org/v2/guide/installation.html#%E6%9C%AF%E8%AF%AD)

* **Full**: builds that contain both the compiler and the runtime.
* **完整版**：同时包含编译器和运行时的版本。
* **Compiler**: code that is responsible for compiling template strings into JavaScript render functions.
* **编译器**：用来将模板字符串编译成为 JavaScript 渲染函数的代码。
* **Runtime**: code that is responsible for creating Vue instances, rendering and patching virtual DOM, etc. Basically everything minus the compiler.
* **运行时**：用来创建 Vue 实例、渲染并处理虚拟 DOM 等的代码。基本上就是除去编译器的其它一切。
* [**UMD**](https://github.com/umdjs/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**](https://cdn.jsdelivr.net/npm/vue) is the Runtime + Compiler UMD build (vue.js).
* [**UMD**](https://github.com/umdjs/umd)：UMD 版本可以通过 <script> 标签直接用在浏览器中。jsDelivr CDN 的 [**https://cdn.jsdelivr.net/npm/vue**](https://cdn.jsdelivr.net/npm/vue) 默认文件就是运行时 + 编译器的 UMD 版本 (vue.js)。
* [**CommonJS**](http://wiki.commonjs.org/wiki/Modules/1.1): CommonJS builds are intended for use with older bundlers like [**browserify**](http://browserify.org/) or [**webpack 1**](https://webpack.github.io/). The default file for these bundlers (pkg.main) is the Runtime only CommonJS build (vue.runtime.common.js).
* [**CommonJS**](http://wiki.commonjs.org/wiki/Modules/1.1)：CommonJS 版本用来配合老的打包工具比如 [**Browserify**](http://browserify.org/) 或 [**webpack 1**](https://webpack.github.io/)。这些打包工具的默认文件 (pkg.main) 是只包含运行时的 CommonJS 版本 (vue.runtime.common.js)。
* [**ES Module**](http://exploringjs.com/es6/ch_modules.html): starting in 2.6 Vue provides two ES Modules (ESM) builds:
* [**ES Module**](http://exploringjs.com/es6/ch_modules.html)：从 2.6 开始 Vue 会提供两个 ES Modules (ESM) 构建文件：
  + ESM for bundlers: intended for use with modern bundlers like [**webpack 2**](https://webpack.js.org/) or [**Rollup**](https://rollupjs.org/). ESM format is designed to be statically analyzable so the bundlers can take advantage of that to perform “tree-shaking” and eliminate unused code from your final bundle. The default file for these bundlers (pkg.module) is the Runtime only ES Module build (vue.runtime.esm.js).
  + 为打包工具提供的 ESM：为诸如 [**webpack 2**](https://webpack.js.org/) 或 [**Rollup**](https://rollupjs.org/) 提供的现代打包工具。ESM 格式被设计为可以被静态分析，所以打包工具可以利用这一点来进行“tree-shaking”并将用不到的代码排除出最终的包。为这些打包工具提供的默认文件 (pkg.module) 是只有运行时的 ES Module 构建 (vue.runtime.esm.js)。
  + ESM for browsers (2.6+ only): intended for direct imports in modern browsers via <script type="module">.
  + 为浏览器提供的 ESM (2.6+)：用于在现代浏览器中通过 <script type="module"> 直接导入。

### [Runtime + Compiler vs. Runtime-only](https://vuejs.org/v2/guide/installation.html#Runtime-Compiler-vs-Runtime-only) [运行时 + 编译器 vs. 只包含运行时](https://cn.vuejs.org/v2/guide/installation.html#%E8%BF%90%E8%A1%8C%E6%97%B6-%E7%BC%96%E8%AF%91%E5%99%A8-vs-%E5%8F%AA%E5%8C%85%E5%90%AB%E8%BF%90%E8%A1%8C%E6%97%B6)

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:

如果你需要在客户端编译模板 (比如传入一个字符串给 template 选项，或挂载到一个元素上并以其 DOM 内部的 HTML 作为模板)，就将需要加上编译器，即完整版：

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.

当使用 vue-loader 或 vueify 的时候，\*.vue 文件内部的模板会在构建时预编译成 JavaScript。你在最终打好的包里实际上是不需要编译器的，所以只用运行时版本即可。

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:

因为运行时版本相比完整版体积要小大约 30%，所以应该尽可能使用这个版本。如果你仍然希望使用完整版，则需要在打包工具里配置一个别名：

### [Development vs. Production Mode](https://vuejs.org/v2/guide/installation.html#Development-vs-Production-Mode) [开发环境 vs. 生产环境模式](https://cn.vuejs.org/v2/guide/installation.html#%E5%BC%80%E5%8F%91%E7%8E%AF%E5%A2%83-vs-%E7%94%9F%E4%BA%A7%E7%8E%AF%E5%A2%83%E6%A8%A1%E5%BC%8F)

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

对于 UMD 版本来说，开发环境/生产环境模式是硬编码好的：开发环境下用未压缩的代码，生产环境下使用压缩后的代码。

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 和 ES Module 版本是用于打包工具的，因此我们不提供压缩后的版本。你需要自行将最终的包进行压缩。

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.

CommonJS 和 ES Module 版本同时保留原始的 process.env.NODE\_ENV 检测，以决定它们应该运行在什么模式下。你应该使用适当的打包工具配置来替换这些环境变量以便控制 Vue 所运行的模式。把 process.env.NODE\_ENV 替换为字符串字面量同时可以让 UglifyJS 之类的压缩工具完全丢掉仅供开发环境的代码块，以减少最终的文件尺寸。

### [CSP environments](https://vuejs.org/v2/guide/installation.html#CSP-environments) [CSP 环境](https://cn.vuejs.org/v2/guide/installation.html#CSP-%E7%8E%AF%E5%A2%83)

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.

有些环境，如 Google Chrome Apps，会强制应用内容安全策略 (CSP)，不能使用 new Function() 对表达式求值。这时可以用 CSP 兼容版本。完整版本依赖于该功能来编译模板，所以无法在这些环境下使用。

On the other hand, the runtime-only build is fully CSP-compliant. When using the runtime-only build with [**Webpack + vue-loader**](https://github.com/vuejs-templates/webpack-simple) or [**Browserify + vueify**](https://github.com/vuejs-templates/browserify-simple), your templates will be precompiled into renderfunctions which work perfectly in CSP environments.

另一方面，运行时版本则是完全兼容 CSP 的。当通过 [**webpack + vue-loader**](https://github.com/vuejs-templates/webpack-simple) 或者 [**Browserify + vueify**](https://github.com/vuejs-templates/browserify-simple) 构建时，模板将被预编译为 render 函数，可以在 CSP 环境中完美运行。

## [Dev Build](https://vuejs.org/v2/guide/installation.html#Dev-Build) [开发版本](https://cn.vuejs.org/v2/guide/installation.html#%E5%BC%80%E5%8F%91%E7%89%88%E6%9C%AC)

**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!

**重要**: GitHub 仓库的 /dist 文件夹只有在新版本发布时才会提交。如果想要使用 GitHub 上 Vue 最新的源码，你需要自己构建！

## [AMD Module Loaders](https://vuejs.org/v2/guide/installation.html#AMD-Module-Loaders) [AMD 模块加载器](https://cn.vuejs.org/v2/guide/installation.html#AMD-%E6%A8%A1%E5%9D%97%E5%8A%A0%E8%BD%BD%E5%99%A8)

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

所有 UMD 版本都可以直接用作 AMD 模块。

# Introduction介绍

## What is Vue.js? [Vue.js 是什么](https://cn.vuejs.org/v2/guide/index.html#Vue-js-%E6%98%AF%E4%BB%80%E4%B9%88)

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

Vue (读音 /vjuː/，类似于 **view**) 是一套用于构建用户界面的**渐进式框架**。与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与[**现代化的工具链**](https://cn.vuejs.org/v2/guide/single-file-components.html)以及各种[**支持类库**](https://github.com/vuejs/awesome-vue#libraries--plugins)结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。

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

如果你想在深入学习 Vue 之前对它有更多了解，我们[**制作了一个视频**](https://cn.vuejs.org/v2/guide/index.html)，带您了解其核心概念和一个示例工程。

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

如果你已经是有经验的前端开发者，想知道 Vue 与其它库/框架有哪些区别，请查看[**对比其它框架**](https://cn.vuejs.org/v2/guide/comparison.html)。

## Getting Started [起步](https://cn.vuejs.org/v2/guide/index.html#%E8%B5%B7%E6%AD%A5)

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.

官方指南假设你已了解关于 HTML、CSS 和 JavaScript 的中级知识。如果你刚开始学习前端开发，将框架作为你的第一步可能不是最好的主意——掌握好基础知识再来吧！之前有其它框架的使用经验会有帮助，但这不是必需的。

The easiest way to try out Vue.js is using the [**JSFiddle Hello World example**](https://jsfiddle.net/chrisvfritz/50wL7mdz/). Feel free to open it in another tab and follow along as we go through some basic examples. Or, you can [**create an index.html file**](https://gist.githubusercontent.com/chrisvfritz/7f8d7d63000b48493c336e48b3db3e52/raw/ed60c4e5d5c6fec48b0921edaed0cb60be30e87c/index.html) and include Vue with:

尝试 Vue.js 最简单的方法是使用 [**JSFiddle 上的 Hello World 例子**](https://jsfiddle.net/chrisvfritz/50wL7mdz/)。你可以在浏览器新标签页中打开它，跟着例子学习一些基础用法。或者你也可以[**创建一个 .html 文件**](https://gist.githubusercontent.com/chrisvfritz/7f8d7d63000b48493c336e48b3db3e52/raw/ed60c4e5d5c6fec48b0921edaed0cb60be30e87c/index.html)，然后通过如下方式引入 Vue：

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

[**安装教程**](https://cn.vuejs.org/guide/installation.html)给出了更多安装 Vue 的方式。请注意我们**不推荐**新手直接使用 vue-cli，尤其是在你还不熟悉基于 Node.js 的构建工具时。

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

如果你喜欢交互式的东西，你也可以查阅[**这个 Scrimba 上的系列教程**](https://scrimba.com/playlist/pXKqta)，它揉合了录屏和代码试验田，并允许你随时暂停和播放。

## Declarative Rendering [声明式渲染](https://cn.vuejs.org/v2/guide/index.html#%E5%A3%B0%E6%98%8E%E5%BC%8F%E6%B8%B2%E6%9F%93)

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

Vue.js 的核心是一个允许采用简洁的模板语法来声明式地将数据渲染进 DOM 的系统：

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.

我们已经成功创建了第一个 Vue 应用！看起来这跟渲染一个字符串模板非常类似，但是 Vue 在背后做了大量工作。现在数据和 DOM 已经被建立了关联，所有东西都是**响应式的**。我们要怎么确认呢？打开你的浏览器的 JavaScript 控制台 (就在这个页面打开)，并修改 app.message 的值，你将看到上例相应地更新。

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

除了文本插值，我们还可以像这样来绑定元素特性：

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

鼠标悬停几秒钟查看此处动态绑定的提示信息！

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.”

这里我们遇到了一点新东西。你看到的 v-bind 特性被称为**指令**。指令带有前缀 v-，以表示它们是 Vue 提供的特殊特性。可能你已经猜到了，它们会在渲染的 DOM 上应用特殊的响应式行为。在这里，该指令的意思是：“将这个元素节点的 title 特性和 Vue 实例的 message 属性保持一致”。

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.

如果你再次打开浏览器的 JavaScript 控制台，输入 app2.message = '新消息'，就会再一次看到这个绑定了 title 特性的 HTML 已经进行了更新。

## Conditionals and Loops [条件与循环](https://cn.vuejs.org/v2/guide/index.html#%E6%9D%A1%E4%BB%B6%E4%B8%8E%E5%BE%AA%E7%8E%AF)

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

控制切换一个元素是否显示也相当简单：

Now you see me

现在你看到我了

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

继续在控制台输入 app3.seen = false，你会发现之前显示的消息消失了。

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

这个例子演示了我们不仅可以把数据绑定到 DOM 文本或特性，还可以绑定到 DOM **结构**。此外，Vue 也提供一个强大的过渡效果系统，可以在 Vue 插入/更新/移除元素时自动应用[**过渡效果**](https://cn.vuejs.org/v2/guide/transitions.html)。

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

还有其它很多指令，每个都有特殊的功能。例如，v-for 指令可以绑定数组的数据来渲染一个项目列表：

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

在控制台里，输入 app4.todos.push({ text: '新项目' })，你会发现列表最后添加了一个新项目。

## Handling User Input [处理用户输入](https://cn.vuejs.org/v2/guide/index.html#%E5%A4%84%E7%90%86%E7%94%A8%E6%88%B7%E8%BE%93%E5%85%A5)

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:

为了让用户和你的应用进行交互，我们可以用 v-on 指令添加一个事件监听器，通过它调用在 Vue 实例中定义的方法：

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.

注意在 reverseMessage 方法中，我们更新了应用的状态，但没有触碰 DOM——所有的 DOM 操作都由 Vue 来处理，你编写的代码只需要关注逻辑层面即可。

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

Vue 还提供了 v-model 指令，它能轻松实现表单输入和应用状态之间的双向绑定。

## Composing with Components [组件化应用构建](https://cn.vuejs.org/v2/guide/index.html#%E7%BB%84%E4%BB%B6%E5%8C%96%E5%BA%94%E7%94%A8%E6%9E%84%E5%BB%BA)

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:

组件系统是 Vue 的另一个重要概念，因为它是一种抽象，允许我们使用小型、独立和通常可复用的组件构建大型应用。仔细想想，几乎任意类型的应用界面都可以抽象为一个组件树：

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

在 Vue 里，一个组件本质上是一个拥有预定义选项的一个 Vue 实例。在 Vue 中注册组件很简单：

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

现在你可以用它构建另一个组件模板：

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

但是这样会为每个待办项渲染同样的文本，这看起来并不炫酷。我们应该能从父作用域将数据传到子组件才对。让我们来修改一下组件的定义，使之能够接受一个 [**prop**](https://cn.vuejs.org/v2/guide/components.html#%E9%80%9A%E8%BF%87-Prop-%E5%90%91%E5%AD%90%E7%BB%84%E4%BB%B6%E4%BC%A0%E9%80%92%E6%95%B0%E6%8D%AE)：

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

现在，我们可以使用 v-bind 指令将待办项传到循环输出的每个组件中：

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.

尽管这只是一个刻意设计的例子，但是我们已经设法将应用分割成了两个更小的单元。子单元通过 prop 接口与父单元进行了良好的解耦。我们现在可以进一步改进 <todo-item> 组件，提供更为复杂的模板和逻辑，而不会影响到父单元。

In a large application, it is necessary to divide the whole app into components to make development manageable. We will talk a lot more about components [**later in the guide**](https://vuejs.org/v2/guide/components.html), but here’s an (imaginary) example of what an app’s template might look like with components:  
在一个大型应用中，有必要将整个应用程序划分为组件，以使开发更易管理。在[**后续教程**](https://cn.vuejs.org/v2/guide/components.html)中我们将详述组件，不过这里有一个 (假想的) 例子，以展示使用了组件的应用模板是什么样的：

### Relation to Custom Elements [与自定义元素的关系](https://cn.vuejs.org/v2/guide/index.html#%E4%B8%8E%E8%87%AA%E5%AE%9A%E4%B9%89%E5%85%83%E7%B4%A0%E7%9A%84%E5%85%B3%E7%B3%BB)

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

你可能已经注意到 Vue 组件非常类似于**自定义元素**——它是 [**Web 组件规范**](https://www.w3.org/wiki/WebComponents/)的一部分，这是因为 Vue 的组件语法部分参考了该规范。例如 Vue 组件实现了 [**Slot API**](https://github.com/w3c/webcomponents/blob/gh-pages/proposals/Slots-Proposal.md) 与 is 特性。但是，还是有几个关键差别：

1. The Web Components Spec has been finalized, but is not natively implemented in every browser. Safari 10.1+, Chrome 54+ and Firefox 63+ natively support web components. In comparison, Vue components don’t require any polyfills and work consistently in all supported browsers (IE9 and above). When needed, Vue components can also be wrapped inside a native custom element.
2. Web Components 规范已经完成并通过，但未被所有浏览器原生实现。目前 Safari 10.1+、Chrome 54+ 和 Firefox 63+ 原生支持 Web Components。相比之下，Vue 组件不需要任何 polyfill，并且在所有支持的浏览器 (IE9 及更高版本) 之下表现一致。必要时，Vue 组件也可以包装于原生自定义元素之内。
3. 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.
4. Vue 组件提供了纯自定义元素所不具备的一些重要功能，最突出的是跨组件数据流、自定义事件通信以及构建工具集成。

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

虽然 Vue 内部没有使用自定义元素，不过在应用使用自定义元素、或以自定义元素形式发布时，[**依然有很好的互操作性**](https://custom-elements-everywhere.com/#vue)。Vue CLI 也支持将 Vue 组件构建成为原生的自定义元素。

## Ready for More? [准备好了吗？](https://cn.vuejs.org/v2/guide/index.html#%E5%87%86%E5%A4%87%E5%A5%BD%E4%BA%86%E5%90%97%EF%BC%9F)

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!

我们刚才简单介绍了 Vue 核心最基本的功能——本教程的其余部分将更加详细地涵盖这些功能以及其它高级功能，所以请务必读完整个教程！

# The Vue Instance Vue 实例

## [Creating a Vue Instance](https://vuejs.org/v2/guide/instance.html#Creating-a-Vue-Instance) [创建一个 Vue 实例](https://cn.vuejs.org/v2/guide/instance.html#%E5%88%9B%E5%BB%BA%E4%B8%80%E4%B8%AA-Vue-%E5%AE%9E%E4%BE%8B)

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

每个 Vue 应用都是通过用 Vue 函数创建一个新的 **Vue 实例**开始的：

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

虽然没有完全遵循 [**MVVM 模型**](https://zh.wikipedia.org/wiki/MVVM)，但是 Vue 的设计也受到了它的启发。因此在文档中经常会使用 vm (ViewModel 的缩写) 这个变量名表示 Vue 实例。

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

当创建一个 Vue 实例时，你可以传入一个**选项对象**。这篇教程主要描述的就是如何使用这些选项来创建你想要的行为。作为参考，你也可以在 [**API 文档**](https://cn.vuejs.org/v2/api/#%E9%80%89%E9%A1%B9-%E6%95%B0%E6%8D%AE) 中浏览完整的选项列表。

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:

一个 Vue 应用由一个通过 new Vue 创建的**根 Vue 实例**，以及可选的嵌套的、可复用的组件树组成。举个例子，一个 todo 应用的组件树可以是这样的：

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

我们会在稍后的[**组件系统**](https://cn.vuejs.org/v2/guide/components.html)章节具体展开。不过现在，你只需要明白所有的 Vue 组件都是 Vue 实例，并且接受相同的选项对象 (一些根实例特有的选项除外)。

## [Data and Methods](https://vuejs.org/v2/guide/instance.html#Data-and-Methods) [数据与方法](https://cn.vuejs.org/v2/guide/instance.html#%E6%95%B0%E6%8D%AE%E4%B8%8E%E6%96%B9%E6%B3%95)

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.

当一个 Vue 实例被创建时，它将 data 对象中的所有的属性加入到 Vue 的**响应式系统**中。当这些属性的值发生改变时，视图将会产生“响应”，即匹配更新为新的值。

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:

当这些数据改变时，视图会进行重渲染。值得注意的是只有当实例被创建时 data 中存在的属性才是**响应式**的。也就是说如果你添加一个新的属性，比如：

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:

那么对 b 的改动将不会触发任何视图的更新。如果你知道你会在晚些时候需要一个属性，但是一开始它为空或不存在，那么你仅需要设置一些初始值。比如：

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.

这里唯一的例外是使用 Object.freeze()，这会阻止修改现有的属性，也意味着响应系统无法再追踪变化。

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:

除了数据属性，Vue 实例还暴露了一些有用的实例属性与方法。它们都有前缀 $，以便与用户定义的属性区分开来。例如：

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

以后你可以在 [**API 参考**](https://cn.vuejs.org/v2/api/#%E5%AE%9E%E4%BE%8B%E5%B1%9E%E6%80%A7)中查阅到完整的实例属性和方法的列表。

## [Instance Lifecycle Hooks](https://vuejs.org/v2/guide/instance.html#Instance-Lifecycle-Hooks) [实例生命周期钩子](https://cn.vuejs.org/v2/guide/instance.html#%E5%AE%9E%E4%BE%8B%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F%E9%92%A9%E5%AD%90)

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.

每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做**生命周期钩子**的函数，这给了用户在不同阶段添加自己的代码的机会。

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

比如 [**created**](https://cn.vuejs.org/v2/api/#created) 钩子可以用来在一个实例被创建之后执行代码：

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

也有一些其它的钩子，在实例生命周期的不同阶段被调用，如 [**mounted**](https://cn.vuejs.org/v2/api/#mounted)、[**updated**](https://cn.vuejs.org/v2/api/#updated)和 [**destroyed**](https://cn.vuejs.org/v2/api/#destroyed)。生命周期钩子的 this 上下文指向调用它的 Vue 实例。

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

不要在选项属性或回调上使用[**箭头函数**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/Arrow_functions)，比如 created: () => console.log(this.a) 或 vm.$watch('a', newValue => this.myMethod())。因为箭头函数并没有 this，this 会作为变量一直向上级词法作用域查找，直至找到为止，经常导致 Uncaught TypeError: Cannot read property of undefined 或 Uncaught TypeError: this.myMethod is not a function 之类的错误。

## [Lifecycle Diagram](https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram) [生命周期图示](https://cn.vuejs.org/v2/guide/instance.html#%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F%E5%9B%BE%E7%A4%BA)

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.

