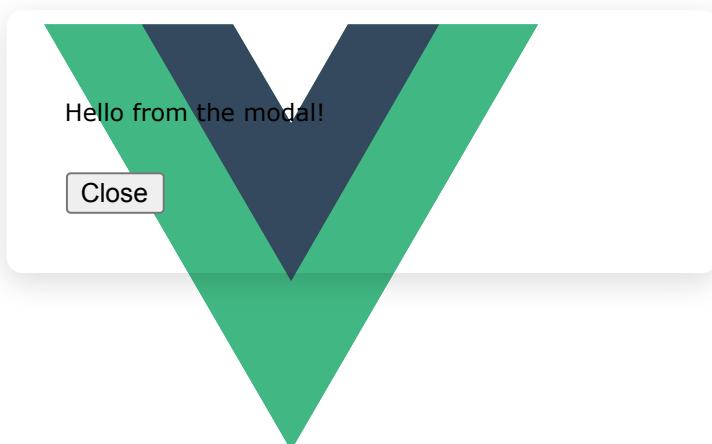


# VUE Docs - English



## Table of contents

• Accessibility	4
• Performance	16
• Production deployment	21
• Security	23
• Keep alive	28
• Suspense	Hello from the modal!
• Teleport	31
• Transition group	35
• Transition	39
• Async	42
• Attrs	53
• Events	56
• Props	60
• Provide inject	71
• Registration	80
• Slots	88
• Application	91
• Class and /style	101
• Component basics	104
• Computed	109
• Conditional	120
• Event handling	125
• Forms	128
• Lifecycle	134
• List	141
• Reactivity fundamentals	145
• Template refs	152
• Template syntax	163
• Watchers	168
• Animation	174
• Composition api /faq	182
• Custom renderer	185
• Reactivity in /depth	192
• Reactivity transform	193
	201

• Render function	208
• Rendering mechanism	219
• Ways of /using /vue	224
• Web components	227
• Introduction	233
• Quick start	238
• Composables	243
• Custom directives	Hello from the modal!
• Plugins	255
• Routing	258
• Sfc	260
• Ssr	262
• State management	271
• Testing	276
• Tooling	286
• Composition api	291
• Options api	298
• Overview	304

# Accessibility

Web accessibility (also known as a11y) refers to the practice of creating websites that can be used by anyone — be that a person with a disability, a slow connection, outdated or broken hardware or simply someone in an unfavorable environment. For example, adding subtitles to a video would help both your deaf and hard-of-hearing users and your users who are in a loud environment and can't hear their phone. Similarly, making sure your text isn't too low contrast will help both your low-vision users and your users who are trying to use their phone in bright sunlight from the modal!

Ready to start but aren't sure where?

Checkout the [Planning and managing web accessibility guide](#) provided by [World Wide Web Consortium \(W3C\)](#)

## Skip link

You should add a link at the top of each page that goes directly to the main content area so users can skip content that is repeated on multiple Web pages.

Typically this is done on the top of `App.vue` as it will be the first focusable element on all your pages:

```
<ul class="skip-links">
  <li>
    <a href="#main" ref="skipLink">Skip to main content</a>
  </li>
</ul>
```

To hide the link unless it is focused, you can add the following style:

```
.skipLink {
  white-space: nowrap;
  margin: 1em auto;
  top: 0;
  position: fixed;
  left: 50%;
  margin-left: -72px;
  opacity: 0;
}
.skipLink:focus {
  opacity: 1;
  background-color: white;
  padding: 0.5em;
  border: 1px solid black;
}
```

Once a user changes route, bring focus back to the skip link. This can be achieved by calling `focus` on the skip link's template ref (assuming usage of `vue-router`):

```
<script>
export default {
  watch: {
```

```

$route() {
  this.$refs.skipLink.focus()
}
}
</script>

```

```

<script setup>
import { ref, watch } from 'vue'
import { useRoute } from 'vue-router'

const route = useRoute()
const skipLink = ref()  Hello from the modal!

watch(
  () => route.path,
  () => {
    skipLink.value.focus()
  }
)
</script>

```

[Read documentation on skip link to main content](#)

## Content Structure

One of the most important pieces of accessibility is making sure that design can support accessible implementation. Design should consider not only color contrast, font selection, text sizing, and language, but also how the content is structured in the application.

### Headings

Users can navigate an application through headings. Having descriptive headings for every section of your application makes it easier for users to predict the content of each section. When it comes to headings, there are a couple of recommended accessibility practices:

- Nest headings in their ranking order: `<h1>` - `<h6>`
- Don't skip headings within a section
- Use actual heading tags instead of styling text to give the visual appearance of headings

[Read more about headings](#)

```

<main role="main" aria-labelledby="main-title">
  <h1 id="main-title">Main title</h1>
  <section aria-labelledby="section-title">
    <h2 id="section-title"> Section Title </h2>
    <h3>Section Subtitle</h3>
    <!-- Content -->
  </section>
  <section aria-labelledby="section-title">
    <h2 id="section-title"> Section Title </h2>
    <h3>Section Subtitle</h3>
    <!-- Content -->
    <h3>Section Subtitle</h3>
  </section>
</main>

```

```
<!-- Content -->
</section>
</main>
```

## Landmarks

**Landmarks** provide programmatic access to sections within an application. Users who rely on assistive technology can navigate to each section of the application and skip over content. You can use [ARIA roles](#) to help you achieve this.

HTML	ARIA Role	Hello from the modal!	Landmark Purpose
header	role="banner"	Prime heading: title of the page	
nav	role="navigation"	Collection of links suitable for use when navigating the document or related documents	
main	role="main"	The main or central content of the document.	
footer	role="contentinfo"	Information about the parent document: footnotes/copyrights/links to privacy statement	
aside	role="complementary"	Supports the main content, yet is separated and meaningful on its own content	
<i>Not available</i>	role="search"	This section contains the search functionality for the application	
form	role="form"	Collection of form-associated elements	
section	role="region"	Content that is relevant and that users will likely want to navigate to. Label must be provided for this element	

:::tip Tip: It is recommended to use landmark HTML elements with redundant landmark role attributes in order to maximize compatibility with legacy [browsers that don't support HTML5 semantic elements](#). :::

[Read more about landmarks](#)

## Semantic Forms

When creating a form, you can use the following elements: `<form>`, `<label>`, `<input>`, `<textarea>`, and `<button>`

Labels are typically placed on top or to the left of the form fields:

```
<form action="/dataCollectionLocation" method="post" autocomplete="on">
  <div v-for="item in formItems" :key="item.id" class="form-item">
    <label :for="item.id">{{ item.label }}: </label>
    <input
      :type="item.type"
      :id="item.id"
      :name="item.id"
      v-model="item.value"
    />
```

```
</div>
<button type="submit">Submit</button>
</form>
```

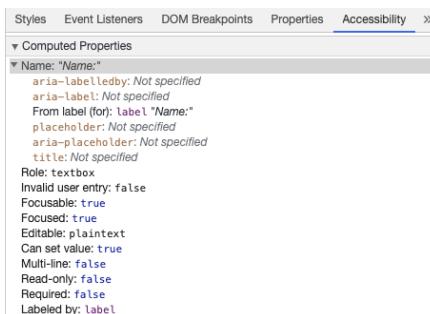
Notice how you can include `autocomplete='on'` on the form element and it will apply to all inputs in your form. You can also set different values for `autocomplete` attribute for each input.

## Labels

Provide labels to describe the purpose of all form control; linking `for` and `id`:

```
<label for="name">Name</label>
<input type="text" name="name" id="name" v-model="name" />
```

If you inspect this element in your chrome developer tools and open the Accessibility tab inside the Elements tab, you will see how the input gets its name from the label:



Chrome Developer Tools showing input accessible name from label

:::warning Warning: Though you might have seen labels wrapping the input fields like this:

```
<label>
  Name:
  <input type="text" name="name" id="name" v-model="name" />
</label>
```

Explicitly setting the labels with a matching id is better supported by assistive technology. :::

`aria-label {#aria-label}`

You can also give the input an accessible name with `aria-label`.

```
<label for="name">Name</label>
<input
  type="text"
  name="name"
  id="name"
  v-model="name"
  :aria-label="nameLabel"
/>
```

Feel free to inspect this element in Chrome DevTools to see how the accessible name has changed:

Styles Event Listeners DOM Breakpoints Properties Accessibility >

▼ ARIA Attributes

**aria-label:** This label will take over the accessible name

▼ Computed Properties

**Name:** "This label will take over the accessible name"

aria-labelledby: Not specified  
From-label-for: Label "Name"  
placeholder: Not specified  
aria-placeholder: Not specified  
title: Not specified

Role: textbox  
Invalid user entry: false  
Focusable: true  
Editable: plaintext  
Can set value: true  
Multi-line: false  
Read-only: false  
Required: false

Hello from the modal!  
Chrome Developer Tools showing input accessible name from aria-label

### aria-labelledby {#aria-labelledby}

Using `aria-labelledby` is similar to `aria-label`, except it is used if the label text is visible on screen. It is paired to other elements by their `id` and you can link multiple `ids`:

```
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on"
>
  <h1 id="billing">Billing</h1>
  <div class="form-item">
    <label for="name">Name:</label>
    <input
      type="text"
      name="name"
      id="name"
      v-model="name"
      aria-labelledby="billing name"
    />
  </div>
  <button type="submit">Submit</button>
</form>
```

Styles Event Listeners DOM Breakpoints Properties Accessibility >

▼ ARIA Attributes

**aria-labelledby:** billing name

▼ Computed Properties

**Name:** "Billing Name:"

▼ **aria-labelledby:**  
hi#billing "Billing"  
input#name "Name:"  
aria-label: Not specified  
From-label-for: Label "Name"  
placeholder: Not specified  
aria-placeholder: Not specified  
title: Not specified

Role: textbox  
Invalid user entry: false  
Focusable: true  
Editable: plaintext  
Can set value: true  
Multi-line: false  
Read-only: false  
Required: false

▼ Labeled by:  
hi#billing "Billing"  
input#name "Name:"

Chrome Developer Tools showing input accessible name from aria-labelledby

## aria-describedby {#aria-describedby}

`aria-describedby` is used the same way as `aria-labelledby` except provides a description with additional information that the user might need. This can be used to describe the criteria for any input:

```
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on">
  <h1 id="billing">Billing<small>Help</small> from the modal!
  <div class="form-item">
    <label for="name">Full Name:</label>
    <input
      type="text"
      name="name"
      id="name"
      v-model="name"
      aria-labelledby="billing"
      aria-describedby="nameDescription"
    />
    <p id="nameDescription">Please provide first and last name.</p>
  </div>
  <button type="submit">Submit</button>
</form>
```

You can see the description by inspecting Chrome DevTools:

The screenshot shows the Chrome Developer Tools Accessibility panel. The 'Accessibility' tab is selected. Under 'Computed Properties', there is a section for the input field with the ID 'name'. It shows the following properties:

- Name:** "Billing Full Name"
- aria-labelledby:** billing
- aria-describedby:** nameDescription
- Description:** "Please provide first and last name."
- Role:** textbox
- Invalid user entry:** false
- Focusable:** true
- Editable:** plaintext
- Can set value:** true
- Multi-line:** false
- Read-only:** false
- Required:** false
- Described by:** nameDescription
- Labeled by:** h1#billing

Chrome Developer Tools showing input accessible name from `aria-labelledby` and description with `aria-describedby`

## Placeholder

Avoid using placeholders as they can confuse many users.

One of the issues with placeholders is that they don't meet the [color contrast criteria](#) by default; fixing the color contrast makes the placeholder look like pre-populated data in the input fields. Looking at the following example, you can see that the Last Name placeholder which meets the color contrast criteria looks like pre-populated data:

**First Name:**

EvanHello from the modal!

**Last Name:**

You

**Submit**

Accessible placeholder

```
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on"
>
  <div v-for="item in formItems" :key="item.id" class="form-item">
    <label :for="item.id">{{ item.label }}: </label>
    <input
      type="text"
      :id="item.id"
      :name="item.id"
      v-model="item.value"
      :placeholder="item.placeholder"
    />
  </div>
  <button type="submit">Submit</button>
</form>
```

```
/* https://www.w3schools.com/howto/howto_css_placeholder.asp */

#lastName::placeholder {
  /* Chrome, Firefox, Opera, Safari 10.1+ */
  color: black;
  opacity: 1; /* Firefox */
}

#lastName:-ms-input-placeholder {
  /* Internet Explorer 10-11 */
```

```

    color: black;
}

#lastName::-ms-input-placeholder {
    /* Microsoft Edge */
    color: black;
}

```

It is best to provide all the information the user needs to fill out forms outside any inputs.

## Instructions

Hello from the modal!

When adding instructions for your input fields, make sure to link it correctly to the input. You can provide additional instructions and bind multiple ids inside an `aria-labelledby`. This allows for more flexible design.

```

<fieldset>
    <legend>Using aria-labelledby</legend>
    <label id="date-label" for="date">Current Date:</label>
    <input
        type="date"
        name="date"
        id="date"
        aria-labelledby="date-label date-instructions"
    />
    <p id="date-instructions">MM/DD/YYYY</p>
</fieldset>

```

Alternatively, you can attach the instructions to the input with `aria-describedby`:

```

<fieldset>
    <legend>Using aria-describedby</legend>
    <label id="dob" for="dob">Date of Birth:</label>
    <input type="date" name="dob" id="dob" aria-describedby="dob-instructions" />
    <p id="dob-instructions">MM/DD/YYYY</p>
</fieldset>

```

## Hiding Content

Usually it is not recommended to visually hide labels, even if the input has an accessible name. However, if the functionality of the input can be understood with surrounding content, then we can hide the visual label.

Let's look at this search field:

```

<form role="search">
    <label for="search" class="hidden-visually">Search: </label>
    <input type="text" name="search" id="search" v-model="search" />
    <button type="submit">Search</button>
</form>

```

We can do this because the search button will help visual users identify the purpose of the input field.

We can use CSS to visually hide elements but keep them available for assistive technology:

```
.hidden-visually {
  position: absolute;
  overflow: hidden;
  white-space: nowrap;
  margin: 0;
  padding: 0;
  height: 1px;
  width: 1px;
  clip: rect(0 0 0 0);
  clip-path: inset(100%);
}
```

`aria-hidden="true"` #aria-hidden=true) the modal!

Adding `aria-hidden="true"` will hide the element from assistive technology but leave it visually available for other users. Do not use it on focusable elements, purely on decorative, duplicated or offscreen content.

```
<p>This is not hidden from screen readers.</p>
<p aria-hidden="true">This is hidden from screen readers.</p>
```

## Buttons

When using buttons inside a form, you must set the type to prevent submitting the form. You can also use an input to create buttons:

```
<form action="/dataCollectionLocation" method="post" autocomplete="on">
  <!-- Buttons -->
  <button type="button">Cancel</button>
  <button type="submit">Submit</button>

  <!-- Input buttons -->
  <input type="button" value="Cancel" />
  <input type="submit" value="Submit" />
</form>
```

## Functional Images

You can use this technique to create functional images.

- Input fields
  - These images will act as a submit type button on forms

```
<form role="search">
  <label for="search" class="hidden-visually">Search: </label>
  <input type="text" name="search" id="search" v-model="search" />
  <input
    type="image"
    class="btnImg"
    src="https://img.icons8.com/search"
    alt="Search"
  />
</form>
```

- Icons

```
<form role="search">
  <label for="searchIcon" class="hidden-Visually">Search: </label>
  <input type="text" name="searchIcon" id="searchIcon" v-model="searchIcon" />
  <button type="submit">
    <i class="fas fa-search aria-hidden="true"></i>
    <span class="hidden-Visually">Search</span>
  </button>
</form>
```

## Standards

The World Wide Web Consortium ([W3C Web Accessibility Initiative](#)) develops web accessibility standards for the different components:

- [User Agent Accessibility Guidelines \(UAAG\)](#)
  - web browsers and media players, including some aspects of assistive technologies
- [Authoring Tool Accessibility Guidelines \(ATAG\)](#)
  - authoring tools
- [Web Content Accessibility Guidelines \(WCAG\)](#)
  - web content - used by developers, authoring tools, and accessibility evaluation tools

## Web Content Accessibility Guidelines (WCAG)

WCAG 2.1 extends on WCAG 2.0 and allows implementation of new technologies by addressing changes to the web. The W3C encourages use of the most current version of WCAG when developing or updating Web accessibility policies.

### WCAG 2.1 Four Main Guiding Principles (abbreviated as POUR):

- [Perceivable](#)
  - Users must be able to perceive the information being presented
- [Operable](#)
  - Interface forms, controls, and navigation are operable
- [Understandable](#)
  - Information and the operation of user interface must be understandable to all users
- [Robust](#)
  - Users must be able to access the content as technologies advance

### Web Accessibility Initiative – Accessible Rich Internet Applications (WAI-ARIA)

W3C's WAI-ARIA provides guidance on how to build dynamic content and advanced user interface controls.

- [Accessible Rich Internet Applications \(WAI-ARIA\) 1.2](#)
- [WAI-ARIA Authoring Practices 1.2](#)

## Resources

### Documentation

- [WCAG 2.0](#)

- [WCAG 2.1](#)
- [Accessible Rich Internet Applications \(WAI-ARIA\) 1.2](#)
- [WAI-ARIA Authoring Practices 1.2](#)

## Assistive Technologies

- Screen Readers
  - [NVDA](#)
  - [VoiceOver](#)
  - [JAWS](#)
  - [ChromeVox](#)
- Zooming Tools
  - [MAGIC](#)
  - [ZoomText](#)
  - [Magnifier](#)

Hello from the modal!

## Testing

- Automated Tools
  - [Lighthouse](#)
  - [WAVE](#)
- Color Tools
  - [WebAim Color Contrast](#)
  - [WebAim Link Color Contrast](#)
- Other Helpful Tools
  - [HeadingMap](#)
  - [Color Oracle](#)
  - [Focus Indicator](#)
  - [NerdeFocus](#)

## Users

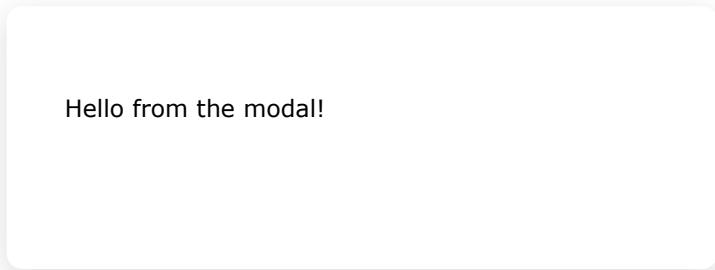
The World Health Organization estimates that 15% of the world's population has some form of disability, 2-4% of them severely so. That is an estimated 1 billion people worldwide; making people with disabilities the largest minority group in the world.

There are a huge range of disabilities, which can be divided roughly into four categories:

- *Visual* - These users can benefit from the use of screen readers, screen magnification, controlling screen contrast, or braille display.
- *Auditory* - These users can benefit from captioning, transcripts or sign language video.
- *Motor* - These users can benefit from a range of [assistive technologies for motor impairments](#): voice recognition software, eye tracking, single-switch access, head wand, sip and puff switch, oversized track-ball mouse, adaptive keyboard or other assistive technologies.
- *Cognitive* - These users can benefit from supplemental media, structural organization of content, clear and simple writing.

Check out the following links from WebAim to understand from users:

- [Web Accessibility Perspectives: Explore the Impact and Benefits for Everyone](#)
- [Stories of Web Users](#)



Hello from the modal!

---

[Go to TOC](#)

# Performance

## Overview

Vue is designed to be performant for most common use cases without much need for manual optimizations. However, there are always challenging scenarios where extra fine-tuning is needed. In this section, we will discuss what you should pay attention to when it comes to performance in a Vue application.

Hello from the modal!

First, let's discuss the two major aspects of web performance:

- **Page Load Performance:** how fast the application shows content and becomes interactive on the initial visit. This is usually measured using web vital metrics like [Largest Contentful Paint \(LCP\)](#) and [First Input Delay \(FID\)](#).
- **Update Performance:** how fast the application updates in response to user input. For example, how fast a list updates when the user types in a search box, or how fast the page switches when the user clicks a navigation link in a Single-Page Application (SPA).

While it would be ideal to maximize both, different frontend architectures tend to affect how easy it is to attain desired performance in these aspects. In addition, the type of application you are building greatly influences what you should prioritize in terms of performance. Therefore, the first step of ensuring optimal performance is picking the right architecture for the type of application you are building:

- Consult [Ways of Using Vue](#) to see how you can leverage Vue in different ways.
- Jason Miller discusses the types of web applications and their respective ideal implementation / delivery in [Application Holotypes](#).

## Profiling Options

To improve performance, we need to first know how to measure it. There are a number of great tools that can help in this regard:

For profiling load performance of production deployments:

- [PageSpeed Insights](#)
- [WebPageTest](#)

For profiling performance during local development:

- [Chrome DevTools Performance Panel](#)
  - `app.config.performance` enables Vue-specific performance markers in Chrome DevTools' performance timeline.
- [Vue DevTools Extension](#) also provides a performance profiling feature.

# Page Load Optimizations

There are many framework-agnostic aspects for optimizing page load performance - check out [this web.dev guide](#) for a comprehensive round up. Here, we will primarily focus on techniques that are specific to Vue.

## Choosing the Right Architecture

If your use case is sensitive to page load performance, avoid shipping it as a pure client-side SPA. You want your server to be directly sending HTML containing the content the users want to see. Pure client-side rendering suffers from slow time-to-content. This can be mitigated with [Server-Side Rendering \(SSR\)](#) or [Static Site Generation \(SSG\)](#). Check out the [SSR Guide](#) to learn about performing SSR with Vue. If your app doesn't have rich interactivity requirements, you can also use a traditional backend server to render the HTML and enhance it with Vue on the client.

If your main application has to be an SPA, but has marketing pages (landing, about, blog), ship them separately! Your marketing pages should ideally be deployed as static HTML with minimal JS, by using SSG.

## Bundle Size and Tree-shaking

One of the most effective ways to improve page load performance is shipping smaller JavaScript bundles. Here are a few ways to reduce bundle size when using Vue:

- Use a build step if possible.
  - Many of Vue's APIs are "[tree-shakable](#)" if bundled via a modern build tool. For example, if you don't use the built-in `<Transition>` component, it won't be included in the final production bundle. Tree-shaking can also remove other unused modules in your source code.
  - When using a build step, templates are pre-compiled so we don't need to ship the Vue compiler to the browser. This saves **14kb** min+gzipped JavaScript and avoids the runtime compilation cost.
- Be cautious of size when introducing new dependencies! In real world applications, bloated bundles are most often a result of introducing heavy dependencies without realizing it.
  - If using a build step, prefer dependencies that offer ES module formats and are tree-shaking friendly. For example, prefer `lodash-es` over `lodash`.
  - Check a dependency's size and evaluate whether it is worth the functionality it provides. Note if the dependency is tree-shaking friendly, the actual size increase will depend on the APIs you actually import from it. Tools like [bundlejs.com](#) can be used for quick checks, but measuring with your actual build setup will always be the most accurate.
- If you are using Vue primarily for progressive enhancement and prefer to avoid a build step, consider using [petite-vue](#) (only **6kb**) instead.

## Code Splitting

Code splitting is where a build tool splits the application bundle into multiple smaller chunks, which can then be loaded on demand or in parallel. With proper code splitting, features required at page load can be downloaded immediately, with additional chunks being lazy loaded only when needed, thus improving performance.

Bundlers like Rollup (which Vite is based upon) or webpack can automatically create split chunks by detecting the ESM dynamic import syntax:

```
// lazy.js and its dependencies will be split into a separate chunk
// and only loaded when `loadLazy()` is called.
function loadLazy() {
  return import('./lazy.js')
}
```

Lazy loading is best used on features that are not immediately needed after initial page load. In Vue applications, this can be used in combination with Vue's [Async Component](#) feature to create split chunks for component trees:

```
import { defineAsyncComponent } from 'vue'

// a separate chunk is created for Foo.vue and its dependencies.
// it is only fetched on demand when the async component is
// rendered on the page.
const Foo = defineAsyncComponent(() => import('./Foo.vue'))
```

For applications using Vue Router, it is strongly recommended to use lazy loading for route components. Vue Router has explicit support for lazy loading, separate from `defineAsyncComponent`. See [Lazy Loading Routes](#) for more details.

## Update Optimizations

### Props Stability

In Vue, a child component only updates when at least one of its received props has changed. Consider the following example:

```
<ListItem
  v-for="item in list"
  :id="item.id"
  :active-id="activeId" />
```

Inside the `<ListItem>` component, it uses its `id` and `activeId` props to determine whether it is the currently active item. While this works, the problem is that whenever `activeId` changes, **every** `<ListItem>` in the list has to update!

Ideally, only the items whose active status changed should update. We can achieve that by moving the active status computation into the parent, and make `<ListItem>` directly accept an `active` prop instead:

```
<ListItem
  v-for="item in list"
  :id="item.id"
  :active="item.id === activeId" />
```

Now, for most components the `active` prop will remain the same when `activeId` changes, so they no longer need to update. In general, the idea is keeping the props passed to child components as stable as possible.

### v-once {#v-once}

Hello from the modal!

`v-once` is a built-in directive that can be used to render content that relies on runtime data but never needs to update. The entire sub-tree it is used on will be skipped for all future updates. Consult its [API reference](#) for more details.

### v-memo {#v-memo}

`v-memo` is a built-in directive that can be used to conditionally skip the update of large sub-trees or `v-for` lists. Consult its [API reference](#) for more details.

## General Optimizations

The following tips affect both page load and update performance.

### Virtualize Large Lists

One of the most common performance issues in all frontend applications is rendering large lists. No matter how performant a framework is, rendering a list with thousands of items **will** be slow due to the sheer number of DOM nodes that the browser needs to handle.

However, we don't necessarily have to render all these nodes upfront. In most cases, the user's screen size can display only a small subset of our large list. We can greatly improve the performance with **list virtualization**, the technique of only rendering the items that are currently in or close to the viewport in a large list.

Implementing list virtualization isn't easy, luckily there are existing community libraries that you can directly use:

- [vue-virtual-scroller](#)
- [vue-virtual-scroll-grid](#)
- [vueuc/VVirtualList](#)

## Reduce Reactivity Overhead for Large Immutable Structures

Vue's reactivity system is deep by default. While this makes state management intuitive, it does create a certain level of overhead when the data size is large, because every property access triggers proxy traps that perform dependency tracking. This typically becomes noticeable when dealing with large arrays of deeply nested objects, where a single render needs to access 100,000+ properties, so it should only affect very specific use cases.

Vue does provide an escape hatch to opt-out of deep reactivity by using `shallowRef()` and `shallowReactive()`. Shallow APIs create state that is reactive only at the root level, and exposes all nested objects untouched. This keeps nested property access fast, with the trade-off being that we must now treat all nested objects as immutable, and updates can only be triggered by replacing the root state:

```
const shallowArray = shallowRef([
  /* big list of deep objects */
])

// this won't trigger updates...
shallowArray.value.push(newObject)
// this does:
shallowArray.value = [...shallowArray.value, newObject]

// this won't trigger updates...
shallowArray.value[0].foo = 1
// this does:
shallowArray.value = [
  {
    ...shallowArray.value[0],
    foo: 1
  },
  ...shallowArray.value.slice(1)
]
```

## Avoid Unnecessary Component Abstractions

Sometimes we may create [renderless components](#) or higher-order components (i.e. components that render other components with extra props) for better abstraction or code organization. While there is nothing wrong with this, do keep in mind that component instances are much more expensive than plain DOM nodes, and creating too many of them due to abstraction patterns will incur performance costs.

Note that reducing only a few instances won't have noticeable effect, so don't sweat it if the component is rendered only a few times in the app. The best scenario to consider this optimization is again in large lists. Imagine a list of 100 items where each item component contains many child components. Removing one unnecessary component abstraction here could result in a reduction of hundreds of component instances.

# Production Deployment

## Development vs. Production

During development, Vue provides a number of features to improve the development experience:

- Warning for common errors and pitfalls
- Props / events validation
- [Reactivity debugging hooks](#)
- Devtools integration

Hello from the modal!

However, these features become useless in production. Some of the warning checks can also incur a small amount of performance overhead. When deploying to production, we should drop all the unused, development-only code branches for smaller payload size and better performance.

## Without Build Tools

If you are using Vue without a build tool by loading it from a CDN or self-hosted script, make sure to use the production build (dist files that end in `.prod.js`) when deploying to production. Production builds are pre-minified with all development-only code branches removed.

- If using global build (accessing via the `Vue` global): use `vue.global.prod.js`.
- If using ESM build (accessing via native ESM imports): use `vue.esm-browser.prod.js`.

Consult the [dist file guide](#) for more details.

## With Build Tools

Projects scaffolded via `create-vue` (based on Vite) or Vue CLI (based on webpack) are pre-configured for production builds.

If using a custom setup, make sure that:

1. `vue` resolves to `vue.runtime.esm-bundler.js`.
2. The `compile time feature flags` are properly configured.
3. `process.env.NODE_ENV` is replaced with `"production"` during build.

Additional references:

- [Vite production build guide](#)
- [Vite deployment guide](#)
- [Vue CLI deployment guide](#)

## Tracking Runtime Errors

The [app-level error handler](#) can be used to report errors to tracking services:

```
import { createApp } from 'vue'  
  
const app = createApp(...)  
  
app.config.errorHandler = (err, instance, info) => {  
  // report error to tracking services  
}
```

Services such as [Sentry](#) and [Bugsnag](#) also provide official integrations for Vue.

Hello from the modal!

---

[Go to TOC](#)

# Security

## Reporting Vulnerabilities

When a vulnerability is reported, it immediately becomes our top concern, with a full-time contributor dropping everything to work on it. To report a vulnerability, please email [security@vuejs.org](mailto:security@vuejs.org).

While the discovery of new vulnerabilities is rare, we also recommend always using the latest versions of Vue and its official companion libraries to ensure your application remains as secure as possible.

## Rule No.1: Never Use Non-trusted Templates

The most fundamental security rule when using Vue is **never use non-trusted content as your component template**. Doing so is equivalent to allowing arbitrary JavaScript execution in your application - and worse, could lead to server breaches if the code is executed during server-side rendering. An example of such usage:

```
Vue.createApp({
  template: `<div>` + userProvidedString + `</div>` // NEVER DO THIS
}).mount('#app')
```

Vue templates are compiled into JavaScript, and expressions inside templates will be executed as part of the rendering process. Although the expressions are evaluated against a specific rendering context, due to the complexity of potential global execution environments, it is impractical for a framework like Vue to completely shield you from potential malicious code execution without incurring unrealistic performance overhead. The most straightforward way to avoid this category of problems altogether is to make sure the contents of your Vue templates are always trusted and entirely controlled by you.

## What Vue Does to Protect You

### HTML content

Whether using templates or render functions, content is automatically escaped. That means in this template:

```
<h1>{{ userProvidedString }}</h1>
```

if `userProvidedString` contained:

```
'<script>alert("hi")</script>'
```

then it would be escaped to the following HTML:

```
&lt;script&gt;alert("hi")&lt;/script&gt;
```

thus preventing the script injection. This escaping is done using native browser APIs, like `textContent`, so a vulnerability can only exist if the browser itself is vulnerable.

## Attribute bindings

Similarly, dynamic attribute bindings are also automatically escaped. That means in this template:

```
<h1 :title="userProvidedString">
  hello
</h1>
```

Hello from the modal!

if `userProvidedString` contained:

```
" onclick="alert('hi')"
```

then it would be escaped to the following HTML:

```
&quot; onclick=&quot;alert('hi')"
```

thus preventing the close of the `title` attribute to inject new, arbitrary HTML. This escaping is done using native browser APIs, like `setAttribute`, so a vulnerability can only exist if the browser itself is vulnerable.

## Potential Dangers

In any web application, allowing unsanitized, user-provided content to be executed as HTML, CSS, or JavaScript is potentially dangerous, so should be avoided wherever possible. There are times when some risk may be acceptable though.

For example, services like CodePen and JSFiddle allow user-provided content to be executed, but it's in a context where this is expected and sandboxed to some extent inside iframes. In the cases when an important feature inherently requires some level of vulnerability, it's up to your team to weigh the importance of the feature against the worst-case scenarios the vulnerability enables.

## HTML Injection

As you learned earlier, Vue automatically escapes HTML content, preventing you from accidentally injecting executable HTML into your application. However, **in cases where you know the HTML is safe**, you can explicitly render HTML content:

- Using a template:

```
<div v-html="userProvidedHtml"></div>
```

- Using a render function:

```
h('div', {
  innerHTML: this.userProvidedHtml
})
```

- Using a render function with JSX:

```
<div innerHTML={this.userProvidedHtml}></div>
```

:::warning User-provided HTML can never be considered 100% safe unless it's in a sandboxed iframe or in a part of the app where only the user who wrote that HTML can ever be exposed to it. Additionally, allowing users to write their own Vue templates brings similar dangers. :::

## URL Injection

In a URL like this:

```
<a :href="userProvidedUrl'Hello from the modal!
      click me
    </a>
```

There's a potential security issue if the URL has not been "sanitized" to prevent JavaScript execution using `javascript:`. There are libraries such as [sanitize-url](#) to help with this, but note: if you're ever doing URL sanitization on the frontend, you already have a security issue. **User-provided URLs should always be sanitized by your backend before even being saved to a database.** Then the problem is avoided for every client connecting to your API, including native mobile apps. Also note that even with sanitized URLs, Vue cannot help you guarantee that they lead to safe destinations.

## Style Injection

Looking at this example:

```
<a
  :href="sanitizedUrl"
  :style="userProvidedStyles"
>
  click me
</a>
```

let's assume that `sanitizedUrl` has been sanitized, so that it's definitely a real URL and not JavaScript. With the `userProvidedStyles`, malicious users could still provide CSS to "click jack", e.g. styling the link into a transparent box over the "Log in" button. Then if <https://user-controlled-website.com/> is built to resemble the login page of your application, they might have just captured a user's real login information.

You may be able to imagine how allowing user-provided content for a `<style>` element would create an even greater vulnerability, giving that user full control over how to style the entire page. That's why Vue prevents rendering of style tags inside templates, such as:

```
<style>{{ userProvidedStyles }}</style>
```

To keep your users fully safe from click jacking, we recommend only allowing full control over CSS inside a sandboxed iframe. Alternatively, when providing user control through a style binding, we recommend using its [object syntax](#) and only allowing users to provide values for specific properties it's safe for them to control, like this:

```
<a
  :href="sanitizedUrl"
  :style="{
    color: userProvidedColor,
    background: userProvidedBackground
  }"
>
  click me
</a>
```

## JavaScript Injection

We strongly discourage ever rendering a `<script>` element with Vue, since templates and render functions should never have side effects. However, this isn't the only way to include strings that would be evaluated as JavaScript at runtime.

Every HTML element has attributes with values accepting strings of JavaScript, such as `onclick`, `onfocus`, and `onmouseenter`. Binding user-provided JavaScript to any of these event attributes is a potential security risk, so should be avoided.

:::warning User-provided JavaScript can never be considered 100% safe unless it's in a sandboxed iframe or in a part of the app where only the user who wrote that JavaScript can ever be exposed to it. :::

Sometimes we receive vulnerability reports on how it's possible to do cross-site scripting (XSS) in Vue templates. In general, we do not consider such cases to be actual vulnerabilities, because there's no practical way to protect developers from the two scenarios that would allow XSS:

1. The developer is explicitly asking Vue to render user-provided, unsanitized content as Vue templates.  
This is inherently unsafe and there's no way for Vue to know the origin.
2. The developer is mounting Vue to an entire HTML page which happens to contain server-rendered and user-provided content. This is fundamentally the same problem as #1, but sometimes devs may do it without realizing. This can lead to possible vulnerabilities where the attacker provides HTML which is safe as plain HTML but unsafe as a Vue template. The best practice is to **never mount Vue on nodes that may contain server-rendered and user-provided content**.

## Best Practices

The general rule is that if you allow unsanitized, user-provided content to be executed (as either HTML, JavaScript, or even CSS), you might be opening yourself up to attacks. This advice actually holds true whether using Vue, another framework, or even no framework.

Beyond the recommendations made above for [Potential Dangers](#), we also recommend familiarizing yourself with these resources:

- [HTML5 Security Cheat Sheet](#)
- [OWASP's Cross Site Scripting \(XSS\) Prevention Cheat Sheet](#)

Then use what you learn to also review the source code of your dependencies for potentially dangerous patterns, if any of them include 3rd-party components or otherwise influence what's rendered to the DOM.