下图展示了实例的生命周期。你不需要立马弄明白所有的东西，不过随着你的不断学习和使用，它的参考价值会越来越高。

# Template Syntax模板语法

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.

Vue.js 使用了基于 HTML 的模板语法，允许开发者声明式地将 DOM 绑定至底层 Vue 实例的数据。所有 Vue.js 的模板都是合法的 HTML ，所以能被遵循规范的浏览器和 HTML 解析器解析。

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.

在底层的实现上，Vue 将模板编译成虚拟 DOM 渲染函数。结合响应系统，Vue 能够智能地计算出最少需要重新渲染多少组件，并把 DOM 操作次数减到最少。

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

如果你熟悉虚拟 DOM 并且偏爱 JavaScript 的原始力量，你也可以不用模板，[**直接写渲染 (render) 函数**](https://cn.vuejs.org/v2/guide/render-function.html)，使用可选的 JSX 语法。

## [Interpolations](https://vuejs.org/v2/guide/syntax.html#Interpolations) [插值](https://cn.vuejs.org/v2/guide/syntax.html#%E6%8F%92%E5%80%BC)

### [Text](https://vuejs.org/v2/guide/syntax.html#Text) [文本](https://cn.vuejs.org/v2/guide/syntax.html#%E6%96%87%E6%9C%AC)

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

数据绑定最常见的形式就是使用“Mustache”语法 (双大括号) 的文本插值：

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.

Mustache 标签将会被替代为对应数据对象上 msg 属性的值。无论何时，绑定的数据对象上 msg 属性发生了改变，插值处的内容都会更新。

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

通过使用 [**v-once 指令**](https://cn.vuejs.org/v2/api/#v-once)，你也能执行一次性地插值，当数据改变时，插值处的内容不会更新。但请留心这会影响到该节点上的其它数据绑定：

### [Raw HTML](https://vuejs.org/v2/guide/syntax.html#Raw-HTML) [原始 HTML](https://cn.vuejs.org/v2/guide/syntax.html#%E5%8E%9F%E5%A7%8B-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:

双大括号会将数据解释为普通文本，而非 HTML 代码。为了输出真正的 HTML，你需要使用 v-html 指令：

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.

这个 span 的内容将会被替换成为属性值 rawHtml，直接作为 HTML——会忽略解析属性值中的数据绑定。注意，你不能使用 v-html 来复合局部模板，因为 Vue 不是基于字符串的模板引擎。反之，对于用户界面 (UI)，组件更适合作为可重用和可组合的基本单位。

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

你的站点上动态渲染的任意 HTML 可能会非常危险，因为它很容易导致 [**XSS 攻击**](https://en.wikipedia.org/wiki/Cross-site_scripting)。请只对可信内容使用 HTML 插值，**绝不要**对用户提供的内容使用插值。

### [Attributes](https://vuejs.org/v2/guide/syntax.html#Attributes) [特性](https://cn.vuejs.org/v2/guide/syntax.html#%E7%89%B9%E6%80%A7)

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

Mustache 语法不能作用在 HTML 特性上，遇到这种情况应该使用 [**v-bind 指令**](https://cn.vuejs.org/v2/api/#v-bind)：

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

对于布尔特性 (它们只要存在就意味着值为 true)，v-bind 工作起来略有不同，在这个例子中：

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

如果 isButtonDisabled 的值是 null、undefined 或 false，则 disabled 特性甚至不会被包含在渲染出来的 <button> 元素中。

### [Using JavaScript Expressions](https://vuejs.org/v2/guide/syntax.html#Using-JavaScript-Expressions) [使用 JavaScript 表达式](https://cn.vuejs.org/v2/guide/syntax.html#%E4%BD%BF%E7%94%A8-JavaScript-%E8%A1%A8%E8%BE%BE%E5%BC%8F)

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:

迄今为止，在我们的模板中，我们一直都只绑定简单的属性键值。但实际上，对于所有的数据绑定，Vue.js 都提供了完全的 JavaScript 表达式支持。

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:

这些表达式会在所属 Vue 实例的数据作用域下作为 JavaScript 被解析。有个限制就是，每个绑定都只能包含**单个表达式**，所以下面的例子都**不会**生效。

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.

模板表达式都被放在沙盒中，只能访问全局变量的一个白名单，如 Math 和 Date 。你不应该在模板表达式中试图访问用户定义的全局变量。

## [Directives](https://vuejs.org/v2/guide/syntax.html#Directives) [指令](https://cn.vuejs.org/v2/guide/syntax.html#%E6%8C%87%E4%BB%A4)

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

指令 (Directives) 是带有 v- 前缀的特殊特性。指令特性的值预期是**单个 JavaScript 表达式** (v-for 是例外情况，稍后我们再讨论)。指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM。回顾我们在介绍中看到的例子：

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

这里，v-if 指令将根据表达式 seen 的值的真假来插入/移除 <p> 元素。

### [Arguments](https://vuejs.org/v2/guide/syntax.html#Arguments) [参数](https://cn.vuejs.org/v2/guide/syntax.html#%E5%8F%82%E6%95%B0)

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:

一些指令能够接收一个“参数”，在指令名称之后以冒号表示。例如，v-bind 指令可以用于响应式地更新 HTML 特性：

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.

在这里 href 是参数，告知 v-bind 指令将该元素的 href 特性与表达式 url 的值绑定。

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

另一个例子是 v-on 指令，它用于监听 DOM 事件：

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

在这里参数是监听的事件名。我们也会更详细地讨论事件处理。

### [Dynamic Arguments](https://vuejs.org/v2/guide/syntax.html#Dynamic-Arguments) [动态参数](https://cn.vuejs.org/v2/guide/syntax.html#%E5%8A%A8%E6%80%81%E5%8F%82%E6%95%B0)

**New in 2.6.0+ 2.6.0 新增**

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

从 2.6.0 开始，可以用方括号括起来的 JavaScript 表达式作为一个指令的参数：

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

这里的 attributeName 会被作为一个 JavaScript 表达式进行动态求值，求得的值将会作为最终的参数来使用。例如，如果你的 Vue 实例有一个 data 属性 attributeName，其值为 "href"，那么这个绑定将等价于 v-bind:href。

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

同样地，你可以使用动态参数为一个动态的事件名绑定处理函数：

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

同样地，当 eventName 的值为 "focus" 时，v-on:[eventName] 将等价于 v-on:focus。

#### Dynamic Argument Value Constraints 对动态参数的值的约束

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

动态参数预期会求出一个字符串，异常情况下值为 null。这个特殊的 null 值可以被显性地用于移除绑定。任何其它非字符串类型的值都将会触发一个警告。

#### Dynamic Argument Expression Constraints 对动态参数表达式的约束

Dynamic argument expressions have some syntax constraints because certain characters are invalid inside HTML attribute names, such as spaces and quotes. You also need to avoid uppercase keys when using in-DOM templates.

动态参数表达式有一些语法约束，因为某些字符，例如空格和引号，放在 HTML 特性名里是无效的。同样，在 DOM 中使用模板时你需要回避大写键名。

For example, the following is invalid:

例如，下面的代码是无效的：

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

变通的办法是使用没有空格或引号的表达式，或用计算属性替代这种复杂表达式。

In addition, if you are using in-DOM templates (templates directly written in an HTML file), you have to be aware that browsers will coerce attribute names into lowercase:

另外，如果你在 DOM 中使用模板 (直接在一个 HTML 文件里撰写模板)，需要留意浏览器会把特性名全部强制转为小写：

### [Modifiers](https://vuejs.org/v2/guide/syntax.html#Modifiers) [修饰符](https://cn.vuejs.org/v2/guide/syntax.html#%E4%BF%AE%E9%A5%B0%E7%AC%A6)

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:

修饰符 (modifier) 是以半角句号 . 指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。例如，.prevent 修饰符告诉 v-on 指令对于触发的事件调用 event.preventDefault()：

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

在接下来对 [**v-on**](https://cn.vuejs.org/v2/guide/events.html#%E4%BA%8B%E4%BB%B6%E4%BF%AE%E9%A5%B0%E7%AC%A6) 和 [**v-for**](https://cn.vuejs.org/v2/guide/forms.html#%E4%BF%AE%E9%A5%B0%E7%AC%A6) 等功能的探索中，你会看到修饰符的其它例子。

## [Shorthands](https://vuejs.org/v2/guide/syntax.html#Shorthands) [缩写](https://cn.vuejs.org/v2/guide/syntax.html#%E7%BC%A9%E5%86%99)

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

v- 前缀作为一种视觉提示，用来识别模板中 Vue 特定的特性。当你在使用 Vue.js 为现有标签添加动态行为 (dynamic behavior) 时，v- 前缀很有帮助，然而，对于一些频繁用到的指令来说，就会感到使用繁琐。同时，在构建由 Vue 管理所有模板的[**单页面应用程序 (SPA - single page application)**](https://en.wikipedia.org/wiki/Single-page_application) 时，v- 前缀也变得没那么重要了。因此，Vue 为 v-bind 和 v-on 这两个最常用的指令，提供了特定简写：

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

它们看起来可能与普通的 HTML 略有不同，但 : 与 @ 对于特性名来说都是合法字符，在所有支持 Vue 的浏览器都能被正确地解析。而且，它们不会出现在最终渲染的标记中。缩写语法是完全可选的，但随着你更深入地了解它们的作用，你会庆幸拥有它们。

# Computed Properties and Watchers 计算属性和侦听器

## [Computed Properties](https://vuejs.org/v2/guide/computed.html#Computed-Properties) [计算属性](https://cn.vuejs.org/v2/guide/computed.html#%E8%AE%A1%E7%AE%97%E5%B1%9E%E6%80%A7)

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:

模板内的表达式非常便利，但是设计它们的初衷是用于简单运算的。在模板中放入太多的逻辑会让模板过重且难以维护。例如：

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.

在这个地方，模板不再是简单的声明式逻辑。你必须看一段时间才能意识到，这里是想要显示变量 message 的翻转字符串。当你想要在模板中多次引用此处的翻转字符串时，就会更加难以处理。

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

所以，对于任何复杂逻辑，你都应当使用**计算属性**。

### [Basic Example](https://vuejs.org/v2/guide/computed.html#Basic-Example) [基础例子](https://cn.vuejs.org/v2/guide/computed.html#%E5%9F%BA%E7%A1%80%E4%BE%8B%E5%AD%90)

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

这里我们声明了一个计算属性 reversedMessage。我们提供的函数将用作属性 vm.reversedMessage 的 getter 函数：

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

你可以打开浏览器的控制台，自行修改例子中的 vm。vm.reversedMessage 的值始终取决于 vm.message 的值。

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.

你可以像绑定普通属性一样在模板中绑定计算属性。Vue 知道 vm.reversedMessage依赖于 vm.message，因此当 vm.message 发生改变时，所有依赖 vm.reversedMessage 的绑定也会更新。而且最妙的是我们已经以声明的方式创建了这种依赖关系：计算属性的 getter 函数是没有副作用 (side effect) 的，这使它更易于测试和理解。

### [Computed Caching vs Methods](https://vuejs.org/v2/guide/computed.html#Computed-Caching-vs-Methods) [算属性缓存 vs 方法](https://cn.vuejs.org/v2/guide/computed.html#%E8%AE%A1%E7%AE%97%E5%B1%9E%E6%80%A7%E7%BC%93%E5%AD%98-vs-%E6%96%B9%E6%B3%95)

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

你可能已经注意到我们可以通过在表达式中调用方法来达到同样的效果：

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 reactive dependencies.** A computed property will only re-evaluate when some of its reactive dependencies have changed. This means as long as message has not changed, multiple access to the reversedMessage computed property will immediately return the previously computed result without having to run the function again.

我们可以将同一函数定义为一个方法而不是一个计算属性。两种方式的最终结果确实是完全相同的。然而，不同的是**计算属性是基于它们的响应式依赖进行缓存的**。只在相关响应式依赖发生改变时它们才会重新求值。这就意味着只要 message 还没有发生改变，多次访问 reversedMessage 计算属性会立即返回之前的计算结果，而不必再次执行函数。

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

这也同样意味着下面的计算属性将不再更新，因为 Date.now() 不是响应式依赖：

In comparison, a method invocation will **always** run the function whenever a re-render happens.

相比之下，每当触发重新渲染时，调用方法将**总会**再次执行函数。

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.

我们为什么需要缓存？假设我们有一个性能开销比较大的计算属性 **A**，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于 **A** 。如果没有缓存，我们将不可避免的多次执行 **A** 的 getter！如果你不希望有缓存，请用方法来替代。

### [Computed vs Watched Property](https://vuejs.org/v2/guide/computed.html#Computed-vs-Watched-Property) [计算属性 vs 侦听属性](https://cn.vuejs.org/v2/guide/computed.html#%E8%AE%A1%E7%AE%97%E5%B1%9E%E6%80%A7-vs-%E4%BE%A6%E5%90%AC%E5%B1%9E%E6%80%A7)

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:

Vue 提供了一种更通用的方式来观察和响应 Vue 实例上的数据变动：**侦听属性**。当你有一些数据需要随着其它数据变动而变动时，你很容易滥用 watch——特别是如果你之前使用过 AngularJS。然而，通常更好的做法是使用计算属性而不是命令式的 watch 回调。细想一下这个例子：

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

上面代码是命令式且重复的。将它与计算属性的版本进行比较：

Much better, isn’t it?

好得多了，不是吗？

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

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

// ...

computed: {

fullName: {

// getter

get: function () {

return this.firstName + ' ' + this.lastName

},

// setter

set: function (newValue) {

var names = newValue.split(' ')

this.firstName = names[0]

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

}

}

}

// ...

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

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

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

For example:

<div id="watch-example">

<p>

Ask a yes/no question:

<input v-model="question">

</p>

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

</div>

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

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

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

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

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

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

<script>

var watchExampleVM = new Vue({

el: '#watch-example',

data: {

question: '',

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

},

watch: {

// whenever question changes, this function will run

question: function (newQuestion, oldQuestion) {

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

this.debouncedGetAnswer()

}

},

created: function () {

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

// often a particularly expensive operation can be run.

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

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

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

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

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

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

},

methods: {

getAnswer: function () {

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

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

return

}

this.answer = 'Thinking...'

var vm = this

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

.then(function (response) {

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

})

.catch(function (error) {

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

})

}

}

})

</script>

Result:

Ask a yes/no question: 

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

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

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

# Class and Style Bindings

A common need for data binding is manipulating an element’s class list and its inline styles. Since they are both attributes, we can use v-bind to handle them: we only need to calculate a final string with our expressions. However, meddling with string concatenation is annoying and error-prone. For this reason, Vue provides special enhancements when v-bind is used with class and style. In addition to strings, the expressions can also evaluate to objects or arrays.

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

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

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

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

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

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

<div

class="static"

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

></div>

And the following data:

data: {

isActive: true,

hasError: false

}

It will render:

<div class="static active"></div>

When isActive or hasError changes, the class list will be updated accordingly. For example, if hasError becomes true, the class list will become "static active text-danger".

The bound object doesn’t have to be inline:

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

data: {

classObject: {

active: true,

'text-danger': false

}

}

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

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

data: {

isActive: true,

error: null

},

computed: {

classObject: function () {

return {

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

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

}

}

}

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

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

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

data: {

activeClass: 'active',

errorClass: 'text-danger'

}

Which will render:

<div class="active text-danger"></div>

If you would like to also toggle a class in the list conditionally, you can do it with a ternary expression:

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

This will always apply errorClass, but will only apply activeClass when isActive is truthy.

However, this can be a bit verbose if you have multiple conditional classes. That’s why it’s also possible to use the object syntax inside array syntax:

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

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

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

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

For example, if you declare this component:

Vue.component('my-component', {

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

})

Then add some classes when using it:

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

The rendered HTML will be:

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

The same is true for class bindings:

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

When isActive is truthy, the rendered HTML will be:

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

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

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

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

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

data: {

activeColor: 'red',

fontSize: 30

}

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

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

data: {

styleObject: {

color: 'red',

fontSize: '13px'

}

}

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

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

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

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

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

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

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

**2.3.0+**

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

<div v-bind:style="{ display: ['-webkit-box', '-ms-flexbox', 'flex'] }"></div>

This will only render the last value in the array which the browser supports. In this example, it will render display: flex for browsers that support the unprefixed version of flexbox.

# Conditional Rendering

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

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

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

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

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

<h1 v-else>Oh no 😢</h1>

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

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

<template v-if="ok">

<h1>Title</h1>

<p>Paragraph 1</p>

<p>Paragraph 2</p>

</template>

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

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

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

Now you see me

</div>

<div v-else>

Now you don't

</div>

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

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

**New in 2.1.0+**

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

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

A

</div>

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

B

</div>

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

C

</div>

<div v-else>

Not A/B/C

</div>

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

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

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

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

<label>Username</label>

<input placeholder="Enter your username">

</template>

<template v-else>

<label>Email</label>

<input placeholder="Enter your email address">

</template>

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

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

Username 

Toggle login type

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

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

<label>Username</label>

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

</template>

<template v-else>

<label>Email</label>

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

</template>

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

Username 

Toggle login type

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# List Rendering

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

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

* Foo
* Bar

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

<ul id="example-2">

<li v-for="(item, index) in items">

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

</li>

</ul>

var example2 = new Vue({

el: '#example-2',

data: {

parentMessage: 'Parent',

items: [

{ message: 'Foo' },

{ message: 'Bar' }

]

}

})

Result:

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

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

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

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

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

<ul id="v-for-object" class="demo">

<li v-for="value in object">

{{ value }}

</li>

</ul>

new Vue({

el: '#v-for-object',

data: {

object: {

title: 'How to do lists in Vue',

author: 'Jane Doe',

publishedAt: '2016-04-10'

}

}

})

Result:

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

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

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

{{ name }}: {{ value }}

</div>

title: How to do lists in Vue

author: Jane Doe

publishedAt: 2016-04-10

And another for the index:

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

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

</div>

0. title: How to do lists in Vue

1. author: Jane Doe

2. publishedAt: 2016-04-10

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

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

When Vue is updating a list of elements rendered with v-for, by default it uses an “in-place patch” strategy. If the order of the data items has changed, instead of moving the DOM elements to match the order of the items, Vue will patch each element in-place and make sure it reflects what should be rendered at that particular index. This is similar to the behavior of track-by="$index" in Vue 1.x.

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

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

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

<!-- content -->

</div>

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

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

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

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

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

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

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

* push()
* pop()
* shift()
* unshift()
* splice()
* sort()
* reverse()

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

### [Replacing an Array](https://vuejs.org/v2/guide/list.html#Replacing-an-Array)

Mutation methods, as the name suggests, mutate the original array they are called on. In comparison, there are also non-mutating methods, e.g. filter(), concat() and slice(), which do not mutate the original array but **always return a new array**. When working with non-mutating methods, you can replace the old array with the new one:

example1.items = example1.items.filter(function (item) {

return item.message.match(/Foo/)

})

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

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

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

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

For example:

var vm = new Vue({

data: {

items: ['a', 'b', 'c']

}

})

vm.items[1] = 'x' // is NOT reactive

vm.items.length = 2 // is NOT reactive

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

// Vue.set

Vue.set(vm.items, indexOfItem, newValue)

// Array.prototype.splice

vm.items.splice(indexOfItem, 1, newValue)

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

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

To deal with caveat 2, you can use splice:

vm.items.splice(newLength)

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

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

var vm = new Vue({

data: {

a: 1

}

})

// `vm.a` is now reactive

vm.b = 2

// `vm.b` is NOT reactive

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

var vm = new Vue({

data: {

userProfile: {

name: 'Anika'

}

}

})

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

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

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

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

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

Object.assign(vm.userProfile, {

age: 27,

favoriteColor: 'Vue Green'

})

You would add new, reactive properties with:

vm.userProfile = Object.assign({}, vm.userProfile, {

age: 27,

favoriteColor: 'Vue Green'

})

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

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

For example:

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

data: {

numbers: [ 1, 2, 3, 4, 5 ]

},

computed: {

evenNumbers: function () {

return this.numbers.filter(function (number) {

return number % 2 === 0

})

}

}

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

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

data: {

numbers: [ 1, 2, 3, 4, 5 ]

},

methods: {

even: function (numbers) {

return numbers.filter(function (number) {

return number % 2 === 0

})

}

}

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

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

<div>

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

</div>

Result:

1 2 3 4 5 6 7 8 9 10

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

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

<ul>

<template v-for="item in items">

<li>{{ item.msg }}</li>

<li class="divider" role="presentation"></li>

</template>

</ul>

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

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

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

<li v-for="todo in todos" v-if="!todo.isComplete">

{{ todo }}

</li>

The above only renders the todos that are not complete.

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

<ul v-if="todos.length">

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

{{ todo }}

</li>

</ul>

<p v-else>No todos left!</p>

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

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

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

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

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

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

<my-component

v-for="(item, index) in items"

v-bind:item="item"

v-bind:index="index"

v-bind:key="item.id"

></my-component>

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

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

<div id="todo-list-example">

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

<label for="new-todo">Add a todo</label>

<input

v-model="newTodoText"

id="new-todo"

placeholder="E.g. Feed the cat"

>

<button>Add</button>

</form>

<ul>

<li

is="todo-item"

v-for="(todo, index) in todos"

v-bind:key="todo.id"

v-bind:title="todo.title"

v-on:remove="todos.splice(index, 1)"

></li>

</ul>

</div>

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

Vue.component('todo-item', {

template: '\

<li>\

{{ title }}\

<button v-on:click="$emit(\'remove\')">Remove</button>\

</li>\

',

props: ['title']

})

new Vue({

el: '#todo-list-example',

data: {

newTodoText: '',

todos: [

{

id: 1,

title: 'Do the dishes',

},

{

id: 2,

title: 'Take out the trash',

},

{

id: 3,

title: 'Mow the lawn'

}

],

nextTodoId: 4

},

methods: {

addNewTodo: function () {

this.todos.push({

id: this.nextTodoId++,

title: this.newTodoText

})

this.newTodoText = ''

}

}

})

窗体顶端

Add a todo  Add

窗体底端

* Do the dishes Remove
* Take out the trash Remove
* Mow the lawn Remove

# Event Handling

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

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

For example:

<div id="example-1">

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

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

</div>

var example1 = new Vue({

el: '#example-1',

data: {

counter: 0

}

})

Result:

Add 1

The button above has been clicked 0 times.

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

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

For example:

<div id="example-2">

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

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

</div>

var example2 = new Vue({

el: '#example-2',

data: {

name: 'Vue.js'

},

// define methods under the `methods` object

methods: {

greet: function (event) {

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

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

// `event` is the native DOM event

if (event) {

alert(event.target.tagName)

}

}

}

})

// you can invoke methods in JavaScript too

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

Result:

Greet

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

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

<div id="example-3">

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

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

</div>

new Vue({

el: '#example-3',

methods: {

say: function (message) {

alert(message)

}

}

})

Result:

Say hi Say what

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

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

Submit

</button>

// ...

methods: {

warn: function (message, event) {

// now we have access to the native event

if (event) event.preventDefault()

alert(message)

}

}

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

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

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

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

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

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

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

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

<!-- modifiers can be chained -->

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

<!-- just the modifier -->

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

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

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

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

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

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

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

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

**New in 2.1.4+**

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

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

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

**New in 2.3.0+**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Using keyCode attributes is also permitted:

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

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

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

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

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

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

Vue.config.keyCodes.f1 = 112

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

**New in 2.1.0+**

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

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

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

For example:

<!-- Alt + C -->

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

**New in 2.5.0+**

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

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

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

**New in 2.2.0+**

* .left
* .right
* .middle

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

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

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

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

# Form Input Bindings

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

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

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

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

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

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

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

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

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



Message is:

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

<span>Multiline message is:</span>

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

<br>

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

Multiline message is:



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

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

Single checkbox, boolean value:

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

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

 false

Multiple checkboxes, bound to the same Array:

<div id='example-3'>

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

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

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

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

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

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

<br>

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

</div>

new Vue({

el: '#example-3',

data: {

checkedNames: []

}

})

 Jack  John  Mike   
Checked names: []

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

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

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

<br>

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

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

<br>

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

 One   
 Two   
Picked:

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

Single select:

<select v-model="selected">

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

<option>A</option>

<option>B</option>

<option>C</option>

</select>

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

new Vue({

el: '...',

data: {

selected: ''

}

})

 Selected:

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

Multiple select (bound to Array):

<select v-model="selected" multiple>

<option>A</option>

<option>B</option>

<option>C</option>

</select>

<br>

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

   
Selected: []

Dynamic options rendered with v-for:

<select v-model="selected">

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

{{ option.text }}

</option>

</select>

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

new Vue({

el: '...',

data: {

selected: 'A',

options: [

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

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

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

]

}

})

 Selected: A

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

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

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

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

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

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

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

<select v-model="selected">

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

</select>

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

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

<input

type="checkbox"

v-model="toggle"

true-value="yes"

false-value="no"

>

// when checked:

vm.toggle === 'yes'

// when unchecked:

vm.toggle === 'no'

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

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

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

// when checked:

vm.pick === vm.a

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

<select v-model="selected">

<!-- inline object literal -->

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

</select>

// when selected:

typeof vm.selected // => 'object'

vm.selected.number // => 123

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

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

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

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

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

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

If you want user input to be automatically typecast as a number, you can add the 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. If the value cannot be parsed with parseFloat(), then the original value is returned.

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

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

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

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

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

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

# Components Basics

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

Here’s an example of a Vue component:

// Define a new component called button-counter

Vue.component('button-counter', {

data: function () {

return {

count: 0

}

},

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

})

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

<div id="components-demo">

<button-counter></button-counter>

</div>

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

You clicked me 0 times.

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

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

Components can be reused as many times as you want:

<div id="components-demo">

<button-counter></button-counter>

<button-counter></button-counter>

<button-counter></button-counter>

</div>

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

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

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

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

data: {

count: 0

}

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

data: function () {

return {

count: 0

}

}

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

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

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

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



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

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

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

// ... options ...

})

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

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

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

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

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

Vue.component('blog-post', {

props: ['title'],

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

})

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

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

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

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

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

### My journey with Vue

### Blogging with Vue

### Why Vue is so fun

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

new Vue({

el: '#blog-post-demo',

data: {

posts: [

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

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

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

]

}

})

Then want to render a component for each one:

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:title="post.title"

></blog-post>

Above, you’ll see that we can use v-bind to dynamically pass props. This is especially useful when you don’t know the exact content you’re going to render ahead of time, like when [**fetching posts from an API**](https://jsfiddle.net/chrisvfritz/sbLgr0ad).

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

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

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

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

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

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

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

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

<div class="blog-post">

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

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

</div>

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

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:title="post.title"

v-bind:content="post.content"

v-bind:publishedAt="post.publishedAt"

v-bind:comments="post.comments"

></blog-post>

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

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:post="post"

></blog-post>

Vue.component('blog-post', {

props: ['post'],

template: `

<div class="blog-post">

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

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

</div>

`

})

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

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

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

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

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

new Vue({

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

data: {

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

postFontSize: 1

}

})

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

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

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

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:post="post"

></blog-post>

</div>

</div>

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

Vue.component('blog-post', {

props: ['post'],

template: `

<div class="blog-post">

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

<button>

Enlarge text

</button>

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

</div>

`

})

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

<button>

Enlarge text

</button>

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

<blog-post

...

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

></blog-post>

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

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

Enlarge text

</button>

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

### My journey with Vue

Enlarge text

...content...

### Blogging with Vue

Enlarge text

...content...

### Why Vue is so fun

Enlarge text

...content...

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

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

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

Enlarge text

</button>

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

<blog-post

...

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

></blog-post>

Or, if the event handler is a method:

<blog-post

...

v-on:enlarge-text="onEnlargeText"

></blog-post>

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

methods: {

onEnlargeText: function (enlargeAmount) {

this.postFontSize += enlargeAmount

}

}

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

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

<input v-model="searchText">

does the same thing as:

<input

v-bind:value="searchText"

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

>

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

<custom-input

v-bind:value="searchText"

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

></custom-input>

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

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

Here’s that in action:

Vue.component('custom-input', {

props: ['value'],

template: `

<input

v-bind:value="value"

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

>

`

})

Now v-model should work perfectly with this component:

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

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

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

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

<alert-box>

Something bad happened.

</alert-box>

Which might render something like:

**Error!** Something bad happened.

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

Vue.component('alert-box', {

template: `

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

<strong>Error!</strong>

<slot></slot>

</div>

`

})

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

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

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

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

HomePostsArchive

Home component

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

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

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

In the example above, currentTabComponent can contain either:

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

See [**this fiddle**](https://jsfiddle.net/chrisvfritz/o3nycadu/) to experiment with the full code, or [**this version**](https://jsfiddle.net/chrisvfritz/b2qj69o1/) 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**](https://vuejs.org/v2/guide/components-dynamic-async.html).

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

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

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

<table>

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

</table>

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

<table>

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

</table>

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

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

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

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

### Components In-Depth

# Component Registration

**This page assumes you’ve already read the**[**Components Basics**](https://vuejs.org/v2/guide/components.html)**. Read that first if you are new to components.**

## [Component Names](https://vuejs.org/v2/guide/components-registration.html#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**](https://vuejs.org/v2/guide/single-file-components.html)), we strongly recommend following the [**W3C rules**](https://html.spec.whatwg.org/multipage/custom-elements.html#valid-custom-element-name) 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**](https://vuejs.org/v2/style-guide/#Base-component-names-strongly-recommended).

### [Name Casing](https://vuejs.org/v2/guide/components-registration.html#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](https://vuejs.org/v2/guide/components-registration.html#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](https://vuejs.org/v2/guide/components-registration.html#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](https://vuejs.org/v2/guide/components-registration.html#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](https://vuejs.org/v2/guide/components-registration.html#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](https://vuejs.org/v2/guide/components-registration.html#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**](https://vuejs.org/v2/style-guide/#Base-component-names-strongly-recommended) 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+**](https://github.com/vuejs/vue-cli), 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(

// Gets the file name regardless of folder depth

fileName

.split('/')

.pop()

.replace(/\.\w+$/, '')

)

)

// 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**](https://github.com/chrisvfritz/vue-enterprise-boilerplate/blob/master/src/components/_globals.js) of this pattern in a real project context.

# Props

**This page assumes you’ve already read the**[**Components Basics**](https://vuejs.org/v2/guide/components.html)**. Read that first if you are new to components.**

## [Prop Casing (camelCase vs kebab-case)](https://vuejs.org/v2/guide/components-props.html#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](https://vuejs.org/v2/guide/components-props.html#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,

callback: Function,

contactsPromise: Promise // or any other constructor

}

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**](https://vuejs.org/v2/guide/components-props.html#Prop-Validation)further down this page.

## [Passing Static or Dynamic Props](https://vuejs.org/v2/guide/components-props.html#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](https://vuejs.org/v2/guide/components-props.html#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](https://vuejs.org/v2/guide/components-props.html#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](https://vuejs.org/v2/guide/components-props.html#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](https://vuejs.org/v2/guide/components-props.html#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](https://vuejs.org/v2/guide/components-props.html#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](https://vuejs.org/v2/guide/components-props.html#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:
2. props: ['initialCounter'],
3. data: function () {
4. return {
5. counter: this.initialCounter
6. }

}

1. **The prop is passed in as a raw value that needs to be transformed.** In this case, it’s best to define a computed property using the prop’s value:
2. props: ['size'],
3. computed: {
4. normalizedSize: function () {
5. return this.size.trim().toLowerCase()
6. }

}

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](https://vuejs.org/v2/guide/components-props.html#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` and `undefined` values will pass any type validation)

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](https://vuejs.org/v2/guide/components-props.html#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](https://vuejs.org/v2/guide/components-props.html#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](https://vuejs.org/v2/guide/components-props.html#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](https://vuejs.org/v2/guide/components-props.html#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:

{

required: true,

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**](https://vuejs.org/v2/style-guide/#Base-component-names-strongly-recommended):

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>

`

})

Note that inheritAttrs: false option does **not** affect style and class bindings.

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"

required

placeholder="Enter your username"

></base-input>

# Custom Events

**This page assumes you’ve already read the**[**Components Basics**](https://vuejs.org/v2/guide/components.html)**. Read that first if you are new to components.**

## [Event Names](https://vuejs.org/v2/guide/components-custom-events.html#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:

<!-- Won't work -->

<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](https://vuejs.org/v2/guide/components-custom-events.html#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**](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/checkbox#Value). 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](https://vuejs.org/v2/guide/components-custom-events.html#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 .nativemodifier.

## [.sync Modifier](https://vuejs.org/v2/guide/components-custom-events.html#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:myPropName. 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>

Note that v-bind with the .sync modifier does **not** work with expressions (e.g. v-bind:title.sync=”doc.title + ‘!’” is invalid). Instead, you must only provide the name of the property you want to bind, similar to v-model.

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-onupdate 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.

# Slots

**This page assumes you’ve already read the**[**Components Basics**](https://vuejs.org/v2/guide/components.html)**. Read that first if you are new to components.**

**In 2.6.0, we introduced a new unified syntax (the v-slot directive) for named and scoped slots. It replaces the slot and slot-scope attributes, which are now deprecated, but have notbeen removed and are still documented**[**here**](https://vuejs.org/v2/guide/components-slots.html#Deprecated-Syntax)**. The rationale for introducing the new syntax is described in this**[**RFC**](https://github.com/vuejs/rfcs/blob/master/active-rfcs/0001-new-slot-syntax.md)**.**

## [Slot Content](https://vuejs.org/v2/guide/components-slots.html#Slot-Content)

Vue implements a content distribution API inspired by the [**Web Components spec draft**](https://github.com/w3c/webcomponents/blob/gh-pages/proposals/Slots-Proposal.md), 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, <slot></slot> 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>‘s template did **not** contain a <slot> element, any content provided between its opening and closing tag would be discarded.

## [Compilation Scope](https://vuejs.org/v2/guide/components-slots.html#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 urlwould not work:

<navigation-link url="/profile">

Clicking here will send you to: {{ url }}

<!--

The `url` will be undefined, because this content is passed

\_to\_ <navigation-link>, rather than defined \_inside\_ the

<navigation-link> component.

-->

</navigation-link>

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.**

## [Fallback Content](https://vuejs.org/v2/guide/components-slots.html#Fallback-Content)

There are cases when it’s useful to specify fallback (i.e. default) content for a slot, to be rendered only when no content is provided. For example, in a <submit-button> component:

<button type="submit">

<slot></slot>

</button>

We might want the text “Submit” to be rendered inside the <button> most of the time. To make “Submit” the fallback content, we can place it in between the <slot> tags:

<button type="submit">

<slot>Submit</slot>

</button>

Now when we use <submit-button> in a parent component, providing no content for the slot:

<submit-button></submit-button>

will render the fallback content, “Submit”:

<button type="submit">

Submit

</button>

But if we provide content:

<submit-button>

Save

</submit-button>

Then the provided content will be rendered instead:

<button type="submit">

Save

</button>

## [Named Slots](https://vuejs.org/v2/guide/components-slots.html#Named-Slots)

**Updated in 2.6.0+.**[**See here**](https://vuejs.org/v2/guide/components-slots.html#Deprecated-Syntax)**for the deprecated syntax using the slot attribute.**

There are times when it’s useful to have multiple slots. For example, in a <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>

A <slot> outlet without name implicitly has the name “default”.

To provide content to named slots, we can use the v-slot directive on a <template>, providing the name of the slot as v-slot‘s argument:

<base-layout>

<template v-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 v-slot:footer>

<p>Here's some contact info</p>

</template>

</base-layout>

Now everything inside the <template> elements will be passed to the corresponding slots. Any content not wrapped in a <template> using v-slot is assumed to be for the default slot.

However, you can still wrap default slot content in a <template> if you wish to be explicit:

<base-layout>

<template v-slot:header>

<h1>Here might be a page title</h1>

</template>

<template v-slot:default>

<p>A paragraph for the main content.</p>

<p>And another one.</p>

</template>

<template v-slot:footer>

<p>Here's some contact info</p>

</template>

</base-layout>

Either way, the rendered HTML will 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>

Note that **v-slot can only be added to a <template>** (with [**one exception**](https://vuejs.org/v2/guide/components-slots.html#Abbreviated-Syntax-for-Lone-Default-Slots)), unlike the deprecated [**slot attribute**](https://vuejs.org/v2/guide/components-slots.html#Deprecated-Syntax).

## [Scoped Slots](https://vuejs.org/v2/guide/components-slots.html#Scoped-Slots)

**Updated in 2.6.0+.**[**See here**](https://vuejs.org/v2/guide/components-slots.html#Deprecated-Syntax)**for the deprecated syntax using the slot-scope attribute.**

Sometimes, it’s useful for slot content to have access to data only available in the child component. For example, imagine a <current-user> component with the following template:

<span>

<slot>{{ user.lastName }}</slot>

</span>

We might want to replace this fallback content to display the user’s first name, instead of last, like this:

<current-user>

{{ user.firstName }}

</current-user>

That won’t work, however, because only the <current-user> component has access to the userand the content we’re providing is rendered in the parent.

To make user available to the slot content in the parent, we can bind user as an attribute to the <slot> element:

<span>

<slot v-bind:user="user">

{{ user.lastName }}

</slot>

</span>

Attributes bound to a <slot> element are called **slot props**. Now, in the parent scope, we can use v-slot with a value to define a name for the slot props we’ve been provided:

<current-user>

<template v-slot:default="slotProps">

{{ slotProps.user.firstName }}

</template>

</current-user>

In this example, we’ve chosen to name the object containing all our slot props slotProps, but you can use any name you like.

### [Abbreviated Syntax for Lone Default Slots](https://vuejs.org/v2/guide/components-slots.html#Abbreviated-Syntax-for-Lone-Default-Slots)

In cases like above, when only the default slot is provided content, the component’s tags can be used as the slot’s template. This allows us to use v-slot directly on the component:

<current-user v-slot:default="slotProps">

{{ slotProps.user.firstName }}

</current-user>

This can be shortened even further. Just as non-specified content is assumed to be for the default slot, v-slot without an argument is assumed to refer to the default slot:

<current-user v-slot="slotProps">

{{ slotProps.user.firstName }}

</current-user>

Note that the abbreviated syntax for default slot **cannot** be mixed with named slots, as it would lead to scope ambiguity:

<!-- INVALID, will result in warning -->

<current-user v-slot="slotProps">

{{ slotProps.user.firstName }}

<template v-slot:other="otherSlotProps">

slotProps is NOT available here

</template>

</current-user>

Whenever there are multiple slots, use the full <template> based syntax for all slots:

<current-user>

<template v-slot:default="slotProps">

{{ slotProps.user.firstName }}

</template>

<template v-slot:other="otherSlotProps">

...

</template>

</current-user>

### [Destructuring Slot Props](https://vuejs.org/v2/guide/components-slots.html#Destructuring-Slot-Props)

Internally, scoped slots work by wrapping your slot content in a function passed a single argument:

function (slotProps) {

// ... slot content ...

}

That means the value of v-slot can actually accept any valid JavaScript expression that can appear in the argument position of a function definition. So in supported environments ([**single-file components**](https://vuejs.org/v2/guide/single-file-components.html) or [**modern browsers**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Browser_compatibility)), you can also use [**ES2015 destructuring**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Object_destructuring) to pull out specific slot props, like so:

<current-user v-slot="{ user }">

{{ user.firstName }}

</current-user>

This can make the template much cleaner, especially when the slot provides many props. It also opens other possibilities, such as renaming props, e.g. user to person:

<current-user v-slot="{ user: person }">

{{ person.firstName }}

</current-user>

You can even define fallbacks, to be used in case a slot prop is undefined:

<current-user v-slot="{ user = { firstName: 'Guest' } }">

{{ user.firstName }}

</current-user>

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

**New in 2.6.0+**

[**Dynamic directive arguments**](https://vuejs.org/v2/guide/syntax.html#Dynamic-Arguments) also work on v-slot, allowing the definition of dynamic slot names:

<base-layout>

<template v-slot:[dynamicSlotName]>

...

</template>

</base-layout>

## [Named Slots Shorthand](https://vuejs.org/v2/guide/components-slots.html#Named-Slots-Shorthand)

**New in 2.6.0+**

Similar to v-on and v-bind, v-slot also has a shorthand, replacing everything before the argument (v-slot:) with the special symbol #. For example, v-slot:header can be rewritten as #header:

<base-layout>

<template #header>

<h1>Here might be a page title</h1>

</template>

<p>A paragraph for the main content.</p>

<p>And another one.</p>

<template #footer>

<p>Here's some contact info</p>

</template>

</base-layout>

However, just as with other directives, the shorthand is only available when an argument is provided. That means the following syntax is invalid:

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

<current-user #="{ user }">

{{ user.firstName }}

</current-user>

Instead, you must always specify the name of the slot if you wish to use the shorthand:

<current-user #default="{ user }">

{{ user.firstName }}

</current-user>

## [Other Examples](https://vuejs.org/v2/guide/components-slots.html#Other-Examples)

**Slot props allow us to turn slots into reusable templates that can render different content based on input props.** This is most useful when you are designing a reusable component that encapsulates data logic while allowing the consuming parent component to customize part of its layout.

For example, we are implementing a <todo-list> component that contains the layout and filtering logic for a list:

<ul>

<li

v-for="todo in filteredTodos"

v-bind:key="todo.id"

>

{{ todo.text }}

</li>

</ul>

Instead of hard-coding the content for each todo, we can let the parent component take control by making every todo a slot, then binding todo as a slot prop:

<ul>

<li

v-for="todo in filteredTodos"

v-bind:key="todo.id"

>

<!--

We have a slot for each todo, passing it the

`todo` object as a slot prop.

-->

<slot name="todo" 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:

<todo-list v-bind:todos="todos">

<template v-slot:todo="{ todo }">

<span v-if="todo.isComplete">✓</span>

{{ todo.text }}

</template>

</todo-list>

However, even this barely scratches the surface of what scoped slots are capable of. For real-life, powerful examples of scoped slot usage, we recommend browsing libraries such as [**Vue Virtual Scroller**](https://github.com/Akryum/vue-virtual-scroller), [**Vue Promised**](https://github.com/posva/vue-promised), and [**Portal Vue**](https://github.com/LinusBorg/portal-vue).