## Backend Coordination

HTTP security vulnerabilities, such as cross-site request forgery (CSRF/XSRF) and cross-site script inclusion (XSSI), are primarily addressed on the backend, so aren't a concern of Vue's. However, it's still a good idea to communicate with your backend team to learn how to best interact with their API, e.g. by submitting CSRF tokens with form submissions.

## Server-Side Rendering (SSR)

There are some additional security concerns when using SSR, so make sure to follow the best practices outlined throughout [our SSR documentation](#) to avoid vulnerabilities.

# KeepAlive

`<KeepAlive>` is a built-in component that allows us to conditionally cache component instances when dynamically switching between multiple components.

## Basic Usage

In the Component Basics chapter, we introduced the syntax for [Dynamic Components](#), using the `<component>` special element:

```
<component :is="activeComponent" />
```

By default, an active component instance will be unmounted when switching away from it. This will cause any changed state it holds to be lost. When this component is displayed again, a new instance will be created with only the initial state.

In the example below, we have two stateful components - A contains a counter, while B contains a message synced with an input via `v-model`. Try updating the state of one of them, switch away, and then switch back to it:

You'll notice that when switched back, the previous changed state would have been reset.

Creating fresh component instance on switch is normally useful behavior, but in this case, we'd really like the two component instances to be preserved even when they are inactive. To solve this problem, we can wrap our dynamic component with the `<KeepAlive>` built-in component:

```
<!-- Inactive components will be cached! -->
<KeepAlive>
  <component :is="activeComponent" />
</KeepAlive>
```

Now, the state will be persisted across component switches:

[Try it in the Playground](#)

[Try it in the Playground](#)

:::tip When used in [DOM templates](#), it should be referenced as `<keep-alive>`. :::

## Include / Exclude

By default, `<KeepAlive>` will cache any component instance inside. We can customize this behavior via the `include` and `exclude` props. Both props can be a comma-delimited string, a `RegExp`, or an array containing either types:

```
<!-- comma-delimited string -->
<KeepAlive include="a,b">
  <component :is="view" />
```

```
</KeepAlive>

<!-- regex (use `v-bind`) -->
<KeepAlive :include="/alb/">
  <component :is="view" />
</KeepAlive>

<!-- Array (use `v-bind`) -->
<KeepAlive :include="['a', 'b']">
  <component :is="view" />
</KeepAlive>
```

The match is checked against the `:include` option, so components that need to be conditionally cached by `KeepAlive` must explicitly declare a `name` option.

:::tip Since version 3.2.34, a single-file component using `<script setup>` will automatically infer its `name` option based on the filename, removing the need to manually declare the name. :::

## Max Cached Instances

We can limit the maximum number of component instances that can be cached via the `max` prop. When `max` is specified, `<KeepAlive>` behaves like an [LRU cache](#): if the number of cached instances is about to exceed the specified max count, the least recently accessed cached instance will be destroyed to make room for the new one.

```
<KeepAlive :max="10">
  <component :is="activeComponent" />
</KeepAlive>
```

## Lifecycle of Cached Instance

When a component instance is removed from the DOM but is part of a component tree cached by `<KeepAlive>`, it goes into a **deactivated** state instead of being unmounted. When a component instance is inserted into the DOM as part of a cached tree, it is **activated**.

A kept-alive component can register lifecycle hooks for these two states using `onActivated()` and `onDeactivated()`:

```
<script setup>
import { onActivated, onDeactivated } from 'vue'

onActivated() => {
  // called on initial mount
  // and every time it is re-inserted from the cache
}

onDeactivated() => {
  // called when removed from the DOM into the cache
  // and also when unmounted
}
</script>
```

A kept-alive component can register lifecycle hooks for these two states using `activated` and `deactivated` hooks:

```
export default {
  activated() {
    // called on initial mount
    // and every time it is re-inserted from the cache
  },
  deactivated() {
    // called when removed from the DOM into the cache
    // and also when unmounted
  }
}
```

Hello from the modal!

Note that:

- `onActivated activated` is also called on mount, and `onDeactivated deactivated` on unmount.
- Both hooks work for not only the root component cached by `<KeepAlive>`, but also descendant components in the cached tree.

---

## Related

- [`<KeepAlive>` API reference](#)

---

[Go to TOC](#)

# Suspense

:::warning Experimental Feature `<Suspense>` is an experimental feature. It is not guaranteed to reach stable status and the API may change before it does. :::

`<Suspense>` is a built-in component for orchestrating async dependencies in a component tree. It can render a loading state while waiting for multiple nested async dependencies down the component tree to be resolved.

Hello from the modal!

## Async Dependencies

To explain the problem `<Suspense>` is trying to solve and how it interacts with these async dependencies, let's imagine a component hierarchy like the following:

```

<Suspense>
└─ <Dashboard>
  ├─ <Profile>
  |  └─ <FriendStatus> (component with async setup())
  └─ <Content>
    ├─ <ActivityFeed> (async component)
    └─ <Stats> (async component)
  
```

In the component tree there are multiple nested components whose rendering depends on some async resource to be resolved first. Without `<Suspense>`, each of them will need to handle its own loading / error and loaded states. In the worst case scenario, we may see three loading spinners on the page, with content displayed at different times.

The `<Suspense>` component gives us the ability to display top-level loading / error states while we wait on these nested async dependencies to be resolved.

There are two types of async dependencies that `<Suspense>` can wait on:

1. Components with an async `setup()` hook. This includes components using `<script setup>` with top-level `await` expressions.
2. [Async Components](#).

### `async setup() { #async-setup }`

A Composition API component's `setup()` hook can be async:

```

export default {
  async setup() {
    const res = await fetch(...)
    const posts = await res.json()
    return {
      posts
    }
  }
}
  
```

If using `<script setup>`, the presence of top-level `await` expressions automatically makes the component an async dependency:

```
<script setup>
const res = await fetch(...)
const posts = await res.json()
</script>

<template>
  {{ posts }}
</template>
```

Hello from the modal!

## Async Components

Async components are "**suspensible**" by default. This means that if it has a `<Suspense>` in the parent chain, it will be treated as an async dependency of that `<Suspense>`. In this case, the loading state will be controlled by the `<Suspense>`, and the component's own loading, error, delay and timeout options will be ignored.

The async component can opt-out of `Suspense` control and let the component always control its own loading state by specifying `suspensible: false` in its options.

## Loading State

The `<Suspense>` component has two slots: `#default` and `#fallback`. Both slots only allow for **one** immediate child node. The node in the default slot is shown if possible. If not, the node in the fallback slot will be shown instead.

```
<Suspense>
  <!-- component with nested async dependencies -->
  <Dashboard />

  <!-- loading state via #fallback slot -->
  <template #fallback>
    Loading...
  </template>
</Suspense>
```

On initial render, `<Suspense>` will render its default slot content in memory. If any async dependencies are encountered during the process, it will enter a **pending** state. During the pending state, the fallback content will be displayed. When all encountered async dependencies have been resolved, `<Suspense>` enters a **resolved** state and the resolved default slot content is displayed.

If no async dependencies were encountered during the initial render, `<Suspense>` will directly go into a resolved state.

Once in a resolved state, `<Suspense>` will only revert to a pending state if the root node of the `#default` slot is replaced. New async dependencies nested deeper in the tree will **not** cause the `<Suspense>` to revert to a pending state.

When a revert happens, fallback content will not be immediately displayed. Instead, `<Suspense>` will display the previous `#default` content while waiting for the new content and its async dependencies to be resolved. This behavior can be configured with the `timeout` prop: `<Suspense>` will switch to fallback content if it takes longer than `timeout` to render the new default content. A `timeout` value of `0` will cause the fallback content to be displayed immediately when default content is replaced.

## Events

The `<Suspense>` component emits 3 events: `pending`, `resolve` and `fallback`. The `pending` event occurs when entering a pending state. The `resolve` event is emitted when new content has finished resolving in the `default` slot. The `fallback` event is fired when the contents of the `fallback` slot are shown.

The events could be used, for example, to show a loading indicator in front of the old DOM while new components are loading.

## Error Handling

`<Suspense>` currently does not provide error handling via the component itself - however, you can use the `errorCaptured` option or the `onErrorCaptured()` hook to capture and handle async errors in the parent component of `<Suspense>`.

## Combining with Other Components

It is common to want to use `<Suspense>` in combination with the `<Transition>` and `<KeepAlive>` components. The nesting order of these components is important to get them all working correctly.

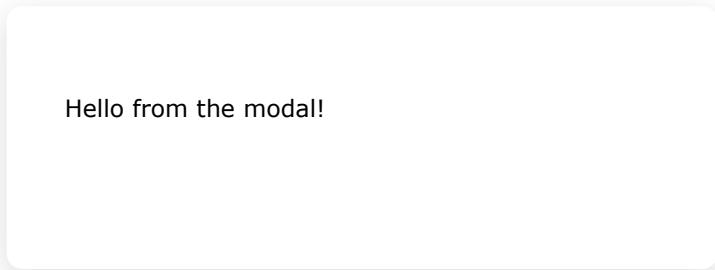
In addition, these components are often used in conjunction with the `<RouterView>` component from [Vue Router](#).

The following example shows how to nest these components so that they all behave as expected. For simpler combinations you can remove the components that you don't need:

```
<RouterView v-slot="{ Component }">
  <template v-if="Component">
    <Transition mode="out-in">
      <KeepAlive>
        <Suspense>
          <!-- main content -->
          <component :is="Component"></component>

          <!-- loading state -->
          <template #fallback>
            Loading...
          </template>
        </Suspense>
      </KeepAlive>
    </Transition>
  </template>
</RouterView>
```

Vue Router has built-in support for [lazily loading components](#) using dynamic imports. These are distinct from async components and currently they will not trigger `<Suspense>`. However, they can still have async components as descendants and those can trigger `<Suspense>` in the usual way.



Hello from the modal!

---

[Go to TOC](#)

# Teleport

`<Teleport>` is a built-in component that allows us to "teleport" a part of a component's template into a DOM node that exists outside the DOM hierarchy of that component.

## Basic Usage

Sometimes we may run into the following scenario: a part of a component's template belongs to it logically, but from a visual standpoint, it should be displayed somewhere else in the DOM, outside of the Vue application.

The most common example of this is when building a full-screen modal. Ideally, we want the modal's button and the modal itself to live within the same component, since they are both related to the open / close state of the modal. But that means the modal will be rendered alongside the button, deeply nested in the application's DOM hierarchy. This can create some tricky issues when positioning the modal via CSS.

Consider the following HTML structure.

```
<div class="outer">
  <h3>Vue Teleport Example</h3>
  <div>
    <MyModal />
  </div>
</div>
```

And here is the implementation of `<MyModal>`:

```
<script setup>
import { ref } from 'vue'

const open = ref(false)
</script>

<template>
  <button @click="open = true">Open Modal</button>

  <div v-if="open" class="modal">
    <p>Hello from the modal!</p>
    <button @click="open = false">Close</button>
  </div>
</template>

<style scoped>
.modal {
  position: fixed;
  z-index: 999;
  top: 20%;
  left: 50%;
  width: 300px;
  margin-left: -150px;
}
</style>
```

```

<script>
export default {
  data() {
    return {
      open: false
    }
  }
}
</script>

<template>
  <button @click="open = true">Open Modal</button>
  Hello from the modal!
  <div v-if="open" class="modal">
    <p>Hello from the modal!</p>
    <button @click="open = false">Close</button>
  </div>
</template>

<style scoped>
.modal {
  position: fixed;
  z-index: 999;
  top: 20%;
  left: 50%;
  width: 300px;
  margin-left: -150px;
}
</style>

```

The component contains a `<button>` to trigger the opening of the modal, and a `<div>` with a class of `.modal`, which will contain the modal's content and a button to self-close.

When using this component inside the initial HTML structure, there are a number of potential issues:

- `position: fixed` only places the element relative to the viewport when no ancestor element has `transform`, `perspective` or `filter` property set. If, for example, we intend to animate the ancestor `<div class="outer">` with a CSS transform, it would break the modal layout!
- The modal's `z-index` is constrained by its containing elements. If there is another element that overlaps with `<div class="outer">` and has a higher `z-index`, it would cover our modal.

`<Teleport>` provides a clean way to work around these, by allowing us to break out of the nested DOM structure. Let's modify `<MyModal>` to use `<Teleport>`:

```

<button @click="open = true">Open Modal</button>

<Teleport to="body">
  <div v-if="open" class="modal">
    <p>Hello from the modal!</p>
    <button @click="open = false">Close</button>
  </div>
</Teleport>

```

The `to` target of `<Teleport>` expects a CSS selector string or an actual DOM node. Here, we are essentially telling Vue to "teleport" this template fragment **to** the `body` tag".

You can click the button below and inspect the `<body>` tag via your browser's devtools:

[Open Modal](#)

You can combine `<Teleport>` with `<Transition>` to create animated modals - see [Example here](#).

:::tip The teleport `to` target must be already in the DOM when the `<Teleport>` component is mounted. Ideally, this should be an element outside the entire Vue application. If targeting another element rendered by Vue, you need to make sure that element is mounted before the `<Teleport>`. :::

## Using with Components

Hello from the modal!

`<Teleport>` only alters the rendered DOM structure - it does not affect the logical hierarchy of the components. That is to say, if `<Teleport>` contains a component, that component will remain a logical child of the parent component containing the `<Teleport>`. Props passing and event emitting will continue to work the same way.

This also means that injections from a parent component work as expected, and that the child component will be nested below the parent component in the Vue Devtools, instead of being placed where the actual content moved to.

## Disabling Teleport

In some cases, we may want to conditionally disable `<Teleport>`. For example, we may want to render a component as an overlay for desktop, but inline on mobile. `<Teleport>` supports the `disabled` prop which can be dynamically toggled:

```
<Teleport :disabled="isMobile">
  ...
</Teleport>
```

Where the `isMobile` state can be dynamically updated by detecting media query changes.

## Multiple Teleports on the Same Target

A common use case would be a reusable `<Modal>` component, with the potential for multiple instances to be active at the same time. For this kind of scenario, multiple `<Teleport>` components can mount their content to the same target element. The order will be a simple append - later mounts will be located after earlier ones within the target element.

Given the following usage:

```
<Teleport to="#modals">
  <div>A</div>
</Teleport>
<Teleport to="#modals">
  <div>B</div>
</Teleport>
```

The rendered result would be:

```
<div id="modals">
  <div>A</div>
  <div>B</div>
</div>
```

## Related

- [<Teleport> API reference](#)
- [Handling Teleports in SSR](#)

Hello from the modal!

---

[Go to TOC](#)

# TransitionGroup

`<TransitionGroup>` is a built-in component designed for animating the insertion, removal, and order change of elements or components that are rendered in a list.

## Differences from `<Transition>` {#differences-from-transition}

Hello from the modal!

`<TransitionGroup>` supports the same props, CSS transition classes, and JavaScript hook listeners as `<Transition>`, with the following differences:

- By default, it doesn't render a wrapper element. But you can specify an element to be rendered with the `tag` prop.
- `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.
- CSS transition classes will be applied to individual elements in the list, **not** to the group / container itself.

:::tip When used in [DOM templates](#), it should be referenced as `<transition-group>`. :::

## Enter / Leave Transitions

Here is an example of applying enter / leave transitions to a `v-for` list using `<TransitionGroup>`:

```
<TransitionGroup name="list" tag="ul">
  <li v-for="item in items" :key="item">
    {{ item }}
  </li>
</TransitionGroup>
```

```
.list-enter-active,
.list-leave-active {
  transition: all 0.5s ease;
}
.list-enter-from,
.list-leave-to {
  opacity: 0;
  transform: translateX(30px);
}
```

## Move Transitions

The above demo has some obvious flaws: when an item is inserted or removed, its surrounding items instantly "jump" into place instead of moving smoothly. We can fix this by adding a few additional CSS rules:

```
.list-move, /* apply transition to moving elements */
.list-enter-active,
.list-leave-active {
  transition: all 0.5s ease;
}

.list-enter-from,
.list-leave-to {
  opacity: 0;
  transform: translateX(30px);
}

/* ensure leaving items are taken out of layout flow so that moving
   animations can be calculated correctly. */
.list-leave-active {
  position: absolute;
}
```

Now it looks much better - even animating smoothly when the whole list is shuffled:

[Full Example](#)

## Staggering List Transitions

By communicating with JavaScript transitions through data attributes, it's also possible to stagger transitions in a list. First, we render the index of an item as a data attribute on the DOM element:

```
<TransitionGroup
  tag="ul"
  :css="false"
  @before-enter="onBeforeEnter"
  @enter="onEnter"
  @leave="onLeave"
>
  <li
    v-for="(item, index) in computedList"
    :key="item.msg"
    :data-index="index"
  >
    {{ item.msg }}
  </li>
</TransitionGroup>
```

Then, in JavaScript hooks, we animate the element with a delay based on the data attribute. This example is using the [GreenSock library](#) to perform the animation:

```
function onEnter(el, done) {
  gsap.to(el, {
    opacity: 1,
    height: '1.6em',
    delay: el.dataset.index * 0.15,
    onComplete: done
  })
}
```

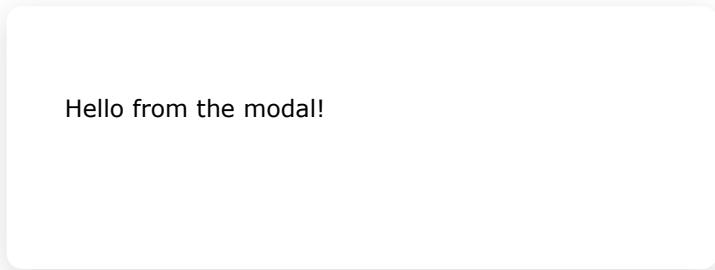
[Full Example in the Playground](#)

## Full Example in the Playground

---

### Related

- `<TransitionGroup>` [API reference](#)



Hello from the modal!

# Transition

Vue offers two built-in components that can help work with transitions and animations in response to changing state:

- `<Transition>` for applying animations when an element or component is entering and leaving the DOM. This is covered on this page.
- `<TransitionGroup>` for applying Hello from the modal! an element or component is inserted into, removed from, or moved within a `v-for` list. This is covered in [the next chapter](#).

Aside from these two components, we can also apply animations in Vue using other techniques such as toggling CSS classes or state-driven animations via style bindings. These additional techniques are covered in the [Animation Techniques](#) chapter.

## The `<Transition>` Component

`<Transition>` is a built-in component: this means it is available in any component's template without having to register it. It can be used to apply enter and leave animations on elements or components passed to it via its default slot. The enter or leave can be triggered by one of the following:

- Conditional rendering via `v-if`
- Conditional display via `v-show`
- Dynamic components toggling via the `<component>` special element

This is an example of the most basic usage:

```
<button @click="show = !show">Toggle</button>
<Transition>
  <p v-if="show">hello</p>
</Transition>
```

```
/* we will explain what these classes do next! */
.v-enter-active,
.v-leave-active {
  transition: opacity 0.5s ease;
}

.v-enter-from,
.v-leave-to {
  opacity: 0;
}
```

[Try it in the Playground](#)

[Try it in the Playground](#)

:::tip `<Transition>` only supports a single element or component as its slot content. If the content is a component, the component must also have only one single root element. :::

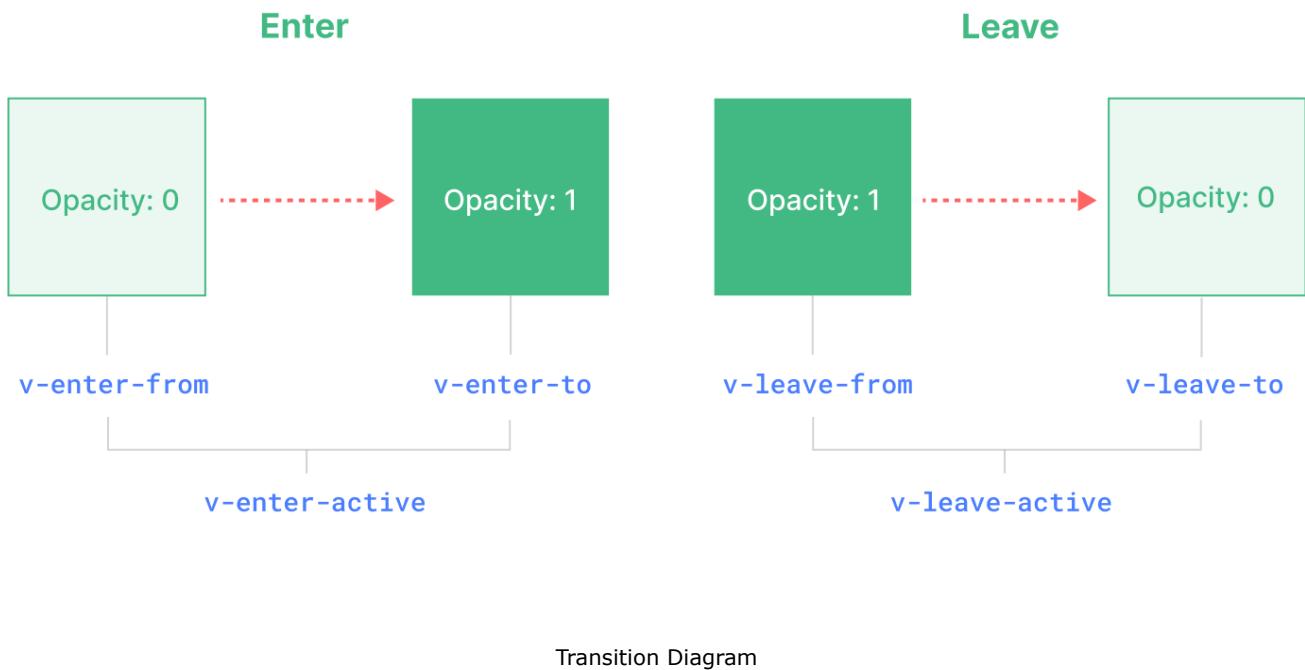
When an element in a `<Transition>` component is inserted or removed, this is what happens:

1. Vue will automatically sniff whether the target element has CSS transitions or animations applied. If it does, a number of [CSS transition classes](#) will be added / removed at appropriate timings.
2. If there are listeners for [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 on the browser's next animation frame.

## CSS-Based Transitions

### Transition Classes

There are six classes applied for enter / leave transitions.



1. `v-enter-from`: Starting state for enter. Added before the element is inserted, removed one frame after the element is inserted.
2. `v-enter-active`: Active state for enter. Applied during the entire entering phase. Added before the element is inserted, removed when the transition/animation finishes. This class can be used to define the duration, delay and easing curve for the entering transition.
3. `v-enter-to`: Ending state for enter. Added one frame after the element is inserted (at the same time `v-enter-from` is removed), removed when the transition/animation finishes.

4. `v-leave-from`: 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 a 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`: Ending state for leave. Added one frame after a leaving transition is triggered (at the same time `v-leave-from` is removed), removed when the transition/animation finishes.

`v-enter-active` and `v-leave-active` from the modal give the modality to specify different easing curves for enter / leave transitions, which we'll see an example of in the following sections.

## Named Transitions

A transition can be named via the `name` prop:

```
<Transition name="fade">
  ...
</Transition>
```

For a named transition, its transition classes will be prefixed with its name instead of `v`. For example, the applied class for the above transition will be `fade-enter-active` instead of `v-enter-active`. The CSS for the fade transition should look like this:

```
.fade-enter-active,
.fade-leave-active {
  transition: opacity 0.5s ease;
}

.fade-enter-from,
.fade-leave-to {
  opacity: 0;
}
```

## CSS Transitions

`<Transition>` is most commonly used in combination with [native CSS transitions](#), as seen in the basic example above. The `transition` CSS property is a shorthand that allows us to specify multiple aspects of a transition, including properties that should be animated, duration of the transition, and [easing curves](#).

Here is a more advanced example that transitions multiple properties, with different durations and easing curves for enter and leave:

```
<Transition name="slide-fade">
  <p v-if="show">hello</p>
</Transition>
```

```
/*
  Enter and leave animations can use different
  durations and timing functions.
*/
.slide-fade-enter-active {
```

```

    transition: all 0.3s ease-out;
}

.slide-fade-leave-active {
  transition: all 0.8s cubic-bezier(1, 0.5, 0.8, 1);
}

.slide-fade-enter-from,
.slide-fade-leave-to {
  transform: translateX(20px);
  opacity: 0;
}

```

Hello from the modal!

[Try it in the Playground](#)

[Try it in the Playground](#)

## CSS Animations

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

For most CSS animations, we can simply declare them under the `*-enter-active` and `*-leave-active` classes. Here's an example:

```

<Transition name="bounce">
  <p v-if="show" style="text-align: center;">
    Hello here is some bouncy text!
  </p>
</Transition>

```

```

.bounce-enter-active {
  animation: bounce-in 0.5s;
}
.bounce-leave-active {
  animation: bounce-in 0.5s reverse;
}
@keyframes bounce-in {
  0% {
    transform: scale(0);
  }
  50% {
    transform: scale(1.25);
  }
  100% {
    transform: scale(1);
  }
}

```

[Try it in the Playground](#)

[Try it in the Playground](#)

## Custom Transition Classes

You can also specify custom transition classes by passing the following props to `<Transition>`:

- `enter-from-class`
- `enter-active-class`
- `enter-to-class`
- `leave-from-class`
- `leave-active-class`
- `leave-to-class`

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](#):

Hello from the modal!

```
<!-- assuming Animate.css is included on the page -->
<Transition
  name="custom-classes"
  enter-active-class="animate__animated animate__tada"
  leave-active-class="animate__animated animate__bounceOutRight"
>
  <p v-if="show">hello</p>
</Transition>
```

[Try it in the Playground](#)

[Try it in the Playground](#)

## 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 by passing the `type` prop, with a value of either `animation` or `transition`:

```
<Transition type="animation">...</Transition>
```

## Nested Transitions and Explicit Transition Durations

Although the transition classes are only applied to the direct child element in `<Transition>`, we can transition nested elements using nested CSS selectors:

```
<Transition name="nested">
  <div v-if="show" class="outer">
    <div class="inner">
      Hello
    </div>
  </div>
</Transition>
```

```
/* rules that target nested elements */
.nested-enter-active .inner,
.nested-leave-active .inner {
```

```

    transition: all 0.3s ease-in-out;
}

.nested-enter-from .inner,
.nested-leave-to .inner {
  transform: translateX(30px);
  opacity: 0;
}

/* ... other necessary CSS omitted */

```

We can even add a transition delay to the nested element on enter, which creates a staggered enter animation sequence:

```

/* delay enter of nested element for staggered effect */
.nested-enter-active .inner {
  transition-delay: 0.25s;
}

```

However, this creates a small issue. By default, the `<Transition>` component attempts to automatically figure out when the transition has finished by listening to the `first transitionend` or `animationend` event on the root transition element. With a nested transition, the desired behavior should be waiting until the transitions of all inner elements have finished.

In such cases you can specify an explicit transition duration (in milliseconds) using the `duration` prop on the `<transition>` component. The total duration should match the delay plus transition duration of the inner element:

```
<Transition :duration="550">...</Transition>
```

[Try it in the Playground](#)

If necessary, you can also specify separate values for enter and leave durations using an object:

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

## Performance Considerations

You may notice that the animations shown above are mostly using properties like `transform` and `opacity`. These properties are efficient to animate because:

1. They do not affect the document layout during the animation, so they do not trigger expensive CSS layout calculation on every animation frame.
2. Most modern browsers can leverage GPU hardware acceleration when animating `transform`.

In comparison, properties like `height` or `margin` will trigger CSS layout, so they are much more expensive to animate, and should be used with caution. We can check resources like [CSS-Triggers](#) to see which properties will trigger layout if we animate them.

## JavaScript Hooks

You can hook into the transition process with JavaScript by listening to events on the `<Transition>` component:

```
<Transition
  @before-enter="onBeforeEnter"
  @enter="onEnter"
  @after-enter="onAfterEnter"
  @enter-cancelled="onEnterCancelled"
  @before-leave="onBeforeLeave"
  @leave="onLeave"
  @after-leave="onAfterLeave"
  @leave-cancelled="onLeaveCancelled"
>
  <!-- ... -->
</Transition>
```

```
// called before the element is inserted into the DOM.
// use this to set the "enter-from" state of the element
function onBeforeEnter(el) {}

// called one frame after the element is inserted.
// use this to start the entering animation.
function onEnter(el, done) {
  // call the done callback to indicate transition end
  // optional if used in combination with CSS
  done()
}

// called when the enter transition has finished.
function onAfterEnter(el) {}
function onEnterCancelled(el) {}

// called before the leave hook.
// Most of the time, you should just use the leave hook
function onBeforeLeave(el) {}

// called when the leave transition starts.
// use this to start the leaving animation.
function onLeave(el, done) {
  // call the done callback to indicate transition end
  // optional if used in combination with CSS
  done()
}

// called when the leave transition has finished and the
// element has been removed from the DOM.
function onAfterLeave(el) {}

// only available with v-show transitions
function onLeaveCancelled(el) {}
```

```
export default {
  // ...
  methods: {
    // called before the element is inserted into the DOM.
    // use this to set the "enter-from" state of the element
    onBeforeEnter(el) {},
```

```

// called one frame after the element is inserted.
// use this to start the animation.
onEnter(el, done) {
  // call the done callback to indicate transition end
  // optional if used in combination with CSS
  done()
},

// called when the enter transition has finished.
onAfterEnter(el) {},
onEnterCancelled(el) {},


// called before the leave hook.
// Most of the time, you should just use the leave hook.
onBeforeLeave(el),


// called when the leave transition starts.
// use this to start the leaving animation.
onLeave(el, done) {
  // call the done callback to indicate transition end
  // optional if used in combination with CSS
  done()
},


// called when the leave transition has finished and the
// element has been removed from the DOM.
onAfterLeave(el) {},


// only available with v-show transitions
onLeaveCancelled(el) {}
}
}

```

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

When using JavaScript-only transitions, it is usually a good idea to add the `:css="false"` prop. This explicitly tells Vue to skip auto CSS transition detection. Aside from being slightly more performant, this also prevents CSS rules from accidentally interfering with the transition:

```

<Transition
  ...
  :css="false"
>
  ...
</Transition>

```

With `:css="false"`, we are also fully responsible for controlling when the transition ends. In this case, the `done` callbacks are required for the `@enter` and `@leave` hooks. Otherwise, the hooks will be called synchronously and the transition will finish immediately.

Here's a demo using the [GreenSock library](#) to perform the animations. You can, of course, use any other animation library you want, for example [Anime.js](#) or [Motion One](#).

[Try it in the Playground](#)

[Try it in the Playground](#)

## Reusable Transitions

Transitions can be reused through Vue's component system. To create a reusable transition, we can create a component that wraps the `<Transition>` component and passes down the slot content:

```
<!-- MyTransition.vue -->
<script>
// JavaScript hooks logic...
</script>

<template>
<!-- wrap the built-in Transition component -->
<Transition
  name="my-transition"
  @enter="onEnter"
  @leave="onLeave">
  <slot></slot> <!-- pass down slot content -->
</Transition>
</template>

<style>
/*
  Necessary CSS...
  Note: avoid using <style scoped> here since it
  does not apply to slot content.
*/
</style>
```

Now `MyTransition` can be imported and used just like the built-in version:

```
<MyTransition>
  <div v-if="show">Hello</div>
</MyTransition>
```

## Transition on Appear

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

```
<Transition appear>
  ...
</Transition>
```

## Transition Between Elements

In addition to toggling an element with `v-if` / `v-show`, we can also transition between two elements using `v-if` / `v-else` / `v-else-if`, as long as we make sure that there is only one element being shown at any given moment:

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

[Try it in the Playground](#)

## Transition Modes

In the previous example, the entering and leaving elements are animated at the same time, and we had to make them `position: absolute` to avoid the layout issue when both elements are present in the DOM.

However, in some cases this isn't an option, or simply isn't the desired behavior. We may want the leaving element to be animated out first, and for the entering element to only be inserted **after** the leaving animation has finished. Orchestrating such animations manually would be very complicated - luckily, we can enable this behavior by passing `<Transition>` a `mode` prop:

```
<Transition mode="out-in">
  ...
</Transition>
```

Here's the previous demo with `mode="out-in"`:

`<Transition>` also supports `mode="in-out"`, although it's much less frequently used.

## Transition Between Components

`<Transition>` can also be used around [dynamic components](#):

```
<Transition name="fade" mode="out-in">
  <component :is="activeComponent"></component>
</Transition>
```

[Try it in the Playground](#)

[Try it in the Playground](#)

## Dynamic Transitions

`<Transition>` props like `name` can also be dynamic! It allows us to dynamically apply different transitions based on state change:

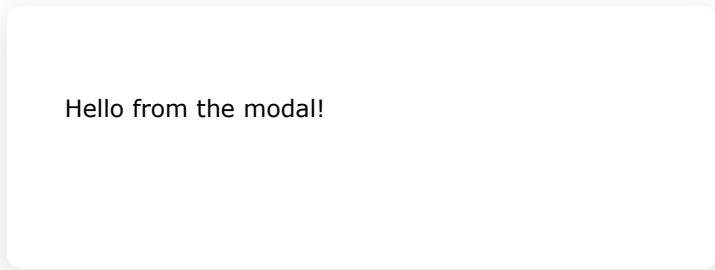
```
<Transition :name="transitionName">
  <!-- ... -->
</Transition>
```

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

You can also apply different behavior in JavaScript transition hooks based on the current state of your component. Finally, the ultimate way of creating dynamic transitions is through [reusable transition 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.

## Related

- <Transition> [API reference](#)



Hello from the modal!

---

[Go to TOC](#)

# Async Components

## Basic Usage

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 possible, Vue has a `defineAsyncComponent` function:

```
import { defineAsyncComponent } from 'vue'

const AsyncComp = defineAsyncComponent(() => {
  return new Promise((resolve, reject) => {
    // ...load component from server
    resolve(/* loaded component */)
  })
})
// ... use `AsyncComp` like a normal component
```

As you can see, `defineAsyncComponent` accepts a loader function that returns a Promise. The Promise's `resolve` callback 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.

`ES module dynamic import` also returns a Promise, so most of the time we will use it in combination with `defineAsyncComponent`. Bundlers like Vite and webpack also support the syntax (and will use it as bundle split points), so we can use it to import Vue SFCs:

```
import { defineAsyncComponent } from 'vue'

const AsyncComp = defineAsyncComponent(() =>
  import('./components/MyComponent.vue')
)
```

The resulting `AsyncComp` is a wrapper component that only calls the loader function when it is actually rendered on the page. In addition, it will pass along any props and slots to the inner component, so you can use the async wrapper to seamlessly replace the original component while achieving lazy loading.

As with normal components, async components can be registered globally using `app.component()`:

```
app.component('MyComponent', defineAsyncComponent(() =>
  import('./components/MyComponent.vue')
))
```

You can also use `defineAsyncComponent` when registering a component locally:

```
<script>
import { defineAsyncComponent } from 'vue'

export default {
  components: {
    AdminPage: defineAsyncComponent(() =>
      import('./components/AdminPageComponent.vue')
    )
  }
}
```

```

        }
    }
</script>

<template>
    <AdminPage />
</template>

```

They can also be defined directly inside their parent component:

```

<script setup>
import { defineAsyncComponent } from 'vue'

const AdminPage = defineAsyncComponent(() =>
    import('./components/AdminPageComponent.vue')
)
</script>

<template>
    <AdminPage />
</template>

```

## Loading and Error States

Asynchronous operations inevitably involve loading and error states - `defineAsyncComponent()` supports handling these states via advanced options:

```

const AsyncComp = defineAsyncComponent({
    // the loader function
    loader: () => import('./Foo.vue'),

    // A component to use while the async component is loading
    loadingComponent: LoadingComponent,
    // Delay before showing the loading component. Default: 200ms.
    delay: 200,

    // A component to use if the load fails
    errorComponent: ErrorComponent,
    // The error component will be displayed if a timeout is
    // provided and exceeded. Default: Infinity.
    timeout: 3000
})

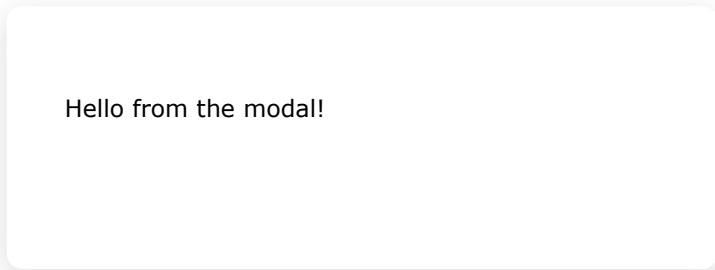
```

If a loading component is provided, it will be displayed first while the inner component is being loaded. There is a default 200ms delay before the loading component is shown - this is because on fast networks, an instant loading state may get replaced too fast and end up looking like a flicker.