## [Deprecated Syntax](https://vuejs.org/v2/guide/components-slots.html#Deprecated-Syntax)

**The v-slot directive was introduced in Vue 2.6.0, offering an improved, alternative API to the still-supported slot and slot-scope attributes. The full rationale for introducing v-slot is described in this**[**RFC**](https://github.com/vuejs/rfcs/blob/master/active-rfcs/0001-new-slot-syntax.md)**. The slot and slot-scope attributes will continue to be supported in all future 2.x releases, but are officially deprecated and will eventually be removed in Vue 3.**

### [Named Slots with the slot Attribute](https://vuejs.org/v2/guide/components-slots.html#Named-Slots-with-the-slot-Attribute)

**Deprecated in 2.6.0+. See**[**here**](https://vuejs.org/v2/guide/components-slots.html#Named-Slots)**for the new, recommended syntax.**

To pass content to named slots from the parent, use the special slot attribute on <template>(using the <base-layout> component described [**here**](https://vuejs.org/v2/guide/components-slots.html#Named-Slots) as example):

<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 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>

### [Scoped Slots with the slot-scope Attribute](https://vuejs.org/v2/guide/components-slots.html#Scoped-Slots-with-the-slot-scope-Attribute)

**Deprecated in 2.6.0+. See**[**here**](https://vuejs.org/v2/guide/components-slots.html#Scoped-Slots)**for the new, recommended syntax.**

To receive props passed to a slot, the parent component can use <template> with the slot-scopeattribute (using the <slot-example> described [**here**](https://vuejs.org/v2/guide/components-slots.html#Scoped-Slots) as example):

<slot-example>

<template slot="default" slot-scope="slotProps">

{{ slotProps.msg }}

</template>

</slot-example>

Here, slot-scope declares the received props object as the slotProps variable, and makes it available inside the <template> scope. You can name slotProps anything you like similar to naming function arguments in JavaScript.

Here slot="default" can be omitted as it is implied:

<slot-example>

<template slot-scope="slotProps">

{{ slotProps.msg }}

</template>

</slot-example>

The slot-scope attribute can also be used directly on a non-<template> element (including components):

<slot-example>

<span slot-scope="slotProps">

{{ slotProps.msg }}

</span>

</slot-example>

The value of slot-scope can accept any valid JavaScript expression that can appear in the argument position of a function definition. This means in supported environments ([**single-file components**](https://vuejs.org/v2/guide/single-file-components.html) or [**modern browsers**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Browser_compatibility)) you can also use [**ES2015 destructuring**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Object_destructuring) in the expression, like so:

<slot-example>

<span slot-scope="{ msg }">

{{ msg }}

</span>

</slot-example>

Using the <todo-list> described [**here**](https://vuejs.org/v2/guide/components-slots.html#Other-Examples) as an example, here’s the equivalent usage using slot-scope:

<todo-list v-bind:todos="todos">

<template slot="todo" slot-scope="{ todo }">

<span v-if="todo.isComplete">✓</span>

{{ todo.text }}

</template>

</todo-list>

# Dynamic & Async Components

**This page assumes you’ve already read the**[**Components Basics**](https://vuejs.org/v2/guide/components.html)**. Read that first if you are new to components.**

## [keep-alive with Dynamic Components](https://vuejs.org/v2/guide/components-dynamic-async.html#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:

PostsArchive

* Cat Ipsum
* Hipster Ipsum
* Cupcake Ipsum

**Click on a blog title to the left to view it.**

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:

PostsArchive

* Cat Ipsum
* Hipster Ipsum
* Cupcake Ipsum

**Click on a blog title to the left to view it.**

Now the Posts tab maintains its state (the selected post) even when it’s not rendered. See [**this fiddle**](https://jsfiddle.net/chrisvfritz/Lp20op9o/) 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**](https://vuejs.org/v2/api/#keep-alive).

## [Async Components](https://vuejs.org/v2/guide/components-dynamic-async.html#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**](https://webpack.js.org/guides/code-splitting/):

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**](https://vuejs.org/v2/guide/components-registration.html#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**](https://github.com/substack/node-browserify/issues/58#issuecomment-21978224) that async loading “is not something that Browserify will ever support.” Officially, at least. The Browserify community has found [**some workarounds**](https://github.com/vuejs/vuejs.org/issues/620), 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](https://vuejs.org/v2/guide/components-dynamic-async.html#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**](https://github.com/vuejs/vue-router)**2.4.0+ if you wish to use the above syntax for route components.**

# Handling Edge Cases

**This page assumes you’ve already read the**[**Components Basics**](https://vuejs.org/v2/guide/components.html)**. 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](https://vuejs.org/v2/guide/components-edge-cases.html#Element-amp-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](https://vuejs.org/v2/guide/components-edge-cases.html#Accessing-the-Root-Instance)

In every subcomponent of a new Vue instance, this root instance can be accessed with the $rootproperty. 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**](https://github.com/vuejs/vuex) to manage state in most cases.

### [Accessing the Parent Component Instance](https://vuejs.org/v2/guide/components-edge-cases.html#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**](https://jsfiddle.net/chrisvfritz/ttzutdxh/).

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**](https://vuejs.org/v2/guide/components-edge-cases.html#Dependency-Injection).

### [Accessing Child Component Instances & Child Elements](https://vuejs.org/v2/guide/components-edge-cases.html#Accessing-Child-Component-Instances-amp-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 are only populated after the component has been rendered, and they are not reactive. It is only meant as an escape hatch for direct child manipulation - you should avoid accessing $refs from within templates or computed properties.

### [Dependency Injection](https://vuejs.org/v2/guide/components-edge-cases.html#Dependency-Injection)

Earlier, when we described [**Accessing the Parent Component Instance**](https://vuejs.org/v2/guide/components-edge-cases.html#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**](https://jsfiddle.net/chrisvfritz/tdv8dt3s/). The advantage over using $parent is that we can access getMapin 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**](https://vuejs.org/v2/guide/components-edge-cases.html#Accessing-the-Root-Instance) 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**](https://github.com/vuejs/vuex) instead.

Learn more about dependency injection in [**the API doc**](https://vuejs.org/v2/api/#provide-inject).

## [Programmatic Event Listeners](https://vuejs.org/v2/guide/components-edge-cases.html#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**](https://jsfiddle.net/chrisvfritz/1Leb7up8/) 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**](https://vuejs.org/v2/api/#Instance-Methods-Events).

Note that Vue’s event system is different from the browser’s [**EventTarget API**](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget). Though they work similarly, $emit, $on, and $off are **not** aliases for dispatchEvent, addEventListener, and removeEventListener.

## [Circular References](https://vuejs.org/v2/guide/components-edge-cases.html#Circular-References)

### [Recursive Components](https://vuejs.org/v2/guide/components-edge-cases.html#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](https://vuejs.org/v2/guide/components-edge-cases.html#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 andancestor 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').default

}

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](https://vuejs.org/v2/guide/components-edge-cases.html#Alternate-Template-Definitions)

### [Inline Templates](https://vuejs.org/v2/guide/components-edge-cases.html#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>

Your inline template needs to be defined inside the DOM element to which Vue is attached.

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](https://vuejs.org/v2/guide/components-edge-cases.html#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'

})

Your x-template needs to be defined outside the DOM element to which Vue is attached.

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](https://vuejs.org/v2/guide/components-edge-cases.html#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](https://vuejs.org/v2/guide/components-edge-cases.html#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**](https://vuejs.org/v2/guide/list.html#Caveats) or [**objects**](https://vuejs.org/v2/guide/list.html#Object-Change-Detection-Caveats), 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**](https://vuejs.org/v2/api/#vm-forceUpdate).

### [Cheap Static Components with v-once](https://vuejs.org/v2/guide/components-edge-cases.html#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.

### Transitions & Animation

# Enter/Leave & List Transitions

## [Overview](https://vuejs.org/v2/guide/transitions.html#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**](https://vuejs.org/v2/guide/transitioning-state.html).

## [Transitioning Single Elements/Components](https://vuejs.org/v2/guide/transitions.html#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;

}

Toggle

hello

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**](https://vuejs.org/v2/guide/transitions.html#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](https://vuejs.org/v2/guide/transitions.html#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.

### [CSS Transitions](https://vuejs.org/v2/guide/transitions.html#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;

}

Toggle render

hello

### [CSS Animations](https://vuejs.org/v2/guide/transitions.html#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 enim libero, at lacinia diam fermentum id. Pellentesque habitant morbi tristique senectus et netus.</p>

</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);

}

}

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.

### [Custom Transition Classes](https://vuejs.org/v2/guide/transitions.html#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**](https://daneden.github.io/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

}

})

Toggle render

hello

### [Using Transitions and Animations Together](https://vuejs.org/v2/guide/transitions.html#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](https://vuejs.org/v2/guide/transitions.html#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](https://vuejs.org/v2/guide/transitions.html#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 })

}

}

})

Toggle

## [Transitions on Initial Render](https://vuejs.org/v2/guide/transitions.html#Transitions-on-Initial-Render)

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

<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>

In the example above, either appear attribute or v-on:appear hook will cause an appear transition.

## [Transitioning Between Elements](https://vuejs.org/v2/guide/transitions.html#Transitioning-Between-Elements)

We discuss [**transitioning between components**](https://vuejs.org/v2/guide/transitions.html#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-ifs 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](https://vuejs.org/v2/guide/transitions.html#Transition-Modes)

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

off

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:

off

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

off

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>

off

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:

off

Pretty cool, right?

## [Transitioning Between Components](https://vuejs.org/v2/guide/transitions.html#Transitioning-Between-Components)

Transitioning between components is even simpler - we don’t even need the key attribute. Instead, we wrap a [**dynamic component**](https://vuejs.org/v2/guide/components.html#Dynamic-Components):

<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;

}

A B

Component A

## [List Transitions](https://vuejs.org/v2/guide/transitions.html#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**](https://vuejs.org/v2/guide/transitions.html#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](https://vuejs.org/v2/guide/transitions.html#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);

}

Add Remove

123456789

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](https://vuejs.org/v2/guide/transitions.html#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;

}

Shuffle

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

This might seem like magic, but under the hood, Vue is using an animation technique called [**FLIP**](https://aerotwist.com/blog/flip-your-animations/) 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;

}

Shuffle Add Remove

123456789

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**](https://jsfiddle.net/chrisvfritz/sLrhk1bc/):

**Lazy Sudoku**

Keep hitting the shuffle button until you win.

Shuffle

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

### [Staggering List Transitions](https://vuejs.org/v2/guide/transitions.html#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)

}

}

})



* Bruce Lee
* Jackie Chan
* Chuck Norris
* Jet Li
* Kung Fury

## [Reusable Transitions](https://vuejs.org/v2/guide/transitions.html#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**](https://vuejs.org/v2/guide/render-function.html#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](https://vuejs.org/v2/guide/transitions.html#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="maxFadeDuration">

<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

}

}

)

}

}

})

Fade In:  Fade Out:

hello

Start animating

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.

# State Transitions

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](https://vuejs.org/v2/guide/transitioning-state.html#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**](https://greensock.com/):

<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 });

}

}

})

0

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**](https://github.com/tweenjs/tween.js) and [**Color.js**](https://github.com/brehaut/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;

}

 Update

Preview:

#000000

## [Dynamic State Transitions](https://vuejs.org/v2/guide/transitioning-state.html#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.

Sides: 10Minimum Radius: 50%Update Interval: 500 milliseconds

See [**this fiddle**](https://jsfiddle.net/chrisvfritz/65gLu2b6/) for the complete code behind the above demo.

## [Organizing Transitions into Components](https://vuejs.org/v2/guide/transitioning-state.html#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 () {

vm.tweeningValue = this.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

}

}

})

 +  = 60

20 + 40 = 60

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**](https://vuejs.org/v2/guide/transitions.html). Together, there are very few limits to what can be accomplished.

## [Bringing Designs to Life](https://vuejs.org/v2/guide/transitioning-state.html#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:

### Reusability & Composition

# Mixins

## [Basics](https://vuejs.org/v2/guide/mixins.html#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.

[**Watch a video explanation on Vue Mastery**](https://www.vuemastery.com/courses/next-level-vue/mixins)

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](https://vuejs.org/v2/guide/mixins.html#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 recursive merge, 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](https://vuejs.org/v2/guide/mixins.html#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**](https://vuejs.org/v2/guide/plugins.html)to avoid duplicate application.

## [Custom Option Merge Strategies](https://vuejs.org/v2/guide/mixins.html#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**](https://github.com/vuejs/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)

}

}

# Custom Directives

## [Intro](https://vuejs.org/v2/guide/custom-directive.html#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:



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](https://vuejs.org/v2/guide/custom-directive.html#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).

We’ll cover VNodes in more detail [**later**](https://vuejs.org/v2/guide/render-function.html#The-Virtual-DOM), when we discuss [**render functions**](https://vuejs.org/v2/guide/render-function.html).

* 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](https://vuejs.org/v2/guide/custom-directive.html#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**](https://vuejs.org/v2/api/#VNode-Interface) for full details.
* oldVnode: The previous virtual node, only available in the update and componentUpdatedhooks.

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**](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/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!'

}

})

name: "demo"  
value: "hello!"  
expression: "message"  
argument: "foo"  
modifiers: {"a":true,"b":true}  
vnode keys: tag, data, children, text, elm, ns, context, fnContext, fnOptions, fnScopeId, key, componentOptions, componentInstance, parent, raw, isStatic, isRootInsert, isComment, isCloned, isOnce, asyncFactory, asyncMeta, isAsyncPlaceholder

Directive arguments can be dynamic. For example, in v-mydirective:argument=[dataproperty], argument is the string value assigned to the arg property in your directive hook binding parameter and dataproperty is a reference to a data property on your component instance assigned to the value property in the same binding parameter. As directive hooks are invoked, the value property of the binding parameter will dynamically change based on the value of dataproperty.

An example of a custom directive using a dynamic argument:

<div id="app">

<p>Scroll down the page</p>

<p v-tack:left="[dynamicleft]">I’ll now be offset from the left instead of the top</p>

</div>

Vue.directive('tack', {

bind(el, binding, vnode) {

el.style.position = 'fixed';

const s = (binding.arg == 'left' ? 'left' : 'top');

el.style[s] = binding.value + 'px';

}

})

// start app

new Vue({

el: '#app',

data() {

return {

dynamicleft: 500

}

}

})

## [Function Shorthand](https://vuejs.org/v2/guide/custom-directive.html#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](https://vuejs.org/v2/guide/custom-directive.html#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!"

})

# Render Functions & JSX

## [Basics](https://vuejs.org/v2/guide/render-function.html#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 v-slot directive 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**](https://vuejs.org/v2/api/#Instance-Properties)**before diving into render functions.**

## [Nodes, Trees, and the Virtual DOM](https://vuejs.org/v2/guide/render-function.html#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>

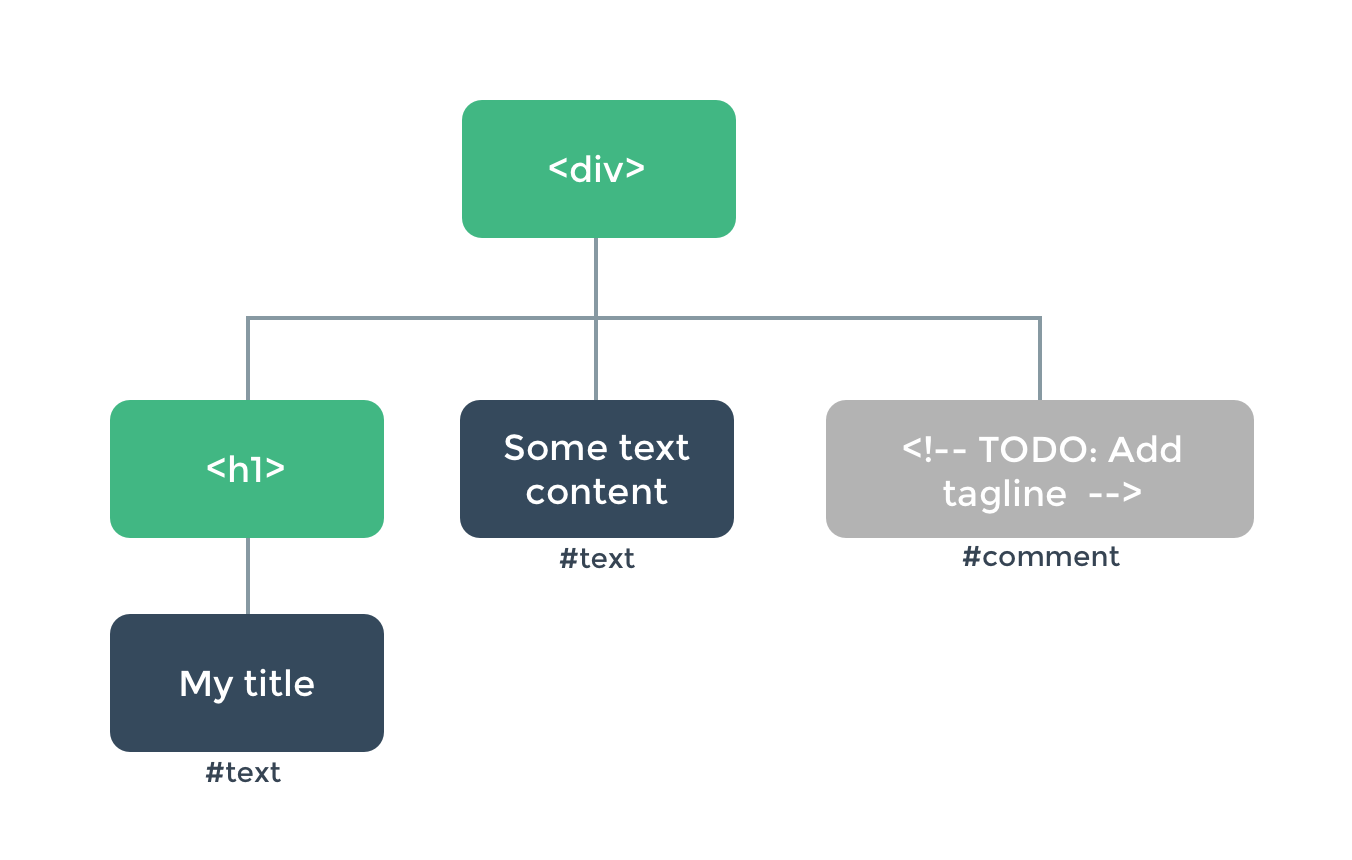
Some text content

<!-- TODO: Add tagline -->

</div>

When a browser reads this code, it builds a [**tree of “DOM nodes”**](https://javascript.info/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:



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](https://vuejs.org/v2/guide/render-function.html#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](https://vuejs.org/v2/guide/render-function.html#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](https://vuejs.org/v2/guide/render-function.html#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',

// If you are applying the same ref name to multiple

// elements in the render function. This will make `$refs.myRef` become an

// array

refInFor: true

}

### [Complete Example](https://vuejs.org/v2/guide/render-function.html#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 kebab-case 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](https://vuejs.org/v2/guide/render-function.html#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](https://vuejs.org/v2/guide/render-function.html#Replacing-Template-Features-with-Plain-JavaScript)

### [v-if and v-for](https://vuejs.org/v2/guide/render-function.html#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](https://vuejs.org/v2/guide/render-function.html#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](https://vuejs.org/v2/guide/render-function.html#Event-amp-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**](http://keycode.info/) 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](https://vuejs.org/v2/guide/render-function.html#Slots)

You can access static slot contents as Arrays of VNodes from [**this.$slots**](https://vuejs.org/v2/api/#vm-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**](https://vuejs.org/v2/api/#vm-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](https://vuejs.org/v2/guide/render-function.html#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**](https://github.com/vuejs/jsx) 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. Starting with [**version 3.4.0**](https://github.com/vuejs/babel-plugin-transform-vue-jsx#h-auto-injection) of the Babel plugin for Vue, we automatically inject const h = this.$createElement in any method and getter (not functions or arrow functions), declared in ES2015 syntax that has JSX, so you can drop the (h) parameter. With prior versions of the plugin, your app would throw an error if h was not available in the scope.

For more on how JSX maps to JavaScript, see the [**usage docs**](https://github.com/vuejs/jsx#installation).

## [Functional Components](https://vuejs.org/v2/guide/render-function.html#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**](https://vuejs.org/v2/api/#Options-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**](https://vuejs.org/v2/guide/single-file-components.html), 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
* scopedSlots: (2.6.0+) An object that exposes passed-in scoped slots. Also exposes normal slots as functions.
* data: The entire [**data object**](https://vuejs.org/v2/guide/render-function.html#The-Data-Object-In-Depth), 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**](https://vuejs.org/v2/api/#provide-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.

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](https://vuejs.org/v2/guide/render-function.html#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**](https://vuejs.org/v2/guide/class-and-style.html) 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](https://vuejs.org/v2/guide/render-function.html#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 v-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](https://vuejs.org/v2/guide/render-function.html#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.compileto live-compile a template string:

# Plugins

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

1. Add some global methods or properties. e.g. [**vue-custom-element**](https://github.com/karol-f/vue-custom-element)
2. Add one or more global assets: directives/filters/transitions etc. e.g. [**vue-touch**](https://github.com/vuejs/vue-touch)
3. Add some component options by global mixin. e.g. [**vue-router**](https://github.com/vuejs/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**](https://github.com/vuejs/vue-router)

## [Using a Plugin](https://vuejs.org/v2/guide/plugins.html#Using-a-Plugin)

Use plugins by calling the Vue.use() global method. This has to be done before you start your app by calling new Vue():

// calls `MyPlugin.install(Vue)`

Vue.use(MyPlugin)

new Vue({

//... options

})

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**](https://github.com/vuejs/awesome-vue#components--libraries) for a huge collection of community-contributed plugins and libraries.

## [Writing a Plugin](https://vuejs.org/v2/guide/plugins.html#Writing-a-Plugin)

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

MyPlugin.install = function (Vue, options) {

// 1. add global method or property

Vue.myGlobalMethod = function () {

// some logic ...

}

// 2. add a global asset

Vue.directive('my-directive', {

bind (el, binding, vnode, oldVnode) {

// some logic ...

}

...

})

// 3. inject some component options

Vue.mixin({

created: function () {

// some logic ...

}

...

})

// 4. add an instance method

Vue.prototype.$myMethod = function (methodOptions) {

// some logic ...

}

}

# Filters

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:



John

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.

### Tooling

# Single File Components

## [Introduction](https://vuejs.org/v2/guide/single-file-components.html#Introduction)

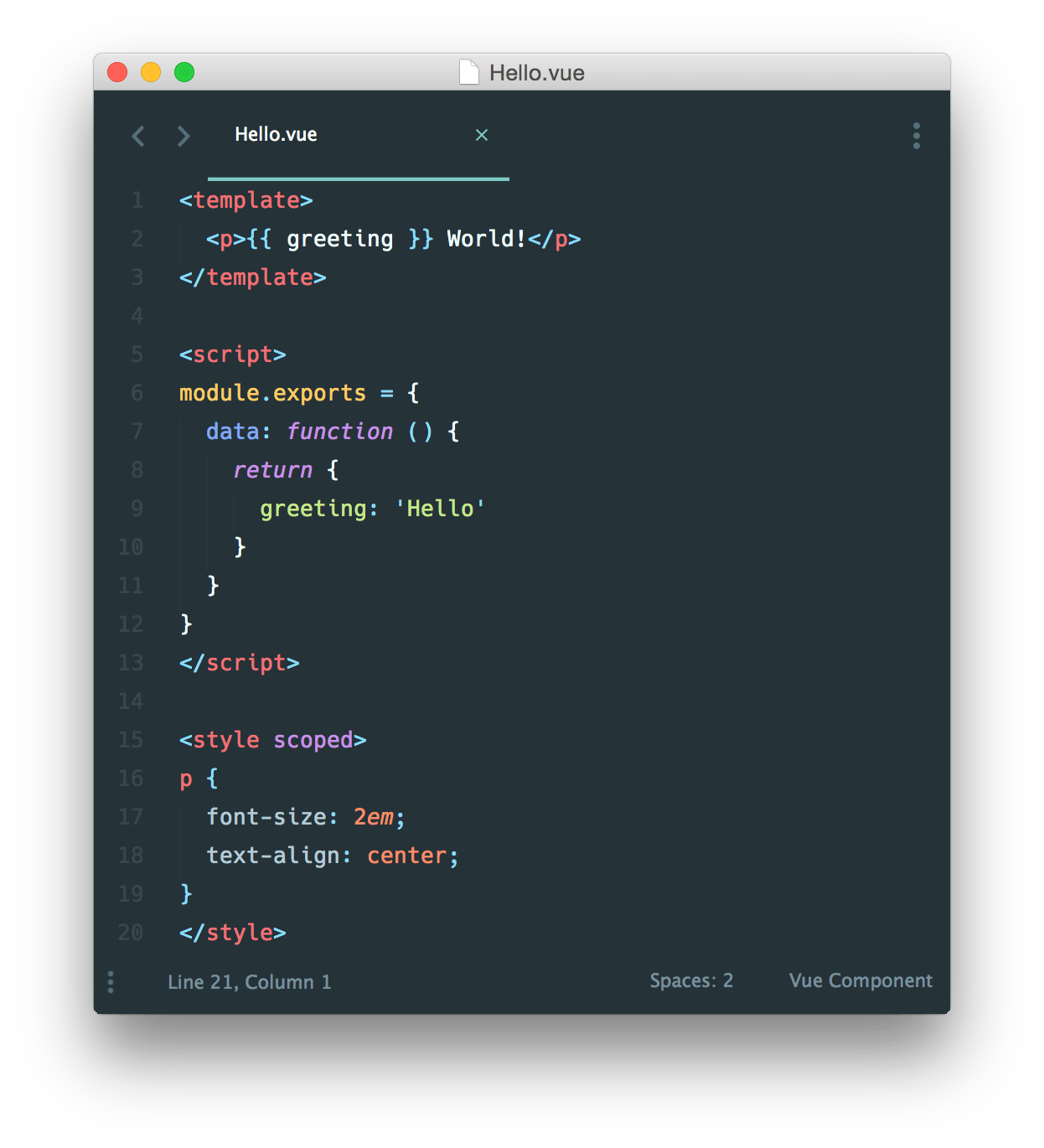
In many Vue projects, global components will be defined using Vue.component, followed by new Vue({ el: '#container' }) to target a container element in the body of every page.

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:

**[](https://gist.github.com/chrisvfritz/e2b6a6110e0829d78fa4aedf7cf6b235)**

Now we get:

* [**Complete syntax highlighting**](https://github.com/vuejs/awesome-vue#source-code-editing)
* [**CommonJS modules**](https://webpack.js.org/concepts/modules/#what-is-a-webpack-module)
* [**Component-scoped CSS**](https://vue-loader.vuejs.org/en/features/scoped-css.html)

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

**[](https://gist.github.com/chrisvfritz/1c9f2daea9bc078dcb47e9a82e5f7587)**

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?](https://vuejs.org/v2/guide/single-file-components.html#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](https://vuejs.org/v2/guide/single-file-components.html#Getting-Started)

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

If you want to dive right in and start playing with single-file components, check out [**this simple todo app**](https://codesandbox.io/s/o29j95wx9) on CodeSandbox.

### [For Users New to Module Build Systems in JavaScript](https://vuejs.org/v2/guide/single-file-components.html#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**](https://docs.npmjs.com/getting-started/what-is-npm) through section 10: Uninstalling global packages.
* **Modern JavaScript with ES2015/16**: Read through Babel’s [**Learn ES2015 guide**](https://babeljs.io/docs/learn-es2015/). 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 [**Vue CLI 3**](https://cli.vuejs.org/). Follow the instructions and you should have a Vue project with .vue components, ES2015, Webpack and hot-reloading in no time!

### [For Advanced Users](https://vuejs.org/v2/guide/single-file-components.html#For-Advanced-Users)

The CLI takes care of most of the tooling configurations for you, but also allows fine-grained customization through its own [**config options**](https://cli.vuejs.org/config/).

In case you prefer setting up your own build setup from scratch, you will need to manually configure webpack with [**vue-loader**](https://vue-loader.vuejs.org/). To learn more about webpack itself, check out [**their official docs**](https://webpack.js.org/configuration/) and [**Webpack Academy**](https://webpack.academy/p/the-core-concepts).

# Unit Testing

[**Vue CLI**](https://cli.vuejs.org/)**has built-in options for unit testing with**[**Jest**](https://github.com/facebook/jest)**or**[**Mocha**](https://mochajs.org/)**that works out of the box. We also have the official**[**Vue Test Utils**](https://vue-test-utils.vuejs.org/)**which provides more detailed guidance for custom setups.**

## [Simple Assertions](https://vuejs.org/v2/guide/unit-testing.html#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 (here we are using Jasmine/Jest style expect assertions just as an example):

// 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](https://vuejs.org/v2/guide/unit-testing.html#Writing-Testable-Components)

A component’s render output is primarily determined by the props it receives. 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](https://vuejs.org/v2/guide/unit-testing.html#Asserting-Asynchronous-Updates)

Since Vue [**performs DOM updates asynchronously**](https://vuejs.org/v2/guide/reactivity.html#Async-Update-Queue), 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()

})

})

For more in-depth information on unit testing in Vue, check out [**Vue Test Utils**](https://vue-test-utils.vuejs.org/) and our cookbook entry about [**unit testing vue components**](https://vuejs.org/v2/cookbook/unit-testing-vue-components.html).

# TypeScript Support

[**Vue CLI**](https://cli.vuejs.org/)**provides built-in TypeScript tooling support. In our next major version of Vue (3.x), we are also planning to considerably improve our TypeScript support with built-in class based components API and TSX support.**

## [Official Declaration in NPM Packages](https://vuejs.org/v2/guide/typescript.html#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**](https://github.com/vuejs/vue/tree/dev/types) for [**TypeScript**](https://www.typescriptlang.org/) - not only in Vue core, but also for [**vue-router**](https://github.com/vuejs/vue-router/tree/dev/types) and [**vuex**](https://github.com/vuejs/vuex/tree/dev/types) as well.

Since these are [**published on NPM**](https://cdn.jsdelivr.net/npm/vue/types/), 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](https://vuejs.org/v2/guide/typescript.html#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**](https://www.typescriptlang.org/docs/handbook/compiler-options.html) for more details.

## [Development Tooling](https://vuejs.org/v2/guide/typescript.html#Development-Tooling)

### [Project Creation](https://vuejs.org/v2/guide/typescript.html#Project-Creation)

[**Vue CLI 3**](https://github.com/vuejs/vue-cli) 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](https://vuejs.org/v2/guide/typescript.html#Editor-Support)

For developing Vue applications with TypeScript, we strongly recommend using [**Visual Studio Code**](https://code.visualstudio.com/), which provides great out-of-the-box support for TypeScript. If you are using [**single-file components**](https://vuejs.org/v2/guide/single-file-components.html)(SFCs), get the awesome [**Vetur extension**](https://github.com/vuejs/vetur), which provides TypeScript inference inside SFCs and many other great features.

[**WebStorm**](https://www.jetbrains.com/webstorm/) also provides out-of-the-box support for both TypeScript and Vue.

## [Basic Usage](https://vuejs.org/v2/guide/typescript.html#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](https://vuejs.org/v2/guide/typescript.html#Class-Style-Vue-Components)

If you prefer a class-based API when declaring components, you can use the officially maintained [**vue-class-component**](https://github.com/vuejs/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](https://vuejs.org/v2/guide/typescript.html#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**](https://www.typescriptlang.org/docs/handbook/declaration-merging.html#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](https://vuejs.org/v2/guide/typescript.html#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.

# Production Deployment

**Most of the tips below are enabled by default if you are using**[**Vue CLI**](https://cli.vuejs.org/)**. This section is only relevant if you are using a custom build setup.**

## [Turn on Production Mode](https://vuejs.org/v2/guide/deployment.html#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](https://vuejs.org/v2/guide/deployment.html#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**](https://vuejs.org/v2/guide/installation.html#Direct-lt-script-gt-Include).

### [With Build Tools](https://vuejs.org/v2/guide/deployment.html#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**](https://webpack.js.org/plugins/define-plugin/):

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**](https://github.com/hughsk/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**](https://github.com/hughsk/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**](https://github.com/hughsk/envify) with Grunt and [**grunt-browserify**](https://github.com/jmreidy/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**](https://github.com/rollup/rollup-plugin-replace):

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

rollup({

// ...

plugins: [

replace({

'process.env.NODE\_ENV': JSON.stringify( 'production' )

})

]

}).then(...)

## [Pre-Compiling Templates](https://vuejs.org/v2/guide/deployment.html#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**](https://vuejs.org/v2/guide/single-file-components.html) - 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**](https://github.com/ktsn/vue-template-loader), which also transforms the template files into JavaScript render functions during the build step.

## [Extracting Component CSS](https://vuejs.org/v2/guide/deployment.html#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**](https://vue-loader.vuejs.org/en/configurations/extract-css.html) (the vue-cli webpack template has this pre-configured)
* [**Browserify + vueify**](https://github.com/vuejs/vueify#css-extraction)
* [**Rollup + rollup-plugin-vue**](https://vuejs.github.io/rollup-plugin-vue/#/en/2.3/?id=custom-handler)

## [Tracking Runtime Errors](https://vuejs.org/v2/guide/deployment.html#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**](https://sentry.io/), which provides [**an official integration**](https://sentry.io/for/vue/) for Vue.

### Scaling Up

# Routing

## [Official Router](https://vuejs.org/v2/guide/routing.html#Official-Router)

For most Single Page Applications, it’s recommended to use the officially-supported [**vue-router library**](https://github.com/vuejs/vue-router). For more details, see vue-router’s [**documentation**](https://router.vuejs.org/).

## [Simple Routing From Scratch](https://vuejs.org/v2/guide/routing.html#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**](https://github.com/chrisvfritz/vue-2.0-simple-routing-example).

## [Integrating 3rd-Party Routers](https://vuejs.org/v2/guide/routing.html#Integrating-3rd-Party-Routers)

If there’s a 3rd-party router you prefer to use, such as [**Page.js**](https://github.com/visionmedia/page.js) or [**Director**](https://github.com/flatiron/director), integration is [**similarly easy**](https://github.com/chrisvfritz/vue-2.0-simple-routing-example/compare/master...pagejs). Here’s a [**complete example**](https://github.com/chrisvfritz/vue-2.0-simple-routing-example/tree/pagejs) using Page.js.

# State Management

## [Official Flux-Like Implementation](https://vuejs.org/v2/guide/state-management.html#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**](https://github.com/vuejs/vuex): our own Elm-inspired state management library. It even integrates into [**vue-devtools**](https://github.com/vuejs/vue-devtools), providing zero-setup access to [**time travel debugging**](https://raw.githubusercontent.com/vuejs/vue-devtools/master/media/demo.gif).

[**Watch a video explanation on Vue Mastery**](https://www.vuemastery.com/courses/mastering-vuex/intro-to-vuex/)

### [Information for React Developers](https://vuejs.org/v2/guide/state-management.html#Information-for-React-Developers)

If you’re coming from React, you may be wondering how vuex compares to [**redux**](https://github.com/reactjs/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**](https://yarnpkg.com/en/packages?q=redux%20vue&p=1). 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](https://vuejs.org/v2/guide/state-management.html#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 they are triggered. Now when something goes wrong, we’ll also have a log of what happened leading up to the bug.

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

var vmA = new Vue({

data: {

privateState: {},

sharedState: store.state

}

})

var vmB = new Vue({

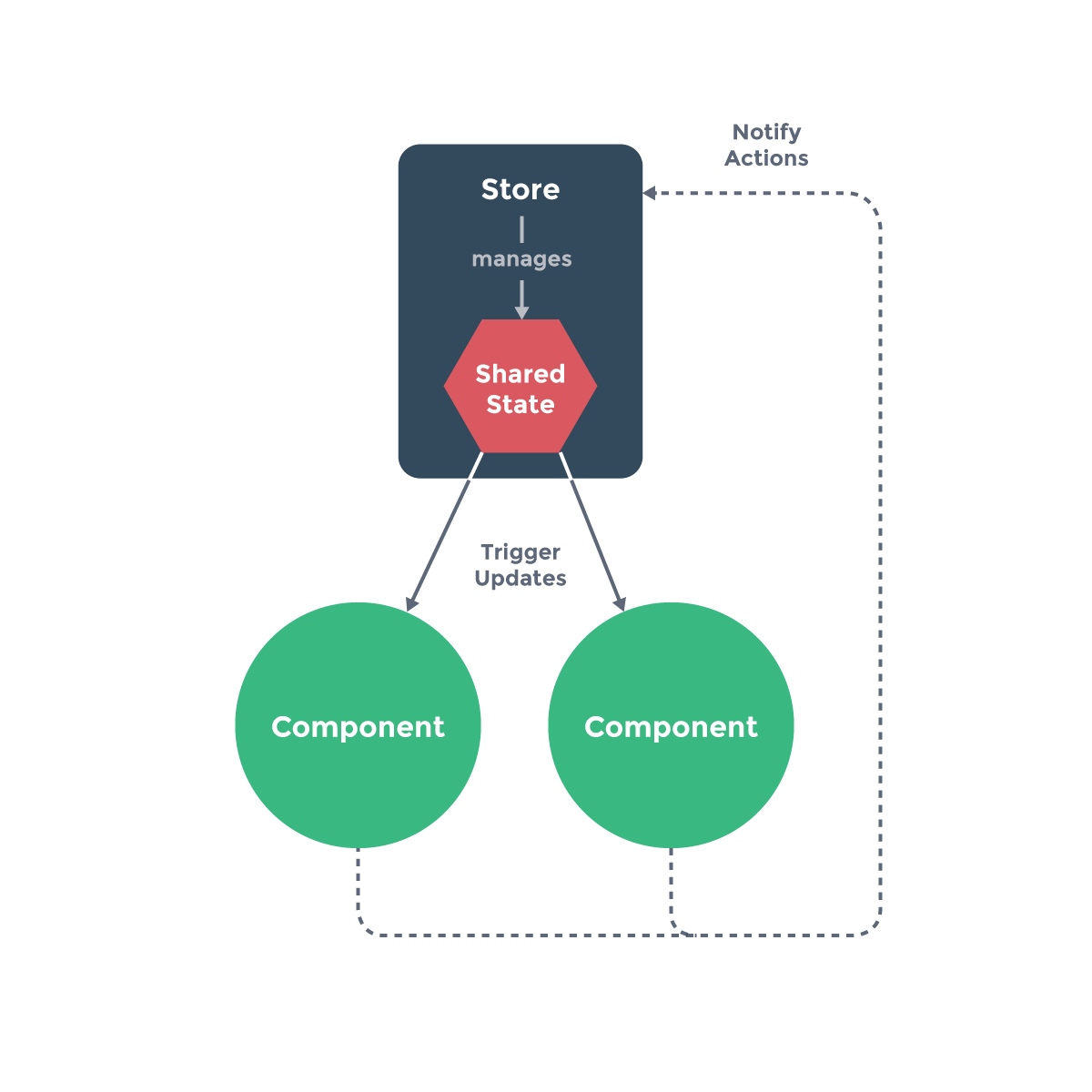
data: {

privateState: {},

sharedState: store.state

}

})



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

As we continue developing the convention where components are never allowed to directly mutate state that belongs to a store, but should instead dispatch events that notify the store to perform actions, we eventually arrive at the [**Flux**](https://facebook.github.io/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**](https://github.com/vuejs/vuex), so if you’ve read this far it’s probably time to try it out!

# Server-Side Rendering

## [The Complete SSR Guide](https://vuejs.org/v2/guide/ssr.html#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**](https://ssr.vuejs.org/).

## [Nuxt.js](https://vuejs.org/v2/guide/ssr.html#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**](https://nuxtjs.org/). 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.

## [Quasar Framework SSR + PWA](https://vuejs.org/v2/guide/ssr.html#Quasar-Framework-SSR-PWA)

[**Quasar Framework**](https://quasar-framework.org/) will generate an SSR app (with optional PWA handoff) that leverages its best-in-class build system, sensible configuration and developer extensibility to make designing and building your idea a breeze. With over one hundred specific “Material Design 2.0”-compliant components, you can decide which ones to execute on the server, which are available in the browser - and even manage the <meta> tags of your site. Quasar is a node.js and webpack based development environment that supercharges and streamlines rapid development of SPA, PWA, SSR, Electron and Cordova apps - all from one codebase.

### Internals

# Reactivity in Depth

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.

[**Watch a video explanation on Vue Mastery**](https://www.vuemastery.com/courses/advanced-components/build-a-reactivity-system)

## [How Changes Are Tracked](https://vuejs.org/v2/guide/reactivity.html#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**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Defining_getters_and_setters) using [**Object.defineProperty**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/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**](https://github.com/vuejs/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.



## [Change Detection Caveats](https://vuejs.org/v2/guide/reactivity.html#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, propertyName, 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**](https://vuejs.org/v2/guide/list.html#Caveats).

## [Declaring Reactive Properties](https://vuejs.org/v2/guide/reactivity.html#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](https://vuejs.org/v2/guide/reactivity.html#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, MutationObserver, and setImmediate 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'

})

}

}

})

Since $nextTick() returns a promise, you can achieve the same as the above using the new [**ES2016 async/await**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) syntax:

methods: {

updateMessage: async function () {

this.message = 'updated'

console.log(this.$el.textContent) // => 'not updated'

await this.$nextTick()

console.log(this.$el.textContent) // => 'updated'

}

}

### Migrating

# Migration from Vue 1.x

## [FAQ](https://vuejs.org/v2/guide/migration.html#FAQ)

**Woah - this is a super long page! Does that mean 2.0 is completely different, I’ll have to learn the basics all over again, and migrating will be practically impossible?**

I’m glad you asked! The answer is no. About 90% of the API is the same and the core concepts haven’t changed. It’s long because we like to offer very detailed explanations and include a lot of examples. Rest assured, **this is not something you have to read from top to bottom!**

**Where should I start in a migration?**

1. Start by running the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on a current project. We’ve carefully minified and compressed a senior Vue dev into a simple command line interface. Whenever they recognize an obsolete feature, they’ll let you know, offer suggestions, and provide links to more info.
2. After that, browse through the table of contents for this page in the sidebar. If you see a topic you may be affected by, but the migration helper didn’t catch, check it out.
3. If you have any tests, run them and see what still fails. If you don’t have tests, just open the app in your browser and keep an eye out for warnings or errors as you navigate around.
4. By now, your app should be fully migrated. If you’re still hungry for more though, you can read the rest of this page - or dive in to the new and improved guide from [**the beginning**](https://vuejs.org/v2/guide/index.html). Many parts will be skimmable, since you’re already familiar with the core concepts.

**How long will it take to migrate a Vue 1.x app to 2.0?**

It depends on a few factors:

* The size of your app (small to medium-sized apps will probably be less than a day)
* How many times you get distracted and start playing with a cool new feature. 😉  Not judging, it also happened to us while building 2.0!
* Which obsolete features you’re using. Most can be upgraded with find-and-replace, but others might take a few minutes. If you’re not currently following best practices, Vue 2.0 will also try harder to force you to. This is a good thing in the long run, but could also mean a significant (though possibly overdue) refactor.

**If I upgrade to Vue 2, will I also have to upgrade Vuex and Vue Router?**

Only Vue Router 2 is compatible with Vue 2, so yes, you’ll have to follow the [**migration path for Vue Router**](https://vuejs.org/v2/guide/migration-vue-router.html) as well. Fortunately, most applications don’t have a lot of router code, so this likely won’t take more than an hour.