If an error component is provided, it will be displayed when the Promise returned by the loader function is rejected. You can also specify a timeout to show the error component when the request is taking too long.

## Using with Suspense

Async components can be used with the `<Suspense>` built-in component. The interaction between `<Suspense>` and async components is documented in the [dedicated chapter for `<Suspense>`](#).



Hello from the modal!

# Fallthrough Attributes

This page assumes you've already read the [Components Basics](#). Read that first if you are new to components.

## Attribute Inheritance

click me from the modal!

A "fallthrough attribute" is an attribute or `v-on` event listener that is passed to a component, but is not explicitly declared in the receiving component's `props` or `emits`. Common examples of this include `class`, `style`, and `id` attributes.

When a component renders a single root element, fallthrough attributes will be automatically added to the root element's attributes. For example, given a `<MyButton>` component with the following template:

```
<!-- template of <MyButton> -->
<button>click me</button>
```

And a parent using this component with:

```
<MyButton class="large" />
```

The final rendered DOM would be:

```
<button class="large">click me</button>
```

Here, `<MyButton>` did not declare `class` as an accepted prop. Therefore, `class` is treated as a fallthrough attribute and automatically added to `<MyButton>`'s root element.

## class and style Merging

If the child component's root element already has existing `class` or `style` attributes, it will be merged with the `class` and `style` values that are inherited from the parent. Suppose we change the template of `<MyButton>` in the previous example to:

```
<!-- template of <MyButton> -->
<button class="btn">click me</button>
```

Then the final rendered DOM would now become:

```
<button class="btn large">click me</button>
```

## v-on Listener Inheritance

The same rule applies to `v-on` event listeners:

```
<MyButton @click="onClick" />
```

The `click` listener will be added to the root element of `<MyButton>`, i.e. the native `<button>` element. When the native `<button>` is clicked, it will trigger the `onClick` method of the parent component. If the native `<button>` already has a `click` listener bound with `v-on`, then both listeners will trigger.

## Nested Component Inheritance

If a component renders another component as its root node, for example, we refactored `<MyButton>` to render a `<BaseButton>` as its root:

Hello from the modal!

```
<!-- template of <MyButton/> that simply renders another component -->
<BaseButton />
```

Then the fallthrough attributes received by `<MyButton>` will be automatically forwarded to `<BaseButton>`.

Note that:

1. Forwarded attributes do not include any attributes that are declared as props, or `v-on` listeners of declared events by `<MyButton>` - in other words, the declared props and listeners have been "consumed" by `<MyButton>`.
2. Forwarded attributes may be accepted as props by `<BaseButton>`, if declared by it.

## Disabling Attribute Inheritance

If you do **not** want a component to automatically inherit attributes, you can set `inheritAttrs: false` in the component's options.

If using `<script setup>`, you will need to declare this option using a separate, normal `<script>` block:

```
<script>
// use normal <script> to declare options
export default {
  inheritAttrs: false
}
</script>

<script setup>
// ...setup logic
</script>
```

The common scenario for disabling attribute inheritance is when attributes need to be applied to other elements besides the root node. By setting the `inheritAttrs` option to `false`, you can take full control over where the fallthrough attributes should be applied.

These fallthrough attributes can be accessed directly in template expressions as `$attrs`:

```
<span>Fallthrough attributes: {{ $attrs }}</span>
```

The `$attrs` object includes all attributes that are not declared by the component's `props` or `emits` options (e.g., `class`, `style`, `v-on` listeners, etc.).

Some notes:

- Unlike props, fallthrough attributes preserve their original casing in JavaScript, so an attribute like `foo-bar` needs to be accessed as `$attrs['foo-bar']`.
- A `v-on` event listener like `@click` will be exposed on the object as a function under `$attrs.onClick`.

Using our `<MyButton>` component example from the [previous section](#) - sometimes we may need to wrap the actual `<button>` element with an extra `<div>` for styling purposes:

```
<div class="btn-wrapper">
  <button class="btn">click me</button>
</div>
```

We want all fallthrough attributes like `class` and `v-on` listeners to be applied to the inner `<button>`, not the outer `<div>`. We can achieve this with `inheritAttrs: false` and `v-bind="$attrs"`:

```
<div class="btn-wrapper">
  <button class="btn" v-bind="$attrs">click me</button>
</div>
```

Remember that `v-bind` [without an argument](#) binds all the properties of an object as attributes of the target element.

## Attribute Inheritance on Multiple Root Nodes

Unlike components with a single root node, components with multiple root nodes do not have an automatic attribute fallthrough behavior. If `$attrs` are not bound explicitly, a runtime warning will be issued.

```
<CustomLayout id="custom-layout" @click="changeValue" />
```

If `<CustomLayout>` has the following multi-root template, there will be a warning because Vue cannot be sure where to apply the fallthrough attributes:

```
<header>...</header>
<main>...</main>
<footer>...</footer>
```

The warning will be suppressed if `$attrs` is explicitly bound:

```
<header>...</header>
<main v-bind="$attrs">...</main>
<footer>...</footer>
```

# Accessing Fallthrough Attributes in JavaScript

If needed, you can access a component's fallthrough attributes in `<script setup>` using the `useAttrs()` API:

```
<script setup>
import { useAttrs } from 'vue'

const attrs = useAttrs()
</script>
```

Hello from the modal!

If not using `<script setup>`, `attrs` will be exposed as a property of the `setup()` context:

```
export default {
  setup(props, ctx) {
    // fallthrough attributes are exposed as ctx.attrs
    console.log(ctx.attrs)
  }
}
```

Note that although the `attrs` object here always reflects the latest fallthrough attributes, it isn't reactive (for performance reasons). You cannot use watchers to observe its changes. If you need reactivity, use a prop. Alternatively, you can use `onUpdated()` to perform side effects with the latest `attrs` on each update.

If needed, you can access a component's fallthrough attributes via the `$attrs` instance property:

```
export default {
  created() {
    console.log(this.$attrs)
  }
}
```

# Component Events

This page assumes you've already read the [Components Basics](#). Read that first if you are new to components.

## Emitting and Listening to Events

A component can emit custom events directly in template expressions (e.g. in a `v-on` handler) using the built-in `$emit` method:

```
<!-- MyComponent -->
<button @click="$emit('someEvent')">click me</button>
```

The `$emit()` method is also available on the component instance as `this.$emit()`:

```
export default {
  methods: {
    submit() {
      this.$emit('someEvent')
    }
  }
}
```

The parent can then listen to it using `v-on`:

```
<MyComponent @some-event="callback" />
```

The `.once` modifier is also supported on component event listeners:

```
<MyComponent @some-event.once="callback" />
```

Like components and props, event names provide an automatic case transformation. Notice we emitted a camelCase event, but can listen for it using a kebab-cased listener in the parent. As with [props casing](#), we recommend using kebab-cased event listeners in templates.

**:::tip** Unlike native DOM events, component emitted events do **not** bubble. You can only listen to the events emitted by a direct child component. If there is a need to communicate between sibling or deeply nested components, use an external event bus or a [global state management solution](#). **:::**

## Event Arguments

It's sometimes useful to emit a specific value with an event. For example, we may want the `<BlogPost>` component to be in charge of how much to enlarge the text by. In those cases, we can pass extra arguments to `$emit` to provide this value:

```
<button @click="$emit('increaseBy', 1)">
  Increase by 1
</button>
```

Then, when we listen to the event in the parent, we can use an inline arrow function as the listener, which allows us to access the event argument:

```
<MyButton @increase-by="(n) => count += n" />
```

Or, if the event handler is a method:

```
<MyButton @increase-by="increaseCount" />
```

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

```
methods: {
  increaseCount(n) {
    this.count += n
  }
}
```

```
function increaseCount(n) {
  count.value += n
}
```

:::tip All extra arguments passed to `$emit()` after the event name will be forwarded to the listener. For example, with `$emit('foo', 1, 2, 3)` the listener function will receive three arguments. :::

## Declaring Emitted Events

Emitted events can be explicitly declared on the component via the `defineEmits()` macro `emits` option:

```
<script setup>
defineEmits(['inFocus', 'submit'])
</script>
```

The `$emit` method that we used in the `<template>` isn't accessible within the `<script setup>` section of a component, but `defineEmits()` returns an equivalent function that we can use instead:

```
<script setup>
const emit = defineEmits(['inFocus', 'submit'])

function buttonClick() {
  emit('submit')
}
</script>
```

The `defineEmits()` macro **cannot** be used inside a function, it must be placed directly within `<script setup>`, as in the example above.

If you're using an explicit `setup` function instead of `<script setup>`, events should be declared using the `emits` option, and the `emit` function is exposed on the `setup()` context:

```
export default {
  emits: ['inFocus', 'submit'],
  setup(props, ctx) {
    ctx.emit('submit')
  }
}
```

As with other properties of the `setup()` context, `emit` can safely be destructured:

```
export default {
  emits: ['inFocus', 'submit'],
  setup(props, { emit }) {
    emit('submit')
  }
}
```

```
export default {
  emits: ['inFocus', 'submit']
}
```

The `emits` option also supports an object syntax, which allows us to perform runtime validation of the payload of the emitted events:

```
<script setup>
const emit = defineEmits({
  submit(payload) {
    // return `true` or `false` to indicate
    // validation pass / fail
  }
})
</script>
```

If you are using TypeScript with `<script setup>`, it's also possible to declare emitted events using pure type annotations:

```
<script setup lang="ts">
const emit = defineEmits<{
  (e: 'change', id: number): void
  (e: 'update', value: string): void
}>()
</script>
```

More details: [Typing Component Emits](#)

```
export default {
  emits: {
    submit(payload) {
      // return `true` or `false` to indicate
      // validation pass / fail
    }
  }
}
```

See also: [Typing Component Emits](#)

Although optional, it is recommended to define all emitted events in order to better document how a component should work. It also allows Vue to exclude known listeners from [fallthrough attributes](#), avoiding edge cases caused by DOM events manually dispatched by 3rd party code.

:::tip If a native event (e.g., `click`) is defined in the `emits` option, the listener will now only listen to component-emitted `click` events and no longer respond to native `click` events. :::

## Events Validation

Similar to prop type validation, an emitted event can be validated if it is defined with the object syntax instead of the array syntax.

To add validation, the event is assigned a function that receives the arguments passed to the `this.$emit emit` call and returns a boolean to indicate whether the event is valid or not.

```
<script setup>
const emit = defineEmits({
  // No validation
  click: null,

  // Validate submit event
  submit: ({ email, password }) => {
    if (email && password) {
      return true
    } else {
      console.warn('Invalid submit event payload!')
      return false
    }
  }
})

function submitForm(email, password) {
  emit('submit', { email, password })
}
</script>
```

```
export default {
  emits: {
    // No validation
    click: null,

    // Validate submit event
    submit: ({ email, password }) => {
      if (email && password) {
        return true
      } else {
        console.warn('Invalid submit event payload!')
        return false
      }
    }
  },
  methods: {
    submitForm(email, password) {
      this.$emit('submit', { email, password })
    }
  }
}
```

## Usage with `v-model` {#usage-with-v-model}

Custom events can also be used to create custom inputs that work with `v-model`. Let's revisit how `v-model` is used on a native element:

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

Under the hood, the template compiler expands `v-model` to the more verbose equivalent for us. So the above code does the same as the following:

Hello from the modal!

```
<input  
  :value="searchText"  
  @input="searchText = $event.target.value"  
/>
```

When used on a component, `v-model` instead expands to this:

```
<CustomInput  
  :modelValue="searchText"  
  @update:modelValue="newValue => searchText = newValue"  
/>
```

For this to actually work though, the `<CustomInput>` component must do two things:

1. Bind the `value` attribute of a native `<input>` element to the `modelValue` prop
2. When a native `input` event is triggered, emit an `update:modelValue` custom event with the new value

Here's that in action:

```
<!-- CustomInput.vue -->  
<script>  
export default {  
  props: ['modelValue'],  
  emits: ['update:modelValue']  
}  
</script>  
  
<template>  
  <input  
    :value="modelValue"  
    @input="$emit('update:modelValue', $event.target.value)"  
  />  
</template>
```

```
<!-- CustomInput.vue -->  
<script setup>  
defineProps(['modelValue'])  
defineEmits(['update:modelValue'])  
</script>  
  
<template>  
  <input  
    :value="modelValue"  
    @input="$emit('update:modelValue', $event.target.value)"  
  />  
</template>
```

Now `v-model` should work perfectly with this component:

```
<CustomInput v-model="searchText" />
```

[Try it in the Playground](#)

[Try it in the Playground](#)

Another way of implementing `v-model` within this component is to use a writable `computed` property with both a getter and a setter. The `get` method should return the `modelValue` property and the `set` method should emit the corresponding `eventHello` from the modal!

```
<!-- CustomInput.vue -->
<script>
export default {
  props: ['modelValue'],
  emits: ['update:modelValue'],
  computed: {
    value: {
      get() {
        return this.modelValue
      },
      set(value) {
        this.$emit('update:modelValue', value)
      }
    }
  }
</script>

<template>
  <input v-model="value" />
</template>
```

```
<!-- CustomInput.vue -->
<script setup>
import { computed } from 'vue'

const props = defineProps(['modelValue'])
const emit = defineEmits(['update:modelValue'])

const value = computed({
  get() {
    return props.modelValue
  },
  set(value) {
    emit('update:modelValue', value)
  }
})
</script>

<template>
  <input v-model="value" />
</template>
```

## v-model arguments

By default, `v-model` on a component uses `modelValue` as the prop and `update:modelValue` as the event. We can modify these names passing an argument to `v-model`:

```
<MyComponent v-model:title="bookTitle" />
```

In this case, the child component should expect a `title` prop and emit an `update:title` event to update the parent value:

```
<!-- MyComponent.vue -->
<script setup>
defineProps(['title'])
defineEmits(['update:title'])
</script>

<template>
  <input
    type="text"
    :value="title"
    @input="$emit('update:title', $event.target.value)"
  />
</template>
```

[Try it in the Playground](#)

```
<!-- MyComponent.vue -->
<script>
export default {
  props: ['title'],
  emits: ['update:title']
}
</script>

<template>
  <input
    type="text"
    :value="title"
    @input="$emit('update:title', $event.target.value)"
  />
</template>
```

[Try it in the Playground](#)

## Multiple v-model bindings

By leveraging the ability to target a particular prop and event as we learned before with [v-model arguments](#), we can now create multiple v-model bindings on a single component instance.

Each v-model will sync to a different prop, without the need for extra options in the component:

```
<UserName
  v-model:first-name="first"
  v-model:last-name="last"
/>
```

```
<script setup>
defineProps({
  firstName: String,
  lastName: String
})

defineEmits(['update:firstName', 'update:lastName'])
</script>

<template>
  <input
    type="text"
    :value="firstName"
    @input="$emit('update:firstName', $event.target.value)"
  />
  <input
    type="text"
    :value="lastName"
    @input="$emit('update:lastName', $event.target.value)"
  />
</template>
```

[Try it in the Playground](#)

```
<script>
export default {
  props: {
    firstName: String,
    lastName: String
  },
  emits: ['update:firstName', 'update:lastName']
}
</script>

<template>
  <input
    type="text"
    :value="firstName"
    @input="$emit('update:firstName', $event.target.value)"
  />
  <input
    type="text"
    :value="lastName"
    @input="$emit('update:lastName', $event.target.value)"
  />
</template>
```

[Try it in the Playground](#)

## Handling `v-model` modifiers

When we were learning about form input bindings, we saw that `v-model` has [built-in modifiers](#) - `.trim`, `.number` and `.lazy`. In some cases, you might also want the `v-model` on your custom input component to support custom modifiers.

Let's create an example custom modifier, `capitalize`, that capitalizes the first letter of the string provided by the `v-model` binding:

```
<MyComponent v-model.capitalize="myText" />
```

Modifiers added to a component `v-model` will be provided to the component via the `modelModifiers` prop. In the below example, we have created a component that contains a `modelModifiers` prop that defaults to an empty object:

```
<script setup>
const props = defineProps({
  modelValue: String,
  modelModifiers: { default: () => ({}), }
})

defineEmits(['update:modelValue'])

console.log(props.modelModifiers) // { capitalize: true }
</script>

<template>
  <input
    type="text"
    :value="modelValue"
    @input="$emit('update:modelValue', $event.target.value)"
  />
</template>
```

```
<script>
export default {
  props: {
    modelValue: String,
    modelModifiers: {
      default: () => ({}),
    },
  },
  emits: ['update:modelValue'],
  created() {
    console.log(this.modelModifiers) // { capitalize: true }
  }
}
</script>

<template>
  <input
    type="text"
    :value="modelValue"
    @input="$emit('update:modelValue', $event.target.value)"
  />
</template>
```

Notice the component's `modelModifiers` prop contains `capitalize` and its value is `true` - due to it being set on the `v-model` binding `v-model.capitalize="myText"`.

Now that we have our prop set up, we can check the `modelModifiers` object keys and write a handler to change the emitted value. In the code below we will capitalize the string whenever the `<input />` element fires an `input` event.

```
<script setup>
const props = defineProps({
  modelValue: String,
```

```

    modelModifiers: { default: () => ({})}
  })

const emit = defineEmits(['update:modelValue'])

function emitValue(e) {
  let value = e.target.value
  if (props.modelModifiers.capitalize) {
    value = value.charAt(0).toUpperCase() + value.slice(1)
  }
  emit('update:modelValue', value)
}
</script>

<template>
  <input type="text" :value="modelValue" @input="emitValue" />
</template>

```

[Try it in the Playground](#)

```

<script>
export default {
  props: {
    modelValue: String,
    modelModifiers: {
      default: () => ({})
    },
    emits: ['update:modelValue'],
    methods: {
      emitValue(e) {
        let value = e.target.value
        if (this.modelModifiers.capitalize) {
          value = value.charAt(0).toUpperCase() + value.slice(1)
        }
        this.$emit('update:modelValue', value)
      }
    }
  }
</script>

<template>
  <input type="text" :value="modelValue" @input="emitValue" />
</template>

```

[Try it in the Playground](#)

For `v-model` bindings with both argument and modifiers, the generated prop name will be `arg + "Modifiers"`. For example:

```
<MyComponent v-model:title.capitalize="myText">
```

The corresponding declarations should be:

```

const props = defineProps(['title', 'titleModifiers'])
defineEmits(['update:title'])

console.log(props.titleModifiers) // { capitalize: true }

```

```
export default {
  props: ['title', 'titleModifiers'],
  emits: ['update:title'],
  created() {
    console.log(this.titleModifiers) // { capitalize: true }
  }
}
```

Hello from the modal!

# Props

This page assumes you've already read the [Components Basics](#). Read that first if you are new to components.

## Props Declaration

Hello from the modal!

Vue components require explicit props declaration so that Vue knows what external props passed to the component should be treated as fallthrough attributes (which will be discussed in [its dedicated section](#)).

In SFCs using `<script setup>`, props can be declared using the `defineProps()` macro:

```
<script setup>
const props = defineProps(['foo'])

console.log(props.foo)
</script>
```

In non-`<script setup>` components, props are declared using the `props` option:

```
export default {
  props: ['foo'],
  setup(props) {
    // setup() receives props as the first argument.
    console.log(props.foo)
  }
}
```

Notice the argument passed to `defineProps()` is the same as the value provided to the `props` options: the same props options API is shared between the two declaration styles.

Props are declared using the `props` option:

```
export default {
  props: ['foo'],
  created() {
    // props are exposed on `this`
    console.log(this.foo)
  }
}
```

In addition to declaring props using an array of strings, we can also use the object syntax:

```
export default {
  props: {
    title: String,
    likes: Number
  }
}
```

```
// in <script setup>
defineProps({
  title: String,
  likes: Number
})

// in non-<script setup>
export default {
  props: {
    title: String,
    likes: Number
  }
}
```

For each property in the object declaration syntax, the key is the name of the prop, while the value should be the constructor function of the expected type.

This not only documents your component, but will also warn other developers using your component in the browser console if they pass the wrong type. We will discuss more details about [prop validation](#) further down this page.

See also: [Typing Component Props](#)

If you are using TypeScript with `<script setup>`, it's also possible to declare props using pure type annotations:

```
<script setup lang="ts">
defineProps<{
  title?: string
  likes?: number
}>()
</script>
```

More details: [Typing Component Props](#)

## Prop Passing Details

### Prop Name Casing

We declare long prop names using camelCase because this avoids having to use quotes when using them as property keys, and allows us to reference them directly in template expressions because they are valid JavaScript identifiers:

```
defineProps({
  greetingMessage: String
})

export default {
  props: {
    greetingMessage: String
  }
}

<span>{{ greetingMessage }}</span>
```

Technically, you can also use camelCase when passing props to a child component (except in [DOM templates](#)). However, the convention is using kebab-case in all cases to align with HTML attributes:

```
<MyComponent greeting-message="hello" />
```

We use [PascalCase for component tags](#) when possible because it improves template readability by differentiating Vue components from native elements. However, there isn't as much practical benefit in using camelCase when passing props, so we choose to follow each language's conventions.

## Static vs. Dynamic Props

Hello from the modal!

So far, you've seen props passed as static values, like in:

```
<BlogPost title="My journey with Vue" />
```

You've also seen props assigned dynamically with `v-bind` or its `:` shortcut, such as in:

```
<!-- Dynamically assign the value of a variable -->
<BlogPost :title="post.title" />

<!-- Dynamically assign the value of a complex expression -->
<BlogPost :title="post.title + ' by ' + post.author.name" />
```

## Passing Different Value Types

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

### Number

```
<!-- Even though `42` is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string. -->
<BlogPost :likes="42" />

<!-- Dynamically assign to the value of a variable. -->
<BlogPost :likes="post.likes" />
```

### Boolean

```
<!-- Including the prop with no value will imply `true`. -->
<BlogPost is-published />

<!-- Even though `false` is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string. -->
<BlogPost :is-published="false" />

<!-- Dynamically assign to the value of a variable. -->
<BlogPost :is-published="post.isPublished" />
```

### Array

```
<!-- Even though the array is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string. -->
<BlogPost :comment-ids="[234, 266, 273]" />
```

```
<!-- Dynamically assign to the value of a variable. -->
<BlogPost :comment-ids="post.commentIds" />
```

## Object

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

<!-- Dynamically assign to the value of a variable. -->
<BlogPost :author="post.author" />
```

## Binding Multiple Properties Using 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 `:prop-name`). For example, given a `post` object:

```
export default {
  data() {
    return {
      post: {
        id: 1,
        title: 'My Journey with Vue'
      }
    }
}
```

```
const post = {
  id: 1,
  title: 'My Journey with Vue'
```

The following template:

```
<BlogPost v-bind="post" />
```

Will be equivalent to:

```
<BlogPost :id="post.id" :title="post.title" />
```

## 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:

```
const props = defineProps(['foo'])
// ✖️ warning, props are readonly!
props.foo = 'bar'

export default {
  props: ['foo'],
  created() {
    // ✖️ warning, props are readonly!
    this.foo = 'bar'
  }
}
```

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:

```
const props = defineProps(['initialCounter'])

// counter only uses props.initialCounter as the initial value;
// it is disconnected from future prop updates.
const counter = ref(props.initialCounter)

export default {
  props: ['initialCounter'],
  data() {
    return {
      // counter only uses this.initialCounter as the initial value;
      // it is disconnected from future prop updates.
      counter: this.initialCounter
    }
  }
}
```

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

```
const props = defineProps(['size'])

// computed property that auto-updates when the prop changes
const normalizedSize = computed(() => props.size.trim().toLowerCase())

export default {
  props: ['size'],
  computed: {
    // computed property that auto-updates when the prop changes
    normalizedSize() {
      return this.size.trim().toLowerCase()
    }
  }
}
```

## Mutating Object / Array Props

When objects and arrays are passed as props, while the child component cannot mutate the prop binding, it **will** be able to mutate the object or array's nested properties. This is because in JavaScript objects and arrays are passed by reference, and it is unreasonably expensive for Vue to prevent such mutations.

The main drawback of such mutations is that it allows the child component to affect parent state in a way that isn't obvious to the parent component, potentially making it more difficult to reason about the data flow in the future. As a best practice, you should avoid such mutations unless the parent and child are tightly coupled by design. In most cases, the child should [emit an event](#) to let the parent perform the mutation. Hello from the modal!

## Prop Validation

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

To specify prop validations, you can provide an object with validation requirements to the `defineProps()` macro `props` option, instead of an array of strings. For example:

```
defineProps({
  // Basic type check
  // (`null` and `undefined` values will allow any type)
  propA: Number,
  // Multiple possible types
  propB: [String, Number],
  // Required string
  propC: {
    type: String,
    required: true
  },
  // Number with a default value
  propD: {
    type: Number,
    default: 100
  },
  // Object with a default value
  propE: {
    type: Object,
    // Object or array defaults must be returned from
    // a factory function. The function receives the raw
    // props received by the component as the argument.
    default(rawProps) {
      return { message: 'hello' }
    }
  },
  // Custom validator function
  propF: {
    validator(value) {
      // The value must match one of these strings
      return ['success', 'warning', 'danger'].includes(value)
    }
  },
  // Function with a default value
  propG: {
    type: Function,
```

```
// Unlike object or array default, this is not a factory function - this is a
// function to serve as a default value
default() {
  return 'Default function'
}
})
```

:::tip Code inside the `defineProps()` argument **cannot access other variables declared in `<script setup>`**, because the entire expression is moved to an outer function scope when compiled. :::

```
export default {
  props: {
    // Basic type check
    // (`null` and `undefined` values will allow any type)
    propA: Number,
    // Multiple possible types
    propB: [String, Number],
    // Required string
    propC: {
      type: String,
      required: true
    },
    // Number with a default value
    propD: {
      type: Number,
      default: 100
    },
    // Object with a default value
    propE: {
      type: Object,
      // Object or array defaults must be returned from
      // a factory function. The function receives the raw
      // props received by the component as the argument.
      default(rawProps) {
        return { message: 'hello' }
      }
    },
    // Custom validator function
    propF: {
      validator(value) {
        // The value must match one of these strings
        return ['success', 'warning', 'danger'].includes(value)
      }
    },
    // Function with a default value
    propG: {
      type: Function,
      // Unlike object or array default, this is not a factory function - this is
      // a function to serve as a default value
      default() {
        return 'Default function'
      }
    }
}
```

Additional details:

- All props are optional by default, unless `required: true` is specified.

- An absent optional prop other than `Boolean` will have `undefined` value.
- The `Boolean` absent props will be cast to `false`. You can change this by setting a `default` for it — i.e.: `default: undefined` to behave as a non-Boolean prop.
- If a `default` value is specified, it will be used if the resolved prop value is `undefined` — this includes both when the prop is absent, or an explicit `undefined` value is passed.

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

If using [Type-based props declarations](#), Vue will try its best to compile the type annotations into equivalent runtime prop declarations. For example, `defineProps<{ msg: string }>` will be compiled into `{ msg: { type: String, required: true } }`.

::: tip Note 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. :::

## Runtime 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 class or constructor function and the assertion will be made with an `instanceof` check. For example, given the following class:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }
}
```

You could use it as a prop's type:

```
defineProps({
  author: Person
})
```

```
export default {
  props: {
    author: Person
  }
}
```

Vue will use `instanceof Person` to validate whether the value of the `author` prop is indeed an instance of the `Person` class.

## Boolean Casting

Props with `Boolean` type has special casting rules to mimic the behavior of native boolean attributes. Given a `<MyComponent>` with the following declaration:

```
defineProps({
  disabled: Boolean
})
```

```
export default {
  props: {
    disabled: Boolean
  }
}
```

The component can be used like this:

```
<!-- equivalent of passing :disabled="true" -->
<MyComponent disabled />

<!-- equivalent of passing :disabled="false" -->
<MyComponent />
```

When a prop is declared to allow multiple types, e.g.

```
defineProps({
  disabled: [Boolean, Number]
})
```

```
export default {
  props: {
    disabled: [Boolean, Number]
  }
}
```

The casting rules for `Boolean` will apply regardless of type appearance order.

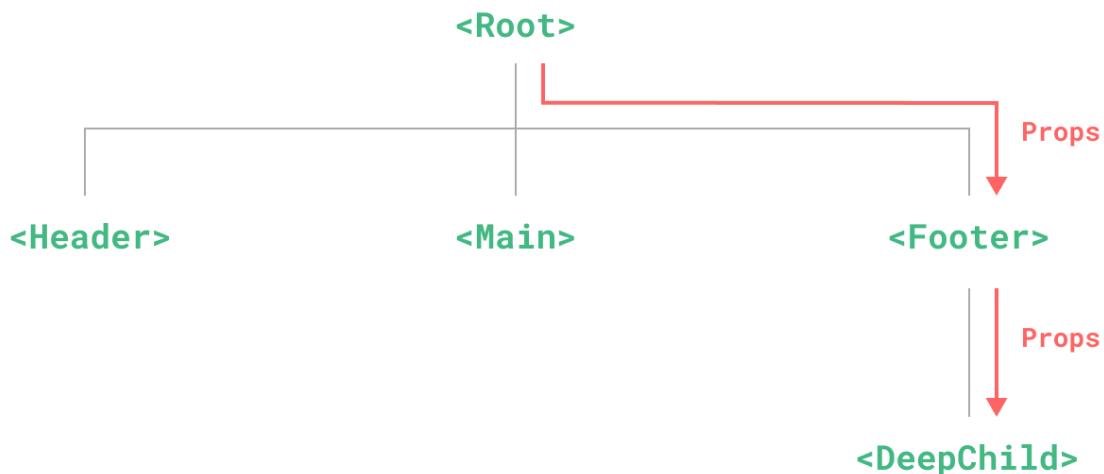
# Provide / Inject

This page assumes you've already read the [Components Basics](#). Read that first if you are new to components.

## Prop Drilling

Hello from the modal!

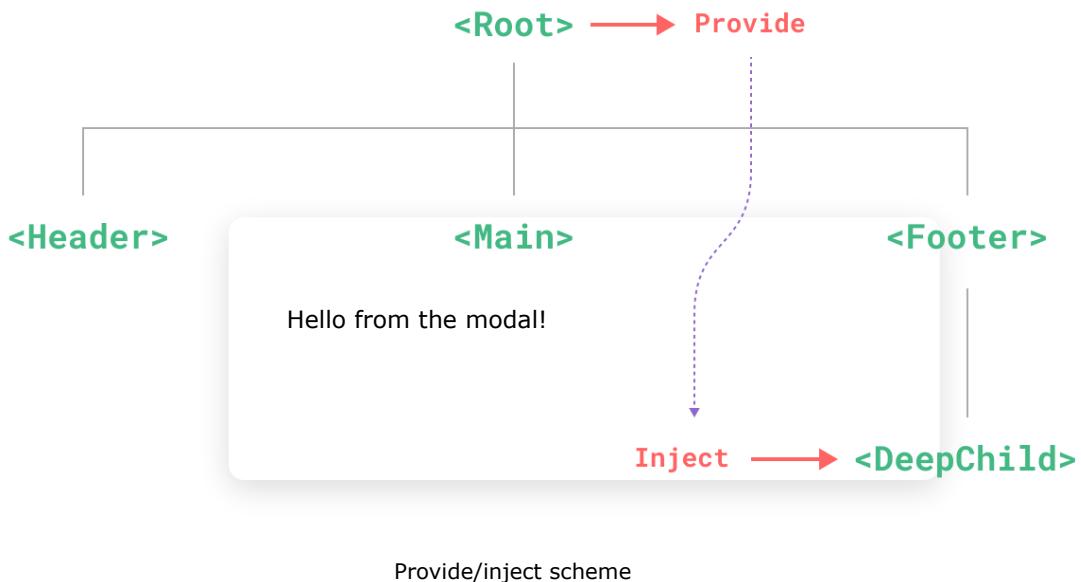
Usually, when we need to pass data from the parent to a child component, we use `props`. However, imagine the case where we have a large component tree, and a deeply nested component needs something from a distant ancestor component. With only props, we would have to pass the same prop across the entire parent chain:



prop drilling diagram

Notice although the `<Footer>` component may not care about these props at all, it still needs to declare and pass them along just so `<DeepChild>` can access them. If there is a longer parent chain, more components would be affected along the way. This is called "props drilling" and definitely isn't fun to deal with.

We can solve props drilling with `provide` and `inject`. A parent component can serve as a **dependency provider** for all its descendants. Any component in the descendant tree, regardless of how deep it is, can **inject** dependencies provided by components up in its parent chain.



## Provide

To provide data to a component's descendants, use the `provide()` function:

```
<script setup>
import { provide } from 'vue'

provide(/* key */ 'message', /* value */ 'hello!')
</script>
```

If not using `<script setup>`, make sure `provide()` is called synchronously inside `setup()`:

```
import { provide } from 'vue'

export default {
  setup() {
    provide(/* key */ 'message', /* value */ 'hello!')
  }
}
```

The `provide()` function accepts two arguments. The first argument is called the **injection key**, which can be a string or a `Symbol`. The injection key is used by descendant components to lookup the desired value to inject. A single component can call `provide()` multiple times with different injection keys to provide different values.

The second argument is the provided value. The value can be of any type, including reactive state such as `refs`:

```
import { ref, provide } from 'vue'

const count = ref(0)
provide('key', count)
```

Providing reactive values allows the descendant components using the provided value to establish a reactive connection to the provider component.

To provide data to a component's descendants, use the `provide` option:

```
export default {
  provide: {
    message: 'hello!'
  }
}
```

For each property in the `provide` object, the key is used by child components to locate the correct value to inject, while the value is what ends up being injected.

If we need to provide per-instance state, for example data declared via the `data()`, then `provide` must use a function value:

```
export default {
  data() {
    return {
      message: 'hello!'
    }
  },
  provide() {
    // use function syntax so that we can access `this`
    return {
      message: this.message
    }
  }
}
```

However, do note this does **not** make the injection reactive. We will discuss [making injections reactive](#) below.

## App-level Provide

In addition to providing data in a component, we can also provide at the app level:

```
import { createApp } from 'vue'
const app = createApp({})
app.provide(/* key */ 'message', /* value */ 'hello!')
```

App-level provides are available to all components rendered in the app. This is especially useful when writing [plugins](#), as plugins typically wouldn't be able to provide values using components.

## Inject

To inject data provided by an ancestor component, use the `inject()` function:

```
<script setup>
import { inject } from 'vue'

const message = inject('message')
</script>
```

If the provided value is a ref, it will be injected as-is and will **not** be automatically unwrapped. This allows the injector component to retain the reactivity connection to the provider component.

#### Full provide + inject Example with Reactivity

Again, if not using `<script setup>`, `inject()` should only be called synchronously inside `setup()`:

```
import { inject } from 'vue'

export default {
  setup() {
    const message = inject('message')
    return { message }
  }
}
```

To inject data provided by an ancestor component, use the `inject` option:

```
export default {
  inject: ['message'],
  created() {
    console.log(this.message) // injected value
  }
}
```

Injections are resolved **before** the component's own state, so you can access injected properties in `data()`:

```
export default {
  inject: ['message'],
  data() {
    return {
      // initial data based on injected value
      fullMessage: this.message
    }
  }
}
```

#### Full provide + inject example

## Injection Aliasing \* {#injection-aliasing}

When using the array syntax for `inject`, the injected properties are exposed on the component instance using the same key. In the example above, the property was provided under the key `"message"`, and injected as `this.message`. The local key is the same as the injection key.

If we want to inject the property using a different local key, we need to use the object syntax for the `inject` option:

```
export default {
  inject: {
    /* local key */ localMessage: {
      from: /* injection key */ 'message'
    }
  }
}
```

Here, the component will locate a property provided with the key "message", and then expose it as `this.localMessage`.

## Injection Default Values

By default, `inject` assumes that the injected key is provided somewhere in the parent chain. In the case where the key is not provided, there will be a runtime warning.

If we want to make an injected property work with optional providers, we need to declare a default value, similar to props:

```
// `value` will be "default value"
// if no data matching "message" was provided
const value = inject('message', 'default value')
```

In some cases, the default value may need to be created by calling a function or instantiating a new class. To avoid unnecessary computation or side effects in case the optional value is not used, we can use a factory function for creating the default value:

```
const value = inject('key', () => new ExpensiveClass())
```

```
export default {
  // object syntax is required
  // when declaring default values for injections
  inject: {
    message: {
      from: 'message', // this is optional if using the same key for injection
      default: 'default value'
    },
    user: {
      // use a factory function for non-primitive values that are expensive
      // to create, or ones that should be unique per component instance.
      default: () => ({ name: 'John' })
    }
  }
}
```

## Working with Reactivity

When using reactive provide / inject values, **it is recommended to keep any mutations to reactive state inside of the provider whenever possible**. This ensures that the provided state and its possible mutations are co-located in the same component, making it easier to maintain in the future.

There may be times when we need to update the data from an injector component. In such cases, we recommend providing a function that is responsible for mutating the state:

```
<!-- inside provider component -->
<script setup>
import { provide, ref } from 'vue'

const location = ref('North Pole')

function updateLocation() {
  location.value = 'South Pole'
}

provide('location', {
  location,
  updateLocation
})
</script>
```

```
<!-- in injector component -->
<script setup>
import { inject } from 'vue'

const { location, updateLocation } = inject('location')
</script>

<template>
  <button @click="updateLocation">{{ location }}</button>
</template>
```

Finally, you can wrap the provided value with `readonly()` if you want to ensure that the data passed through `provide` cannot be mutated by the injected component.

```
<script setup>
import { ref, provide, readonly } from 'vue'

const count = ref(0)
provide('read-only-count', readonly(count))
</script>
```

In order to make injections reactively linked to the provider, we need to provide a computed property using the `computed()` function:

```
import { computed } from 'vue'

export default {
  data() {
    return {
      message: 'hello!'
    }
  },
  provide() {
    return {
      // explicitly provide a computed property
      message: computed(() => this.message)
    }
  }
}
```

[Full provide + inject Example with Reactivity](#)

The `computed()` function is typically used in Composition API components, but can also be used to complement certain use cases in Options API. You can learn more about its usage by reading the [Reactivity Fundamentals](#) and [Computed Properties](#) with the API Preference set to Composition API.

**:::warning Temporary Config Required** The above usage requires setting `app.config.unwrapInjectedRef = true` to make injections automatically unwrap computed refs. This will become the default behavior in Vue 3.3 and this config is introduced temporarily to avoid breakage. It will no longer be required after 3.3.  
:::

## Working with Symbol Keys

So far, we have been using string injection keys in the examples. If you are working in a large application with many dependency providers, or you are authoring components that are going to be used by other developers, it is best to use Symbol injection keys to avoid potential collisions.

It's recommended to export the Symbols in a dedicated file:

```
// keys.js
export const myInjectionKey = Symbol()
```

```
// in provider component
import { provide } from 'vue'
import { myInjectionKey } from './keys.js'

provide(myInjectionKey, {
  /* data to provide */
})
```

```
// in injector component
import { inject } from 'vue'
import { myInjectionKey } from './keys.js'

const injected = inject(myInjectionKey)
```

See also: [Typing Provide / Inject](#)

```
// in provider component
import { myInjectionKey } from './keys.js'

export default {
  provide() {
    return {
      [myInjectionKey]: {
        /* data to provide */
      }
    }
  }
}
```

```
// in injector component
import { myInjectionKey } from './keys.js'

export default {
  inject: {
```

```
    injected: { from: myInjectionKey }  
}  
}
```

Hello from the modal!

# Component Registration

This page assumes you've already read the [Components Basics](#). Read that first if you are new to components.

A Vue component needs to be "registered" so that Vue knows where to locate its implementation when it is encountered in a template. There are two ways to register components: global and local.

## Global Registration

We can make components available globally in the current [Vue application](#) using the `app.component()` method:

```
import { createApp } from 'vue'

const app = createApp({})

app.component(
  // the registered name
  'MyComponent',
  // the implementation
  {
    /* ... */
  }
)
```

If using SFCs, you will be registering the imported `.vue` files:

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

app.component('MyComponent', MyComponent)
```

The `app.component()` method can be chained:

```
app
  .component('ComponentA', ComponentA)
  .component('ComponentB', ComponentB)
  .component('ComponentC', ComponentC)
```

Globally registered components can be used in the template of any component within this application:

```
<!-- this will work in any component inside the app -->
<ComponentA/>
<ComponentB/>
<ComponentC/>
```

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

# Local Registration

While convenient, global registration has a few drawbacks:

1. Global registration prevents build systems from removing unused components (a.k.a "tree-shaking"). If you globally register a component but end up not using it anywhere in your app, it will still be included in the final bundle.
2. Global registration makes dependency relationships less explicit in large applications. It makes it difficult to locate a child component's implementation from a parent component using it. This can affect long-term maintainability similar to using too many global variables.

Local registration scopes the availability of the registered components to the current component only. It makes the dependency relationship more explicit, and is more tree-shaking friendly.

When using SFC with `<script setup>`, imported components can be locally used without registration:

```
<script setup>
import ComponentA from './ComponentA.vue'
</script>

<template>
  <ComponentA />
</template>
```

In non-`<script setup>`, you will need to use the `components` option:

```
import ComponentA from './ComponentA.js'

export default {
  components: {
    ComponentA
  },
  setup() {
    // ...
  }
}
```

Local registration is done using the `components` option:

```
<script>
import ComponentA from './ComponentA.vue'

export default {
  components: {
    ComponentA
  }
}
</script>

<template>
  <ComponentA />
</template>
```

For each property in the `components` object, the key will be the registered name of the component, while the value will contain the implementation of the component. The above example is using the ES2015 property shorthand and is equivalent to:

```
export default {
  components: {
    ComponentA: ComponentA
  }
  // ...
}
```

Hello from the modal!

Note that **locally registered components are *not* also available in descendant components**. In this case, `ComponentA` will be made available to the current component only, not any of its child or descendant components.

## Component Name Casing

Throughout the guide, we are using PascalCase names when registering components. This is because:

1. PascalCase names are valid JavaScript identifiers. This makes it easier to import and register components in JavaScript. It also helps IDEs with auto-completion.
2. `<PascalCase />` makes it more obvious that this is a Vue component instead of a native HTML element in templates. It also differentiates Vue components from custom elements (web components).

This is the recommended style when working with SFC or string templates. However, as discussed in [DOM Template Parsing Caveats](#), PascalCase tags are not usable in DOM templates.

Luckily, Vue supports resolving kebab-case tags to components registered using PascalCase. This means a component registered as `MyComponent` can be referenced in the template via both `<MyComponent>` and `<my-component>`. This allows us to use the same JavaScript component registration code regardless of template source.

# Slots

This page assumes you've already read the [Components Basics](#). Read that first if you are new to components.

## Slot Content and Outlet

New from the modal!

We have learned that components can accept props, which can be JavaScript values of any type. But how about template content? In some cases, we may want to pass a template fragment to a child component, and let the child component render the fragment within its own template.

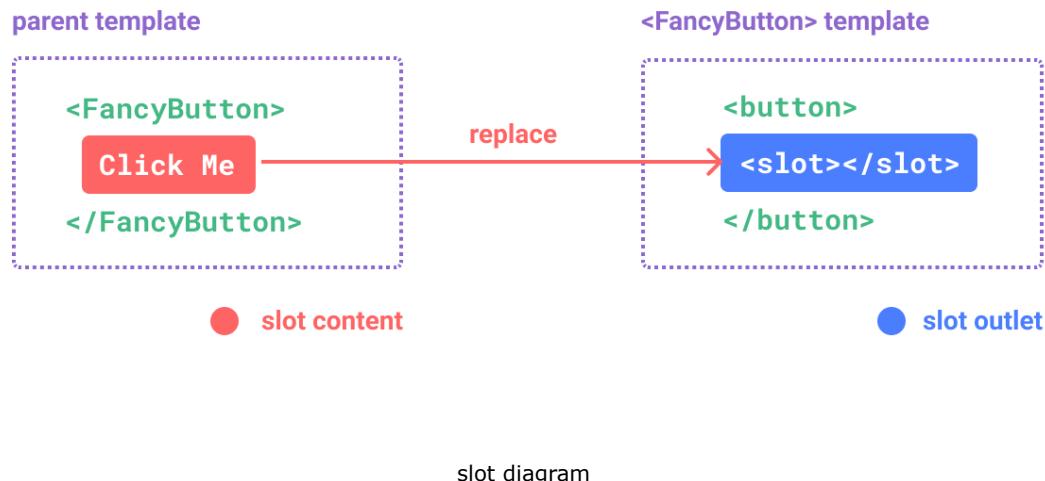
For example, we may have a `<FancyButton>` component that supports usage like this:

```
<FancyButton>
  Click me! <!-- slot content -->
</FancyButton>
```

The template of `<FancyButton>` looks like this:

```
<button class="fancy-btn">
  <slot></slot> <!-- slot outlet -->
</button>
```

The `<slot>` element is a **slot outlet** that indicates where the parent-provided **slot content** should be rendered.



And the final rendered DOM:

```
<button class="fancy-btn">Click me!</button>
```

[Try it in the Playground](#)

[Try it in the Playground](#)

With slots, the `<FancyButton>` is responsible for rendering the outer `<button>` (and its fancy styling), while the inner content is provided by the parent component.

Another way to understand slots is by comparing them to JavaScript functions:

```
// parent component passing slot content
FancyButton('Click me!')

// FancyButton renders slot content in its own template
function FancyButton(slotContent) {
  return `<button class="fancy-btn">
    ${slotContent}
  </button>`
}
```

Slot content is not just limited to text. It can be any valid template content. For example, we can pass in multiple elements, or even other components:

```
<FancyButton>
  <span style="color:red">Click me!</span>
  <AwesomeIcon name="plus" />
</FancyButton>
```

[Try it in the Playground](#)

[Try it in the Playground](#)

By using slots, our `<FancyButton>` is more flexible and reusable. We can now use it in different places with different inner content, but all with the same fancy styling.

Vue components' slot mechanism is inspired by the native Web Component `<slot>` element, but with additional capabilities that we will see later.

## Render Scope

Slot content has access to the data scope of the parent component, because it is defined in the parent. For example:

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

Here both `{{ message }}` interpolations will render the same content.

Slot content does **not** have access to the child component's data. Expressions in Vue templates can only access the scope it is defined in, consistent with JavaScript's lexical scoping. In other words:

Expressions in the parent template only have access to the parent scope; expressions in the child template only have access to the child scope.

## 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 `<SubmitButton>` component:

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

We might want the text "Submit" to be rendered inside the `<button>` if the parent didn't provide any slot content. To make "Submit" the fallback content, we can place it in between the `<slot>` tags:

```
<button type="submit">
  <slot>
    Submit <!-- fallback content -->
  </slot>
</button>
```

Now when we use `<SubmitButton>` in a parent component, providing no content for the slot:

```
<SubmitButton />
```

This will render the fallback content, "Submit":

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

But if we provide content:

```
<SubmitButton>Save</SubmitButton>
```

Then the provided content will be rendered instead:

```
<button type="submit">Save</button>
```

[Try it in the Playground](#)

[Try it in the Playground](#)

## Named Slots

There are times when it's useful to have multiple slot outlets in a single component. For example, in a `<BaseLayout>` 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 assign a unique ID to different slots so you can determine where content should be rendered:

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

Hello from the modal!

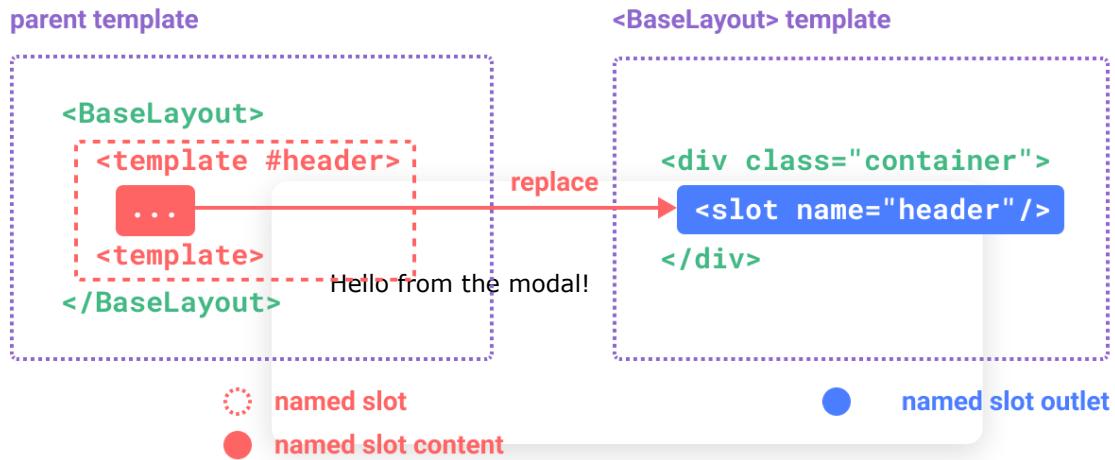
A `<slot>` outlet without `name` implicitly has the name "default".

In a parent component using `<BaseLayout>`, we need a way to pass multiple slot content fragments, each targeting a different slot outlet. This is where **named slots** come in.

To pass a named slot, we need to use a `<template>` element with the `v-slot` directive, and then pass the name of the slot as an argument to `v-slot`:

```
<BaseLayout>
  <template v-slot:header>
    <!-- content for the header slot -->
  </template>
</BaseLayout>
```

`v-slot` has a dedicated shorthand `#`, so `<template v-slot:header>` can be shortened to just `<template #header>`. Think of it as "render this template fragment in the child component's 'header' slot".



named slots diagram

Here's the code passing content for all three slots to `<BaseLayout>` using the shorthand syntax:

```
<BaseLayout>
  <template #header>
    <h1>Here might be a page title</h1>
  </template>

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

  <template #footer>
    <p>Here's some contact info</p>
  </template>
</BaseLayout>
```

When a component accepts both a default slot and named slots, all top-level non-`<template>` nodes are implicitly treated as content for the default slot. So the above can also be written as:

```
<BaseLayout>
  <template #header>
    <h1>Here might be a page title</h1>
  </template>

  <!-- implicit default slot -->
  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <template #footer>
    <p>Here's some contact info</p>
  </template>
</BaseLayout>
```

Now everything inside the `<template>` elements will be passed to the corresponding slots. The final 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>
```

[Try it in the Playground](#)

[Try it in the Playground](#)

Again, it may help you understand named slots better using the JavaScript function analogy:

```
// passing multiple slot fragments with different names
BaseLayout({
  header: `...`,
  default: `...`,
  footer: `...`
})

// <BaseLayout> renders them in different places
function BaseLayout(slots) {
  return `<div class="container">
    <header>${slots.header}</header>
    <main>${slots.default}</main>
    <footer>${slots.footer}</footer>
  </div>`
}
```

## Dynamic Slot Names

Dynamic directive arguments also work on `v-slot`, allowing the definition of dynamic slot names:

```
<base-layout>
  <template v-slot:[dynamicSlotName]>
    ...
  </template>

  <!-- with shorthand -->
  <template #[dynamicSlotName]>
    ...
  </template>
</base-layout>
```

Do note the expression is subject to the [syntax constraints](#) of dynamic directive arguments.

## Scoped Slots

As discussed in [Render Scope](#), slot content does not have access to state in the child component.

However, there are cases where it could be useful if a slot's content can make use of data from both the parent scope and the child scope. To achieve that, we need a way for the child to pass data to a slot when rendering it.

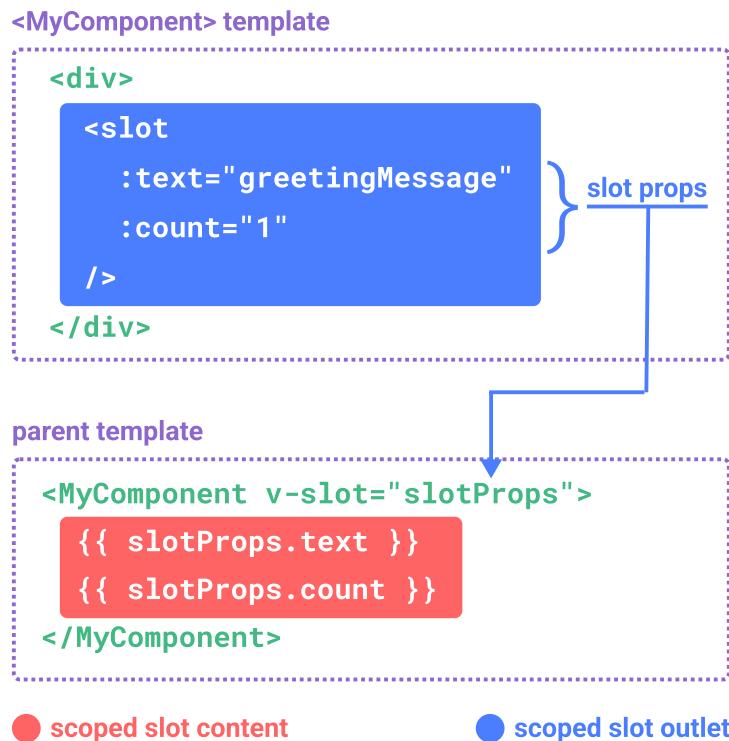
In fact, we can do exactly that - we can pass attributes to a slot outlet just like passing props to a component:

Hello from the modal!

```
<!-- <MyComponent> template -->
<div>
  <slot :text="greetingMessage" :count="1"></slot>
</div>
```

Receiving the slot props is a bit different when using a single default slot vs. using named slots. We are going to show how to receive props using a single default slot first, by using `v-slot` directly on the child component tag:

```
<MyComponent v-slot="slotProps">
  {{ slotProps.text }} {{ slotProps.count }}
</MyComponent>
```



scoped slots diagram

[Try it in the Playground](#)

[Try it in the Playground](#)

The props passed to the slot by the child are available as the value of the corresponding `v-slot` directive, which can be accessed by expressions inside the slot.

You can think of a scoped slot as a function being passed into the child component. The child component then calls it, passing props as arguments:

```
MyComponent({
  // passing the default slot, but as a function
  default: (slotProps) => {
    return `${slotProps.text} ${slotProps.count}`
  }
}

function MyComponent(slots) {
  const greetingMessage = 'hello'
  return `<div>${{
    // call the slot function with props!
    slots.default({ text: greetingMessage, count: 1 })
  }}</div>`
}
```

In fact, this is very close to how scoped slots are compiled, and how you would use scoped slots in manual [render functions](#).

Notice how `v-slot="slotProps"` matches the slot function signature. Just like with function arguments, we can use destructuring in `v-slot`:

```
<MyComponent v-slot="{ text, count }">
  {{ text }} {{ count }}
</MyComponent>
```

## Named Scoped Slots

Named scoped slots work similarly - slot props are accessible as the value of the `v-slot` directive: `v-slot:name="slotProps"`. When using the shorthand, it looks like this:

```
<MyComponent>
  <template #header="headerProps">
    {{ headerProps }}
  </template>

  <template #default="defaultProps">
    {{ defaultProps }}
  </template>

  <template #footer="footerProps">
    {{ footerProps }}
  </template>
</MyComponent>
```

Passing props to a named slot:

```
<slot name="header" message="hello"></slot>
```

Note the `name` of a slot won't be included in the props because it is reserved - so the resulting `header-Props` would be `{ message: 'hello' }`.

If you are mixing named slots with the default scoped slot, you need to use an explicit `<template>` tag for the default slot. Attempting to place the `v-slot` directive directly on the component will result in a compilation error. This is to avoid any ambiguity about the scope of the props of the default slot. For example:

```
<!-- This template won't compile -->
<template>
  <MyComponent v-slot="{ message }">
    <p>{{ message }}</p>
    <template #footer>
      <!-- message belongs to the default slot, and is not available here -->
      <p>{{ message }}</p>
    </template>
  </MyComponent>
</template>
```

Using an explicit `<template>` tag for the default slot helps to make it clear that the `message` prop is not available inside the other slot:

```
<template>
  <MyComponent>
    <!-- Use explicit default slot -->
    <template #default="{ message }">
      <p>{{ message }}</p>
    </template>

    <template #footer>
      <p>Here's some contact info</p>
    </template>
  </MyComponent>
</template>
```

## Fancy List Example

You may be wondering what would be a good use case for scoped slots. Here's an example: imagine a `<FancyList>` component that renders a list of items - it may encapsulate the logic for loading remote data, using the data to display a list, or even advanced features like pagination or infinite scrolling. However, we want it to be flexible with how each item looks and leave the styling of each item to the parent component consuming it. So the desired usage may look like this:

```
<FancyList :api-url="url" :per-page="10">
  <template #item="{ body, username, likes }">
    <div class="item">
      <p>{{ body }}</p>
      <p>by {{ username }} | {{ likes }} likes</p>
    </div>
  </template>
</FancyList>
```

Inside `<FancyList>`, we can render the same `<slot>` multiple times with different item data (notice we are using `v-bind` to pass an object as slot props):

```
<ul>
  <li v-for="item in items">
    <slot name="item" v-bind="item"></slot>
  </li>
</ul>
```

[Try it in the Playground](#)

[Try it in the Playground](#)

Hello from the modal!

## Renderless Components

The `<FancyList>` use case we discussed above encapsulates both reusable logic (data fetching, pagination etc.) and visual output, while delegating part of the visual output to the consumer component via scoped slots.

If we push this concept a bit further, we can come up with components that only encapsulate logic and do not render anything by themselves - visual output is fully delegated to the consumer component with scoped slots. We call this type of component a **Renderless Component**.

An example renderless component could be one that encapsulates the logic of tracking the current mouse position:

```
<MouseTracker v-slot="{ x, y }">
  Mouse is at: {{ x }}, {{ y }}
</MouseTracker>
```

[Try it in the Playground](#)

[Try it in the Playground](#)

While an interesting pattern, most of what can be achieved with Renderless Components can be achieved in a more efficient fashion with Composition API, without incurring the overhead of extra component nesting. Later, we will see how we can implement the same mouse tracking functionality as a [Composable](#).

That said, scoped slots are still useful in cases where we need to both encapsulate logic **and** compose visual output, like in the `<FancyList>` example.

---

[Go to TOC](#)

# Creating a Vue Application

## The application instance

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

```
import { createApp } from 'vue'
const app = createApp({
  /* root component options */
})
```

## The Root Component

The object we are passing into `createApp` is in fact a component. Every app requires a "root component" that can contain other components as its children.

If you are using Single-File Components, we typically import the root component from another file:

```
import { createApp } from 'vue'
// import the root component App from a single-file component.
import App from './App.vue'

const app = createApp(App)
```

While many examples in this guide only need a single component, most real applications are organized into a tree of nested, reusable components. For example, a Todo application's component tree might look like this:

```
App (root component)
├ TodoList
| └ TodoItem
|   └ TodoDeleteButton
|   └ TodoEditButton
└ TodoFooter
  └ TodoClearButton
  └ TodoStatistics
```

In later sections of the guide, we will discuss how to define and compose multiple components together. Before that, we will focus on what happens inside a single component.

## Mounting the App

An application instance won't render anything until its `.mount()` method is called. It expects a "container" argument, which can either be an actual DOM element or a selector string:

```
<div id="app"></div>
```

```
app.mount('#app')
```

The content of the app's root component will be rendered inside the container element. The container element itself is not considered part of the app.

The `.mount()` method should always be called after all app configurations and asset registrations are done. Also note that its return value, unlike the asset registration methods, is the root component instance instead of the application instance.

## In-DOM Root Component Template

When using Vue without a build step, we can write our root component's template directly inside the mount container:

```
<div id="app">
  <button @click="count++">{{ count }}</button>
</div>
```

```
import { createApp } from 'vue'

const app = createApp({
  data() {
    return {
      count: 0
    }
  }
})

app.mount('#app')
```

Vue will automatically use the container's `innerHTML` as the template if the root component does not already have a `template` option.

## App Configurations

The application instance exposes a `.config` object that allows us to configure a few app-level options, for example, defining an app-level error handler that captures errors from all descendant components:

```
app.config.errorHandler = (err) => {
  /* handle error */
}
```

The application instance also provides a few methods for registering app-scoped assets. For example, registering a component:

```
app.component('TodoDeleteButton', TodoDeleteButton)
```

This makes the `TodoDeleteButton` available for use anywhere in our app. We will discuss registration for components and other types of assets in later sections of the guide. You can also browse the full list of application instance APIs in its [API reference](#).

Make sure to apply all app configurations before mounting the app!

## Multiple application instances

You are not limited to a single application instance on the same page. The `createApp` API allows multiple Vue applications to co-exist on the same page, each with its own scope for configuration and global assets:

```
const app1 = createApp({
  /* ... */
})
app1.mount('#container-1')

const app2 = createApp({
  /* ... */
})
app2.mount('#container-2')
```

If you are using Vue to enhance server-rendered HTML and only need Vue to control specific parts of a large page, avoid mounting a single Vue application instance on the entire page. Instead, create multiple small application instances and mount them on the elements they are responsible for.

# Class and Style Bindings

A common need for data binding is manipulating an element's class list and inline styles. Since `class` and `style` are both attributes, we can use `v-bind` to assign them a string value dynamically, much like with other attributes. However, trying to generate those values using string concatenation can be 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.

Hello from the modal!

## Binding HTML Classes

### Binding to Objects

We can pass an object to `:class` (short for `v-bind:class`) to dynamically toggle classes:

```
<div :class="{ active: isActive }"></div>
```

The above syntax means the presence of the `active` class will be determined by the [truthiness](#) of the data property `isActive`.

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

```
const isActive = ref(true)
const hasError = ref(false)
```

```
data() {
  return {
    isActive: true,
    hasError: false
  }
}
```

And the following template:

```
<div
  class="static"
  :class="{ active: isActive, 'text-danger': hasError }"
></div>
```

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:

```
const classObject = reactive({
  active: true,
  'text-danger': false
})
```

```
data() {
  return {
    classObject: {
      active: true,
      'text-danger': false
    }
  }
}
```

```
<div :class="classObject"></div>
```

This will render the same result. We can also bind to a [computed property](#) that returns an object. This is a common and powerful pattern:

```
const isActive = ref(true)
const error = ref(null)

const classObject = computed(() => ({
  active: isActive.value && !error.value,
  'text-danger': error.value && error.value.type === 'fatal'
}))
```

```
data() {
  return {
    isActive: true,
    error: null
  }
},
computed: {
  classObject() {
    return {
      active: this.isActive && !this.error,
      'text-danger': this.error && this.error.type === 'fatal'
    }
  }
}
```

```
<div :class="classObject"></div>
```

## Binding to Arrays

We can bind `:class` to an array to apply a list of classes:

```
const activeClass = ref('active')
const errorClass = ref('text-danger')
```

```
data() {
  return {
    activeClass: 'active',
    errorClass: 'text-danger'
  }
}
```

```
<div :class="[activeClass, errorClass]"></div>
```

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

This will always apply `errorClass`. Help `activeClass` will only be applied 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 :class="[{ active: isActive }, errorClass]"></div>
```

## With Components

This section assumes knowledge of [Components](#). Feel free to skip it and come back later.

When you use the `class` attribute on a component with a single root element, those classes will be added to the component's root element, and merged with any existing class already on it.

For example, if we have a component named `MyComponent` with the following template:

```
<!-- child component template -->
<p class="foo bar">Hi!</p>
```

Then add some classes when using it:

```
<!-- when using the component -->
<MyComponent class="baz boo" />
```

The rendered HTML will be:

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

The same is true for class bindings:

```
<MyComponent :class="{ active: isActive }" />
```

When `isActive` is truthy, the rendered HTML will be:

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

If your component has multiple root elements, you would need to define which element will receive this class. You can do this using the `$attrs` component property:

```
<!-- MyComponent template using $attrs -->
<p :class="$attrs.class">Hi!</p>
<span>This is a child component</span>
```

```
<MyComponent class="baz" />
```

Will render:

```
<p class="baz">Hi!</p>
<span>This is a child component</span>
```

You can learn more about component inheritance from the model in [Fallthrough Attributes](#) section.

## Binding Inline Styles

### Binding to Objects

`:style` supports binding to JavaScript object values - it corresponds to an [HTML element's `style` property](#):

```
const activeColor = ref('red')
const fontSize = ref(30)
```

```
data() {
  return {
    activeColor: 'red',
    fontSize: 30
  }
}
```

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

Although camelCase keys are recommended, `:style` also supports kebab-cased CSS property keys (corresponds to how they are used in actual CSS) - for example:

```
<div :style="{ 'font-size': fontSize + 'px' }"></div>
```

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

```
const styleObject = reactive({
  color: 'red',
  fontSize: '13px'
})
```

```
data() {
  return {
    styleObject: {
      color: 'red',
      fontSize: '13px'
    }
  }
}
```

```
<div :style="styleObject"></div>
```

Again, object style binding is often used in conjunction with computed properties that return objects.

## Binding to Arrays

We can bind `:style` to an array of multiple style objects. These objects will be merged and applied to the same element:

```
<div :style="[baseStyles, overridingStyles]"></div>
```

## Auto-prefixing

Hello from the modal!

When you use a CSS property that requires a [vendor prefix](#) in `:style`, Vue will automatically add the appropriate prefix. Vue does this by checking at runtime to see which style properties are supported in the current browser. If the browser doesn't support a particular property then various prefixed variants will be tested to try to find one that is supported.

## Multiple Values

You can provide an array of multiple (prefixed) values to a style property, for example:

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

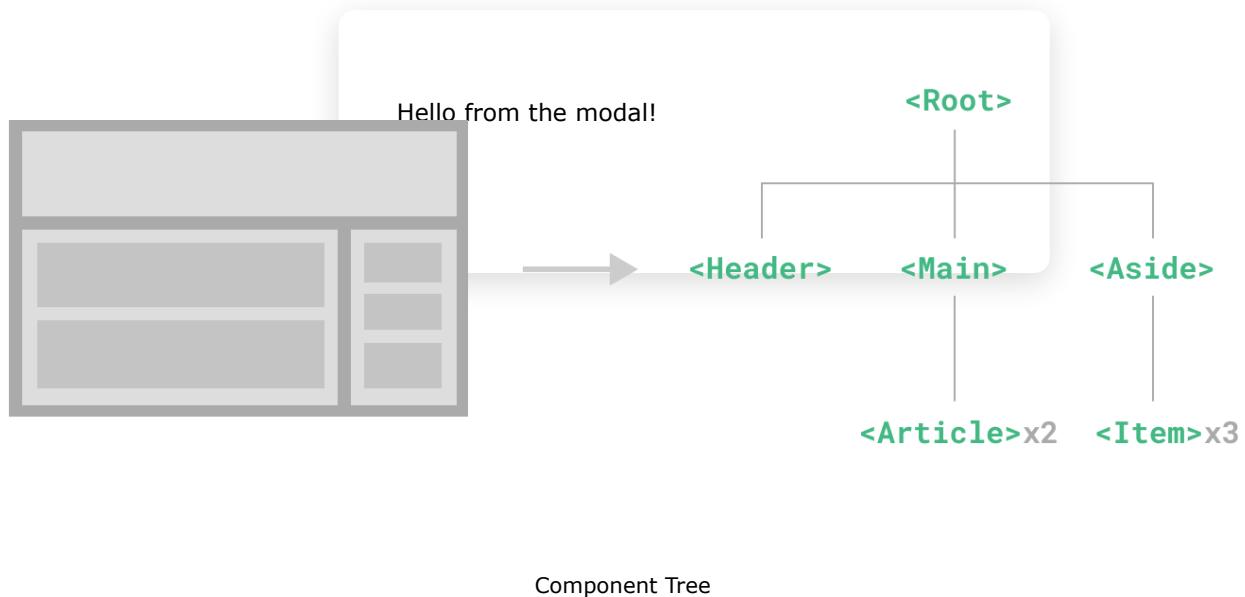
This will only render the last value in the array which the browser supports. In this example, it will render `display: flex` for browsers that support the unprefixed version of flexbox.

---

[Go to TOC](#)

# Components Basics

Components allow us to split the UI into independent and reusable pieces, and think about each piece in isolation. It's common for an app to be organized into a tree of nested components:



This is very similar to how we nest native HTML elements, but Vue implements its own component model that allow us to encapsulate custom content and logic in each component. Vue also plays nicely with native Web Components. If you are curious about the relationship between Vue Components and native Web Components, [read more here](#).

## Defining a Component

When using a build step, we typically define each Vue component in a dedicated file using the `.vue` extension - known as a [Single-File Component](#) (SFC for short):

```

<script>
export default {
  data() {
    return {
      count: 0
    }
  }
}
</script>

<template>
  <button @click="count++">You clicked me {{ count }} times.</button>
</template>
  
```

```

<script setup>
import { ref } from 'vue'
  
```

```

const count = ref(0)
</script>

<template>
  <button @click="count++">You clicked me {{ count }} times.</button>
</template>

```

When not using a build step, a Vue component can be defined as a plain JavaScript object containing Vue-specific options:

```

export default {
  data() {
    return {
      count: 0
    }
  },
  template: `
    <button @click="count++">
      You clicked me {{ count }} times.
    </button>
  `
}

```

```

import { ref } from 'vue'

export default {
  setup() {
    const count = ref(0)
    return { count }
  },
  template: `
    <button @click="count++">
      You clicked me {{ count }} times.
    </button>
    // or `template: '#my-template-element'`
  `
}

```

The template is inlined as a JavaScript string here, which Vue will compile on the fly. You can also use an ID selector pointing to an element (usually native `<template>` elements) - Vue will use its content as the template source.

The example above defines a single component and exports it as the default export of a `.js` file, but you can use named exports to export multiple components from the same file.

## Using a Component

:::tip We will be using SFC syntax for the rest of this guide - the concepts around components are the same regardless of whether you are using a build step or not. The [Examples](#) section shows component usage in both scenarios. :::

To use a child component, we need to import it in the parent component. Assuming we placed our counter component inside a file called `ButtonCounter.vue`, the component will be exposed as the file's default export:

```
<script>
import ButtonCounter from './ButtonCounter.vue'

export default {
  components: {
    ButtonCounter
  }
}
</script>

<template>
  <h1>Here is a child component!</h1>
  <ButtonCounter />
</template>
```

To expose the imported component to our template, we need to [register](#) it with the `components` option. The component will then be available as a tag using the key it is registered under.

```
<script setup>
import ButtonCounter from './ButtonCounter.vue'
</script>

<template>
  <h1>Here is a child component!</h1>
  <ButtonCounter />
</template>
```

With `<script setup>`, imported components are automatically made available to the template.

It's also possible to globally register a component, making it available to all components in a given app without having to import it. The pros and cons of global vs. local registration is discussed in the dedicated [Component Registration](#) section.

Components can be reused as many times as you want:

```
<h1>Here are many child components!</h1>
<ButtonCounter />
<ButtonCounter />
<ButtonCounter />
```

[Try it in the Playground](#)

[Try it in the Playground](#)

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.

In SFCs, it's recommended to use `PascalCase` tag names for child components to differentiate from native HTML elements. Although native HTML tag names are case-insensitive, Vue SFC is a compiled format so we are able to use case-sensitive tag names in it. We are also able to use `/>` to close a tag.

If you are authoring your templates directly in a DOM (e.g. as the content of a native `<template>` element), the template will be subject to the browser's native HTML parsing behavior. In such cases, you will need to use `kebab-case` and explicit closing tags for components:

```
<!-- if this template is written in the DOM -->
<button-counter></button-counter>
<button-counter></button-counter>
<button-counter></button-counter>
```

See [DOM template parsing caveats](#) for more details.

## Passing Props

If we are building a blog, we will likely need a component representing a blog post. We want all the blog posts to share the same visual layout ~~but with different content~~. Such a 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. To pass a title to our blog post component, we must declare it in the list of props this component accepts, using the `props` option `defineProps` macro:

```
<!-- BlogPost.vue -->
<script>
export default {
  props: ['title']
}
</script>

<template>
  <h4>{{ title }}</h4>
</template>
```

When a value is passed to a prop attribute, it becomes a property on that component instance. The value of that property is accessible within the template and on the component's `this` context, just like any other component property.