As for Vuex, even version 0.8 is compatible with Vue 2, so you’re not forced to upgrade. The only reason you may want to upgrade immediately is to take advantage of the new features in Vuex 2, such as modules and reduced boilerplate.

## [Templates](https://vuejs.org/v2/guide/migration.html#Templates)

### [Fragment Instances removed](https://vuejs.org/v2/guide/migration.html#Fragment-Instances-removed)

Every component must have exactly one root element. Fragment instances are no longer allowed. If you have a template like this:

<p>foo</p>

<p>bar</p>

It’s recommended to wrap the entire contents in a new element, like this:

<div>

<p>foo</p>

<p>bar</p>

</div>

#### Upgrade Path

Run your end-to-end test suite or app after upgrading and look for **console warnings** about multiple root elements in a template.

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

### [beforeCompile removed](https://vuejs.org/v2/guide/migration.html#beforeCompile-removed)

Use the created hook instead.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find all examples of this hook.

### [compiled replaced](https://vuejs.org/v2/guide/migration.html#compiled-replaced)

Use the new mounted hook instead.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find all examples of this hook.

### [attached removed](https://vuejs.org/v2/guide/migration.html#attached-removed)

Use a custom in-DOM check in other hooks. For example, to replace:

attached: function () {

doSomething()

}

You could use:

mounted: function () {

this.$nextTick(function () {

doSomething()

})

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find all examples of this hook.

### [detached removed](https://vuejs.org/v2/guide/migration.html#detached-removed)

Use a custom in-DOM check in other hooks. For example, to replace:

detached: function () {

doSomething()

}

You could use:

destroyed: function () {

this.$nextTick(function () {

doSomething()

})

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find all examples of this hook.

### [init renamed](https://vuejs.org/v2/guide/migration.html#init-renamed)

Use the new beforeCreate hook instead, which is essentially the same thing. It was renamed for consistency with other lifecycle methods.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find all examples of this hook.

### [ready replaced](https://vuejs.org/v2/guide/migration.html#ready-replaced)

Use the new mounted hook instead. It should be noted though that with mounted, there’s no guarantee to be in-document. For that, also include Vue.nextTick/vm.$nextTick. For example:

mounted: function () {

this.$nextTick(function () {

// code that assumes this.$el is in-document

})

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find all examples of this hook.

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

### [v-for Argument Order for Arrays changed](https://vuejs.org/v2/guide/migration.html#v-for-Argument-Order-for-Arrays-changed)

When including an index, the argument order for arrays used to be (index, value). It is now (value, index) to be more consistent with JavaScript’s native array methods such as forEachand map.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the obsolete argument order. Note that if you name your index arguments something unusual like position or num, the helper will not flag them.

### [v-for Argument Order for Objects changed](https://vuejs.org/v2/guide/migration.html#v-for-Argument-Order-for-Objects-changed)

When including a property name/key, the argument order for objects used to be (name, value). It is now (value, name) to be more consistent with common object iterators such as lodash’s.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the obsolete argument order. Note that if you name your key arguments something like name or property, the helper will not flag them.

### [$index and $key removed](https://vuejs.org/v2/guide/migration.html#index-and-key-removed)

The implicitly assigned $index and $key variables have been removed in favor of explicitly defining them in v-for. This makes the code easier to read for developers less experienced with Vue and also results in much clearer behavior when dealing with nested loops.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of these removed variables. If you miss any, you should also see **console errors** such as: Uncaught ReferenceError: $index is not defined

### [track-by replaced](https://vuejs.org/v2/guide/migration.html#track-by-replaced)

track-by has been replaced with key, which works like any other attribute: without the v-bind:or : prefix, it is treated as a literal string. In most cases, you’d want to use a dynamic binding which expects a full expression instead of a key. For example, in place of:

<div v-for="item in items" track-by="id">

You would now write:

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

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of track-by.

### [v-for Range Values changed](https://vuejs.org/v2/guide/migration.html#v-for-Range-Values-changed)

Previously, v-for="number in 10" would have number starting at 0 and ending at 9. Now it starts at 1 and ends at 10.

#### Upgrade Path

Search your codebase for the regex /\w+ in \d+/. Wherever it appears in a v-for, check to see if you may be affected.

## [Props](https://vuejs.org/v2/guide/migration.html#Props)

### [coerce Prop Option removed](https://vuejs.org/v2/guide/migration.html#coerce-Prop-Option-removed)

If you want to coerce a prop, setup a local computed value based on it instead. For example, instead of:

props: {

username: {

type: String,

coerce: function (value) {

return value

.toLowerCase()

.replace(/\s+/, '-')

}

}

}

You could write:

props: {

username: String,

},

computed: {

normalizedUsername: function () {

return this.username

.toLowerCase()

.replace(/\s+/, '-')

}

}

There are a few advantages:

* You still have access to the original value of the prop.
* You are forced to be more explicit, by giving your coerced value a name that differentiates it from the value passed in the prop.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the coerce option.

### [twoWay Prop Option removed](https://vuejs.org/v2/guide/migration.html#twoWay-Prop-Option-removed)

Props are now always one-way down. To produce side effects in the parent scope, a component needs to explicitly emit an event instead of relying on implicit binding. For more information, see:

* [**Custom component events**](https://vuejs.org/v2/guide/components.html#Custom-Events)
* [**Custom input components**](https://vuejs.org/v2/guide/components.html#Form-Input-Components-using-Custom-Events) (using component events)
* [**Global state management**](https://vuejs.org/v2/guide/state-management.html)

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the twoWay option.

### [.once and .sync Modifiers on v-bind removed](https://vuejs.org/v2/guide/migration.html#once-and-sync-Modifiers-on-v-bind-removed)

Props are now always one-way down. To produce side effects in the parent scope, a component needs to explicitly emit an event instead of relying on implicit binding. For more information, see:

* [**Custom component events**](https://vuejs.org/v2/guide/components.html#Custom-Events)
* [**Custom input components**](https://vuejs.org/v2/guide/components.html#Form-Input-Components-using-Custom-Events) (using component events)
* [**Global state management**](https://vuejs.org/v2/guide/state-management.html)

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the .once and .syncmodifiers.

### [Prop Mutation deprecated](https://vuejs.org/v2/guide/migration.html#Prop-Mutation-deprecated)

Mutating a prop locally is now considered an anti-pattern, e.g. declaring a prop and then setting this.myProp = 'someOtherValue' in the component. Due to the new rendering mechanism, whenever the parent component re-renders, the child component’s local changes will be overwritten.

Most use cases of mutating a prop can be replaced by one of these options:

* a data property, with the prop used to set its default value
* a computed property

#### Upgrade Path

Run your end-to-end test suite or app after upgrading and look for **console warnings** about prop mutations.

### [Props on a Root Instance replaced](https://vuejs.org/v2/guide/migration.html#Props-on-a-Root-Instance-replaced)

On root Vue instances (i.e. instances created with new Vue({ ... })), you must use propsDatainstead of props.

#### Upgrade Path

Run your end-to-end test suite, if you have one. The **failed tests** should alert to you to the fact that props passed to root instances are no longer working.

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

### [cache: false deprecated](https://vuejs.org/v2/guide/migration.html#cache-false-deprecated)

Caching invalidation of computed properties will be removed in future major versions of Vue. Replace any uncached computed properties with methods, which will have the same result.

For example:

template: '<p>message: {{ timeMessage }}</p>',

computed: {

timeMessage: {

cache: false,

get: function () {

return Date.now() + this.message

}

}

}

Or with component methods:

template: '<p>message: {{ getTimeMessage() }}</p>',

methods: {

getTimeMessage: function () {

return Date.now() + this.message

}

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the cache: false option.

## [Built-In Directives](https://vuejs.org/v2/guide/migration.html#Built-In-Directives)

### [Truthiness/Falsiness with v-bind changed](https://vuejs.org/v2/guide/migration.html#Truthiness-Falsiness-with-v-bind-changed)

When used with v-bind, the only falsy values are now: null, undefined, and false. This means 0 and empty strings will render as truthy. So for example, v-bind:draggable="''" will render as draggable="true".

For enumerated attributes, in addition to the falsy values above, the string "false" will also render as attr="false".

Note that for other directives (e.g. v-if and v-show), JavaScript’s normal truthiness still applies.

#### Upgrade Path

Run your end-to-end test suite, if you have one. The **failed tests** should alert to you to any parts of your app that may be affected by this change.

### [Listening for Native Events on Components with v-on changed](https://vuejs.org/v2/guide/migration.html#Listening-for-Native-Events-on-Components-with-v-on-changed)

When used on a component, v-on now only listens to custom events $emitted by that component. To listen for a native DOM event on the root element, you can use the .native modifier. For example:

<my-component v-on:click.native="doSomething"></my-component>

#### Upgrade Path

Run your end-to-end test suite, if you have one. The **failed tests** should alert to you to any parts of your app that may be affected by this change.

### [debounce Param Attribute for v-model removed](https://vuejs.org/v2/guide/migration.html#debounce-Param-Attribute-for-v-model-removed)

Debouncing is used to limit how often we execute Ajax requests and other expensive operations. Vue’s debounce attribute parameter for v-model made this easy for very simple cases, but it actually debounced **state updates** rather than the expensive operations themselves. It’s a subtle difference, but it comes with limitations as an application grows.

These limitations become apparent when designing a search indicator, like this one for example:

 **✓ Done**

Using the debounce attribute, there’d be no way to detect the “Typing” state, because we lose access to the input’s real-time state. By decoupling the debounce function from Vue however, we’re able to debounce only the operation we want to limit, removing the limits on features we can develop:

<!--

By using the debounce function from lodash or another dedicated

utility library, we know the specific debounce implementation we

use will be best-in-class - and we can use it ANYWHERE. Not only

in our template.

-->

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

<div id="debounce-search-demo">

<input v-model="searchQuery" placeholder="Type something">

<strong>{{ searchIndicator }}</strong>

</div>

new Vue({

el: '#debounce-search-demo',

data: {

searchQuery: '',

searchQueryIsDirty: false,

isCalculating: false

},

computed: {

searchIndicator: function () {

if (this.isCalculating) {

return '⟳ Fetching new results'

} else if (this.searchQueryIsDirty) {

return '... Typing'

} else {

return '✓ Done'

}

}

},

watch: {

searchQuery: function () {

this.searchQueryIsDirty = true

this.expensiveOperation()

}

},

methods: {

// This is where the debounce actually belongs.

expensiveOperation: \_.debounce(function () {

this.isCalculating = true

setTimeout(function () {

this.isCalculating = false

this.searchQueryIsDirty = false

}.bind(this), 1000)

}, 500)

}

})

Another advantage of this approach is there will be times when debouncing isn’t quite the right wrapper function. For example, when hitting an API for search suggestions, waiting to offer suggestions until after the user has stopped typing for a period of time isn’t an ideal experience. What you probably want instead is a **throttling** function. Now since you’re already using a utility library like lodash, refactoring to use its throttle function instead takes only a few seconds.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the debounce attribute.

### [lazy or number Param Attributes for v-model replaced](https://vuejs.org/v2/guide/migration.html#lazy-or-number-Param-Attributes-for-v-model-replaced)

The lazy and number param attributes are now modifiers, to make it more clear what That means instead of:

<input v-model="name" lazy>

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

You would use:

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

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

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the these param attributes.

### [value Attribute with v-model removed](https://vuejs.org/v2/guide/migration.html#value-Attribute-with-v-model-removed)

v-model no longer cares about the initial value of an inline value attribute. For predictability, it will instead always treat the Vue instance data as the source of truth.

That means this element:

<input v-model="text" value="foo">

backed by this data:

data: {

text: 'bar'

}

will render with a value of “bar” instead of “foo”. The same goes for a <textarea> with existing content. Instead of:

<textarea v-model="text">

hello world

</textarea>

You should ensure your initial value for text is “hello world”.

#### Upgrade Path

Run your end-to-end test suite or app after upgrading and look for **console warnings** about inline value attributes with v-model.

### [v-model with v-for Iterated Primitive Values removed](https://vuejs.org/v2/guide/migration.html#v-model-with-v-for-Iterated-Primitive-Values-removed)

Cases like this no longer work:

<input v-for="str in strings" v-model="str">

The reason is this is the equivalent JavaScript that the <input> would compile to:

strings.map(function (str) {

return createElement('input', ...)

})

As you can see, v-model‘s two-way binding doesn’t make sense here. Setting str to another value in the iterator function will do nothing because it’s only a local variable in the function scope.

Instead, you should use an array of **objects** so that v-model can update the field on the object. For example:

<input v-for="obj in objects" v-model="obj.str">

#### Upgrade Path

Run your test suite, if you have one. The **failed tests** should alert to you to any parts of your app that may be affected by this change.

### [v-bind:style with Object Syntax and !important removed](https://vuejs.org/v2/guide/migration.html#v-bind-style-with-Object-Syntax-and-important-removed)

This will no longer work:

<p v-bind:style="{ color: myColor + ' !important' }">hello</p>

If you really need to override another !important, you must use the string syntax:

<p v-bind:style="'color: ' + myColor + ' !important'">hello</p>

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of style bindings with !important in objects.

### [v-el and v-ref replaced](https://vuejs.org/v2/guide/migration.html#v-el-and-v-ref-replaced)

For simplicity, v-el and v-ref have been merged into the ref attribute, accessible on a component instance via $refs. That means v-el:my-element would become ref="myElement"and v-ref:my-component would become ref="myComponent". When used on a normal element, the ref will be the DOM element, and when used on a component, the ref will be the component instance.

Since v-ref is no longer a directive, but a special attribute, it can also be dynamically defined. This is especially useful in combination with v-for. For example:

<p v-for="item in items" v-bind:ref="'item' + item.id"></p>

Previously, v-el/v-ref combined with v-for would produce an array of elements/components, because there was no way to give each item a unique name. You can still achieve this behavior by giving each item the same ref:

<p v-for="item in items" ref="items"></p>

Unlike in 1.x, these $refs are not reactive, because they’re registered/updated during the render process itself. Making them reactive would require duplicate renders for every change.

On the other hand, $refs are designed primarily for programmatic access in JavaScript - it is not recommended to rely on them in templates, because that would mean referring to state that does not belong to the instance itself. This would violate Vue’s data-driven view model.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of v-el and v-ref.

### [v-else with v-show removed](https://vuejs.org/v2/guide/migration.html#v-else-with-v-show-removed)

v-else no longer works with v-show. Use v-if with a negation expression instead. For example, instead of:

<p v-if="foo">Foo</p>

<p v-else v-show="bar">Not foo, but bar</p>

You can use:

<p v-if="foo">Foo</p>

<p v-if="!foo && bar">Not foo, but bar</p>

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the v-else with v-show.

## [Custom Directives simplified](https://vuejs.org/v2/guide/migration.html#Custom-Directives-simplified)

Directives have a greatly reduced scope of responsibility: they are now only used for applying low-level direct DOM manipulations. In most cases, you should prefer using components as the main code-reuse abstraction.

Some of the most notable differences include:

* Directives no longer have instances. This means there’s no more this inside directive hooks. Instead, they receive everything they might need as arguments. If you really must persist state across hooks, you can do so on el.
* Options such as acceptStatement, deep, priority, etc have all been removed. To replace twoWay directives, see [**this example**](https://vuejs.org/v2/guide/migration.html#Two-Way-Filters-replaced).
* Some of the current hooks have different behavior and there are also a couple new hooks.

Fortunately, since the new directives are much simpler, you can master them more easily. Read the new [**Custom Directives guide**](https://vuejs.org/v2/guide/custom-directive.html) to learn more.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of defined directives. The helper will flag all of them, as it's likely in most cases that you'll want to refactor to a component.

### [Directive .literal Modifier removed](https://vuejs.org/v2/guide/migration.html#Directive-literal-Modifier-removed)

The .literal modifier has been removed, as the same can be easily achieved by providing a string literal as the value.

For example, you can update:

<p v-my-directive.literal="foo bar baz"></p>

to:

<p v-my-directive="'foo bar baz'"></p>

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the `.literal` modifier on a directive.

## [Transitions](https://vuejs.org/v2/guide/migration.html#Transitions)

### [transition Attribute replaced](https://vuejs.org/v2/guide/migration.html#transition-Attribute-replaced)

Vue’s transition system has changed quite drastically and now uses <transition> and <transition-group> wrapper elements, rather than the transition attribute. It’s recommended to read the new [**Transitions guide**](https://vuejs.org/v2/guide/transitions.html) to learn more.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the transition attribute.

### [Vue.transition for Reusable Transitions replaced](https://vuejs.org/v2/guide/migration.html#Vue-transition-for-Reusable-Transitions-replaced)

With the new transition system, you can now [**use components for reusable transitions**](https://vuejs.org/v2/guide/transitions.html#Reusable-Transitions).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of Vue.transition.

### [Transition stagger Attribute removed](https://vuejs.org/v2/guide/migration.html#Transition-stagger-Attribute-removed)

If you need to stagger list transitions, you can control timing by setting and accessing a data-index(or similar attribute) on an element. See [**an example here**](https://vuejs.org/v2/guide/transitions.html#Staggering-List-Transitions).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the transition attribute. During your update, you can transition (pun very much intended) to the new staggering strategy as well.

## [Events](https://vuejs.org/v2/guide/migration.html#Events)

### [events option removed](https://vuejs.org/v2/guide/migration.html#events-option-removed)

The events option has been removed. Event handlers should now be registered in the createdhook instead. Check out the [**$dispatch and $broadcast migration guide**](https://vuejs.org/v2/guide/migration.html#dispatch-and-broadcast-replaced) for a detailed example.

### [Vue.directive('on').keyCodes replaced](https://vuejs.org/v2/guide/migration.html#Vue-directive-39-on-39-keyCodes-replaced)

The new, more concise way to configure keyCodes is through Vue.config.keyCodes. For example:

// enable v-on:keyup.f1

Vue.config.keyCodes.f1 = 112

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the the old keyCodeconfiguration syntax.

### [$dispatch and $broadcast replaced](https://vuejs.org/v2/guide/migration.html#dispatch-and-broadcast-replaced)

$dispatch and $broadcast have been removed in favor of more explicitly cross-component communication and more maintainable state management solutions, such as [**Vuex**](https://github.com/vuejs/vuex).

The problem is event flows that depend on a component’s tree structure can be hard to reason about and are very brittle when the tree becomes large. They don’t scale well and only set you up for pain later. $dispatch and $broadcast also do not solve communication between sibling components.

One of the most common uses for these methods is to communicate between a parent and its direct children. In these cases, you can actually [**listen to an $emit from a child with v-on**](https://vuejs.org/v2/guide/components.html#Form-Input-Components-using-Custom-Events). This allows you to keep the convenience of events with added explicitness.

However, when communicating between distant descendants/ancestors, $emit won’t help you. Instead, the simplest possible upgrade would be to use a centralized event hub. This has the added benefit of allowing you to communicate between components no matter where they are in the component tree - even between siblings! Because Vue instances implement an event emitter interface, you can actually use an empty Vue instance for this purpose.

For example, let’s say we have a todo app structured like this:

Todos

├─ NewTodoInput

└─ Todo

└─ DeleteTodoButton

We could manage communication between components with this single event hub:

// This is the event hub we'll use in every

// component to communicate between them.

var eventHub = new Vue()

Then in our components, we can use $emit, $on, $off to emit events, listen for events, and clean up event listeners, respectively:

// NewTodoInput

// ...

methods: {

addTodo: function () {

eventHub.$emit('add-todo', { text: this.newTodoText })

this.newTodoText = ''

}

}

// DeleteTodoButton

// ...

methods: {

deleteTodo: function (id) {

eventHub.$emit('delete-todo', id)

}

}

// Todos

// ...

created: function () {

eventHub.$on('add-todo', this.addTodo)

eventHub.$on('delete-todo', this.deleteTodo)

},

// It's good to clean up event listeners before

// a component is destroyed.

beforeDestroy: function () {

eventHub.$off('add-todo', this.addTodo)

eventHub.$off('delete-todo', this.deleteTodo)

},

methods: {

addTodo: function (newTodo) {

this.todos.push(newTodo)

},

deleteTodo: function (todoId) {

this.todos = this.todos.filter(function (todo) {

return todo.id !== todoId

})

}

}

This pattern can serve as a replacement for $dispatch and $broadcast in simple scenarios, but for more complex cases, it’s recommended to use a dedicated state management layer such as [**Vuex**](https://github.com/vuejs/vuex).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of $dispatch and $broadcast.

## [Filters](https://vuejs.org/v2/guide/migration.html#Filters)

### [Filters Outside Text Interpolations removed](https://vuejs.org/v2/guide/migration.html#Filters-Outside-Text-Interpolations-removed)

Filters can now only be used inside text interpolations ({{ }} tags). In the past we’ve found using filters within directives such as v-model, v-on, etc led to more complexity than convenience. For list filtering on v-for, it’s also better to move that logic into JavaScript as computed properties, so that it can be reused throughout your component.

In general, whenever something can be achieved in plain JavaScript, we want to avoid introducing a special syntax like filters to take care of the same concern. Here’s how you can replace Vue’s built-in directive filters:

#### Replacing the debounce Filter

Instead of:

<input v-on:keyup="doStuff | debounce 500">

methods: {

doStuff: function () {

// ...

}

}

Use [**lodash’s debounce**](https://lodash.com/docs/4.15.0#debounce) (or possibly [**throttle**](https://lodash.com/docs/4.15.0#throttle)) to directly limit calling the expensive method. You can achieve the same as above like this:

<input v-on:keyup="doStuff">

methods: {

doStuff: \_.debounce(function () {

// ...