```
<!-- BlogPost.vue -->
<script setup>
defineProps(['title'])
</script>

<template>
  <h4>{{ title }}</h4>
</template>
```

`defineProps` is a compile-time macro that is only available inside `<script setup>` and does not need to be explicitly imported. Declared props are automatically exposed to the template. `defineProps` also returns an object that contains all the props passed to the component, so that we can access them in JavaScript if needed:

```
const props = defineProps(['title'])
console.log(props.title)
```

See also: [Typing Component Props](#)

If you are not using `<script setup>`, props should be declared using the `props` option, and the props object will be passed to `setup()` as the first argument:

```
export default {
  props: ['title'],
  setup(props) {
    console.log(props.title)
  }
}
```

A component can have as many props as you like and, by default, any value can be passed to any prop.  
Hello from the modal!

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

```
<BlogPost title="My journey with Vue" />
<BlogPost title="Blogging with Vue" />
<BlogPost title="Why Vue is so fun" />
```

In a typical app, however, you'll likely have an array of posts in your parent component:

```
export default {
  // ...
  data() {
    return {
      posts: [
        { id: 1, title: 'My journey with Vue' },
        { id: 2, title: 'Blogging with Vue' },
        { id: 3, title: 'Why Vue is so fun' }
      ]
    }
  }
}
```

```
const posts = ref([
  { 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, using `v-for`:

```
<BlogPost
  v-for="post in posts"
  :key="post.id"
  :title="post.title"
/>
```

[Try it in the Playground](#)

[Try it in the Playground](#)

Notice how `v-bind` is used to pass dynamic prop values. This is especially useful when you don't know the exact content you're going to render ahead of time.

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](#).

## Listening to Events

As we develop our `<BlogPost>` 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 at its default size.

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

```
data() {
  return {
    posts: [
      /* ... */
    ],
    postFontSize: 1
  }
}
```

```
const posts = ref([
  /* ... */
])

const postFontSize = ref(1)
```

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

```
<div :style="{ fontSize: postFontSize + 'em' }">
  <BlogPost
    v-for="post in posts"
    :key="post.id"
    :title="post.title"
  />
</div>
```

Now let's add a button to the `<BlogPost>` component's template:

```
<!-- BlogPost.vue, omitting <script> -->
<template>
  <div class="blog-post">
    <h4>{{ title }}</h4>
    <button>Enlarge text</button>
  </div>
</template>
```

The button doesn't do anything yet - we want clicking the button to communicate to the parent that it should enlarge the text of all posts. To solve this problem, components provide a custom events system. The parent can choose to listen to any event on the child component instance with `v-on` or `@`, just as we would with a native DOM event:

```
<BlogPost
  ...
  @enlarge-text="postFontSize += 0.1"
/>
```

Then the child component can emit an event on itself by calling the built-in `$emit` method, passing the name of the event:

```
<!-- BlogPost.vue, omitting <script> -->
<template>
  <div class="blog-post">
    <h4>{{ title }}</h4>
    <button @click="$emit('enlarge-text')">Enlarge text</button>
  </div>
</template>
```

Thanks to the `@enlarge-text="postFontSize += 0.1"` listener, the parent will receive the event and update the value of `postFontSize`.

[Try it in the Playground](#)

[Try it in the Playground](#)

We can optionally declare emitted events using the `emits` option `defineEmits` macro:

```
<!-- BlogPost.vue -->
<script>
export default {
  props: ['title'],
  emits: ['enlarge-text']
}
</script>
```

```
<!-- BlogPost.vue -->
<script setup>
defineProps(['title'])
defineEmits(['enlarge-text'])
</script>
```

This documents all the events that a component emits and optionally [validates them](#). It also allows Vue to avoid implicitly applying them as native listeners to the child component's root element.

Similar to `defineProps`, `defineEmits` is only usable in `<script setup>` and doesn't need to be imported. It returns an `emit` function that is equivalent to the `$emit` method. It can be used to emit events in the `<script setup>` section of a component, where `$emit` isn't directly accessible:

```
<script setup>
const emit = defineEmits(['enlarge-text'])

emit('enlarge-text')
</script>
```

See also: [Typing Component Emits](#)

If you are not using `<script setup>`, you can declare emitted events using the `emits` option. You can access the `emit` function as a property of the setup context (passed to `setup()` as the second argument):

```
export default {
  emits: ['enlarge-text'],
  setup(props, ctx) {
    ctx.emit('enlarge-text')
  }
}
```

That's all you need to know about custom component events for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on [Custom Events](#).

## Content Distribution with Slots

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

```
<AlertBox>
  Something bad happened.
</AlertBox>
```

Which might render something like:

:::danger This is an Error for Demo Purposes Something bad happened. :::

This can be achieved using Vue's custom `<slot>` element:

```
<template>
  <div class="alert-box">
    <strong>This is an Error for Demo Purposes</strong>
    <slot />
  </div>
</template>

<style scoped>
.alert-box {
  /* ... */
}
</style>
```

As you'll see above, we use the `<slot>` as a placeholder where we want the content to go – and that's it. We're done!

[Try it in the Playground](#)

[Try it in the Playground](#)

That's all you need to know about slots for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on [Slots](#).

## Dynamic Components

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

[Open example in the Playground](#)

[Open example in the Playground](#)

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

```
<!-- Component changes when currentTab changes -->
<component :is="currentTab"></component>
```

```
<!-- Component changes when currentTab changes -->
<component :is="tabs[currentTab]"></component>
```

In the example above, the value passed to `:is` can contain either:  
Hello from the modal!

- the name string of a registered component, OR
- the actual imported component object

You can also use the `:is` attribute to create regular HTML elements.

When switching between multiple components with `<component :is="...">`, a component will be unmounted when it is switched away from. We can force the inactive components to stay "alive" with the built-in `<KeepAlive>` component.

## DOM Template Parsing Caveats

If you are writing your Vue templates directly in the DOM, Vue will have to retrieve the template string from the DOM. This leads to some caveats due to browsers' native HTML parsing behavior.

:::tip It should be noted that the limitations discussed below only apply if you are writing your templates directly in the DOM. They do NOT apply if you are using string templates from the following sources:

- Single-File Components
- Inlined template strings (e.g. `template: '...'`)
- `<script type="text/x-template"> :::`

## Case Insensitivity

HTML tags and attribute names are case-insensitive, so browsers will interpret any uppercase characters as lowercase. That means when you're using in-DOM templates, PascalCase component names and camelCased prop names or `v-on` event names all need to use their kebab-cased (hyphen-delimited) equivalents:

```
// camelCase in JavaScript
const BlogPost = {
  props: ['postTitle'],
  emits: ['updatePost'],
  template:
    `<h3>{{ postTitle }}</h3>
`}
```

```
<!-- kebab-case in HTML -->
<blog-post post-title="hello!" @update-post="onUpdatePost"></blog-post>
```

## Self Closing Tags

We have been using self-closing tags for components in previous code samples:

```
<MyComponent />
```

This is because Vue's template parser respects `/>` as an indication to end any tag, regardless of its type.

In DOM templates, however, we must always include explicit closing tags:

```
<my-component></my-component>
```

This is because the HTML spec only allows [a few specific elements](#) to omit closing tags, the most common being `<input>` and `<img>`. For all other elements, if you omit the closing tag, the native HTML parser will think you never terminated the opening tag. For example, the following snippet:

```
<my-component /> <!-- we intend to close the tag here... -->
<span>hello</span>
```

will be parsed as:

```
<my-component>
  <span>hello</span>
</my-component> <!-- but the browser will close it here. -->
```

## Element Placement Restrictions

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>
```

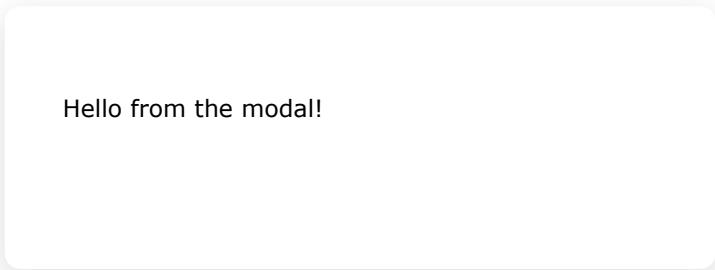
The custom component `<blog-post-row>` will be hoisted out as invalid content, causing errors in the eventual rendered output. We can use the special `is` attribute as a workaround:

```
<table>
  <tr is="vue:blog-post-row"></tr>
</table>
```

:::tip When used on native HTML elements, the value of `is` must be prefixed with `vue:` in order to be interpreted as a Vue component. This is required to avoid confusion with native [customized built-in elements](#).  
:::

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 - build something fun, or check out some of the [Examples](#) if you haven't already.

Once you feel comfortable with the knowledge you've just digested, move on with the guide to learn more about components in depth.



Hello from the modal!

# Computed Properties

## Basic Example

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, if we have an object with a nested array:

Hello from the modal!

```
export default {
  data() {
    return {
      author: {
        name: 'John Doe',
        books: [
          'Vue 2 - Advanced Guide',
          'Vue 3 - Basic Guide',
          'Vue 4 - The Mystery'
        ]
      }
    }
  }
}
```

```
const author = reactive({
  name: 'John Doe',
  books: [
    'Vue 2 - Advanced Guide',
    'Vue 3 - Basic Guide',
    'Vue 4 - The Mystery'
  ]
})
```

And we want to display different messages depending on if `author` already has some books or not:

```
<p>Has published books:</p>
<span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
```

At this point, the template is getting a bit cluttered. We have to look at it for a second before realizing that it performs a calculation depending on `author.books`. More importantly, we probably don't want to repeat ourselves if we need to include this calculation in the template more than once.

That's why for complex logic that includes reactive data, it is recommended to use a **computed property**. Here's the same example, refactored:

```
export default {
  data() {
    return {
      author: {
        name: 'John Doe',
        books: [
          'Vue 2 - Advanced Guide',
          'Vue 3 - Basic Guide',
          'Vue 4 - The Mystery'
        ]
      }
    }
  }
}
```

```

        ],
      },
    },
  },
  computed: {
    // a computed getter
    publishedBooksMessage() {
      // `this` points to the component instance
      return this.author.books.length > 0 ? 'Yes' : 'No'
    }
  }
}

```

Hello from the modal!

```

<p>Has published books:</p>
<span>{{ publishedBooksMessage }}</span>

```

[Try it in the Playground](#)

Here we have declared a computed property `publishedBooksMessage`.

Try to change the value of the `books` array in the application `data` and you will see how `publishedBooksMessage` is changing accordingly.

You can data-bind to computed properties in templates just like a normal property. Vue is aware that `this.publishedBooksMessage` depends on `this.author.books`, so it will update any bindings that depend on `this.publishedBooksMessage` when `this.author.books` changes.

See also: [Typing Computed Properties](#)

```

<script setup>
import { reactive, computed } from 'vue'

const author = reactive({
  name: 'John Doe',
  books: [
    'Vue 2 - Advanced Guide',
    'Vue 3 - Basic Guide',
    'Vue 4 - The Mystery'
  ]
})

// a computed ref
const publishedBooksMessage = computed(() => {
  return author.books.length > 0 ? 'Yes' : 'No'
})
</script>

<template>
  <p>Has published books:</p>
  <span>{{ publishedBooksMessage }}</span>
</template>

```

[Try it in the Playground](#)

Here we have declared a computed property `publishedBooksMessage`. The `computed()` function expects to be passed a getter function, and the returned value is a **computed ref**. Similar to normal refs, you can access the computed result as `publishedBooksMessage.value`. Computed refs are also auto-unwrapped in templates so you can reference them without `.value` in template expressions.

A computed property automatically tracks its reactive dependencies. Vue is aware that the computation of `publishedBooksMessage` depends on `author.books`, so it will update any bindings that depend on `publishedBooksMessage` when `author.books` changes.

See also: [Typing Computed](#)

Hello from the modal!

## Computed Caching vs. Methods

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

```
<p>{{ calculateBooksMessage() }}</p>
```

```
// in component
methods: {
  calculateBooksMessage() {
    return this.author.books.length > 0 ? 'Yes' : 'No'
  }
}
```

```
// in component
function calculateBooksMessage() {
  return author.books.length > 0 ? 'Yes' : 'No'
}
```

Instead of a computed property, we can define the same function as a method. For the end result, the two approaches are indeed exactly the same. However, the difference is that **computed properties are cached based on their reactive dependencies**. A computed property will only re-evaluate when some of its reactive dependencies have changed. This means as long as `author.books` has not changed, multiple access to `publishedBooksMessage` will immediately return the previously computed result without having to run the getter function again.

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

```
computed: {
  now() {
    return Date.now()
  }
}
```

```
const now = computed(() => Date.now())
```

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

Why do we need caching? Imagine we have an expensive computed property `list`, 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 `list`. Without caching, we would be executing `list`'s getter many more times than necessary! In cases where you do not want caching, use a method call instead.

## Writable Computed

Computed properties are by default getter-only. If you attempt to assign a new value to a computed property, you will receive a runtime warning. In the rare cases where you need a "writable" computed property, you can create one by providing both a getter and a setter:

```
export default {
  data() {
    return {
      firstName: 'John',
      lastName: 'Doe'
    }
  },
  computed: {
    fullName: {
      // getter
      get() {
        return this.firstName + ' ' + this.lastName
      },
      // setter
      set(newValue) {
        // Note: we are using destructuring assignment syntax here.
        [this.firstName, this.lastName] = newValue.split(' ')
      }
    }
  }
}
```

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

```
<script setup>
import { ref, computed } from 'vue'

const firstName = ref('John')
const lastName = ref('Doe')

const fullName = computed({
  // getter
  get() {
    return firstName.value + ' ' + lastName.value
  },
  // setter
  set(newValue) {
    // Note: we are using destructuring assignment syntax here.
    [firstName.value, lastName.value] = newValue.split(' ')
  }
})
```

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

## Best Practices

### Getters should be side-effect free

It is important to remember that computed getter functions should only perform pure computation and be free of side effects. For example, **don't make async requests or mutate the DOM inside a computed getter!** Think of a computed property as declaratively describing how to derive a value based on other values - its only responsibility should be computing and returning that value. Later in the guide we will discuss how we can perform side effects in reaction to state changes with [watchers](#).

### Avoid mutating computed value

The returned value from a computed property is derived state. Think of it as a temporary snapshot - every time the source state changes, a new snapshot is created. It does not make sense to mutate a snapshot, so a computed return value should be treated as read-only and never be mutated - instead, update the source state it depends on to trigger new computations.

# Conditional Rendering

## v-if {#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>
```

## v-else {#v-else}

You can use the `v-else` directive to indicate an "else block" for `v-if`:

```
<button @click="awesome = !awesome">Toggle</button>
<h1 v-if="awesome">Vue is awesome!</h1>
<h1 v-else>Oh no 😢 </h1>
```

[Toggle](#)

## Vue is awesome!

## Oh no 😢

[Try it in the Playground](#)

[Try it in the Playground](#)

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 {#v-else-if}

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

```
<div v-if="type === 'A'>
  A
</div>
<div v-else-if="type === 'B'>
  B
</div>
<div v-else-if="type === 'C'>
  C
</div>
```

```
<div v-else>
  Not A/B/C
</div>
```

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

## v-if on <template> {#v-if-on-template}

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

```
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

`v-else` and `v-else-if` can also be used on `<template>`.

## v-show {#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.

`v-show` doesn't support the `<template>` element, nor does it work with `v-else`.

## v-if vs. v-show {#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 {#v-if-with-v-for}

::: warning Note It's **not** recommended to use `v-if` and `v-for` on the same element due to implicit precedence. Refer to [style guide](#) for details. :::

When `v-if` and `v-for` are both used on the same element, `v-if` will be evaluated first. See the [list rendering guide](#) for details.

Hello from the modal!

# Event Handling

## Listening to Events

We can use the `v-on` directive, which we typically shorten to the `@` symbol, to listen to DOM events and run some JavaScript when they're triggered. The usage would be `v-on:click="handler"` or with the shortcut, `@click="handler"`.

Hello from the modal!

The handler value can be one of the following:

1. **Inline handlers:** Inline JavaScript to be executed when the event is triggered (similar to the native `onclick` attribute).
2. **Method handlers:** A property name or path that points to a method defined on the component.

## Inline Handlers

Inline handlers are typically used in simple cases, for example:

```
const count = ref(0)
```

```
data() {
  return {
    count: 0
  }
}
```

```
<button @click="count++">Add 1</button>
<p>Count is: {{ count }}</p>
```

[Try it in the Playground](#)

[Try it in the Playground](#)

## Method Handlers

The logic for many event handlers will be more complex though, and likely isn't feasible with inline handlers. That's why `v-on` can also accept the name or path of a component method you'd like to call.

For example:

```
const name = ref('Vue.js')

function greet(event) {
  alert(`Hello ${name.value}!`)
  // `event` is the native DOM event
  if (event) {
    alert(event.target.tagName)
  }
}
```

```

data() {
  return {
    name: 'Vue.js'
  }
},
methods: {
  greet(event) {
    // `this` inside methods points to the current active instance
    alert(`Hello ${this.name}!`)
    // `event` is the native DOM event
    if (event) {
      alert(event.target.tagName)
    }
  }
}

```

```

<!-- `greet` is the name of the method defined above -->
<button @click="greet">Greet</button>

```

[Try it in the Playground](#)

[Try it in the Playground](#)

A method handler automatically receives the native DOM Event object that triggers it - in the example above, we are able to access the element dispatching the event via `event.target.tagName`.

See also: [Typing Event Handlers](#)

See also: [Typing Event Handlers](#)

## Method vs. Inline Detection

The template compiler detects method handlers by checking whether the `v-on` value string is a valid JavaScript identifier or property access path. For example, `foo`, `foo.bar` and `foo['bar']` are treated as method handlers, while `foo()` and `count++` are treated as inline handlers.

## Calling Methods in Inline Handlers

Instead of binding directly to a method name, we can also call methods in an inline handler. This allows us to pass the method custom arguments instead of the native event:

```

function say(message) {
  alert(message)
}

```

```

methods: {
  say(message) {
    alert(message)
  }
}

```

```

<button @click="say('hello')">Say hello</button>
<button @click="say('bye')">Say bye</button>

```

[Try it in the Playground](#)

[Try it in the Playground](#)

## Accessing Event Argument in Inline Handlers

Sometimes we also need to access the original DOM event in an inline handler. You can pass it into a method using the special `$event` variable, or use an inline arrow function:

```
<!-- using $event special variable -->
<button @click="warn('Form cannot be submitted yet.', $event)">
  Submit
</button>

<!-- using inline arrow function -->
<button @click="(event) => warn('Form cannot be submitted yet.', event)">
  Submit
</button>
```

```
function warn(message, event) {
  // now we have access to the native event
  if (event) {
    event.preventDefault()
  }
  alert(message)
}
```

```
methods: {
  warn(message, event) {
    // now we have access to the native event
    if (event) {
      event.preventDefault()
    }
    alert(message)
  }
}
```

## Event Modifiers

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

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

- `.stop`
- `.prevent`
- `.self`
- `.capture`
- `.once`
- `.passive`

```

<!-- the click event's propagation will be stopped -->
<a @click.stop="doThis"></a>

<!-- the submit event will no longer reload the page -->
<form @submit.prevent="onSubmit"></form>

<!-- modifiers can be chained -->
<a @click.stop.prevent="doThat"></a>

<!-- just the modifier -->
<form @submit.prevent></form>

<!-- only trigger handler if event.target is the element itself -->
<!-- i.e. not from a child element -->
<div @click.self="doThat">...</div>

```

::: tip Order matters when using modifiers because the relevant code is generated in the same order. Therefore using `@click.prevent.self` will prevent **click's default action on the element itself and its children**, while `@click.self.prevent` will only prevent click's default action on the element itself. :::

The `.capture`, `.once`, and `.passive` modifiers mirror the [options of the native `addEventListener` method](#):

```

<!-- 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 @click.capture="doThis">...</div>

<!-- the click event will be triggered at most once -->
<a @click.once="doThis"></a>

<!-- the scroll event's default behavior (scrolling) will happen -->
<!-- immediately, instead of waiting for `onScroll` to complete -->
<!-- in case it contains `event.preventDefault()` -->
<div @scroll.passive="onScroll">...</div>

```

The `.passive` modifier is typically used with touch event listeners for [improving performance on mobile devices](#).

::: tip Do not use `.passive` and `.prevent` together, because `.passive` already indicates to the browser that you *do not* intend to prevent the event's default behavior, and you will likely see a warning from the browser if you do so. :::

## Key Modifiers

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

```

<!-- only call `submit` when the `key` is `Enter` -->
<input @keyup.enter="submit" />

```

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

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

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

## Key Aliases

Vue provides aliases for the most commonly used keys:

- `.enter`
- `.tab`
- `.delete` (captures both "Delete" and "Backspace" keys)
- `.esc`
- `.space`
- `.up`
- `.down`
- `.left`
- `.right`

## System Modifier Keys

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`

::: tip 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 + Enter -->
<input @keyup.alt.enter="clear" />

<!-- Ctrl + Click -->
<div @click.ctrl="doSomething">Do something</div>
```

::: tip 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. :::

## .exact Modifier

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

- `.left`
- `.right`
- `.middle`

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

# Form Input Bindings

When dealing with forms on the frontend, we often need to sync the state of form input elements with corresponding state in JavaScript. It can be cumbersome to manually wire up value bindings and change event listeners:

```
<input  
  :value="text"  
  @input="event => text = event.target.value">
```

The `v-model` directive helps us simplify the above to:

```
<input v-model="text">
```

In addition, `v-model` can be used on inputs of different types, `<textarea>`, and `<select>` elements. It automatically expands to different DOM property and event pairs based on the element it is used on:

- `<input>` with text types and `<textarea>` elements use `value` property and `input` event;
- `<input type="checkbox">` and `<input type="radio">` use `checked` property and `change` event;
- `<select>` use `value` as a prop and `change` as an event.

::: tip Note `v-model` will ignore the initial `value`, `checked` or `selected` attributes found on any form elements. It will always treat the current bound JavaScript state as the source of truth. You should declare the initial value on the JavaScript side, using the `data` option/reactivity APIs. :::

## Basic Usage

### Text

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

Message is: {{ message }}

[Try it in the Playground](#)

[Try it in the Playground](#)

::: tip Note For languages that require an [IME](#) (Chinese, Japanese, Korean etc.), you'll notice that `v-model` doesn't get updated during IME composition. If you want to respond to these updates as well, use your own `input` event listener and `value` binding instead of using `v-model`. :::

## Multiline text

```
<span>Multiline message is:</span>
<p style="white-space: pre-line;">{{ message }}</p>
<textarea v-model="message" placeholder="add multiple lines"></textarea>
```

Multiline message is:

```
{{ multilineText }}
```

```
add multiple lines
```

Hello from the modal!

[Try it in the Playground](#)

[Try it in the Playground](#)

Note that interpolation inside `<textarea>` won't work. Use `v-model` instead.

```
<!-- bad -->
<textarea>{{ text }}</textarea>

<!-- good -->
<textarea v-model="text"></textarea>
```

## Checkbox

Single checkbox, boolean value:

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

{{ checked }}

[Try it in the Playground](#)

[Try it in the Playground](#)

We can also bind multiple checkboxes to the same array or `Set` value:

```
const checkedNames = ref([])
```

```
export default {
  data() {
    return {
      checkedNames: []
    }
  }
}
```

```
<div>Checked names: {{ checkedNames }}</div>
```

```
<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>
```

Checked names: {{ checkedNames }}

Jack  John  Mike

In this case, the `checkedNames` array will always contain the values from the currently checked boxes.

[Try it in the Playground](#)

Hello from the modal!

[Try it in the Playground](#)

## Radio

```
<div>Picked: {{ picked }}</div>

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

<input type="radio" id="two" value="Two" v-model="picked" />
<label for="two">Two</label>
```

Picked: {{ picked }}

One  Two

[Try it in the Playground](#)

[Try it in the Playground](#)

## Select

Single select:

```
<div>Selected: {{ selected }}</div>

<select v-model="selected">
  <option disabled value="">Please select one</option>
  <option>A</option>
  <option>B</option>
  <option>C</option>
</select>
```

Selected: {{ selected }}

A 

[Try it in the Playground](#)

[Try it in the Playground](#)

:::tip Note 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):

```
<div>Selected: {{ selected }}</div>

<select v-model="selected" multiple>
  <option>A</option>
  <option>B</option>
  <option>C</option>
</select>
```

Selected: {{ multiSelected }}

A B C

[Try it in the Playground](#)

[Try it in the Playground](#)

Select options can be dynamically rendered with `v-for`:

```
const selected = ref('A')

const options = ref([
  { text: 'One', value: 'A' },
  { text: 'Two', value: 'B' },
  { text: 'Three', value: 'C' }
])
```

```
export default {
  data() {
    return {
      selected: 'A',
      options: [
        { text: 'One', value: 'A' },
        { text: 'Two', value: 'B' },
        { text: 'Three', value: 'C' }
      ]
    }
  }
}
```

```
<select v-model="selected">
  <option v-for="option in options" :value="option.value">
    {{ option.text }}
  </option>
</select>

<div>Selected: {{ selected }}</div>
```

[Try it in the Playground](#)[Try it in the Playground](#)

## 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 current active instance. We can use `v-bind` to achieve that. In addition, using `v-bind` allows us to bind the input value to non-string values.

## Checkbox

```
<input
  type="checkbox"
  v-model="toggle"
  true-value="yes"
  false-value="no" />
```

`true-value` and `false-value` are Vue-specific attributes that only work with `v-model`. Here the `toggle` property's value will be set to `'yes'` when the box is checked, and set to `'no'` when unchecked. You can also bind them to dynamic values using `v-bind`:

```
<input
  type="checkbox"
  v-model="toggle"
  :true-value="dynamicTrueValue"
  :false-value="dynamicFalseValue" />
```

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

## Radio

```
<input type="radio" v-model="pick" :value="first" />
<input type="radio" v-model="pick" :value="second" />
```

`pick` will be set to the value of `first` when the first radio input is checked, and set to the value of `second` when the second one is checked.

## Select Options

```
<select v-model="selected">
  <!-- inline object literal -->
  <option :value="{ number: 123 }">123</option>
</select>
```

`v-model` supports value bindings of non-string values as well! In the above example, when the option is selected, `selected` will be set to the object literal value of `{ number: 123 }`.

## Modifiers

### .lazy {#lazy}

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

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

### .number {#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" />
```

If the value cannot be parsed with `parseFloat()`, then the original value is used instead.

The `number` modifier is applied automatically if the input has `type="number"`.

### .trim {#trim}

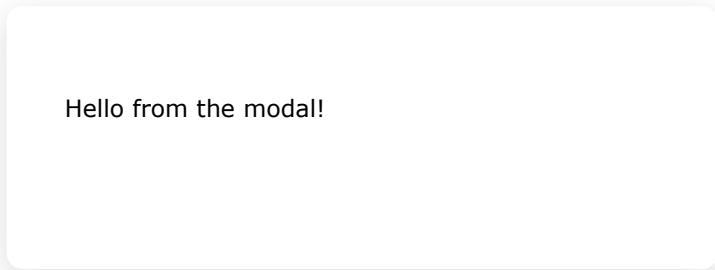
If you want whitespace from user input to be trimmed automatically, you can add the `trim` modifier to your `v-model`-managed inputs:

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

## v-model with Components

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 [Usage with v-model](#) in the Components guide.



Hello from the modal!

---

[Go to TOC](#)

# Lifecycle Hooks

Each Vue component 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.

## Registering Lifecycle Hooks

For example, the `onMounted mounted` hook can be used to run code after the component has finished the initial rendering and created the DOM nodes:

```
<script setup>
import { onMounted } from 'vue'

onMounted(() => {
  console.log(`the component is now mounted.`)
})
</script>

export default {
  mounted() {
    console.log(`the component is now mounted.`)
  }
}
```

There are also other hooks which will be called at different stages of the instance's lifecycle, with the most commonly used being `onMounted`, `onUpdated`, and `onUnmounted`. `mounted`, `updated`, and `unmounted`.

All lifecycle hooks are called with their `this` context pointing to the current active instance invoking it. Note this means you should avoid using arrow functions when declaring lifecycle hooks, as you won't be able to access the component instance via `this` if you do so.

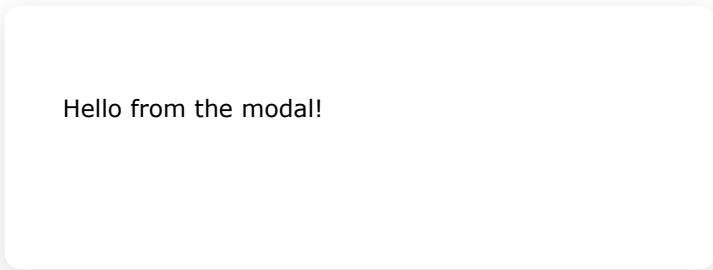
When calling `onMounted`, Vue automatically associates the registered callback function with the current active component instance. This requires these hooks to be registered **synchronously** during component set-up. For example, do not do this:

```
setTimeout(() => {
  onMounted(() => {
    // this won't work.
  })
}, 100)
```

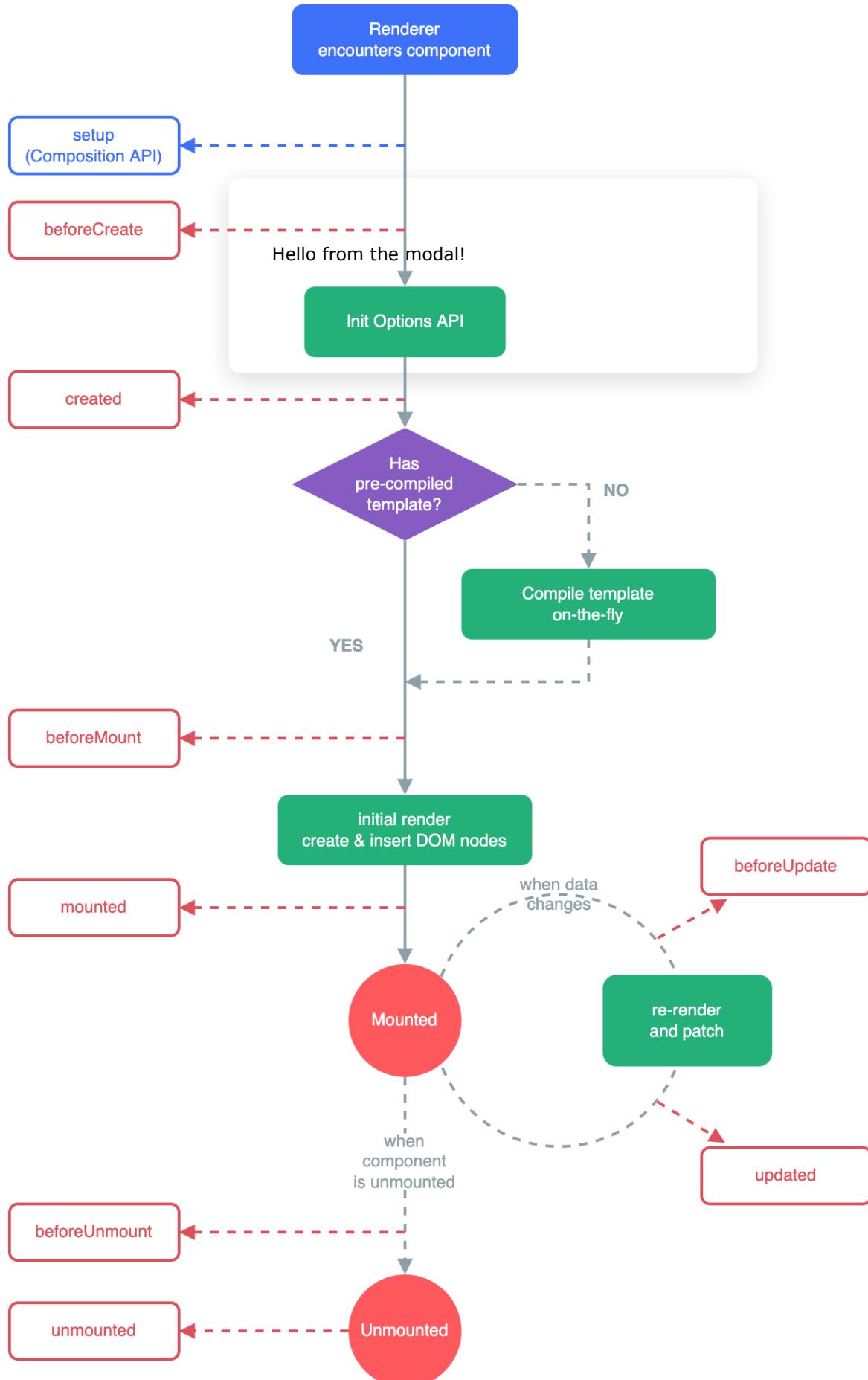
Do note this doesn't mean that the call must be placed lexically inside `setup()` or `<script setup>`. `onMounted()` can be called in an external function as long as the call stack is synchronous and originates from within `setup()`.

## Lifecycle Diagram

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

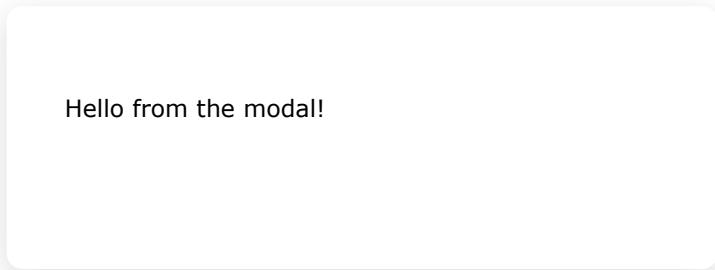


Hello from the modal!



#### Component lifecycle diagram

Consult the [Lifecycle Hooks API reference](#) for details on all lifecycle hooks and their respective use cases.



Hello from the modal!

---

[Go to TOC](#)

# List Rendering

## v-for {#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:

Hello from the modal!

```
const items = ref([{ message: 'Foo' }, { message: 'Bar' }])
```

```
data() {
  return {
    items: [{ message: 'Foo' }, { message: 'Bar' }]
  }
}
```

```
<li v-for="item in items">
  {{ item.message }}
</li>
```

Inside the `v-for` scope, template expressions have access to all parent scope properties. In addition, `v-for` also supports an optional second alias for the index of the current item:

```
const parentMessage = ref('Parent')
const items = ref([{ message: 'Foo' }, { message: 'Bar' }])
```

```
data() {
  return {
    parentMessage: 'Parent',
    items: [{ message: 'Foo' }, { message: 'Bar' }]
  }
}
```

```
<li v-for="(item, index) in items">
  {{ parentMessage }} - {{ index }} - {{ item.message }}
</li>
```

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

[Try it in the Playground](#)

[Try it in the Playground](#)

The variable scoping of `v-for` is similar to the following JavaScript:

```
const parentMessage = 'Parent'
const items = [
  /* ... */
]

items.forEach((item, index) => {
  // has access to outer scope `parentMessage`
```

```
// but `item` and `index` are only available in here
  console.log(parentMessage, item.message, index)
})
```

Notice how the `v-for` value matches the function signature of the `forEach` callback. In fact, you can use destructuring on the `v-for` item alias similar to destructuring function arguments:

```
<li v-for="{ message } in items">
  {{ message }}
</li>

<!-- with index alias --&gt;
&lt;li v-for="({ message }, index) in items"&gt;
  {{ message }} {{ index }}
&lt;/li&gt;</pre>

```

For nested `v-for`, scoping also works similar to nested functions. Each `v-for` scope has access to parent scopes:

```
<li v-for="item in items">
  <span v-for="childItem in item.children">
    {{ item.message }} {{ childItem }}
  </span>
</li>
```

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

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

## v-for with an Object

You can also use `v-for` to iterate through the properties of an object. The iteration order will be based on the result of calling `Object.keys()` on the object:

```
const myObject = reactive({
  title: 'How to do lists in Vue',
  author: 'Jane Doe',
  publishedAt: '2016-04-10'
})
```

```
data() {
  return {
    myObject: {
      title: 'How to do lists in Vue',
      author: 'Jane Doe',
      publishedAt: '2016-04-10'
    }
  }
}
```

```
<ul>
  <li v-for="value in myObject">
    {{ value }}
  </li>
</ul>
```

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

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

And another for the index:

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

Hello from the modal!

[Try it in the Playground](#)

[Try it in the Playground](#)

## v-for with a Range

`v-for` can also take an integer. In this case it will repeat the template that many times, based on a range of `1...n`.

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

Note here `n` starts with an initial value of `1` instead of `0`.

## v-for on <template> {#v-for-on-template}

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

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

## v-for with v-if {#v-for-with-v-if}

:::warning Note It's **not** recommended to use `v-if` and `v-for` on the same element due to implicit precedence. Refer to [style guide](#) for details. :::

When they exist on the same node, `v-if` has a higher priority than `v-for`. That means the `v-if` condition will not have access to variables from the scope of the `v-for`:

```
<!--
This will throw an error because property "todo"
is not defined on instance.
-->
```

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

This can be fixed by moving `v-for` to a wrapping `<template>` tag (which is also more explicit):

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

Hello from the modal!

## Maintaining State with `key` {#maintaining-state-with-key}

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

This 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" :key="item.id">
  <!-- content -->
</div>
```

When using `<template v-for>`, the `key` should be placed on the `<template>` container:

```
<template v-for="todo in todos" :key="todo.name">
  <li>{{ todo.name }}</li>
</template>
```

:::tip Note `key` here is a special attribute being bound with `v-bind`. It should not be confused with the property `key` variable when [using `v-for` with an object](#). :::

[It is recommended](#) to provide a `key` attribute with `v-for` whenever possible, unless the iterated DOM content is simple (i.e. contains no components or stateful DOM elements), or you are intentionally relying on the default behavior for performance gains.

The `key` binding expects primitive values - i.e. strings and numbers. Do not use objects as `v-for` keys. For detailed usage of the `key` attribute, please see the [key API documentation](#).

## v-for with a Component

This section assumes knowledge of [Components](#). Feel free to skip it and come back later.

You can directly use `v-for` on a component, like any normal element (don't forget to provide a `key`):

```
<MyComponent v-for="item in items" :key="item.id" />
```

Hello from the modal!

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:

```
<MyComponent
  v-for="(item, index) in items"
  :item="item"
  :index="index"
  :key="item.id"
/>
```

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.

Check out [this example of a simple todo list](#) to see how to render a list of components using `v-for`, passing different data to each instance.

Check out [this example of a simple todo list](#) to see how to render a list of components using `v-for`, passing different data to each instance.

## Array Change Detection

### Mutation Methods

Vue is able to detect when a reactive array's mutation methods are called and trigger necessary updates. These mutation methods are:

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

## 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, we should replace the old array with the new one:

```
// `items` is a ref with array value
items.value = items.value.filter((item) => item.message.match(/Foo/))

this.items = this.items.filter((item) => 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.

## 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:

```
const numbers = ref([1, 2, 3, 4, 5])

const evenNumbers = computed(() => {
  return numbers.value.filter((n) => n % 2 === 0)
})

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

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

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

```
const sets = ref([
  [1, 2, 3, 4, 5],
  [6, 7, 8, 9, 10]
])

function even(numbers) {
  return numbers.filter((number) => number % 2 === 0)
}
```

```
data() {
  return {
    sets: [[ 1, 2, 3, 4, 5 ], [6, 7, 8, 9, 10]]
  }
},
methods: {
  even(numbers) {
    return numbers.filter(number => number % 2 === 0)
  }
}

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

Be careful with `reverse()` and `sort()` in a computed property! These two methods will mutate the original array, which should be avoided in computed getters. Create a copy of the original array before calling these methods:

```
- return numbers.reverse()
+ return [...numbers].reverse()
```

# Reactivity Fundamentals

:::tip API Preference This page and many other chapters later in the guide contain different content for Options API and Composition API. Your current preference is Options APIComposition API. You can toggle between the API styles using the "API Preference" switches at the top of the left sidebar. :::

## Declaring Reactive State

Hello from the modal!

With Options API, we use the `data` option to declare reactive state of a component. The option value should be a function that returns an object. Vue will call the function when creating a new component instance, and wrap the returned object in its reactivity system. Any top-level properties of this object are proxied on the component instance (`this` in methods and lifecycle hooks):

```
export default {
  data() {
    return {
      count: 1
    },
    // `mounted` is a lifecycle hook which we will explain later
    mounted() {
      // `this` refers to the component instance.
      console.log(this.count) // => 1
      // data can be mutated as well
      this.count = 2
    }
}
```

[Try it in the Playground](#)

These instance properties are only added when the instance is first created, so you need to ensure they are all present in the object returned by the `data` function. Where necessary, use `null`, `undefined` or some other placeholder value for properties where the desired value isn't yet available.

It is possible to add a new property directly to `this` without including it in `data`. However, properties added this way will not be able to trigger reactive updates.

Vue uses a `$` prefix when exposing its own built-in APIs via the component instance. It also reserves the prefix `_` for internal properties. You should avoid using names for top-level `data` properties that start with either of these characters.

## Reactive Proxy vs. Original \* {#reactive-proxy-vs-original}

In Vue 3, data is made reactive by leveraging [JavaScript Proxies](#). Users coming from Vue 2 should be aware of the following edge case:

```

export default {
  data() {
    return {
      someObject: {}
    }
  },
  mounted() {
    const newObject = {}
    this.someObject = newObject

    console.log(newObject === this.someObject) // false
  }
}

```

When you access `this.someObject` after assigning it, the value is a reactive proxy of the original `newObject`. **Unlike in Vue 2, the original `newObject` is left intact and will not be made reactive: make sure to always access reactive state as a property of `this`.**

We can create a reactive object or array with the `reactive()` function:

```

import { reactive } from 'vue'

const state = reactive({ count: 0 })

```

Reactive objects are [JavaScript Proxies](#) and behave just like normal objects. The difference is that Vue is able to track the property access and mutations of a reactive object. If you are curious about the details, we explain how Vue's reactivity system works in [Reactivity in Depth](#) - but we recommend reading it after you have finished the main guide.

See also: [Typing Reactive](#)

To use reactive state in a component's template, declare and return them from a component's `setup()` function:

```

import { reactive } from 'vue'

export default {
  // `setup` is a special hook dedicated for composition API.
  setup() {
    const state = reactive({ count: 0 })

    // expose the state to the template
    return {
      state
    }
  }
}

```

```
<div>{{ state.count }}</div>
```

Similarly, we can declare functions that mutate reactive state in the same scope and expose it as a method alongside the state:

```
import { reactive } from 'vue'

export default {
  setup() {
    const state = reactive({ count: 0 })

    function increment() {
      state.count++
    }

    // don't forget to expose the function as well.
    return {
      state,
      increment
    }
  }
}
```

Exposed methods are typically used as event listeners:

```
<button @click="increment">
  {{ state.count }}
</button>
```

## `<script setup> ** {#script-setup}`

Manually exposing state and methods via `setup()` can be verbose. Luckily, it is only necessary when not using a build step. When using Single-File Components (SFCs), we can greatly simplify the usage with `<script setup>`:

```
<script setup>
import { reactive } from 'vue'

const state = reactive({ count: 0 })

function increment() {
  state.count++
}

</script>

<template>
  <button @click="increment">
    {{ state.count }}
  </button>
</template>
```

[Try it in the Playground](#)

Top-level imports and variables declared in `<script setup>` are automatically usable in the template of the same component.

For the rest of the guide, we will be primarily using SFC + `<script setup>` syntax for Composition API code examples, as that is the most common usage for Vue developers.

## Declaring Methods \* {#declaring-methods}

To add methods to a component instance we use the `methods` option. This should be an object containing the desired methods:

```
export default {
  data() {
    return {
      count: 0
    }
  },
  methods: {
    increment() {
      this.count++
    }
  },
  mounted() {
    // methods can be called in lifecycle hooks, or other methods!
    this.increment()
  }
}
```

Vue automatically binds the `this` value for `methods` so that it always refers to the component instance. This ensures that a method retains the correct `this` value if it's used as an event listener or callback. You should avoid using arrow functions when defining `methods`, as that prevents Vue from binding the appropriate `this` value:

```
export default {
  methods: {
    increment: () => {
      // BAD: no `this` access here!
    }
  }
}
```

Just like all other properties of the component instance, the `methods` are accessible from within the component's template. Inside a template they are most commonly used as event listeners:

```
<button @click="increment">{{ count }}</button>
```

[Try it in the Playground](#)

In the example above, the method `increment` will be called when the `<button>` is clicked.

## DOM Update Timing

When you mutate reactive state, the DOM is updated automatically. However, it should be noted that the DOM updates are not applied synchronously. Instead, Vue buffers them until the "next tick" in the update cycle to ensure that each component updates only once no matter how many state changes you have made.

To wait for the DOM update to complete after a state change, you can use the `nextTick()` global API:

```
import { nextTick } from 'vue'

function increment() {
  state.count++
  nextTick(() => {
    // access updated DOM
  })
}
```

```
import { nextTick } from 'vue'

export default {
  methods: {
    increment() {
      this.count++
      nextTick(() => {
        // access updated DOM
      })
    }
  }
}
```

## Deep Reactivity

In Vue, state is deeply reactive by default. This means you can expect changes to be detected even when you mutate nested objects or arrays:

```
export default {
  data() {
    return {
      obj: {
        nested: { count: 0 },
        arr: ['foo', 'bar']
      }
    },
    methods: {
      mutateDeeply() {
        // these will work as expected.
        this.obj.nested.count++
        this.obj.arr.push('baz')
      }
    }
}
```

```
import { reactive } from 'vue'

const obj = reactive({
  nested: { count: 0 },
  arr: ['foo', 'bar']
})

function mutateDeeply() {
  // these will work as expected.
  obj.nested.count++
  obj.arr.push('baz')
}
```

It is also possible to explicitly create [shallow reactive objects](#) where the reactivity is only tracked at the root-level, but these are typically only needed in advanced use cases.

## Reactive Proxy vs. Original \*\* {#reactive-proxy-vs-original-1}

It is important to note that the returned value from `reactive()` is a [Proxy](#) of the original object, which is not equal to the original object:

```
const raw = []
const proxy = reactive(raw)

// proxy is NOT equal to the original.
console.log(proxy === raw) // false
```

Only the proxy is reactive - mutating the original object will not trigger updates. Therefore, the best practice when working with Vue's reactivity system is to **exclusively use the proxied versions of your state**.

To ensure consistent access to the proxy, calling `reactive()` on the same object always returns the same proxy, and calling `reactive()` on an existing proxy also returns that same proxy:

```
// calling reactive() on the same object returns the same proxy
console.log(reactive(raw) === proxy) // true

// calling reactive() on a proxy returns itself
console.log(reactive(proxy) === proxy) // true
```

This rule applies to nested objects as well. Due to deep reactivity, nested objects inside a reactive object are also proxies:

```
const proxy = reactive({})
const raw = []
proxy.nested = raw

console.log(proxy.nested === raw) // false
```

## Limitations of `reactive()` \*\* {#limitations-of-reactive}

The `reactive()` API has two limitations:

1. It only works for object types (objects, arrays, and [collection types](#) such as `Map` and `Set`). It cannot hold [primitive types](#) such as `string`, `number` or `boolean`.
2. Since Vue's reactivity tracking works over property access, we must always keep the same reference to the reactive object. This means we can't easily "replace" a reactive object because the reactivity connection to the first reference is lost:

```
let state = reactive({ count: 0 })

// the above reference ({ count: 0 }) is no longer being tracked (reactivity
// connection is lost!)
state = reactive({ count: 1 })
```

It also means that when we assign or destructure a reactive object's property into local variables, or when we pass that property into a function, we will lose the reactivity connection:

```
const state = reactive({ count: 0 })

// n is a local variable that is disconnected
// from state.count.
let n = state.count
// does not affect original state
n++

// count is also disconnected from state.count.
let { count } = state
// does not affect original state
count++

// the function receives a plain number and
// won't be able to track changes to state.count
callSomeFunction(state.count)
```

## Reactive Variables with `ref()` \*\* {#reactive-variables-with-ref}

To address the limitations of `reactive()`, Vue also provides a `ref()` function which allows us to create reactive "refs" that can hold any value type:

```
import { ref } from 'vue'

const count = ref(0)
```

`ref()` takes the argument and returns it wrapped within a `ref` object with a `.value` property:

```
const count = ref(0)

console.log(count) // { value: 0 }
console.log(count.value) // 0

count.value++
console.log(count.value) // 1
```

See also: [Typing Refs](#)

Similar to properties on a reactive object, the `.value` property of a `ref` is reactive. In addition, when holding object types, `ref` automatically converts its `.value` with `reactive()`.

A `ref` containing an object value can reactively replace the entire object:

```
const objectRef = ref({ count: 0 })

// this works reactively
objectRef.value = { count: 1 }
```

Refs can also be passed into functions or destructured from plain objects without losing reactivity:

```

const obj = {
  foo: ref(1),
  bar: ref(2)
}

// the function receives a ref
// it needs to access the value via .value but it
// will retain the reactivity connection
callSomeFunction(obj.foo)

// still reactive
const { foo, bar } = obj

```

Hello from the modal!

In other words, `ref()` allows us to create a "reference" to any value and pass it around without losing reactivity. This capability is quite important as it is frequently used when extracting logic into [Composable Functions](#).

## Ref Unwrapping in Templates \*\* {#ref-unwrapping-in-templates}

When refs are accessed as top-level properties in the template, they are automatically "unwrapped" so there is no need to use `.value`. Here's the previous counter example, using `ref()` instead:

```

<script setup>
import { ref } from 'vue'

const count = ref(0)

function increment() {
  count.value++
}

</script>

<template>
  <button @click="increment">
    {{ count }} <!-- no .value needed -->
  </button>
</template>

```

[Try it in the Playground](#)

Note that the unwrapping only applies if the ref is a top-level property on the template render context. As an example, `foo` is a top-level property, but `object.foo` is not.

So, given the following object:

```
const object = { foo: ref(1) }
```

The following expression will **NOT** work as expected:

```
{{ object.foo + 1 }}
```

The rendered result will be `[object Object]` because `object.foo` is a ref object. We can fix that by making `foo` a top-level property:

```
const { foo } = object
```

```
  {{ foo + 1 }}
```

Now the render result will be `2`.

One thing to note is that a `ref` will also be unwrapped if it is the final evaluated value of a text interpolation (i.e. a `{} tag`), so the following will render `1`:

```
  {{ object.foo }}
```

This is just a convenience feature of text interpolation and is equivalent to `{} object.foo.value`.

Hello from the modal!

## Ref Unwrapping in Reactive Objects \*\* {#ref-unwrapping-in-reactive-objects}

When a `ref` is accessed or mutated as a property of a reactive object, it is also automatically unwrapped so it behaves like a normal property:

```
const count = ref(0)
const state = reactive({
  count
})

console.log(state.count) // 0

state.count = 1
console.log(count.value) // 1
```

If a new `ref` is assigned to a property linked to an existing `ref`, it will replace the old `ref`:

```
const otherCount = ref(2)

state.count = otherCount
console.log(state.count) // 2
// original ref is now disconnected from state.count
console.log(count.value) // 1
```

Ref unwrapping only happens when nested inside a deep reactive object. It does not apply when it is accessed as a property of a [shallow reactive object](#).

## Ref Unwrapping in Arrays and Collections

Unlike reactive objects, there is no unwrapping performed when the `ref` is accessed as an element of a reactive array or a native collection type like [Map](#):

```
const books = reactive([ref('Vue 3 Guide')])
// need .value here
console.log(books[0].value)

const map = reactive(new Map([('count', ref(0))]))
// need .value here
console.log(map.get('count').value)
```

## Stateful Methods \* {#stateful-methods}

In some cases, we may need to dynamically create a method function, for example creating a debounced event handler:

```
import { debounce } from 'lodash-es'

export default {
  methods: {
    // Debouncing with Lodash
    click: debounce(function () {
      // ... respond to click ...
    }, 500)
  }
}
```

However, this approach is problematic for components that are reused because a debounced function is **stateful**: it maintains some internal state on the elapsed time. If multiple component instances share the same debounced function, they will interfere with one another.

To keep each component instance's debounced function independent of the others, we can create the debounced version in the `created` lifecycle hook:

```
export default {
  created() {
    // each instance now has its own copy of debounced handler
    this.debouncedClick = _.debounce(this.click, 500)
  },
  unmounted() {
    // also a good idea to cancel the timer
    // when the component is removed
    this.debouncedClick.cancel()
  },
  methods: {
    click() {
      // ... respond to click ...
    }
  }
}
```

## Reactivity Transform \*\* {#reactivity-transform}

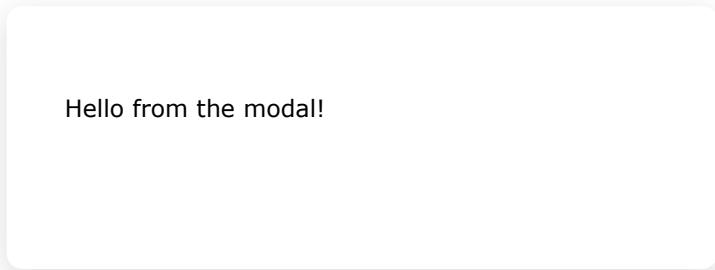
Having to use `.value` with refs is a drawback imposed by the language constraints of JavaScript. However, with compile-time transforms we can improve the ergonomics by automatically appending `.value` in appropriate locations. Vue provides a compile-time transform that allows us to write the earlier "counter" example like this:

```
<script setup>
let count = $ref(0)

function increment() {
  // no need for .value
  count++
}
</script>
```

```
<template>
  <button @click="increment">{{ count }}</button>
</template>
```

You can learn more about [Reactivity Transform](#) in its dedicated section. Do note that it is currently still experimental and may change before being finalized.



Hello from the modal!

---

[Go to TOC](#)

# Template Refs

While Vue's declarative rendering model abstracts away most of the direct DOM operations for you, there may still be cases where we need direct access to the underlying DOM elements. To achieve this, we can use the special `ref` attribute:

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

Hello from the modal!

`ref` is a special attribute, similar to the `key` attribute discussed in the `v-for` chapter. It allows us to obtain a direct reference to a specific DOM element or child component instance after it's mounted. This may be useful when you want to, for example, programmatically focus an input on component mount, or initialize a 3rd party library on an element.

## Accessing the Refs

To obtain the reference with Composition API, we need to declare a ref with the same name:

```
<script setup>
import { ref, onMounted } from 'vue'

// declare a ref to hold the element reference
// the name must match template ref value
const input = ref(null)

onMounted(() => {
  input.value.focus()
})
</script>

<template>
  <input ref="input" />
</template>
```

If not using `<script setup>`, make sure to also return the ref from `setup()`:

```
export default {
  setup() {
    const input = ref(null)
    // ...
    return {
      input
    }
  }
}
```

The resulting ref is exposed on `this.$refs`:

```
<script>
export default {
  mounted() {
    this.$refs.input.focus()
  }
}
```

```

    }
</script>

<template>
  <input ref="input" />
</template>

```

Note that you can only access the ref **after the component is mounted**. If you try to access `$refs.input` in a template expression, it will be `null` on the first render. This is because the element doesn't exist until after the first render!

If you are trying to watch the changes from the template, make sure to account for the case where the ref has `null` value:

```

watchEffect(() => {
  if (input.value) {
    input.value.focus()
  } else {
    // not mounted yet, or the element was unmounted (e.g. by v-if)
  }
})

```

See also: [Typing Template Refs](#)

## Refs inside `v-for` {#refs-inside-v-for}

Requires v3.2.25 or above

When `ref` is used inside `v-for`, the corresponding ref should contain an Array value, which will be populated with the elements after mount:

```

<script setup>
import { ref, onMounted } from 'vue'

const list = ref([
  /* ... */
])

const itemRefs = ref([])

onMounted(() => console.log(itemRefs.value))
</script>

<template>
  <ul>
    <li v-for="item in list" ref="itemRefs">
      {{ item }}
    </li>
  </ul>
</template>

```

[Try it in the Playground](#)

When `ref` is used inside `v-for`, the resulting ref value will be an array containing the corresponding elements:

```
<script>
export default {
  data() {
    return {
      list: [
        /* ... */
      ]
    },
    mounted() {
      console.log(this.$refs.items)
    }
}
</script>

<template>
<ul>
  <li v-for="item in list" ref="items">
    {{ item }}
  </li>
</ul>
</template>
```

[Try it in the Playground](#)

It should be noted that the ref array does **not** guarantee the same order as the source array.

## Function Refs

Instead of a string key, the `ref` attribute can also be bound to a function, which will be called on each component update and gives you full flexibility on where to store the element reference. The function receives the element reference as the first argument:

```
<input :ref="(el) => { /* assign el to a property or ref */ }">
```

Note we are using a dynamic `:ref` binding so we can pass it a function instead of a ref name string. When the element is unmounted, the argument will be `null`. You can, of course, use a method instead of an inline function.

## Ref on Component

This section assumes knowledge of [Components](#). Feel free to skip it and come back later.

`ref` can also be used on a child component. In this case the reference will be that of a component instance:

```
<script setup>
import { ref, onMounted } from 'vue'
import Child from './Child.vue'

const child = ref(null)

onMounted(() => {
  // child.value will hold an instance of <Child />
})
</script>

<template>
  <Child ref="child" />
</template>
```

```
<script>
import Child from './Child.vue'

export default {
  components: {
    Child
  },
  mounted() {
    // this.$refs.child will hold an instance of <Child />
  }
}
</script>

<template>
  <Child ref="child" />
</template>
```

If the child component is using Options API or not using `<script setup>`, the referenced instance will be identical to the child component's `this`, which means the parent component will have full access to every property and method of the child component. This makes it easy to create tightly coupled implementation details between the parent and the child, so component refs should be only used when absolutely needed - in most cases, you should try to implement parent / child interactions using the standard props and emit interfaces first.

An exception here is that components using `<script setup>` are **private by default**: a parent component referencing a child component using `<script setup>` won't be able to access anything unless the child component chooses to expose a public interface using the `defineExpose` macro:

```
<script setup>
import { ref } from 'vue'

const a = 1
const b = ref(2)

defineExpose({
  a,
  b
})
</script>
```

When a parent gets an instance of this component via template refs, the retrieved instance will be of the shape `{ a: number, b: number }` (refs are automatically unwrapped just like on normal instances).

See also: [Typing Component Template Refs](#)

The `expose` option can be used to limit the access to a child instance:

```
export default {
  expose: ['publicData', 'publicMethod'],
  data() {
    return {
      publicData: 'foo',
      privateData: 'bar'
    }
  },
  methods: {
    publicMethod() {
      /* ... */
    },
    privateMethod() {
      /* ... */
    }
  }
}
```

In the above example, a parent referencing this component via template ref will only be able to access `publicData` and `publicMethod`.

# Template Syntax

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

Under the hood, Vue compiles the templates into highly-optimized JavaScript code. Combined with the reactivity system, Vue can intelligently figure out the minimal number of components to re-render and apply the minimal amount of DOM manipulations when the app state changes.

If you are familiar with Virtual DOM concepts and prefer the raw power of JavaScript, you can also [directly write render functions](#) instead of templates, with optional JSX support. However, do note that they do not enjoy the same level of compile-time optimizations as templates.

## Text Interpolation

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

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

The mustache tag will be replaced with the value of the `msg` property from the corresponding component instance. It will also be updated whenever the `msg` property changes.

## Raw HTML

The double mustaches interpret the data as plain text, not HTML. In order to output real HTML, you will need to use the `v-html` directive:

```
<p>Using text interpolation: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

Using text interpolation: {{ rawHtml }}

Using v-html directive:

Here we're encountering something new. The `v-html` attribute you're 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, we're basically saying "keep this element's inner HTML up-to-date with the `rawHtml` property on the current active instance."

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.

:::warning Security Warning Dynamically rendering arbitrary HTML on your website can be very dangerous because it can easily lead to [XSS vulnerabilities](#). Only use `v-html` on trusted content and **never** on user-provided content. :::

## Attribute Bindings

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

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

Hello from the modal!

The `v-bind` directive instructs Vue to keep the element's `id` attribute in sync with the component's `dynamicId` property. If the bound value is `null` or `undefined`, then the attribute will be removed from the rendered element.

### Shorthand

Because `v-bind` is so commonly used, it has a dedicated shorthand syntax:

```
<div :id="dynamicId"></div>
```

Attributes that start with `:` may look a bit different from normal HTML, but it is in fact a valid character 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 optional, but you will likely appreciate it when you learn more about its usage later.

For the rest of the guide, we will be using the shorthand syntax in code examples, as that's the most common usage for Vue developers.

## Boolean Attributes

[Boolean attributes](#) are attributes that can indicate true / false values by its presence on an element. For example, `disabled` is one of the most commonly used boolean attributes.

`v-bind` works a bit differently in this case:

```
<button :disabled="isButtonDisabled">Button</button>
```

The `disabled` attribute will be included if `isButtonDisabled` has a [truthy value](#). It will also be included if the value is an empty string, maintaining consistency with `<button disabled="">`. For other [falsy values](#) the attribute will be omitted.

## Dynamically Binding Multiple Attributes

If you have a JavaScript object representing multiple attributes that looks like this:

```
const objectOfAttrs = {
  id: 'container',
  class: 'wrapper'
}
```

```
data() {
  return {
    objectOfAttrs: {
      id: 'container',
      class: 'wrapper'
    }
  }
}
```

You can bind them to a single element by using `v-bind` without an argument:

```
<div v-bind="objectOfAttrs"></div>
```

## Using JavaScript Expressions

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

```
{{ number + 1 }}
{{ ok ? 'YES' : 'NO' }}
{{ message.split('').reverse().join('') }}
<div :id=`list-${id}`></div>
```

These expressions will be evaluated as JavaScript in the data scope of the current component instance.

In Vue templates, JavaScript expressions can be used in the following positions:

- Inside text interpolations (mustaches)
- In the attribute value of any Vue directives (special attributes that start with `v-`)

## Expressions Only

Each binding can only contain **one single expression**. An expression is a piece of code that can be evaluated to a value. A simple check is whether it can be used after `return`.

Therefore, the following will **NOT** work:

```
<!-- this is a statement, not an expression: -->
{{ var a = 1 }}

<!-- flow control won't work either, use ternary expressions -->
{{ if (ok) { return message } }}
```

## Calling Functions

It is possible to call a component-exposed method inside a binding expression:

```
<span :title="toTitleDate(date)">
  {{ formatDate(date) }}
</span>
```

:::tip Functions called inside binding expressions will be called every time the component updates, so they should **not** have any side effects, such as changing data or triggering asynchronous operations. :::

## Restricted Globals Access

Template expressions are sandboxed and only have access to a [restricted list of globals](#). The list exposes commonly used built-in globals such as `Math` and `Date`.

Globals not explicitly included in the list, for example user-attached properties on `window`, will not be accessible in template expressions. You can, however, explicitly define additional globals for all Vue expressions by adding them to `app.config.globalProperties`.

## Directives

Directives are special attributes with the `v-` prefix. Vue provides a number of [built-in directives](#), including `v-html` and `v-bind` which we have introduced above.

Directive attribute values are expected to be single JavaScript expressions (with the exception of `v-for`, `v-on` and `v-slot`, which will be discussed in their respective sections later). A directive's job is to reactively apply updates to the DOM when the value of its expression changes. Take `v-if` as an example:

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

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

## Arguments

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

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

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

Here, `href` is the argument, which tells the `v-bind` directive to bind the element's `href` attribute to the value of the expression `url`. In the shorthand, everything before the argument (i.e., `v-bind:`) is condensed into a single character, `:`.

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

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

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

Here, the argument is the event name to listen to: `click`. `v-on` has a corresponding shorthand, namely the `@` character. We will talk about event handling in more detail too.

## Dynamic Arguments

It is also possible to use a JavaScript expression in a directive argument by wrapping it with square brackets:

```
<!--
Note that there are some constraints to the argument expression,
as explained in the "Dynamic Argument Value Constraints" and "Dynamic Argument
Syntax Constraints" sections below.
-->
<a v-bind:[attributeName]="url"> ... </a>

<!-- shorthand -->
<a :[attributeName]="url"> ... </a>
```

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

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

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

<!-- shorthand -->
<a @[eventName]="doSomething">
```

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

### Dynamic Argument Value Constraints

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

### Dynamic Argument Syntax Constraints

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

```
<!-- This will trigger a compiler warning. -->
<a :['foo' + bar]="value"> ... </a>
```

If you need to pass a complex dynamic argument, it's probably better to use a [computed property](#), which we will cover shortly.

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

```
<a :[someAttr]="value"> ... </a>
```

The above will be converted to `:[someattr]` in in-DOM templates. If your component has a `someAttr` property instead of `someattr`, your code won't work. Templates inside Single-File Components are **not** subject to this constraint.

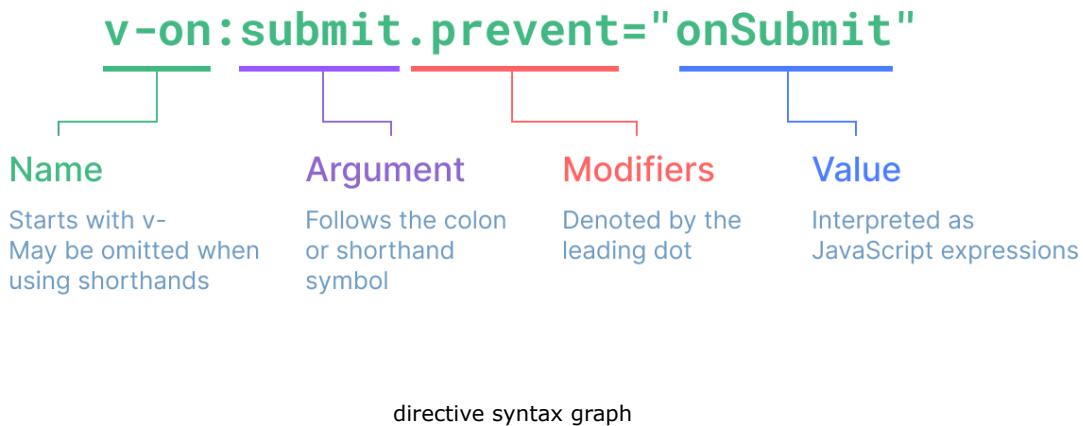
## Modifiers

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

```
<form @submit.prevent="onSubmit">...</form>
```

You'll see other examples of modifiers later, `for v-on` and `for v-model`, when we explore those features.

And finally, here's the full directive syntax visualized:



# Watchers

## Basic Example

Computed properties allow us to declaratively compute derived values. However, there are cases where we need to perform "side effects" in reaction to state changes - for example, mutating the DOM, or changing another piece of state based on the result of an async operation.

Hello from the modal!

With the Options API, we can use the `watch` option to trigger a function whenever a reactive property changes:

```
export default {
  data() {
    return {
      question: '',
      answer: 'Questions usually contain a question mark. ;-)'
    }
  },
  watch: {
    // whenever question changes, this function will run
    question(newQuestion, oldQuestion) {
      if (newQuestion.includes('?')) {
        this.getAnswer()
      }
    }
  },
  methods: {
    async getAnswer() {
      this.answer = 'Thinking...'
      try {
        const res = await fetch('https://yesno.wtf/api')
        this.answer = (await res.json()).answer
      } catch (error) {
        this.answer = 'Error! Could not reach the API. ' + error
      }
    }
  }
}
```

```
<p>
  Ask a yes/no question:
  <input v-model="question" />
</p>
<p>{{ answer }}</p>
```

[Try it in the Playground](#)

The `watch` option also supports a dot-delimited path as the key:

```
export default {
  watch: {
    // Note: only simple paths. Expressions are not supported.
    'some.nested.key'(newValue) {
      // ...
    }
  }
}
```

```

        }
    }
}
```

With Composition API, we can use the `watch` function to trigger a callback whenever a piece of reactive state changes:

```

<script setup>
import { ref, watch } from 'vue'

const question = ref('')
const answer = ref('Questions usually contain a question mark. ;-)')

// watch works directly on a ref
watch(question, async (newQuestion, oldQuestion) => {
    if (newQuestion.indexOf('?') > -1) {
        answer.value = 'Thinking...'
        try {
            const res = await fetch('https://yesno.wtf/api')
            answer.value = (await res.json()).answer
        } catch (error) {
            answer.value = 'Error! Could not reach the API. ' + error
        }
    }
})
</script>

<template>
<p>
    Ask a yes/no question:
    <input v-model="question" />
</p>
<p>{{ answer }}</p>
</template>
```

[Try it in the Playground](#)

## Watch Source Types

`watch`'s first argument can be different types of reactive "sources": it can be a `ref` (including computed `refs`), a reactive object, a getter function, or an array of multiple sources:

```

const x = ref(0)
const y = ref(0)

// single ref
watch(x, (newX) => {
    console.log(`x is ${newX}`)
})

// getter
watch(
    () => x.value + y.value,
    (sum) => {
        console.log(`sum of x + y is: ${sum}`)
    }
)

// array of multiple sources
```

```
watch([x, () => y.value], ([ newX, newY ]) => {
  console.log(`x is ${newX} and y is ${newY}`)
})
```

Do note that you can't watch a property of a reactive object like this:

```
const obj = reactive({ count: 0 })

// this won't work because we are passing a number to watch()
watch(obj.count, (count) => {
  console.log(`count is: ${count}`)
})
```

Hello from the module

Instead, use a getter:

```
// instead, use a getter:
watch(
  () => obj.count,
  (count) => {
    console.log(`count is: ${count}`)
  }
)
```

## Deep Watchers

`watch` is shallow by default: the callback will only trigger when the watched property has been assigned a new value - it won't trigger on nested property changes. If you want the callback to fire on all nested mutations, you need to use a deep watcher:

```
export default {
  watch: [
    someObject: {
      handler(newValue, oldValue) {
        // Note: `newValue` will be equal to `oldValue` here
        // on nested mutations as long as the object itself
        // hasn't been replaced.
      },
      deep: true
    }
  ]
}
```

When you call `watch()` directly on a reactive object, it will implicitly create a deep watcher - the callback will be triggered on all nested mutations:

```
const obj = reactive({ count: 0 })

watch(obj, (newValue, oldValue) => {
  // fires on nested property mutations
  // Note: `newValue` will be equal to `oldValue` here
  // because they both point to the same object!
})

obj.count++
```

This should be differentiated with a getter that returns a reactive object - in the latter case, the callback will only fire if the getter returns a different object:

```
watch(
  () => state.someObject,
  () => {
    // fires only when state.someObject is replaced
  }
)
```

You can, however, force the second case into a deep watcher by explicitly using the `deep` option:  
Hello from the modal!