}, 500)

}

For more on the advantages of this strategy, see [**the example here with v-model**](https://vuejs.org/v2/guide/migration.html#debounce-Param-Attribute-for-v-model-removed).

#### Replacing the limitBy Filter

Instead of:

<p v-for="item in items | limitBy 10">{{ item }}</p>

Use JavaScript’s built-in [**.slice method**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice#Examples) in a computed property:

<p v-for="item in filteredItems">{{ item }}</p>

computed: {

filteredItems: function () {

return this.items.slice(0, 10)

}

}

#### Replacing the filterBy Filter

Instead of:

<p v-for="user in users | filterBy searchQuery in 'name'">{{ user.name }}</p>

Use JavaScript’s built-in [**.filter method**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter#Examples) in a computed property:

<p v-for="user in filteredUsers">{{ user.name }}</p>

computed: {

filteredUsers: function () {

var self = this

return self.users.filter(function (user) {

return user.name.indexOf(self.searchQuery) !== -1

})

}

}

JavaScript’s native .filter can also manage much more complex filtering operations, because you have access to the full power of JavaScript within computed properties. For example, if you wanted to find all active users and case-insensitively match against both their name and email:

var self = this

self.users.filter(function (user) {

var searchRegex = new RegExp(self.searchQuery, 'i')

return user.isActive && (

searchRegex.test(user.name) ||

searchRegex.test(user.email)

)

})

#### Replacing the orderBy Filter

Instead of:

<p v-for="user in users | orderBy 'name'">{{ user.name }}</p>

Use [**lodash’s orderBy**](https://lodash.com/docs/4.15.0#orderBy) (or possibly [**sortBy**](https://lodash.com/docs/4.15.0#sortBy)) in a computed property:

<p v-for="user in orderedUsers">{{ user.name }}</p>

computed: {

orderedUsers: function () {

return \_.orderBy(this.users, 'name')

}

}

You can even order by multiple columns:

\_.orderBy(this.users, ['name', 'last\_login'], ['asc', 'desc'])

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of filters being used inside directives. If you miss any, you should also see **console errors**.

### [Filter Argument Syntax changed](https://vuejs.org/v2/guide/migration.html#Filter-Argument-Syntax-changed)

Filters’ syntax for arguments now better aligns with JavaScript function invocation. So instead of taking space-delimited arguments:

<p>{{ date | formatDate 'YY-MM-DD' timeZone }}</p>

We surround the arguments with parentheses and delimit the arguments with commas:

<p>{{ date | formatDate('YY-MM-DD', timeZone) }}</p>

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the old filter syntax. If you miss any, you should also see **console errors**.

### [Built-In Text Filters removed](https://vuejs.org/v2/guide/migration.html#Built-In-Text-Filters-removed)

Although filters within text interpolations are still allowed, all of the filters have been removed. Instead, it’s recommended to use more specialized libraries for solving problems in each domain (e.g. [**date-fns**](https://date-fns.org/) to format dates and [**accounting**](http://openexchangerates.github.io/accounting.js/) for currencies).

For each of Vue’s built-in text filters, we go through how you can replace them below. The example code could exist in custom helper functions, methods, or computed properties.

#### Replacing the json Filter

You actually don’t need to for debugging anymore, as Vue will nicely format output for you automatically, whether it’s a string, number, array, or plain object. If you want the exact same functionality as JavaScript’s JSON.stringify though, then you can use that in a method or computed property.

#### Replacing the capitalize Filter

text[0].toUpperCase() + text.slice(1)

#### Replacing the uppercase Filter

text.toUpperCase()

#### Replacing the lowercase Filter

text.toLowerCase()

#### Replacing the pluralize Filter

The [**pluralize**](https://www.npmjs.com/package/pluralize) package on NPM serves this purpose nicely, but if you only want to pluralize a specific word or want to have special output for cases like 0, then you can also easily define your own pluralize functions. For example:

function pluralizeKnife (count) {

if (count === 0) {

return 'no knives'

} else if (count === 1) {

return '1 knife'

} else {

return count + 'knives'

}

}

#### Replacing the currency Filter

For a very naive implementation, you could do something like this:

'$' + price.toFixed(2)

In many cases though, you’ll still run into strange behavior (e.g. 0.035.toFixed(2) rounds up to 0.04, but 0.045 rounds down to 0.04). To work around these issues, you can use the [**accounting**](http://openexchangerates.github.io/accounting.js/) library to more reliably format currencies.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the obsolete text filters. If you miss any, you should also see **console errors**.

### [Two-Way Filters replaced](https://vuejs.org/v2/guide/migration.html#Two-Way-Filters-replaced)

Some users have enjoyed using two-way filters with v-model to create interesting inputs with very little code. While seemingly simple however, two-way filters can also hide a great deal of complexity - and even encourage poor UX by delaying state updates. Instead, components wrapping an input are recommended as a more explicit and feature-rich way of creating custom inputs.

As an example, we’ll now walk the migration of a two-way currency filter:

It mostly works well, but the delayed state updates can cause strange behavior. For example, click on the Result tab and try entering 9.999 into one of those inputs. When the input loses focus, its value will update to $10.00. When looking at the calculated total however, you’ll see that 9.999 is what’s stored in our data. The version of reality that the user sees is out of sync!

To start transitioning towards a more robust solution using Vue 2.0, let’s first wrap this filter in a new <currency-input> component:

This allows us add behavior that a filter alone couldn’t encapsulate, such as selecting the content of an input on focus. Now the next step will be to extract the business logic from the filter. Below, we pull everything out into an external [**currencyValidator object**](https://gist.github.com/chrisvfritz/5f0a639590d6e648933416f90ba7ae4e):

This increased modularity not only makes it easier to migrate to Vue 2, but also allows currency parsing and formatting to be:

* unit tested in isolation from your Vue code
* used by other parts of your application, such as to validate the payload to an API endpoint

Having this validator extracted out, we’ve also more comfortably built it up into a more robust solution. The state quirks have been eliminated and it’s actually impossible for users to enter anything wrong, similar to what the browser’s native number input tries to do.

We’re still limited however, by filters and by Vue 1.0 in general, so let’s complete the upgrade to Vue 2.0:

You may notice that:

* Every aspect of our input is more explicit, using lifecycle hooks and DOM events in place of the hidden behavior of two-way filters.
* We can now use v-model directly on our custom inputs, which is not only more consistent with normal inputs, but also means our component is Vuex-friendly.
* Since we’re no longer using filter options that require a value to be returned, our currency work could actually be done asynchronously. That means if we had a lot of apps that had to work with currencies, we could easily refactor this logic into a shared microservice.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of filters used in directives like v-model. If you miss any, you should also see **console errors**.

## [Slots](https://vuejs.org/v2/guide/migration.html#Slots)

### [Duplicate Slots removed](https://vuejs.org/v2/guide/migration.html#Duplicate-Slots-removed)

It is no longer supported to have <slot>s with the same name in the same template. When a slot is rendered it is “used up” and cannot be rendered elsewhere in the same render tree. If you must render the same content in multiple places, pass that content as a prop.

#### Upgrade Path

Run your end-to-end test suite or app after upgrading and look for **console warnings** about duplicate slots v-model.

### [slot Attribute Styling removed](https://vuejs.org/v2/guide/migration.html#slot-Attribute-Styling-removed)

Content inserted via named <slot> no longer preserves the slot attribute. Use a wrapper element to style them, or for advanced use cases, modify the inserted content programmatically using [**render functions**](https://vuejs.org/v2/guide/render-function.html).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find CSS selectors targeting named slots (e.g. [slot="my-slot-name"]).

## [Special Attributes](https://vuejs.org/v2/guide/migration.html#Special-Attributes)

### [keep-alive Attribute replaced](https://vuejs.org/v2/guide/migration.html#keep-alive-Attribute-replaced)

keep-alive is no longer a special attribute, but rather a wrapper component, similar to <transition>. For example:

<keep-alive>

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

</keep-alive>

This makes it possible to use <keep-alive> on multiple conditional children:

<keep-alive>

<todo-list v-if="todos.length > 0"></todo-list>

<no-todos-gif v-else></no-todos-gif>

</keep-alive>

When <keep-alive> has multiple children, they should eventually evaluate to a single child. Any child other than the first one will be ignored.

When used together with <transition>, make sure to nest it inside:

<transition>

<keep-alive>

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

</keep-alive>

</transition>

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find keep-alive attributes.

## [Interpolation](https://vuejs.org/v2/guide/migration.html#Interpolation)

### [Interpolation within Attributes removed](https://vuejs.org/v2/guide/migration.html#Interpolation-within-Attributes-removed)

Interpolation within attributes is no longer valid. For example:

<button class="btn btn-{{ size }}"></button>

Should either be updated to use an inline expression:

<button v-bind:class="'btn btn-' + size"></button>

Or a data/computed property:

<button v-bind:class="buttonClasses"></button>

computed: {

buttonClasses: function () {

return 'btn btn-' + size

}

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of interpolation used within attributes.

### [HTML Interpolation removed](https://vuejs.org/v2/guide/migration.html#HTML-Interpolation-removed)

HTML interpolations ({{{ foo }}}) have been removed in favor of the [**v-html directive**](https://vuejs.org/v2/api/#v-html).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find HTML interpolations.

### [One-Time Bindings replaced](https://vuejs.org/v2/guide/migration.html#One-Time-Bindings-replaced)

One time bindings ({{\* foo }}) have been replaced by the new [**v-once directive**](https://vuejs.org/v2/api/#v-once).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find one-time bindings.

## [Reactivity](https://vuejs.org/v2/guide/migration.html#Reactivity)

### [vm.$watch changed](https://vuejs.org/v2/guide/migration.html#vm-watch-changed)

Watchers created via vm.$watch are now fired before the associated component rerenders. This gives you the chance to further update state before the component rerender, thus avoiding unnecessary updates. For example, you can watch a component prop and update the component’s own data when the prop changes.

If you were previously relying on vm.$watch to do something with the DOM after a component updates, you can instead do so in the updated lifecycle hook.

#### Upgrade Path

Run your end-to-end test suite, if you have one. The **failed tests** should alert to you to the fact that a watcher was relying on the old behavior.

### [vm.$set changed](https://vuejs.org/v2/guide/migration.html#vm-set-changed)

vm.$set is now an alias for [**Vue.set**](https://vuejs.org/v2/api/#Vue-set).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the obsolete usage.

### [vm.$delete changed](https://vuejs.org/v2/guide/migration.html#vm-delete-changed)

vm.$delete is now an alias for [**Vue.delete**](https://vuejs.org/v2/api/#Vue-delete).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the obsolete usage.

### [Array.prototype.$set removed](https://vuejs.org/v2/guide/migration.html#Array-prototype-set-removed)

Use Vue.set instead.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of .$set on an array. If you miss any, you should see **console errors** from the missing method.

### [Array.prototype.$remove removed](https://vuejs.org/v2/guide/migration.html#Array-prototype-remove-removed)

Use Array.prototype.splice instead. For example:

methods: {

removeTodo: function (todo) {

var index = this.todos.indexOf(todo)

this.todos.splice(index, 1)

}

}

Or better yet, pass removal methods an index:

methods: {

removeTodo: function (index) {

this.todos.splice(index, 1)

}

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of .$remove on an array. If you miss any, you should see **console errors** from the missing method.

### [Vue.set and Vue.delete on Vue instances removed](https://vuejs.org/v2/guide/migration.html#Vue-set-and-Vue-delete-on-Vue-instances-removed)

Vue.set and Vue.delete can no longer work on Vue instances. It is now mandatory to properly declare all top-level reactive properties in the data option. If you’d like to delete properties on a Vue instance or its $data, set it to null.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of Vue.set or Vue.deleteon a Vue instance. If you miss any, they'll trigger **console warnings**.

### [Replacing vm.$data removed](https://vuejs.org/v2/guide/migration.html#Replacing-vm-data-removed)

It is now prohibited to replace a component instance’s root $data. This prevents some edge cases in the reactivity system and makes the component state more predictable (especially with type-checking systems).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of overwriting vm.$data. If you miss any, **console warnings** will be emitted.

### [vm.$get removed](https://vuejs.org/v2/guide/migration.html#vm-get-removed)

Instead, retrieve reactive data directly.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of vm.$get. If you miss any, you'll see **console errors**.

## [DOM-Focused Instance Methods](https://vuejs.org/v2/guide/migration.html#DOM-Focused-Instance-Methods)

### [vm.$appendTo removed](https://vuejs.org/v2/guide/migration.html#vm-appendTo-removed)

Use the native DOM API:

myElement.appendChild(vm.$el)

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of vm.$appendTo. If you miss any, you'll see **console errors**.

### [vm.$before removed](https://vuejs.org/v2/guide/migration.html#vm-before-removed)

Use the native DOM API:

myElement.parentNode.insertBefore(vm.$el, myElement)

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of vm.$before. If you miss any, you'll see **console errors**.

### [vm.$after removed](https://vuejs.org/v2/guide/migration.html#vm-after-removed)

Use the native DOM API:

myElement.parentNode.insertBefore(vm.$el, myElement.nextSibling)

Or if myElement is the last child:

myElement.parentNode.appendChild(vm.$el)

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of vm.$after. If you miss any, you'll see **console errors**.

### [vm.$remove removed](https://vuejs.org/v2/guide/migration.html#vm-remove-removed)

Use the native DOM API:

vm.$el.remove()

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of vm.$remove. If you miss any, you'll see **console errors**.

## [Meta Instance Methods](https://vuejs.org/v2/guide/migration.html#Meta-Instance-Methods)

### [vm.$eval removed](https://vuejs.org/v2/guide/migration.html#vm-eval-removed)

No real use. If you do happen to rely on this feature somehow and aren’t sure how to work around it, post on [**the forum**](https://forum.vuejs.org/) for ideas.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of vm.$eval. If you miss any, you'll see **console errors**.

### [vm.$interpolate removed](https://vuejs.org/v2/guide/migration.html#vm-interpolate-removed)

No real use. If you do happen to rely on this feature somehow and aren’t sure how to work around it, post on [**the forum**](https://forum.vuejs.org/) for ideas.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of vm.$interpolate. If you miss any, you'll see **console errors**.

### [vm.$log removed](https://vuejs.org/v2/guide/migration.html#vm-log-removed)

Use the [**Vue Devtools**](https://github.com/vuejs/vue-devtools) for the optimal debugging experience.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of vm.$log. If you miss any, you'll see **console errors**.

## [Instance DOM Options](https://vuejs.org/v2/guide/migration.html#Instance-DOM-Options)

### [replace: false removed](https://vuejs.org/v2/guide/migration.html#replace-false-removed)

Components now always replace the element they’re bound to. To simulate the behavior of replace: false, you can wrap your root component with an element similar to the one you’re replacing. For example:

new Vue({

el: '#app',

template: '<div id="app"> ... </div>'

})

Or with a render function:

new Vue({

el: '#app',

render: function (h) {

h('div', {

attrs: {

id: 'app',

}

}, /\* ... \*/)

}

})

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of replace: false.

## [Global Config](https://vuejs.org/v2/guide/migration.html#Global-Config)

### [Vue.config.debug removed](https://vuejs.org/v2/guide/migration.html#Vue-config-debug-removed)

No longer necessary, since warnings come with stack traces by default now.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of Vue.config.debug.

### [Vue.config.async removed](https://vuejs.org/v2/guide/migration.html#Vue-config-async-removed)

Async is now required for rendering performance.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of Vue.config.async.

### [Vue.config.delimiters replaced](https://vuejs.org/v2/guide/migration.html#Vue-config-delimiters-replaced)

This has been reworked as a [**component-level option**](https://vuejs.org/v2/api/#delimiters). This allows you to use alternative delimiters within your app without breaking 3rd-party components.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of Vue.config.delimiters.

### [Vue.config.unsafeDelimiters removed](https://vuejs.org/v2/guide/migration.html#Vue-config-unsafeDelimiters-removed)

HTML interpolation has been [**removed in favor of v-html**](https://vuejs.org/v2/guide/migration.html#HTML-Interpolation-removed).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of Vue.config.unsafeDelimiters. After this, the helper will also find instances of HTML interpolation so that you can replace them with `v-html`.

## [Global API](https://vuejs.org/v2/guide/migration.html#Global-API)

### [Vue.extend with el removed](https://vuejs.org/v2/guide/migration.html#Vue-extend-with-el-removed)

The el option can no longer be used in Vue.extend. It’s only valid as an instance creation option.

#### Upgrade Path

Run your end-to-end test suite or app after upgrading and look for **console warnings** about the el option with Vue.extend.

### [Vue.elementDirective removed](https://vuejs.org/v2/guide/migration.html#Vue-elementDirective-removed)

Use components instead.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of Vue.elementDirective.

### [Vue.partial removed](https://vuejs.org/v2/guide/migration.html#Vue-partial-removed)

Partials have been removed in favor of more explicit data flow between components, using props. Unless you’re using a partial in a performance-critical area, the recommendation is to use a [**normal component**](https://vuejs.org/v2/guide/components.html) instead. If you were dynamically binding the name of a partial, you can use a [**dynamic component**](https://vuejs.org/v2/guide/components.html#Dynamic-Components).

If you happen to be using partials in a performance-critical part of your app, then you should upgrade to [**functional components**](https://vuejs.org/v2/guide/render-function.html#Functional-Components). They must be in a plain JS/JSX file (rather than in a .vue file) and are stateless and instanceless, like partials. This makes rendering extremely fast.

A benefit of functional components over partials is that they can be much more dynamic, because they grant you access to the full power of JavaScript. There is a cost to this power however. If you’ve never used a component framework with render functions before, they may take a bit longer to learn.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of Vue.partial.

# Migration from Vue Router 0.7.x

**Only Vue Router 2 is compatible with Vue 2, so if you’re updating Vue, you’ll have to update Vue Router as well. That’s why we’ve included details on the migration path here in the main docs. For a complete guide on using the new Vue Router, see the**[**Vue Router docs**](https://router.vuejs.org/en/)**.**

## [Router Initialization](https://vuejs.org/v2/guide/migration-vue-router.html#Router-Initialization)

### [router.start replaced](https://vuejs.org/v2/guide/migration-vue-router.html#router-start-replaced)

There is no longer a special API to initialize an app with Vue Router. That means instead of:

router.start({

template: '<router-view></router-view>'

}, '#app')

You pass a router property to a Vue instance:

new Vue({

el: '#app',

router: router,

template: '<router-view></router-view>'

})

Or, if you’re using the runtime-only build of Vue:

new Vue({

el: '#app',

router: router,

render: h => h('router-view')

})

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of router.start being called.

## [Route Definitions](https://vuejs.org/v2/guide/migration-vue-router.html#Route-Definitions)

### [router.map replaced](https://vuejs.org/v2/guide/migration-vue-router.html#router-map-replaced)

Routes are now defined as an array on a [**routes option**](https://router.vuejs.org/en/essentials/getting-started.html#javascript) at router instantiation. So these routes for example:

router.map({

'/foo': {

component: Foo

},

'/bar': {

component: Bar

}

})

Will instead be defined with:

var router = new VueRouter({

routes: [

{ path: '/foo', component: Foo },

{ path: '/bar', component: Bar }

]

})

The array syntax allows more predictable route matching, since iterating over an object is not guaranteed to use the same property order across browsers.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of router.map being called.

### [router.on removed](https://vuejs.org/v2/guide/migration-vue-router.html#router-on-removed)

If you need to programmatically generate routes when starting up your app, you can do so by dynamically pushing definitions to a routes array. For example:

// Normal base routes

var routes = [

// ...

]

// Dynamically generated routes

marketingPages.forEach(function (page) {

routes.push({

path: '/marketing/' + page.slug

component: {

extends: MarketingComponent

data: function () {

return { page: page }

}

}

})

})

var router = new Router({

routes: routes

})

If you need to add new routes after the router has been instantiated, you can replace the router’s matcher with a new one that includes the route you’d like to add:

router.match = createMatcher(

[{

path: '/my/new/path',

component: MyComponent

}].concat(router.options.routes)

)

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of router.on being called.

### [router.beforeEach changed](https://vuejs.org/v2/guide/migration-vue-router.html#router-beforeEach-changed)

router.beforeEach now works asynchronously and takes a next function as its third argument.

router.beforeEach(function (transition) {

if (transition.to.path === '/forbidden') {

transition.abort()

} else {

transition.next()

}

})

router.beforeEach(function (to, from, next) {

if (to.path === '/forbidden') {

next(false)

} else {

next()

}

})

### [subRoutes renamed](https://vuejs.org/v2/guide/migration-vue-router.html#subRoutes-renamed)

[**Renamed to children**](https://router.vuejs.org/en/essentials/nested-routes.html) for consistency within Vue and with other routing libraries.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the subRoutes option.

### [router.redirect replaced](https://vuejs.org/v2/guide/migration-vue-router.html#router-redirect-replaced)

This is now an [**option on route definitions**](https://router.vuejs.org/en/essentials/redirect-and-alias.html). So for example, you will update:

router.redirect({

'/tos': '/terms-of-service'

})

to a definition like below in your routes configuration:

{

path: '/tos',

redirect: '/terms-of-service'

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of router.redirect being called.

### [router.alias replaced](https://vuejs.org/v2/guide/migration-vue-router.html#router-alias-replaced)

This is now an [**option on the definition for the route**](https://router.vuejs.org/en/essentials/redirect-and-alias.html) you’d like to alias to. So for example, you will update:

router.alias({

'/manage': '/admin'

})

to a definition like below in your routes configuration:

{

path: '/admin',

component: AdminPanel,

alias: '/manage'

}

If you need multiple aliases, you can also use an array syntax:

alias: ['/manage', '/administer', '/administrate']

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of router.alias being called.

### [Arbitrary Route Properties replaced](https://vuejs.org/v2/guide/migration-vue-router.html#Arbitrary-Route-Properties-replaced)

Arbitrary route properties must now be scoped under the new meta property, to avoid conflicts with future features. So for example, if you had defined:

'/admin': {

component: AdminPanel,

requiresAuth: true

}

Then you would now update it to:

{

path: '/admin',

component: AdminPanel,

meta: {

requiresAuth: true

}

}

Then when later accessing this property on a route, you will still go through meta. For example:

if (route.meta.requiresAuth) {

// ...

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of arbitrary route properties not scoped under meta.

### [[] Syntax for Arrays in Queries removed](https://vuejs.org/v2/guide/migration-vue-router.html#Syntax-for-Arrays-in-Queries-removed)

When passing arrays to query parameters the QueryString syntax is no longer /foo?users[]=Tom&users[]=Jerry, instead, the new syntax is /foo?users=Tom&users=Jerry. Internally, $route.query.users will still be an Array, but if there’s only one parameter in the query: /foo?users=Tom, when directly accessing this route, there’s no way for the router to know if we were expecting users to be an Array. Because of this, consider adding a computed property and replacing every reference of $route.query.users with it:

export default {

// ...

computed: {

// users will always be an array

users () {

const users = this.$route.query.users

return Array.isArray(users) ? users : [users]

}

}

}

## [Route Matching](https://vuejs.org/v2/guide/migration-vue-router.html#Route-Matching)

Route matching now uses [**path-to-regexp**](https://github.com/pillarjs/path-to-regexp) under the hood, making it much more flexible than previously.

### [One or More Named Parameters changed](https://vuejs.org/v2/guide/migration-vue-router.html#One-or-More-Named-Parameters-changed)

The syntax has changed slightly, so /category/\*tags for example, should be updated to /category/:tags+.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the obsolete route syntax.

## [Links](https://vuejs.org/v2/guide/migration-vue-router.html#Links)

### [v-link replaced](https://vuejs.org/v2/guide/migration-vue-router.html#v-link-replaced)

The v-link directive has been replaced with a new [**<router-link> component**](https://router.vuejs.org/en/api/router-link.html), as this sort of job is now solely the responsibility of components in Vue 2. That means whenever wherever you have a link like this:

<a v-link="'/about'">About</a>

You’ll need to update it like this:

<router-link to="/about">About</router-link>

Note that target="\_blank" is not supported on <router-link>, so if you need to open a link in a new tab, you have to use <a> instead.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the v-link directive.

### [v-link-active replaced](https://vuejs.org/v2/guide/migration-vue-router.html#v-link-active-replaced)

The v-link-active directive has also been replaced by the tag attribute on [**the <router-link>component**](https://router.vuejs.org/en/api/router-link.html). So for example, you’ll update this:

<li v-link-active>

<a v-link="'/about'">About</a>

</li>

to this:

<router-link tag="li" to="/about">

<a>About</a>

</router-link>

The <a> will be the actual link (and will get the correct href), but the active class will be applied to the outer <li>.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the v-link-activedirective.

## [Programmatic Navigation](https://vuejs.org/v2/guide/migration-vue-router.html#Programmatic-Navigation)

### [router.go changed](https://vuejs.org/v2/guide/migration-vue-router.html#router-go-changed)

For consistency with the [**HTML5 History API**](https://developer.mozilla.org/en-US/docs/Web/API/History_API), router.go is now only used for [**back/forward navigation**](https://router.vuejs.org/en/essentials/navigation.html#routergon), while [**router.push**](https://router.vuejs.org/en/essentials/navigation.html#routerpushlocation) is used to navigate to a specific page.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of router.go being used where router.push should be used instead.

## [Router Options: Modes](https://vuejs.org/v2/guide/migration-vue-router.html#Router-Options-Modes)

### [hashbang: false removed](https://vuejs.org/v2/guide/migration-vue-router.html#hashbang-false-removed)

Hashbangs are no longer required for Google to crawl a URL, so they are no longer the default (or even an option) for the hash strategy.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the hashbang: falseoption.

### [history: true replaced](https://vuejs.org/v2/guide/migration-vue-router.html#history-true-replaced)

All routing mode options have been condensed into a single [**mode option**](https://router.vuejs.org/en/api/options.html#mode). Update:

var router = new VueRouter({

history: 'true'

})

to:

var router = new VueRouter({

mode: 'history'

})

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the history: true option.

### [abstract: true replaced](https://vuejs.org/v2/guide/migration-vue-router.html#abstract-true-replaced)

All routing mode options have been condensed into a single [**mode option**](https://router.vuejs.org/en/api/options.html#mode). Update:

var router = new VueRouter({

abstract: 'true'

})

to:

var router = new VueRouter({

mode: 'abstract'

})

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the abstract: trueoption.

## [Route Options: Misc](https://vuejs.org/v2/guide/migration-vue-router.html#Route-Options-Misc)

### [saveScrollPosition replaced](https://vuejs.org/v2/guide/migration-vue-router.html#saveScrollPosition-replaced)

This has been replaced with a [**scrollBehavior option**](https://router.vuejs.org/en/advanced/scroll-behavior.html) that accepts a function, so that the scroll behavior is completely customizable - even per route. This opens many new possibilities, but to replicate the old behavior of:

saveScrollPosition: true

You can replace it with:

scrollBehavior: function (to, from, savedPosition) {

return savedPosition || { x: 0, y: 0 }

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the saveScrollPosition: true option.

### [root renamed](https://vuejs.org/v2/guide/migration-vue-router.html#root-renamed)

Renamed to base for consistency with [**the HTML <base> element**](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the root option.

### [transitionOnLoad removed](https://vuejs.org/v2/guide/migration-vue-router.html#transitionOnLoad-removed)

This option is no longer necessary now that Vue’s transition system has explicit [**appear transition control**](https://vuejs.org/v2/guide/transitions.html#Transitions-on-Initial-Render).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the transitionOnLoad: true option.

### [suppressTransitionError removed](https://vuejs.org/v2/guide/migration-vue-router.html#suppressTransitionError-removed)

Removed due to hooks simplification. If you really must suppress transition errors, you can use [**try…catch**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch) instead.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the suppressTransitionError: true option.

## [Route Hooks](https://vuejs.org/v2/guide/migration-vue-router.html#Route-Hooks)

### [activate replaced](https://vuejs.org/v2/guide/migration-vue-router.html#activate-replaced)

Use [**beforeRouteEnter**](https://router.vuejs.org/en/advanced/navigation-guards.html#incomponent-guards) in the component instead.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the beforeRouteEnterhook.

### [canActivate replaced](https://vuejs.org/v2/guide/migration-vue-router.html#canActivate-replaced)

Use [**beforeEnter**](https://router.vuejs.org/en/advanced/navigation-guards.html#perroute-guard) in the route instead.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the canActivate hook.

### [deactivate removed](https://vuejs.org/v2/guide/migration-vue-router.html#deactivate-removed)

Use the component’s [**beforeDestroy**](https://vuejs.org/v2/api/#beforeDestroy) or [**destroyed**](https://vuejs.org/v2/api/#destroyed) hooks instead.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the deactivate hook.

### [canDeactivate replaced](https://vuejs.org/v2/guide/migration-vue-router.html#canDeactivate-replaced)

Use [**beforeRouteLeave**](https://router.vuejs.org/en/advanced/navigation-guards.html#incomponent-guards) in the component instead.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the canDeactivate hook.

### [canReuse: false removed](https://vuejs.org/v2/guide/migration-vue-router.html#canReuse-false-removed)

There’s no longer a use case for this in the new Vue Router.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the canReuse: falseoption.

### [data replaced](https://vuejs.org/v2/guide/migration-vue-router.html#data-replaced)

The $route property is now reactive, so you can use a watcher to react to route changes, like this:

watch: {

'$route': 'fetchData'

},

methods: {

fetchData: function () {

// ...

}

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the data hook.

### [$loadingRouteData removed](https://vuejs.org/v2/guide/migration-vue-router.html#loadingRouteData-removed)

Define your own property (e.g. isLoading), then update the loading state in a watcher on the route. For example, if fetching data with [**axios**](https://github.com/mzabriskie/axios):

data: function () {

return {

posts: [],

isLoading: false,

fetchError: null

}

},

watch: {

'$route': function () {

var self = this

self.isLoading = true

self.fetchData().then(function () {

self.isLoading = false

})

}

},

methods: {

fetchData: function () {

var self = this

return axios.get('/api/posts')

.then(function (response) {

self.posts = response.data.posts

})

.catch(function (error) {

self.fetchError = error

})

}

}

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the $loadingRouteDatameta property.

# Migration from Vuex 0.6.x to 1.0

**Vuex 2.0 is released, but this guide only covers the migration to 1.0? Is that a typo? Also, it looks like Vuex 1.0 and 2.0 were released simultaneously. What’s going on? Which one should I use and what’s compatible with Vue 2.0?**

Both Vuex 1.0 and 2.0:

* fully support both Vue 1.0 and 2.0
* will be maintained for the foreseeable future

They have slightly different target users however.

**Vuex 2.0** is a radical redesign and simplification of the API, for those who are starting new projects or want to be on the cutting edge of client-side state management. **It is not covered by this migration guide**, so you should check out [**the Vuex 2.0 docs**](https://vuex.vuejs.org/en/index.html) if you’d like to learn more about it.

**Vuex 1.0** is mostly backwards-compatible, so requires very few changes to upgrade. It is recommended for those with large existing codebases or who want the smoothest possible upgrade path to Vue 2.0. This guide is dedicated to facilitating that process, but only includes migration notes. For the complete usage guide, see [**the Vuex 1.0 docs**](https://github.com/vuejs/vuex/tree/1.0/docs/en).

## [store.watch with String Property Path replaced](https://vuejs.org/v2/guide/migration-vuex.html#store-watch-with-String-Property-Path-replaced)

store.watch now only accept functions. So for example, you would have to replace:

store.watch('user.notifications', callback)

with:

store.watch(

// When the returned result changes...

function (state) {

return state.user.notifications

},

// Run this callback

callback

)

This gives you more complete control over the reactive properties you’d like to watch.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of store.watch with a string as the first argument.

## [Store’s Event Emitter removed](https://vuejs.org/v2/guide/migration-vuex.html#Store%E2%80%99s-Event-Emitter-removed)

The store instance no longer exposes the event emitter interface (on, off, emit). If you were previously using the store as a global event bus, [**see this section**](https://vuejs.org/v2/guide/migration.html#dispatch-and-broadcast-removed) for migration instructions.

Instead of using this interface to watch events emitted by the store itself (e.g. store.on('mutation', callback)), a new method store.subscribe is introduced. Typical usage inside a plugin would be:

var myPlugin = store => {

store.subscribe(function (mutation, state) {

// Do something...

})

}

See example [**the plugins docs**](https://github.com/vuejs/vuex/blob/1.0/docs/en/plugins.md) for more info.

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of store.on, store.off, and store.emit.

## [Middlewares replaced](https://vuejs.org/v2/guide/migration-vuex.html#Middlewares-replaced)

Middlewares are replaced by plugins. A plugin is a function that receives the store as the only argument, and can listen to the mutation event on the store:

const myPlugins = store => {

store.subscribe('mutation', (mutation, state) => {

// Do something...

})

}

For more details, see [**the plugins docs**](https://github.com/vuejs/vuex/blob/1.0/docs/en/plugins.md).

#### Upgrade Path

Run the [**migration helper**](https://github.com/vuejs/vue-migration-helper) on your codebase to find examples of the middlewares option on a store.

### Meta

# Comparison with Other Frameworks

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**](https://github.com/vuejs/vuejs.org/issues/new?title=Inaccuracy+in+comparisons+guide).

## [React](https://vuejs.org/v2/guide/comparison.html#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](https://vuejs.org/v2/guide/comparison.html#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**](https://stefankrause.net/js-frameworks-benchmark8/table.html), 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](https://vuejs.org/v2/guide/comparison.html#HTML-amp-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**](https://vuejs.org/v2/guide/render-function.html) and even [**support JSX**](https://vuejs.org/v2/guide/render-function.html#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 components 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**](https://github.com/gajus/react-css-modules)), scoping CSS in React is often done via CSS-in-JS solutions (e.g. [**styled-components**](https://github.com/styled-components/styled-components), [**glamorous**](https://github.com/paypal/glamorous), and [**emotion**](https://github.com/emotion-js/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**](https://github.com/styled-components/vue-styled-components) and [**vue-emotion**](https://github.com/egoist/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**](https://vuejs.org/v2/guide/single-file-components.html).

[**Single-file components**](https://vuejs.org/v2/guide/single-file-components.html) 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 components is very flexible. Through [**vue-loader**](https://github.com/vuejs/vue-loader), you can use any preprocessor, post-processor, and even deep integration with [**CSS Modules**](https://vue-loader.vuejs.org/en/features/css-modules.html) – all within the <style>element.

### [Scale](https://vuejs.org/v2/guide/comparison.html#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**](https://yarnpkg.com/en/packages?q=redux%20vue&p=1) can be easily integrated into Vue applications. In fact, Vue has even taken this model a step further with [**Vuex**](https://github.com/vuejs/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**](https://github.com/vuejs)) 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**](https://github.com/vuejs/vue-cli) that makes it trivially easy to start a new project using your choice of build system, including [**webpack**](https://github.com/vuejs-templates/webpack), [**Browserify**](https://github.com/vuejs-templates/browserify), or even [**no build system**](https://github.com/vuejs-templates/simple). React is also making strides in this area with [**create-react-app**](https://github.com/facebookincubator/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**](http://yeoman.io/)-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**](https://github.com/facebookincubator/create-react-app#philosophy).

#### 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**](https://vuejs.org/v2/guide/) to learn enough to build non-trivial applications.

### [Native Rendering](https://vuejs.org/v2/guide/comparison.html#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**](https://weex.apache.org/), 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 is [**NativeScript-Vue**](https://nativescript-vue.org/), a [**NativeScript**](https://www.nativescript.org/) plugin for building truly native applications using Vue.js.

### [With MobX](https://vuejs.org/v2/guide/comparison.html#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](https://vuejs.org/v2/guide/comparison.html#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)](https://vuejs.org/v2/guide/comparison.html#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](https://vuejs.org/v2/guide/comparison.html#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](https://vuejs.org/v2/guide/comparison.html#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 full system for rapid Vue.js development. [**Vue CLI**](https://github.com/vuejs/vue-cli) aims to be the standard tooling baseline for the Vue ecosystem. It ensures the various build tools work smoothly together with sensible defaults so you can focus on writing your app instead of spending hours wrangling with configurations. At the same time, it still offers the flexibility to tweak the configuration of each tool to specific needs.

### [Data binding](https://vuejs.org/v2/guide/comparison.html#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](https://vuejs.org/v2/guide/comparison.html#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](https://vuejs.org/v2/guide/comparison.html#Runtime-Performance-1)

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)](https://vuejs.org/v2/guide/comparison.html#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](https://vuejs.org/v2/guide/comparison.html#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**](https://github.com/vuejs/vue/tree/dev/types)and [**official decorator**](https://github.com/vuejs/vue-class-component) 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](https://vuejs.org/v2/guide/comparison.html#Runtime-Performance-2)

Both frameworks are exceptionally fast, with very similar metrics on benchmarks. You can [**browse specific metrics**](https://stefankrause.net/js-frameworks-benchmark8/table.html) for a more granular comparison, but speed is unlikely to be a deciding factor.

### [Size](https://vuejs.org/v2/guide/comparison.html#Size)

Recent versions of Angular, with [**AOT compilation**](https://en.wikipedia.org/wiki/Ahead-of-time_compilation) and [**tree-shaking**](https://en.wikipedia.org/wiki/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](https://vuejs.org/v2/guide/comparison.html#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](https://vuejs.org/v2/guide/comparison.html#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**](https://vuejs.org/v2/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](https://vuejs.org/v2/guide/comparison.html#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**](https://guides.emberjs.com/v2.10.0/templates/handlebars-basics/) and [**object model**](https://guides.emberjs.com/v2.10.0/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**](https://stefankrause.net/js-frameworks-benchmark8/table.html), even after the latest Glimmer engine update in Ember 3.x. Vue automatically batches updates, while in Ember you need to manually manage run loops in performance-critical situations.

## [Knockout](https://vuejs.org/v2/guide/comparison.html#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**](http://knockoutjs.com/documentation/browser-support.html) 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**](https://vuejs.org/v2/guide/components.html#Content-Distribution-with-Slots).

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**](https://gist.github.com/chrisvfritz/9e5f2d6826af00fcbace7be8f6dccb89). It’s definitely somewhat subjective, but many consider Vue’s API to be less complex and better structured.

## [Polymer](https://vuejs.org/v2/guide/comparison.html#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](https://vuejs.org/v2/guide/comparison.html#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**](http://riotjs.com/compare/#virtual-dom-vs-expressions-binding) 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**](https://github.com/vuejs/vue-loader) and [**Browserify**](https://github.com/vuejs/vueify), while Riot relies on community support for build system integration.

# Join the Vue.js Community!

Vue’s community is growing incredibly fast and if you’re reading this, there’s a good chance you’re ready to join it. So… welcome!

Now we’ll answer both what the community can do for you and what you can do for the community.

## [Resources](https://vuejs.org/v2/guide/join.html#Resources)

### [Code of Conduct](https://vuejs.org/v2/guide/join.html#Code-of-Conduct)

Our [**Code of Conduct**](https://vuejs.org/coc) is a guide to make it easier to enrich all of us and the technical communities in which we participate.

### [Get Support](https://vuejs.org/v2/guide/join.html#Get-Support)

* [**Forum**](https://forum.vuejs.org/): The best place to ask questions and get answers about Vue and its ecosystem.
* [**Chat**](https://chat.vuejs.org/): A place for Vue devs to meet and chat in real time.
* [**Meetups**](https://events.vuejs.org/meetups): Want to find local Vue.js enthusiasts like yourself? Interested in becoming a community leader? We have the help and support you need right here!
* [**GitHub**](https://github.com/vuejs): If you have a bug to report or feature to request, that’s what the GitHub issues are for. We also welcome pull requests!

### [Explore the Ecosystem](https://vuejs.org/v2/guide/join.html#Explore-the-Ecosystem)

* [**The Awesome Vue Page**](https://github.com/vuejs/awesome-vue): See what other awesome resources have been published by other awesome people.
* [**The “Show and Tell” Subforum**](https://forum.vuejs.org/c/show-and-tell): Another great place to check out what others have built with and for the growing Vue ecosystem.

## [What You Can Do](https://vuejs.org/v2/guide/join.html#What-You-Can-Do)

### [Contribute Code](https://vuejs.org/v2/guide/join.html#Contribute-Code)

As with any project, there are rules to contributing. To ensure that we can help you or accept your pull request as quickly as possible, please read [**the contributing guide**](https://github.com/vuejs/vue/blob/dev/.github/CONTRIBUTING.md).

After that, you’ll be ready to contribute to Vue’s core repositories:

* [**vue**](https://github.com/vuejs/vue): the core library
* [**vuex**](https://github.com/vuejs/vuex): Flux-inspired state management
* [**vue-router**](https://github.com/vuejs/vue-router): a routing system for SPAs

…as well as many smaller official [**companion libraries**](https://github.com/vuejs).

### [Share (and Build) Your Experience](https://vuejs.org/v2/guide/join.html#Share-and-Build-Your-Experience)

Apart from answering questions and sharing resources in the forum and chat, there are a few other less obvious ways to share and expand what you know:

* **Develop learning materials.** It’s often said that the best way to learn is to teach. If there’s something interesting you’re doing with Vue, strengthen your expertise by writing a blog post, developing a workshop, or even publishing a gist that you share on social media.
* **Watch a repo you care about.** This will send you notifications whenever there’s activity in that repository, giving you insider knowledge about ongoing discussions and upcoming features. It’s a fantastic way to build expertise so that you’re eventually able to help address issues and pull requests.

### [Translate Docs](https://vuejs.org/v2/guide/join.html#Translate-Docs)

Vue has already spread across the globe, with even the core team in at least half a dozen timezones. [**The forum**](https://forum.vuejs.org/) includes 7 languages and counting and many of our docs have [**actively-maintained translations**](https://github.com/vuejs?utf8=%E2%9C%93&query=vuejs.org). We’re very proud of Vue’s international reach, but we can do even better.

I hope that right now, you’re reading this sentence in your preferred language. If not, would you like to help us get there?

If so, please feel free to fork the repo for [**these docs**](https://github.com/vuejs/vuejs.org/) or for any other officially maintained documentation, then start translating. Once you’ve made some progress, open an issue or pull request in the main repo and we’ll put out a call for more contributors to help you out.

### [Become a Community Leader](https://vuejs.org/v2/guide/join.html#Become-a-Community-Leader)

There’s a lot you can do to help Vue grow in your community:

* **Present at your local meetup.** Whether it’s giving a talk or running a workshop, you can bring a lot of value to your community by helping both new and experienced Vue developers continue to grow.
* **Start your own meetup.** If there’s not already a Vue meetup in your area, you can start your own! Use the [**resources at events.vuejs.org**](https://events.vuejs.org/resources/#getting-started) to help you succeed!
* **Help meetup organizers.** There can never be too much help when it comes to running an event, so offer a hand to help out local organizers to help make every event a success.

If you have any questions on how you can get more involved with your local Vue community, reach out at [**@Vuejs\_Events**](https://www.twitter.com/vuejs_events)!