```
watch(
  () => state.someObject,
  (newValue, oldValue) => {
    // Note: `newValue` will be equal to `oldValue` here
    // *unless* state.someObject has been replaced
  },
  { deep: true }
)
```

:::warning Use with Caution Deep watch requires traversing all nested properties in the watched object, and can be expensive when used on large data structures. Use it only when necessary and beware of the performance implications. :::

## Eager Watchers \* {#eager-watchers}

`watch` is lazy by default: the callback won't be called until the watched source has changed. But in some cases we may want the same callback logic to be run eagerly - for example, we may want to fetch some initial data, and then re-fetch the data whenever relevant state changes.

We can force a watcher's callback to be executed immediately by declaring it using an object with a `handler` function and the `immediate: true` option:

```
export default {
  // ...
  watch: {
    question: {
      handler(newQuestion) {
        // this will be run immediately on component creation.
      },
      // force eager callback execution
      immediate: true
    }
  }
  // ...
}
```

The initial execution of the handler function will happen just before the `created` hook. Vue will have already processed the `data`, `computed`, and `methods` options, so those properties will be available on the first invocation.

## watchEffect() \*\* {#watcheffect}

`watch()` is lazy: the callback won't be called until the watched source has changed. But in some cases we may want the same callback logic to be run eagerly - for example, we may want to fetch some initial data, and then re-fetch the data whenever relevant state changes. We may find ourselves doing this:

```
const url = ref('https://...')  
const data = ref(null)  
  
async function fetchData() {  
  const response = await fetch(url.value)  
  data.value = await response.json()  
}  
  
// fetch immediately  
fetchData()  
// ...then watch for url change  
watch(url, fetchData)
```

This can be simplified with `watchEffect()`. `watchEffect()` allows us to perform a side effect immediately while automatically tracking the effect's reactive dependencies. The above example can be rewritten as:

```
watchEffect(async () => {  
  const response = await fetch(url.value)  
  data.value = await response.json()  
})
```

Here, the callback will run immediately. During its execution, it will also automatically track `url.value` as a dependency (similar to computed properties). Whenever `url.value` changes, the callback will be run again.

You can check out [this example](#) with `watchEffect` and reactive data-fetching in action.

:::tip `watchEffect` only tracks dependencies during its **synchronous** execution. When using it with an `async` callback, only properties accessed before the first `await` tick will be tracked. :::

## watch vs. watchEffect {#watch-vs-watcheffect}

`watch` and `watchEffect` both allow us to reactively perform side effects. Their main difference is the way they track their reactive dependencies:

- `watch` only tracks the explicitly watched source. It won't track anything accessed inside the callback. In addition, the callback only triggers when the source has actually changed. `watch` separates dependency tracking from the side effect, giving us more precise control over when the callback should fire.
- `watchEffect`, on the other hand, combines dependency tracking and side effect into one phase. It automatically tracks every reactive property accessed during its synchronous execution. This is more convenient and typically results in terser code, but makes its reactive dependencies less explicit.

# Callback Flush Timing

When you mutate reactive state, it may trigger both Vue component updates and watcher callbacks created by you.

By default, user-created watcher callbacks are called **before** Vue component updates. This means if you attempt to access the DOM inside a watcher callback, the DOM will be in the state before Vue has applied any updates.

If you want to access the DOM in a watcher callback **after** Vue has updated it, you need to specify the `flush: 'post'` option:

```
export default {
  // ...
  watch: {
    key: {
      handler() {},
      flush: 'post'
    }
  }
}
```

```
watch(source, callback, {
  flush: 'post'
})

watchEffect(callback, {
  flush: 'post'
})
```

Post-flush `watchEffect()` also has a convenience alias, `watchPostEffect()`:

```
import { watchPostEffect } from 'vue'

watchPostEffect(() => {
  /* executed after Vue updates */
})
```

## `this.$watch() * {#this-watch}`

It's also possible to imperatively create watchers using the `$watch()` instance method:

```
export default {
  created() {
    this.$watch('question', (newQuestion) => {
      // ...
    })
  }
}
```

This is useful when you need to conditionally set up a watcher, or only watch something in response to user interaction. It also allows you to stop the watcher early.

## Stopping a Watcher

Watchers declared using the `watch` option or the `$watch()` instance method are automatically stopped when the owner component is unmounted, so in most cases you don't need to worry about stopping the watcher yourself.

In the rare case where you need to stop a watcher before the owner component unmounts, the `$watch()` API returns a function for that:

```
const unwatch = this.$watch('foo', callback)
// ...when the watcher is no longer needed:
unwatch()
```

Watchers declared synchronously inside `setup()` or `<script setup>` are bound to the owner component instance, and will be automatically stopped when the owner component is unmounted. In most cases, you don't need to worry about stopping the watcher yourself.

The key here is that the watcher must be created **synchronously**: if the watcher is created in an async callback, it won't be bound to the owner component and must be stopped manually to avoid memory leaks. Here's an example:

```
<script setup>
import { watchEffect } from 'vue'

// this one will be automatically stopped
watchEffect(() => {})

// ...this one will not!
setTimeout(() => {
  watchEffect(() => {})
}, 100)
</script>
```

To manually stop a watcher, use the returned handle function. This works for both `watch` and `watchEffect`:

```
const unwatch = watchEffect(() => {})
// ...later, when no longer needed
unwatch()
```

Note that there should be very few cases where you need to create watchers asynchronously, and synchronous creation should be preferred whenever possible. If you need to wait for some async data, you can make your watch logic conditional instead:

```
// data to be loaded asynchronously
const data = ref(null)

watchEffect(() => {
  if (data.value) {
```

```
// do something when data is loaded  
}  
})
```

Hello from the modal!

# Animation Techniques

Vue provides the `<Transition>` and `<TransitionGroup>` components for handling enter / leave and list transitions. However, there are many other ways of using animations on the web, even in a Vue application. Here we will discuss a few additional techniques.

## Class-based Animations

Hello from the modal!

For elements that are not entering / leaving the DOM, we can trigger animations by dynamically adding a CSS class:

```
const disabled = ref(false)

function warnDisabled() {
  disabled.value = true
  setTimeout(() => {
    disabled.value = false
  }, 1500)
}
```

```
export default {
  data() {
    return {
      disabled: false
    }
  },
  methods: {
    warnDisabled() {
      this.disabled = true
      setTimeout(() => {
        this.disabled = false
      }, 1500)
    }
  }
}
```

```
<div :class="{ shake: disabled }">
  <button @click="warnDisabled">Click me</button>
  <span v-if="disabled">This feature is disabled!</span>
</div>
```

```
.shake {
  animation: shake 0.82s cubic-bezier(0.36, 0.07, 0.19, 0.97) both;
  transform: translate3d(0, 0, 0);
}

@keyframes shake {
  10%,
  90% {
    transform: translate3d(-1px, 0, 0);
  }

  20%,
  80% {
    transform: translate3d(2px, 0, 0);
  }
}
```

```

}
30%,
50%,
70% {
  transform: translate3d(-4px, 0, 0);
}

40%,
60% {
  transform: translate3d(4px, 0, 0);
}

```

## State-driven Animations

Some transition effects can be applied by interpolating values, for instance by binding a style to an element while an interaction occurs. Take this example for instance:

```

const x = ref(0)

function onMousemove(e) {
  x.value = e.clientX
}

```

```

export default {
  data() {
    return {
      x: 0
    }
  },
  methods: {
    onMousemove(e) {
      this.x = e.clientX
    }
  }
}

```

```

<div
  @mousemove="onMousemove"
  :style="`backgroundColor: hsl(${x}, 80%, 50%)`"
  class="movearea"
>
  <p>Move your mouse across this div...</p>
  <p>x: {{ x }}</p>
</div>

```

```

.movearea {
  transition: 0.3s background-color ease;
}

```

In addition to color, you can also use style bindings to animate transform, width, or height. You can even animate SVG paths using spring physics - after all, they are all attribute data bindings:

## Animating with Watchers

With some creativity, we can use watchers to animate anything based on some numerical state. For example, we can animate the number itself:

```
import { ref, reactive, watch } from 'vue'
import gsap from 'gsap'

const number = ref(0)
const tweened = reactive({
  number: 0
})

watch(number, (n) => {
  gsap.to(tweened, { duration: 0.5, number: Number(n) || 0 })
})
```

Type a number: <input v-model.number="number" />  
<p>{{ tweened.number.toFixed(0) }}</p>

```
import gsap from 'gsap'

export default {
  data() {
    return {
      number: 0,
      tweened: 0
    }
  },
  watch: {
    number(n) {
      gsap.to(this, { duration: 0.5, tweened: Number(n) || 0 })
    }
  }
}
```

Type a number: <input v-model.number="number" />  
<p>{{ tweened.toFixed(0) }}</p>

[Try it in the Playground](#)

[Try it in the Playground](#)

---

[Go to TOC](#)

# Composition API FAQ

:::tip This FAQ assumes prior experience with Vue - in particular, experience with Vue 2 while primarily using Options API. :::

## What is Composition API?

Composition API is a set of APIs that allows us to author Vue components using imported functions instead of declaring options. It is an umbrella term that covers the following APIs:

- [Reactivity API](#), e.g. `ref()` and `reactive()`, that allows us to directly create reactive state, computed state, and watchers.
- [Lifecycle Hooks](#), e.g. `onMounted()` and `onUnmounted()`, that allow us to programmatically hook into the component lifecycle.
- [Dependency Injection](#), i.e. `provide()` and `inject()`, that allow us to leverage Vue's dependency injection system while using Reactivity APIs.

Composition API is a built-in feature of Vue 3 and [Vue 2.7](#). For older Vue 2 versions, use the officially maintained `@vue/composition-api` plugin. In Vue 3, it is also primarily used together with the `<script setup>` syntax in Single-File Components. Here's a basic example of a component using Composition API:

```
<script setup>
import { ref, onMounted } from 'vue'

// reactive state
const count = ref(0)

// functions that mutate state and trigger updates
function increment() {
  count.value++
}

// lifecycle hooks
onMounted(() => {
  console.log(`The initial count is ${count.value}.`)
})
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

Despite an API style based on function composition, **Composition API is NOT functional programming**. Composition API is based on Vue's mutable, fine-grained reactivity paradigm, whereas functional programming emphasizes immutability.

If you are interested in learning how to use Vue with Composition API, you can set the site-wide API preference to Composition API using the toggle at the top of the left sidebar, and then go through the guide from the beginning.

## Why Composition API?

### Better Logic Reuse

The primary advantage of Composition API is that it enables clean, efficient logic reuse in the form of [Composable functions](#). It solves [all the drawbacks of mixins](#), the primary logic reuse mechanism for Options API.

Composition API's logic reuse capability has given rise to impressive community projects such as [VueUse](#), an ever-growing collection of composable utilities. It also serves as a clean mechanism for easily integrating stateful third-party services or libraries into Vue's reactivity system, for example [immutable data](#), [state machines](#), and [RxJS](#).

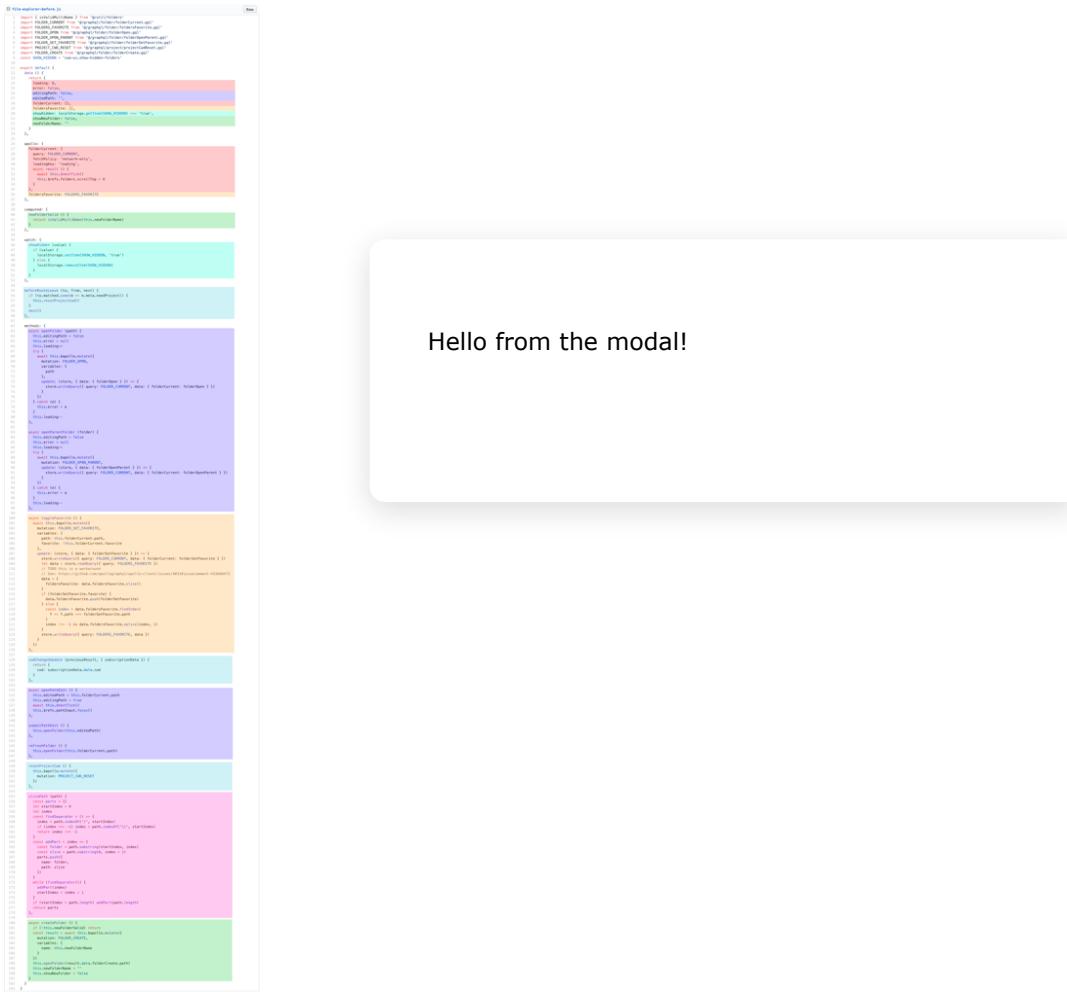
### More Flexible Code Organization

Many users love that we write organized code by default with Options API: everything has its place based on the option it falls under. However, Options API poses serious limitations when a single component's logic grows beyond a certain complexity threshold. This limitation is particularly prominent in components that need to deal with multiple **logical concerns**, which we have witnessed first hand in many production Vue 2 apps.

Take the folder explorer component from Vue CLI's GUI as an example: this component is responsible for the following logical concerns:

- Tracking current folder state and displaying its content
- Handling folder navigation (opening, closing, refreshing...)
- Handling new folder creation
- Toggling show favorite folders only
- Toggling show hidden folders
- Handling current working directory changes

The [original version](#) of the component was written in Options API. If we give each line of code a color based on the logical concern it is dealing with, this is how it looks:



Notice how code dealing with the same logical concern is forced to be split under different options, located in different parts of the file. In a component that is several hundred lines long, understanding and navigating a single logical concern requires constantly scrolling up and down the file, making it much more difficult than it should be. In addition, if we ever intend to extract a logical concern into a reusable utility, it takes quite a bit of work to find and extract the right pieces of code from different parts of the file.

Here's the same component, before and after the [refactor into Composition API](#):

## Options API



## Composition API



folder component after

Notice how the code related to the same logical concern can now be grouped together: we no longer need to jump between different options blocks while working on a specific logical concern. Moreover, we can now move a group of code into an external file with minimal effort, since we no longer need to shuffle the code around in order to extract them. This reduced friction for refactoring is key to the long-term maintainability in large codebases.

## Better Type Inference

In recent years, more and more frontend developers are adopting [TypeScript](#) as it helps us write more robust code, make changes with more confidence, and provides a great development experience with IDE support. However, the Options API, originally conceived in 2013, was designed without type inference in

mind. We had to implement some [absurdly complex type gymnastics](#) to make type inference work with the Options API. Even with all this effort, type inference for Options API can still break down for mixins and dependency injection.

This had led many developers who wanted to use Vue with TS to lean towards Class API powered by `vue-class-component`. However, a class-based API heavily relies on ES decorators, a language feature that was only a stage 2 proposal when Vue 3 was being developed in 2019. We felt it was too risky to base an official API on an unstable proposal. Since then, the decorators proposal has gone through yet another complete overhaul, and finally reached stage 3 in 2022. In addition, class-based API suffers from logic reuse and organization limitations similar to Options API from the modal!

In comparison, Composition API utilizes mostly plain variables and functions, which are naturally type friendly. Code written in Composition API can enjoy full type inference with little need for manual type hints. Most of the time, Composition API code will look largely identical in TypeScript and plain JavaScript. This also makes it possible for plain JavaScript users to benefit from partial type inference.

## Smaller Production Bundle and Less Overhead

Code written in Composition API and `<script setup>` is also more efficient and minification-friendly than Options API equivalent. This is because the template in a `<script setup>` component is compiled as a function inlined in the same scope of the `<script setup>` code. Unlike property access from `this`, the compiled template code can directly access variables declared inside `<script setup>`, without an instance proxy in between. This also leads to better minification because all the variable names can be safely shortened.

## Relationship with Options API

### Trade-offs

Some users moving from Options API found their Composition API code less organized, and concluded that Composition API is "worse" in terms of code organization. We recommend users with such opinions to look at that problem from a different perspective.

It is true that Composition API no longer provides the "guard rails" that guide you to put your code into respective buckets. In return, you get to author component code like how you would write normal JavaScript. This means **you can and should apply any code organization best practices to your Composition API code as you would when writing normal JavaScript**. If you can write well-organized JavaScript, you should also be able to write well-organized Composition API code.

Options API does allow you to "think less" when writing component code, which is why many users love it. However, in reducing the mental overhead, it also locks you into the prescribed code organization pattern with no escape hatch, which can make it difficult to refactor or improve code quality in larger scale projects. In this regard, Composition API provides better long term scalability.

## Does Composition API cover all use cases?

Yes in terms of stateful logic. When using Composition API, there are only a few options that may still be needed: `props`, `emits`, `name`, and `inheritAttrs`. If using `<script setup>`, then `inheritAttrs` is typically the only option that may require a separate normal `<script>` block.

If you intend to exclusively use Composition API (along with the options listed above), you can shave a few kbs off your production bundle via a [compile-time flag](#) that drops Options API related code from Vue. Note this also affects Vue components in your dependencies.

## Can I use both APIs together?

Yes. You can use Composition API via the `setup()` option in an Options API component.

However, we only recommend doing so if you have an existing Options API codebase that needs to integrate with new features / external libraries written with Composition API.

## Will Options API be deprecated?

No, we do not have any plan to do so. Options API is an integral part of Vue and the reason many developers love it. We also realize that many of the benefits of Composition API only manifest in larger-scale projects, and Options API remains a solid choice for many low-to-medium-complexity scenarios.

## Relationship with Class API

We no longer recommend using Class API with Vue 3, given that Composition API provides great TypeScript integration with additional logic reuse and code organization benefits.

## Comparison with React Hooks

Composition API provides the same level of logic composition capabilities as React Hooks, but with some important differences.

React Hooks are invoked repeatedly every time a component updates. This creates a number of caveats that can confuse even seasoned React developers. It also leads to performance optimization issues that can severely affect development experience. Here are some examples:

- Hooks are call-order sensitive and cannot be conditional.
- Variables declared in a React component can be captured by a hook closure and become "stale" if the developer fails to pass in the correct dependencies array. This leads to React developers relying on ESLint rules to ensure correct dependencies are passed. However, the rule is often not smart enough and over-compensates for correctness, which leads to unnecessary invalidation and headaches when edge cases are encountered.
- Expensive computations require the use of `useMemo`, which again requires manually passing in the correct dependencies array.

- Event handlers passed to child components cause unnecessary child updates by default, and require explicit `useCallback` as an optimization. This is almost always needed, and again requires a correct dependencies array. Neglecting this leads to over-rendering apps by default and can cause performance issues without realizing it.
- The stale closure problem, combined with Concurrent features, makes it difficult to reason about when a piece of hooks code is run, and makes working with mutable state that should persist across renders (via `useRef`) cumbersome.

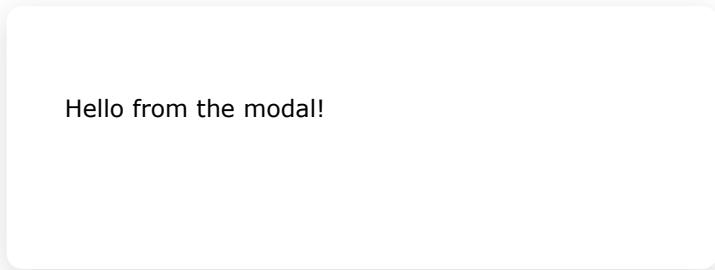
In comparison, Vue Composition API:

Hello from the modal!

- Invokes `setup()` or `<script setup>` code only once. This makes the code align better with the intuitions of idiomatic JavaScript usage as there are no stale closures to worry about. Composition API calls are also not sensitive to call order and can be conditional.
- Vue's runtime reactivity system automatically collects reactive dependencies used in computed properties and watchers, so there's no need to manually declare dependencies.
- No need to manually cache callback functions to avoid unnecessary child updates. In general, Vue's fine-grained reactivity system ensures child components only update when they need to. Manual child-update optimizations are rarely a concern for Vue developers.

We acknowledge the creativity of React Hooks, and it is a major source of inspiration for Composition API. However, the issues mentioned above do exist in its design and we noticed Vue's reactivity model happens to provide a way around them.

# Custom Renderers {#custom-renderers}



Hello from the modal!

---

[Go to TOC](#)

# Reactivity in Depth

One of Vue's most distinctive features is the unobtrusive reactivity system. Component state consists of reactive 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.

## What is Reactivity?

Hello from the modal!

This term comes up in programming quite a bit these days, but what do people mean when they say it? Reactivity is a programming paradigm that allows us to adjust to changes in a declarative manner. The canonical example that people usually show, because it's a great one, is an Excel spreadsheet:

Here cell A2 is defined via a formula of `= A0 + A1` (you can click on A2 to view or edit the formula), so the spreadsheet gives us 3. No surprises there. But if you update A0 or A1, you'll notice that A2 automagically updates too.

JavaScript doesn't usually work like this. If we were to write something comparable in JavaScript:

```
let A0 = 1
let A1 = 2
let A2 = A0 + A1

console.log(A2) // 3

A0 = 2
console.log(A2) // Still 3
```

When we mutate `A0`, `A2` does not change automatically.

So how would we do this in JavaScript? First, in order to re-run the code that updates `A2`, let's wrap it in a function:

```
let A2

function update() {
  A2 = A0 + A1
}
```

Then, we need to define a few terms:

- The `update()` function produces a **side effect**, or **effect** for short, because it modifies the state of the program.
- `A0` and `A1` are considered **dependencies** of the effect, as their values are used to perform the effect. The effect is said to be a **subscriber** to its dependencies.

What we need is a magic function that can invoke `update()` (the **effect**) whenever `A0` or `A1` (the **dependencies**) change:

### whenDepsChange(update)

This `whenDepsChange()` function has the following tasks:

1. Track when a variable is read. E.g. when evaluating the expression `A0 + A1`, both `A0` and `A1` are read.
2. If a variable is read when there is a currently running effect, make that effect a subscriber to that variable. E.g. because `A0` and `A1` are read when `update()` is being executed, `update()` becomes a subscriber to both `A0` and `A1` after the first call.
3. Detect when a variable is mutated. E.g. when `A0` is assigned a new value, notify all its subscriber effects to re-run.

## How Reactivity Works in Vue

We can't really track the reading and writing of local variables like in the example. There's just no mechanism for doing that in vanilla JavaScript. What we **can** do though, is intercept the reading and writing of **object properties**.

There are two ways of intercepting property access in JavaScript: `getter / setters` and `Proxies`. Vue 2 used `getter / setters` exclusively due to browser support limitations. In Vue 3, `Proxies` are used for reactive objects and `getter / setters` are used for refs. Here's some pseudo-code that illustrates how they work:

```
function reactive(obj) {
  return new Proxy(obj, {
    get(target, key) {
      track(target, key)
      return target[key]
    },
    set(target, key, value) {
      target[key] = value
      trigger(target, key)
    }
  })
}

function ref(value) {
  const refObject = {
    get value() {
      track(refObject, 'value')
      return value
    },
    set value(newValue) {
      value = newValue
      trigger(refObject, 'value')
    }
  }
  return refObject
}
```

:::tip Code snippets here and below are meant to explain the core concepts in the simplest form possible, so many details are omitted, and edge cases ignored. :::

This explains a few `limitations of reactive objects` that we have discussed in the fundamentals section:

- When you assign or destructure a reactive object's property to a local variable, the reactivity is "disconnected" because access to the local variable no longer triggers the get / set proxy traps.
- The returned proxy from `reactive()`, although behaving just like the original, has a different identity if we compare it to the original using the `====` operator.

Inside `track()`, we check whether there is a currently running effect. If there is one, we lookup the subscriber effects (stored in a Set) for the property being tracked, and add the effect to the Set:

```
// This will be set right before an effect is about
// to be run. We'll deal with this later.
let activeEffect

function track(target, key) {
  if (activeEffect) {
    const effects = getSubscribersForProperty(target, key)
    effects.add(activeEffect)
  }
}
```

Effect subscriptions are stored in a global `WeakMap<target, Map<key, Set<effect>>>` data structure. If no subscribing effects Set was found for a property (tracked for the first time), it will be created. This is what the `getSubscribersForProperty()` function does, in short. For simplicity, we will skip its details.

Inside `trigger()`, we again lookup the subscriber effects for the property. But this time we invoke them instead:

```
function trigger(target, key) {
  const effects = getSubscribersForProperty(target, key)
  effects.forEach(effect => effect())
}
```

Now let's circle back to the `whenDepsChange()` function:

```
function whenDepsChange(update) {
  const effect = () => {
    activeEffect = effect
    update()
    activeEffect = null
  }
  effect()
}
```

It wraps the raw `update` function in an effect that sets itself as the current active effect before running the actual update. This enables `track()` calls during the update to locate the current active effect.

At this point, we have created an effect that automatically tracks its dependencies, and re-runs whenever a dependency changes. We call this a **Reactive Effect**.

Vue provides an API that allows you to create reactive effects: `watchEffect()`. In fact, you may have noticed that it works pretty similarly to the magical `whenDepsChange()` in the example. We can now rework the original example using actual Vue APIs:

```
import { ref, watchEffect } from 'vue'

const A0 = ref(0)
const A1 = ref(1)
const A2 = ref()

watchEffect(() => {
  // tracks A0 and A1
  A2.value = A0.value + A1.value
})

// triggers the effect
A0.value = 2
```

Using a reactive effect to mutate a ref isn't the most interesting use case - in fact, using a computed property makes it more declarative:

```
import { ref, computed } from 'vue'

const A0 = ref(0)
const A1 = ref(1)
const A2 = computed(() => A0.value + A1.value)

A0.value = 2
```

Internally, `computed` manages its invalidation and re-computation using a reactive effect.

So what's an example of a common and useful reactive effect? Well, updating the DOM! We can implement simple "reactive rendering" like this:

```
import { ref, watchEffect } from 'vue'

const count = ref(0)

watchEffect(() => {
  document.body.innerHTML = `count is: ${count.value}`
})

// updates the DOM
count.value++
```

In fact, this is pretty close to how a Vue component keeps the state and the DOM in sync - each component instance creates a reactive effect to render and update the DOM. Of course, Vue components use much more efficient ways to update the DOM than `innerHTML`. This is discussed in [Rendering Mechanism](#).

The `ref()`, `computed()` and `watchEffect()` APIs are all part of the Composition API. If you have only been using Options API with Vue so far, you'll notice that Composition API is closer to how Vue's reactivity system works under the hood. In fact, in Vue 3 the Options API is implemented on top of the Composition API. All property access on the component instance (`this`) triggers getter / setters for reactivity tracking, and options like `watch` and `computed` invoke their Composition API equivalents internally.

## Runtime vs. Compile-time Reactivity

Vue's reactivity system is primarily runtime-based: the tracking and triggering are all performed while the code is running directly in the browser. The pros of runtime reactivity are that it can work without a build step, and there are fewer edge cases. On the other hand, this makes it constrained by the syntax limitations of JavaScript.

We have already encountered a limitation in the previous example: JavaScript does not provide a way for us to intercept the reading and writing of local variables, so we have to always access reactive state as object properties, using either reactive objects or `getters` and `setters`.

We have been experimenting with the [Reactivity Transform](#) feature to reduce the code verbosity:

```
let A0 = $ref(0)
let A1 = $ref(1)

// track on variable read
const A2 = $computed(() => A0 + A1)

// trigger on variable write
A0 = 2
```

This snippet compiles into exactly what we'd have written without the transform, by automatically appending `.value` after references to the variables. With Reactivity Transform, Vue's reactivity system becomes a hybrid one.

## Reactivity Debugging

It's great that Vue's reactivity system automatically tracks dependencies, but in some cases we may want to figure out exactly what is being tracked, or what is causing a component to re-render.

### Component Debugging Hooks

We can debug what dependencies are used during a component's render and which dependency is triggering an update using the `renderTracked onRenderTracked` and `renderTriggered onRenderTriggered` lifecycle hooks. Both hooks will receive a debugger event which contains information on the dependency in question. It is recommended to place a `debugger` statement in the callbacks to interactively inspect the dependency:

```
<script setup>
import { onRenderTracked, onRenderTriggered } from 'vue'

onRenderTracked((event) => {
  debugger
})

onRenderTriggered((event) => {
  debugger
})
```

```
export default {
  renderTracked(event) {
    debugger
  },
  renderTriggered(event) {
    debugger
  }
}
```

:::tip Component debug hooks only work in development mode. :::

The debug event objects have the following type:  
Hello from the modal!

```
type DebuggerEvent = {
  effect: ReactiveEffect
  target: object
  type:
    | TrackOpTypes /* 'get' | 'has' | 'iterate' */
    | TriggerOpTypes /* 'set' | 'add' | 'delete' | 'clear' */
  key: any
  newValue?: any
  oldValue?: any
  oldTarget?: Map<any, any> | Set<any>
}
```

## Computed Debugging

We can debug computed properties by passing `computed()` a second options object with `onTrack` and `onTrigger` callbacks:

- `onTrack` will be called when a reactive property or ref is tracked as a dependency.
- `onTrigger` will be called when the watcher callback is triggered by the mutation of a dependency.

Both callbacks will receive debugger events in the [same format](#) as component debug hooks:

```
const plusOne = computed(() => count.value + 1, {
  onTrack(e) {
    // triggered when count.value is tracked as a dependency
    debugger
  },
  onTrigger(e) {
    // triggered when count.value is mutated
    debugger
  }
})

// access plusOne, should trigger onTrack
console.log(plusOne.value)

// mutate count.value, should trigger onTrigger
count.value++
```

:::tip `onTrack` and `onTrigger` computed options only work in development mode. :::

## Watcher Debugging

Similar to `computed()`, watchers also support the `onTrack` and `onTrigger` options:

```

watch(source, callback, {
  onTrack(e) {
    debugger
  },
  onTrigger(e) {
    debugger
  }
})

watchEffect(callback, {
  onTrack(e) {
    debugger
  },
  onTrigger(e) {
    debugger
  }
})

```

:::tip `onTrack` and `onTrigger` watcher options only work in development mode. :::

## Integration with External State Systems

Vue's reactivity system works by deeply converting plain JavaScript objects into reactive proxies. The deep conversion can be unnecessary or sometimes unwanted when integrating with external state management systems (e.g. if an external solution also uses Proxies).

The general idea of integrating Vue's reactivity system with an external state management solution is to hold the external state in a `shallowRef`. A shallow ref is only reactive when its `.value` property is accessed - the inner value is left intact. When the external state changes, replace the ref value to trigger updates.

## Immutable Data

If you are implementing an undo / redo feature, you likely want to take a snapshot of the application's state on every user edit. However, Vue's mutable reactivity system isn't best suited for this if the state tree is large, because serializing the entire state object on every update can be expensive in terms of both CPU and memory costs.

[Immutable data structures](#) solve this by never mutating the state objects - instead, it creates new objects that share the same, unchanged parts with old ones. There are different ways of using immutable data in JavaScript, but we recommend using [Immer](#) with Vue because it allows you to use immutable data while keeping the more ergonomic, mutable syntax.

We can integrate Immer with Vue via a simple composable:

```

import produce from 'immer'
import { shallowRef } from 'vue'

export function useImmer(baseState) {
  const state = shallowRef(baseState)
  const update = (updater) => {
    state.value = produce(state.value, updater)
  }
}

```

```

    return [state, update]
}

```

[Try it in the Playground](#)

## State Machines

[State Machine](#) is a model for describing all the possible states an application can be in, and all the possible ways it can transition from one state to another. While it may be overkill for simple components, it can help make complex state flows more robust and manageable.

Hello from the modal!

One of the most popular state machine implementations in JavaScript is [XState](#). Here's a composable that integrates with it:

```

import { createMachine, interpret } from 'xstate'
import { shallowRef } from 'vue'

export function useMachine(options) {
  const machine = createMachine(options)
  const state = shallowRef(machine.initialState)
  const service = interpret(machine)
    .onTransition((newState) => (state.value = newState))
    .start()
  const send = (event) => service.send(event)

  return [state, send]
}

```

[Try it in the Playground](#)

## RxJS

[RxJS](#) is a library for working with asynchronous event streams. The [VueUse](#) library provides the `@vueuse/rxjs` add-on for connecting RxJS streams with Vue's reactivity system.

---

[Go to TOC](#)

# Reactivity Transform

:::warning Experimental Feature Reactivity Transform is currently an experimental feature. It is disabled by default and requires [explicit opt-in](#). It may also change before being finalized. To stay up-to-date, keep an eye on its [proposal and discussion on GitHub](#). :::

:::tip Composition-API-specific Reactivity Transform is a Composition-API-specific feature and requires a build step. :::

Hello from the modal!

## Refs vs. Reactive Variables

Ever since the introduction of the Composition API, one of the primary unresolved questions is the use of refs vs. reactive objects. It's easy to lose reactivity when destructuring reactive objects, while it can be cumbersome to use `.value` everywhere when using refs. Also, `.value` is easy to miss if not using a type system.

[Vue Reactivity Transform](#) is a compile-time transform that allows us to write code like this:

```
<script setup>
let count = $ref(0)

console.log(count)

function increment() {
  count++
}
</script>

<template>
  <button @click="increment">{{ count }}</button>
</template>
```

The `$ref()` method here is a **compile-time macro**: it is not an actual method that will be called at runtime. Instead, the Vue compiler uses it as a hint to treat the resulting `count` variable as a **reactive variable**.

Reactive variables can be accessed and re-assigned just like normal variables, but these operations are compiled into refs with `.value`. For example, the `<script>` part of the above component is compiled into:

```
import { ref } from 'vue'

let count = ref(0)
console.log(count.value)

function increment() {
  count.value++
}
```

Every reactivity API that returns refs will have a `$`-prefixed macro equivalent. These APIs include:

- `ref` -> `$ref`
- `computed` -> `$computed`
- `shallowRef` -> `$shallowRef`
- `customRef` -> `$customRef`
- `toRef` -> `$toRef`

These macros are globally available and do not need to be imported when Reactivity Transform is enabled, but you can optionally import them from `vue/macros` if you want to be more explicit:

```
import { $ref } from 'vue/macros'
let count = $ref(0)
```

## Destructuring with `$()` `{#destructuring-with}`

It is common for a composition function to return an object of refs, and use destructuring to retrieve these refs. For this purpose, reactivity transform provides the `$()` macro:

```
import { useMouse } from '@vueuse/core'
const { x, y } = $(useMouse())
console.log(x, y)
```

Compiled output:

```
import { toRef } from 'vue'
import { useMouse } from '@vueuse/core'

const __temp = useMouse(),
  x = toRef(__temp, 'x'),
  y = toRef(__temp, 'y')

console.log(x.value, y.value)
```

Note that if `x` is already a ref, `toRef(__temp, 'x')` will simply return it as-is and no additional ref will be created. If a destructured value is not a ref (e.g. a function), it will still work - the value will be wrapped in a ref so the rest of the code works as expected.

`$()` destructure works on both reactive objects **and** plain objects containing refs.

## Convert Existing Refs to Reactive Variables with `$()` `{#convert-existing-refs-to-reactive-variables-with}`

In some cases we may have wrapped functions that also return refs. However, the Vue compiler won't be able to know ahead of time that a function is going to return a ref. In such cases, the `$()` macro can also be used to convert any existing refs into reactive variables:

```
function myCreateRef() {
  return ref(0)
}

let count = $(myCreateRef())
```

## Reactive Props Destructure

There are two pain points with the current `defineProps()` usage in `<script setup>`:

1. Similar to `.value`, you need to always access props as `props.x` in order to retain reactivity. This means you cannot destructure `defineProps` because the resulting destructured variables are not reactive and will not update.
2. When using the [type-only props declaration](#), there is no easy way to declare default values for the props. We introduced the `withDefaults()` API for this exact purpose, but it's still clunky to use.

We can address these issues by applying a compile-time transform when `defineProps` is used with de-structuring, similar to what we saw earlier with `$(())`:

```
<script setup lang="ts">
  interface Props {
    msg: string
    count?: number
    foo?: string
  }

  const {
    msg,
    // default value just works
    count = 1,
    // local aliasing also just works
    // here we are aliasing `props.foo` to `bar`
    foo: bar
  } = defineProps<Props>()

  watchEffect(() => {
    // will log whenever the props change
    console.log(msg, count, bar)
  })
</script>
```

The above will be compiled into the following runtime declaration equivalent:

```
export default {
  props: {
    msg: { type: String, required: true },
    count: { type: Number, default: 1 },
    foo: String
  },
  setup(props) {
    watchEffect(() => {
      console.log(props.msg, props.count, props.foo)
    })
  }
}
```

# Retaining Reactivity Across Function Boundaries

While reactive variables relieve us from having to use `.value` everywhere, it creates an issue of "reactivity loss" when we pass reactive variables across function boundaries. This can happen in two cases:

## Passing into function as argument

Given a function that expects a ref as an argument, e.g.:

```
function trackChange(x: Ref<number>) {
  watch(x, (x) => {
    console.log('x changed!')
  })
}

let count = $ref(0)
trackChange(count) // doesn't work!
```

The above case will not work as expected because it compiles to:

```
let count = ref(0)
trackChange(count.value)
```

Here `count.value` is passed as a number, whereas `trackChange` expects an actual ref. This can be fixed by wrapping `count` with `$(...)` before passing it:

```
let count = $ref(0)
- trackChange(count)
+ trackChange($(count))
```

The above compiles to:

```
import { ref } from 'vue'

let count = ref(0)
trackChange(count)
```

As we can see, `$(...)` is a macro that serves as an **escape hint**: reactive variables inside `$(...)` will not get `.value` appended.

## Returning inside function scope

Reactivity can also be lost if reactive variables are used directly in a returned expression:

```
function useMouse() {
  let x = $ref(0)
  let y = $ref(0)

  // listen to mousemove...

  // doesn't work!
  return {
    x,
```

```

    }
  }
}
```

The above return statement compiles to:

```

return {
  x: x.value,
  y: y.value
}
```

In order to retain reactivity, we should return the actual refs, not the current value at return time.

Again, we can use `$$()` to fix this. In this case, `$$()` can be used directly on the returned object - any reference to reactive variables inside the `$$()` call will retain the reference to their underlying refs:

```

function useMouse() {
  let x = $ref(0)
  let y = $ref(0)

  // listen to mousemove...

  // fixed
  return $$({
    x,
    y
  })
}
```

## Using `$()` on destructured props

`$$()` works on destructured props since they are reactive variables as well. The compiler will convert it with `toRef` for efficiency:

```

const { count } = defineProps<{ count: number }>()
passAsRef($$(count))
```

compiles to:

```

setup(props) {
  const __props_count = toRef(props, 'count')
  passAsRef(__props_count)
}
```

## TypeScript Integration [#typescript-integration](#)

Vue provides typings for these macros (available globally) and all types will work as expected. There are no incompatibilities with standard TypeScript semantics, so the syntax will work with all existing tooling.

This also means the macros can work in any files where valid JS / TS are allowed - not just inside Vue SFCs.

Since the macros are available globally, their types need to be explicitly referenced (e.g. in a `env.d.ts` file):

```
//> <reference types="vue/macros-global" />
```

When explicitly importing the macros from `vue/macros`, the type will work without declaring the globals.

## Explicit Opt-in

Reactivity Transform is currently disabled by default and requires explicit opt-in. In addition, all of the following setups require `vue@^3.2.25`.

### Vite

Hello from the modal!

- Requires `@vitejs/plugin-vue@>=2.0.0`
- Applies to SFCs and `js(x)/ts(x)` files. A fast usage check is performed on files before applying the transform so there should be no performance cost for files not using the macros.
- Note `reactivityTransform` is now a plugin root-level option instead of nested as `script.refSugar`, since it affects not just SFCs.

```
// vite.config.js
export default {
  plugins: [
    vue({
      reactivityTransform: true
    })
  ]
}
```

### vue-cli {#vue-cli}

- Currently only affects SFCs
- Requires `vue-loader@>=17.0.0`

```
// vue.config.js
module.exports = {
  chainWebpack: (config) => {
    config.module
      .rule('vue')
      .use('vue-loader')
      .tap((options) => {
        return {
          ...options,
          reactivityTransform: true
        }
      })
  }
}
```

### Plain webpack + vue-loader {#plain-webpack-vue-loader}

- Currently only affects SFCs
- Requires `vue-loader@>=17.0.0`

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.vue$/,
        loader: 'vue-loader',
        options: {
          reactivityTransform: true
        }
      }
    ]
  }
}
```

# Render Functions & JSX

Vue recommends using templates to build applications in the vast majority of cases. However, there are situations where we need the full programmatic power of JavaScript. That's where we can use the **render function**.

If you are new to the concept of virtual DOM and render functions, make sure to read the [Rendering Mechanism](#) chapter first.

## Basic Usage

### Creating Vnodes

Vue provides an `h()` function for creating vnodes:

```
import { h } from 'vue'

const vnode = h(
  'div', // type
  { id: 'foo', class: 'bar' }, // props
  [
    /* children */
  ]
)
```

`h()` is short for **hyperscript** - which means "JavaScript that produces HTML (hypertext markup language)". This name is inherited from conventions shared by many virtual DOM implementations. A more descriptive name could be `createVnode()`, but a shorter name helps when you have to call this function many times in a render function.

The `h()` function is designed to be very flexible:

```
// all arguments except the type are optional
h('div')
h('div', { id: 'foo' })

// both attributes and properties can be used in props
// Vue automatically picks the right way to assign it
h('div', { class: 'bar', innerHTML: 'hello' })

// props modifiers such as .prop and .attr can be added
// with '.' and `^` prefixes respectively
h('div', { '.name': 'some-name', '^width': '100' })

// class and style have the same object / array
// value support that they have in templates
h('div', { class: [foo, { bar }]}, style: { color: 'red' } )

// event listeners should be passed as onXxx
h('div', { onClick: () => {} })
```

```
// children can be a string
h('div', { id: 'foo' }, 'hello')

// props can be omitted when there are no props
h('div', 'hello')
h('div', [h('span', 'hello')])

// children array can contain mixed vnodes and strings
h('div', ['hello', h('span', 'hello')])
```

The resulting vnode has the following shape:

```
const vnode = h('div', { id: 'foo' }, [])
vnode.type // 'div'
vnode.props // { id: 'foo' }
vnode.children // []
vnode.key // null
```

:::warning Note The full `vNode` interface contains many other internal properties, but it is strongly recommended to avoid relying on any properties other than the ones listed here. This avoids unintended breakage in case the internal properties are changed. :::

## Declaring Render Functions

When using templates with Composition API, the return value of the `setup()` hook is used to expose data to the template. When using render functions, however, we can directly return the render function instead:

```
import { ref, h } from 'vue'

export default {
  props: {
    /* ... */
  },
  setup(props) {
    const count = ref(1)

    // return the render function
    return () => h('div', props.msg + count.value)
  }
}
```

The render function is declared inside `setup()` so it naturally has access to the props and any reactive state declared in the same scope.

In addition to returning a single vnode, you can also return strings or arrays:

```
export default {
  setup() {
    return () => 'hello world!'
  }
}
```

```
import { h } from 'vue'
export default {
```

```
setup() {
  // use an array to return multiple root nodes
  return () => [
    h('div'),
    h('div'),
    h('div')
  ]
}
```

:::tip Make sure to return a function instead of directly returning values! The `setup()` function is called only once per component, while the returned render function will be called multiple times. :::  
Hello from the modal!

We can declare render functions using the `render` option:

```
import { h } from 'vue'

export default {
  data() {
    return {
      msg: 'hello'
    }
  },
  render() {
    return h('div', this.msg)
  }
}
```

The `render()` function has access to the component instance via `this`.

In addition to returning a single vnode, you can also return strings or arrays:

```
export default {
  render() {
    return 'hello world!'
  }
}
```

```
import { h } from 'vue'

export default {
  render() {
    // use an array to return multiple root nodes
    return [
      h('div'),
      h('div'),
      h('div')
    ]
  }
}
```

If a render function component doesn't need any instance state, they can also be declared directly as a function for brevity:

```
function Hello() {
  return 'hello world!'
}
```

That's right, this is a valid Vue component! See [Functional Components](#) for more details on this syntax.

## Vnodes Must Be Unique

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

```
function render() {
  const p = h('p', 'hi')
  return h('div', [
    // Yikes - duplicate vnodes!
    p,
    p
  ])
}
```

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:

```
function render() {
  return h(
    'div',
    Array.from({ length: 20 }).map(() => {
      return h('p', 'hi')
    })
  )
}
```

## JSX / TSX

[JSX](#) is an XML-like extension to JavaScript that allows us to write code like this:

```
const vnode = <div>hello</div>
```

Inside JSX expressions, use curly braces to embed dynamic values:

```
const vnode = <div id={dynamicId}>hello, {userName}</div>
```

`create-vue` and Vue CLI both have options for scaffolding projects with pre-configured JSX support. If you are configuring JSX manually, please refer to the documentation of [@vue/babel-plugin-jsx](#) for details.

Although first introduced by React, JSX actually has no defined runtime semantics and can be compiled into various different outputs. If you have worked with JSX before, do note that **Vue JSX transform is different from React's JSX transform**, so you can't use React's JSX transform in Vue applications. Some notable differences from React JSX include:

- You can use HTML attributes such as `class` and `for` as props - no need to use `className` or `htmlFor`.
- Passing children to components (i.e. slots) [works differently](#).

Vue's type definition also provides type inference for TSX usage. When using TSX, make sure to specify `"jsx": "preserve"` in `tsconfig.json` so that TypeScript leaves the JSX syntax intact for Vue JSX transform to process.

## Render Function Recipes

Below we will provide some common recipes for implementing template features as their equivalent render functions / JSX.

### v-if {#v-if}

Hello from the modal!

Template:

```
<div>
  <div v-if="ok">yes</div>
  <span v-else>no</span>
</div>
```

Equivalent render function / JSX:

```
h('div', [ok.value ? h('div', 'yes') : h('span', 'no')])
<div>{ok.value ? <div>yes</div> : <span>no</span>}</div>
h('div', [this.ok ? h('div', 'yes') : h('span', 'no')])
<div>{this.ok ? <div>yes</div> : <span>no</span>}</div>
```

### v-for {#v-for}

Template:

```
<ul>
  <li v-for="{ id, text } in items" :key="id">
    {{ text }}
  </li>
</ul>
```

Equivalent render function / JSX:

```
h(
  'ul',
  // assuming `items` is a ref with array value
  items.value.map(({ id, text }) => {
    return h('li', { key: id }, text)
  })
)

<ul>
  {items.value.map(({ id, text }) => {
    return <li key={id}>{text}</li>
  })}
</ul>
```

```

h(
  'ul',
  this.items.map(({ id, text }) => {
    return h('li', { key: id }, text)
  })
)
  
```

```

<ul>
  {this.items.map(({ id, text }) => {
    return <li key={id}>{text}</li>
  })}
</ul>
  
```

Hello from the modal!

## v-on {#v-on}

Props with names that start with `on` followed by an uppercase letter are treated as event listeners. For example, `onClick` is the equivalent of `@click` in templates.

```

h(
  'button',
  {
    onClick(event) {
      /* ... */
    }
  },
  'click me'
)
  
```

```

<button
  onClick={(event) => {
    /* ... */
  }}
>
  click me
</button>
  
```

## Event Modifiers

For the `.passive`, `.capture`, and `.once` event modifiers, they can be concatenated after the event name using camelCase.

For example:

```

h('input', {
  onClickCapture() {
    /* listener in capture mode */
  },
  onKeyupOnce() {
    /* triggers only once */
  },
  onMouseoverOnceCapture() {
    /* once + capture */
  }
})
  
```

```
<input
  onClickCapture={() => {}}
  onKeyupOnce={() => {}}
  onMouseoverOnceCapture={() => {}}
/>
```

For other event and key modifiers, the `withModifiers` helper can be used:

```
import { withModifiers } from 'vue'

h('div', {
  onClick: withModifiers(() => {}, ['self'])
})

<div onClick={withModifiers(() => {}, ['self'])} />
```

## Components

To create a vnode for a component, the first argument passed to `h()` should be the component definition. This means when using render functions, it is unnecessary to register components - you can just use the imported components directly:

```
import Foo from './Foo.vue'
import Bar from './Bar.jsx'

function render() {
  return h('div', [h(Foo), h(Bar)])
}

function render() {
  return (
    <div>
      <Foo />
      <Bar />
    </div>
  )
}
```

As we can see, `h` can work with components imported from any file format as long as it's a valid Vue component.

Dynamic components are straightforward with render functions:

```
import Foo from './Foo.vue'
import Bar from './Bar.jsx'

function render() {
  return ok.value ? h(Foo) : h(Bar)
}

function render() {
  return ok.value ? <Foo /> : <Bar />
}
```

If a component is registered by name and cannot be imported directly (for example, globally registered by a library), it can be programmatically resolved by using the `resolveComponent()` helper.

## Rendering Slots

In render functions, slots can be accessed from the `setup()` context. Each slot on the `slots` object is a **function that returns an array of vnodes**:

```
export default {
  props: ['message'],
  setup(props, { slots }) {
    return () => [
      // default slot:
      // <div><slot /></div>
      h('div', slots.default()),

      // named slot:
      // <div><slot name="footer" :text="message" /></div>
      h(
        'div',
        slots.footer({
          text: props.message
        })
      )
    ]
  }
}
```

JSX equivalent:

```
// default
<div>{slots.default()}</div>

// named
<div>{slots.footer({ text: props.message })}</div>
```

In render functions, slots can be accessed from `this.$slots`:

```
export default {
  props: ['message'],
  render() {
    return [
      // <div><slot /></div>
      h('div', this.$slots.default()),

      // <div><slot name="footer" :text="message" /></div>
      h(
        'div',
        this.$slots.footer({
          text: this.message
        })
      )
    ]
  }
}
```

JSX equivalent:

```
// <div><slot /></div>
<div>{this.$slots.default()}</div>

// <div><slot name="footer" :text="message" /></div>
<div>{this.$slots.footer({ text: this.message })}</div>
```

## Passing Slots

Passing children to components works a bit differently from passing children to elements. Instead of an array, we need to pass either a slot function, or an object of slot functions. Slot functions can return anything a normal render function can return - which will always be normalized to arrays of vnodes when accessed in the child component.

```
// single default slot
h(MyComponent, () => 'hello')

// named slots
// notice the `null` is required to avoid
// the slots object being treated as props
h(MyComponent, null, {
  default: () => 'default slot',
  foo: () => h('div', 'foo'),
  bar: () => [h('span', 'one'), h('span', 'two')]
})
```

JSX equivalent:

```
// default
<MyComponent>{() => 'hello'}</MyComponent>

// named
<MyComponent>{{
  default: () => 'default slot',
  foo: () => <div>foo</div>,
  bar: () => [<span>one</span>, <span>two</span>]
}}</MyComponent>
```

Passing slots as functions allows them to be invoked lazily by the child component. This leads to the slot's dependencies being tracked by the child instead of the parent, which results in more accurate and efficient updates.

## Built-in Components

Built-in components such as `<KeepAlive>`, `<Transition>`, `<TransitionGroup>`, `<Teleport>` and `<Suspense>` must be imported for use in render functions:

```
import { h, KeepAlive, Teleport, Transition, TransitionGroup } from 'vue'

export default {
  setup () {
    return () => h(Transition, { mode: 'out-in' }, /* ... */)
  }
}
```

```
import { h, KeepAlive, Teleport, Transition, TransitionGroup } from 'vue'

export default {
  render () {
    return h(Transition, { mode: 'out-in' }, /* ... */)
  }
}
```

## v-model {#v-model}

The `v-model` directive is expanded to `modelValue` and `onUpdate:modelValue` props during template compilation—we will have to provide these props ourselves:

```
export default {
  props: ['modelValue'],
  emits: ['update:modelValue'],
  setup(props, { emit }) {
    return () =>
      h(SomeComponent, {
        modelValue: props.modelValue,
        'onUpdate:modelValue': (value) => emit('update:modelValue', value)
      })
  }
}
```

```
export default {
  props: ['modelValue'],
  emits: ['update:modelValue'],
  render() {
    return h(SomeComponent, {
      modelValue: this.modelValue,
      'onUpdate:modelValue': (value) => this.$emit('update:modelValue', value)
    })
  }
}
```

## Custom Directives

Custom directives can be applied to a vnode using `withDirectives`:

```
import { h, withDirectives } from 'vue'

// a custom directive
const pin = {
  mounted() { /* ... */ },
  updated() { /* ... */ }
}

// <div v-pin:top.animate="200"></div>
const vnode = withDirectives(h('div'), [
  [pin, 200, 'top', { animate: true }]
])
```

If the directive is registered by name and cannot be imported directly, it can be resolved using the `resolveDirective` helper.

# Functional Components

Functional components are an alternative form of component that don't have any state of their own. They act like pure functions: props in, vnodes out. They are rendered without creating a component instance (i.e. no `this`), and without the usual component lifecycle hooks.

To create a functional component we use a plain function, rather than an options object. The function is effectively the `render` function for the component.

The signature of a functional component is the same as the `setup()` hook:

```
function MyComponent(props, { slots, emit, attrs }) {
  // ...
}
```

As there is no `this` reference for a functional component, Vue will pass in the `props` as the first argument:

```
function MyComponent(props, context) {
  // ...
}
```

The second argument, `context`, contains three properties: `attrs`, `emit`, and `slots`. These are equivalent to the instance properties `$attrs`, `$emit`, and `$slots` respectively.

Most of the usual configuration options for components are not available for functional components. However, it is possible to define `props` and `emits` by adding them as properties:

```
MyComponent.props = ['value']
MyComponent.emits = ['click']
```

If the `props` option is not specified, then the `props` object passed to the function will contain all attributes, the same as `attrs`. The prop names will not be normalized to camelCase unless the `props` option is specified.

For functional components with explicit `props`, [attribute fallback](#) works much the same as with normal components. However, for functional components that don't explicitly specify their `props`, only the `class`, `style`, and `onXXX` event listeners will be inherited from the `attrs` by default. In either case, `inheritAttrs` can be set to `false` to disable attribute inheritance:

```
MyComponent.inheritAttrs = false
```

Functional components can be registered and consumed just like normal components. If you pass a function as the first argument to `h()`, it will be treated as a functional component.

# Rendering Mechanism

How does Vue take a template and turn it into actual DOM nodes? How does Vue update those DOM nodes efficiently? We will attempt to shed some light on these questions here by diving into Vue's internal rendering mechanism.

## Virtual DOM

Hello from the modal!

You have probably heard about the term "virtual DOM", which Vue's rendering system is based upon.

The virtual DOM (VDOM) is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM. The concept was pioneered by [React](#), and has been adapted in many other frameworks with different implementations, including Vue.

Virtual DOM is more of a pattern than a specific technology, so there is no one canonical implementation. We can illustrate the idea using a simple example:

```
const vnode = {
  type: 'div',
  props: {
    id: 'hello'
  },
  children: [
    /* more vnodes */
  ]
}
```

Here, `vnode` is a plain JavaScript object (a "virtual node") representing a `<div>` element. It contains all the information that we need to create the actual element. It also contains more children vnodes, which makes it the root of a virtual DOM tree.

A runtime renderer can walk a virtual DOM tree and construct a real DOM tree from it. This process is called **mount**.

If we have two copies of virtual DOM trees, the renderer can also walk and compare the two trees, figuring out the differences, and apply those changes to the actual DOM. This process is called **patch**, also known as "diffing" or "reconciliation".

The main benefit of virtual DOM is that it gives the developer the ability to programmatically create, inspect and compose desired UI structures in a declarative way, while leaving the direct DOM manipulation to the renderer.

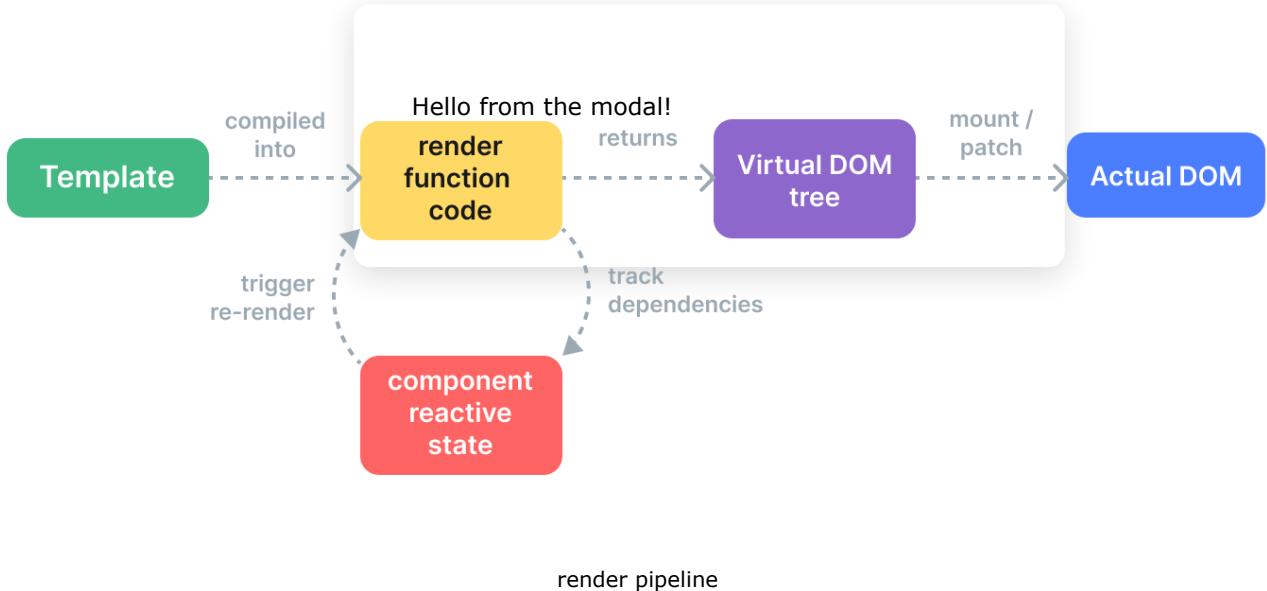
## Render Pipeline

At the high level, this is what happens when a Vue component is mounted:

1. **Compile**: Vue templates are compiled into **render functions**: functions that return virtual DOM trees. This step can be done either ahead-of-time via a build step, or on-the-fly by using the runtime compiler.

2. **Mount:** The runtime renderer invokes the render functions, walks the returned virtual DOM tree, and creates actual DOM nodes based on it. This step is performed as a [reactive effect](#), so it keeps track of all reactive dependencies that were used.

3. **Patch:** When a dependency used during mount changes, the effect re-runs. This time, a new, updated Virtual DOM tree is created. The runtime renderer walks the new tree, compares it with the old one, and applies necessary updates to the actual DOM.



## Templates vs. Render Functions

Vue templates are compiled into virtual DOM render functions. Vue also provides APIs that allow us to skip the template compilation step and directly author render functions. Render functions are more flexible than templates when dealing with highly dynamic logic, because you can work with vnodes using the full power of JavaScript.

So why does Vue recommend templates by default? There are a number of reasons:

1. Templates are closer to actual HTML. This makes it easier to reuse existing HTML snippets, apply accessibility best practices, style with CSS, and for designers to understand and modify.
2. Templates are easier to statically analyze due to their more deterministic syntax. This allows Vue's template compiler to apply many compile-time optimizations to improve the performance of the virtual DOM (which we will discuss below).

In practice, templates are sufficient for most use cases in applications. Render functions are typically only used in reusable components that need to deal with highly dynamic rendering logic. Render function usage is discussed in more detail in [Render Functions & JSX](#).

# Compiler-Informed Virtual DOM

The virtual DOM implementation in React and most other virtual-DOM implementations are purely runtime: the reconciliation algorithm cannot make any assumptions about the incoming virtual DOM tree, so it has to fully traverse the tree and diff the props of every vnode in order to ensure correctness. In addition, even if a part of the tree never changes, new vnodes are always created for them on each re-render, resulting in unnecessary memory pressure. This is one of the most criticized aspect of virtual DOM: the somewhat brute-force reconciliation process sacrifices efficiency in return for declarativeness and correctness.

But it doesn't have to be that way. [Hello from the model!](#) controls both the compiler and the runtime. This allows us to implement many compile-time optimizations that only a tightly-coupled renderer can take advantage of. The compiler can statically analyze the template and leave hints in the generated code so that the runtime can take shortcuts whenever possible. At the same time, we still preserve the capability for the user to drop down to the render function layer for more direct control in edge cases. We call this hybrid approach **Compiler-Informed Virtual DOM**.

Below, we will discuss a few major optimizations done by the Vue template compiler to improve the virtual DOM's runtime performance.

## Static Hoisting

Quite often there will be parts in a template that do not contain any dynamic bindings:

```
<div>
  <div>foo</div> <!-- hoisted -->
  <div>bar</div> <!-- hoisted -->
  <div>{{ dynamic }}</div>
</div>
```

[Inspect in Template Explorer](#)

The `foo` and `bar` divs are static - re-creating vnodes and differencing them on each re-render is unnecessary. The Vue compiler automatically hoists their vnode creation calls out of the render function, and reuses the same vnodes on every render. The renderer is also able to completely skip differencing them when it notices the old vnode and the new vnode are the same one.

In addition, when there are enough consecutive static elements, they will be condensed into a single "static vnode" that contains the plain HTML string for all these nodes ([Example](#)). These static vnodes are mounted by directly setting `innerHTML`. They also cache their corresponding DOM nodes on initial mount - if the same piece of content is reused elsewhere in the app, new DOM nodes are created using native `cloneNode()`, which is extremely efficient.

## Patch Flags

For a single element with dynamic bindings, we can also infer a lot of information from it at compile time:

```
<!-- class binding only -->
<div :class="{ active }"></div>

<!-- id and value bindings only -->
```

```
<input :id="id" :value="value">
<!-- text children only -->
<div>{{ dynamic }}</div>
```

### Inspect in Template Explorer

When generating the render function code for these elements, Vue encodes the type of update each of them needs directly in the vnode creation call:

```
createElementVNode("div", {
  class: _normalizeClass({ active: _ctx.active })
}, null, 2 /* CLASS */)
```

The last argument, `2`, is a [patch flag](#). An element can have multiple patch flags, which will be merged into a single number. The runtime renderer can then check against the flags using [bitwise operations](#) to determine whether it needs to do certain work:

```
if (vnode.patchFlag & PatchFlags.CLASS /* 2 */) {
  // update the element's class
}
```

Bitwise checks are extremely fast. With the patch flags, Vue is able to do the least amount of work necessary when updating elements with dynamic bindings.

Vue also encodes the type of children a vnode has. For example, a template that has multiple root nodes is represented as a fragment. In most cases, we know for sure that the order of these root nodes will never change, so this information can also be provided to the runtime as a patch flag:

```
export function render() {
  return (_openBlock(), _createElementBlock(_Fragment, null, [
    /* children */
  ], 64 /* STABLE_FRAGMENT */))
}
```

The runtime can thus completely skip child-order reconciliation for the root fragment.

## Tree Flattening

Taking another look at the generated code from the previous example, you'll notice the root of the returned virtual DOM tree is created using a special `createElementBlock()` call:

```
export function render() {
  return (_openBlock(), _createElementBlock(_Fragment, null, [
    /* children */
  ], 64 /* STABLE_FRAGMENT */))
}
```

Conceptually, a "block" is a part of the template that has stable inner structure. In this case, the entire template has a single block because it does not contain any structural directives like `v-if` and `v-for`.

Each block tracks any descendant nodes (not just direct children) that have patch flags. For example:

```
<div> <!-- root block -->
  <div>...</div>      <!-- not tracked -->
  <div :id="id"></div>  <!-- tracked -->
  <div>                <!-- not tracked -->
    <div>{{ bar }}</div> <!-- tracked -->
  </div>
</div>
```

The result is a flattened array that contains only the dynamic descendant nodes:

```
div (block root)
- div with :id binding
- div with {{ bar }} binding
```

When this component needs to re-render, it only needs to traverse the flattened tree instead of the full tree. This is called **Tree Flattening**, and it greatly reduces the number of nodes that need to be traversed during virtual DOM reconciliation. Any static parts of the template are effectively skipped.

`v-if` and `v-for` directives will create new block nodes:

```
<div> <!-- root block -->
  <div>
    <div v-if> <!-- if block -->
    ...
    <div>
  </div>
</div>
```

A child block is tracked inside the parent block's array of dynamic descendants. This retains a stable structure for the parent block.

## Impact on SSR Hydration

Both patch flags and tree flattening also greatly improve Vue's [SSR Hydration](#) performance:

- Single element hydration can take fast paths based on the corresponding vnode's patch flag.
- Only block nodes and their dynamic descendants need to be traversed during hydration, effectively achieving partial hydration at the template level.

# Ways of Using Vue

We believe there is no "one size fits all" story for the web. This is why Vue is designed to be flexible and incrementally adoptable. Depending on your use case, Vue can be used in different ways to strike the optimal balance between stack complexity, developer experience and end performance.

## Standalone Script

Hello from the modal!

Vue can be used as a standalone script file - no build step required! If you have a backend framework already rendering most of the HTML, or your frontend logic isn't complex enough to justify a build step, this is the easiest way to integrate Vue into your stack. You can think of Vue as a more declarative replacement of jQuery in such cases.

Vue also provides an alternative distribution called [petite-vue](#) that is specifically optimized for progressively enhancing existing HTML. It has a smaller feature set, but is extremely lightweight and uses an implementation that is more efficient in no-build-step scenarios.

## Embedded Web Components

You can use Vue to [build standard Web Components](#) that can be embedded in any HTML page, regardless of how they are rendered. This option allows you to leverage Vue in a completely consumer-agnostic fashion: the resulting web components can be embedded in legacy applications, static HTML, or even applications built with other frameworks.

## Single-Page Application (SPA)

Some applications require rich interactivity, deep session depth, and non-trivial stateful logic on the front-end. The best way to build such applications is to use an architecture where Vue not only controls the entire page, but also handles data updates and navigation without having to reload the page. This type of application is typically referred to as a Single-Page Application (SPA).

Vue provides core libraries and [comprehensive tooling support](#) with amazing developer experience for building modern SPAs, including:

- Client-side router
- Blazing fast build tool chain
- IDE support
- Browser devtools
- TypeScript integrations
- Testing utilities

SPAs typically require the backend to expose API endpoints - but you can also pair Vue with solutions like [Inertia.js](#) to get the SPA benefits while retaining a server-centric development model.

## Fullstack / SSR

Pure client-side SPAs are problematic when the app is sensitive to SEO and time-to-content. This is because the browser will receive a largely empty HTML page, and has to wait until the JavaScript is loaded before rendering anything.

Vue provides first-class APIs to "render" a Vue app into HTML strings on the server. This allows the server to send back already-rendered HTML, allowing end users to see the content immediately while the JavaScript is being downloaded. Vue will then "hydrate" the application on the client side to make it interactive. This is called [Server-Side Rendering \(SSR\)](#) and it improves Core Web Vital metrics such as [Largest Contentful Paint \(LCP\)](#).

There are higher-level Vue-based frameworks built on top of this paradigm, such as [Nuxt](#), which allow you to develop a fullstack application using Vue and JavaScript.

## JAMStack / SSG

Server-side rendering can be done ahead of time if the required data is static. This means we can pre-render an entire application into HTML and serve them as static files. This improves site performance and makes deployment a lot simpler since we no longer need to dynamically render pages on each request. Vue can still hydrate such applications to provide rich interactivity on the client. This technique is commonly referred to as Static-Site Generation (SSG), also known as [JAMStack](#).

There are two flavors of SSG: single-page and multi-page. Both flavors pre-render the site into static HTML, the difference is that:

- After the initial page load, a single-page SSG "hydrates" the page into an SPA. This requires more up-front JS payload and hydration cost, but subsequent navigations will be faster, since it only needs to partially update the page content instead of reloading the entire page.
- A multi-page SSG loads a new page on every navigation. The upside is that it can ship minimal JS - or no JS at all if the page requires no interaction! Some multi-page SSG frameworks such as [Astro](#) also support "partial hydration" - which allows you to use Vue components to create interactive "islands" inside static HTML.

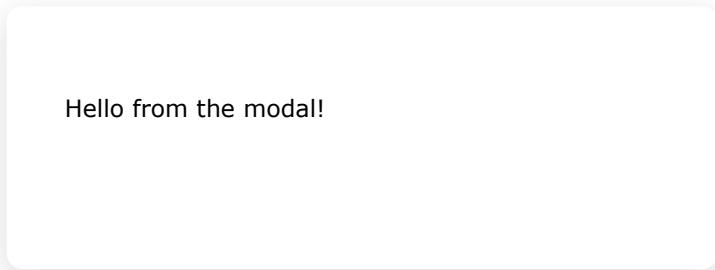
Single-page SSGs are better suited if you expect non-trivial interactivity, deep session lengths, or persisted elements / state across navigations. Otherwise, multi-page SSG would be the better choice.

The Vue team also maintains a static-site generator called [VitePress](#), which powers this website you are reading right now! VitePress supports both flavors of SSG. [Nuxt](#) also supports SSG. You can even mix SSR and SSG for different routes in the same Nuxt app.

## Beyond the Web

Although Vue is primarily designed for building web applications, it is by no means limited to just the browser. You can:

- Build desktop apps with [Electron](#) or [Tauri](#)
- Build mobile apps with [Ionic Vue](#)
- Build desktop and mobile apps from the same codebase with [Quasar](#)
- Use Vue's [Custom Renderer API](#) to build custom renderers targeting [WebGL](#) or even [the terminal!](#)



Hello from the modal!

# Vue and Web Components

[Web Components](#) is an umbrella term for a set of web native APIs that allows developers to create reusable custom elements.

We consider Vue and Web Components to be primarily complementary technologies. Vue has excellent support for both consuming and creating custom elements. Whether you are integrating custom elements into an existing Vue application, or using Vue to build and distribute custom elements, you are in good company.  
Hello from the modal!

## Using Custom Elements in Vue

Vue [scores a perfect 100% in the Custom Elements Everywhere tests](#). Consuming custom elements inside a Vue application largely works the same as using native HTML elements, with a few things to keep in mind:

### Skip Component Resolution

By default, Vue will attempt to resolve a non-native HTML tag as a registered Vue component before falling back to rendering it as a custom element. This will cause Vue to emit a "failed to resolve component" warning during development. To let Vue know that certain elements should be treated as custom elements and skip component resolution, we can specify the `compilerOptions.isCustomElement` option.

If you are using Vue with a build setup, the option should be passed via build configs since it is a compile-time option.

#### Example In-Browser Config

```
// Only works if using in-browser compilation.
// If using build tools, see config examples below.
app.config.compilerOptions.isCustomElement = (tag) => tag.includes('-')
```

#### Example Vite Config

```
// vite.config.js
import vue from '@vitejs/plugin-vue'

export default {
  plugins: [
    vue({
      template: {
        compilerOptions: {
          // treat all tags with a dash as custom elements
          isCustomElement: (tag) => tag.includes('-')
        }
      }
    })
  ]
}
```

## Example Vue CLI Config

```
// vue.config.js
module.exports = {
  chainWebpack: config => {
    config.module
      .rule('vue')
      .use('vue-loader')
      .tap(options => ({
        ...options,
        compilerOptions: {
          // treat any tag that starts with ion- as custom elements
          isCustomElement: tag => tag.startsWith('ion-')
        }
      }))
  }
}
```

## Passing DOM Properties

Since DOM attributes can only be strings, we need to pass complex data to custom elements as DOM properties. When setting props on a custom element, Vue 3 automatically checks DOM-property presence using the `in` operator and will prefer setting the value as a DOM property if the key is present. This means that, in most cases, you won't need to think about this if the custom element follows the [recommended best practices](#).

However, there could be rare cases where the data must be passed as a DOM property, but the custom element does not properly define/reflect the property (causing the `in` check to fail). In this case, you can force a `v-bind` binding to be set as a DOM property using the `.prop` modifier:

```
<my-element :user.prop="{ name: 'jack' }"></my-element>
<!-- shorthand equivalent --&gt;
&lt;my-element .user="{ name: 'jack' }"&gt;&lt;/my-element&gt;</pre>

```

## Building Custom Elements with Vue

The primary benefit of custom elements is that they can be used with any framework, or even without a framework. This makes them ideal for distributing components where the end consumer may not be using the same frontend stack, or when you want to insulate the end application from the implementation details of the components it uses.

### defineCustomElement

Vue supports creating custom elements using exactly the same Vue component APIs via the `defineCustomElement` method. The method accepts the same argument as `defineComponent`, but instead returns a custom element constructor that extends `HTMLElement`:

```
<my-vue-element></my-vue-element>
```

```

import { defineCustomElement } from 'vue'

const MyVueElement = defineCustomElement({
  // normal Vue component options here
  props: {},
  emits: [],
  template: `...`,

  // defineCustomElement only: CSS to be injected into shadow root
  styles: [`/* inlined css */`]
})

// Register the custom element.
// After registration, all `<my-vue-element>` tags
// on the page will be upgraded.
customElements.define('my-vue-element', MyVueElement)

// You can also programmatically instantiate the element:
// (can only be done after registration)
document.body.appendChild(
  new MyVueElement({
    // initial props (optional)
  })
)

```

## Lifecycle

- A Vue custom element will mount an internal Vue component instance inside its shadow root when the element's `connectedCallback` is called for the first time.
- When the element's `disconnectedCallback` is invoked, Vue will check whether the element is detached from the document after a microtask tick.
  - If the element is still in the document, it's a move and the component instance will be preserved;
  - If the element is detached from the document, it's a removal and the component instance will be unmounted.

## Props

- All props declared using the `props` option will be defined on the custom element as properties. Vue will automatically handle the reflection between attributes / properties where appropriate.
  - Attributes are always reflected to corresponding properties.
  - Properties with primitive values (`string`, `boolean` or `number`) are reflected as attributes.
- Vue also automatically casts props declared with `Boolean` or `Number` types into the desired type when they are set as attributes (which are always strings). For example, given the following props declaration:

```

props: {
  selected: Boolean,
  index: Number
}

```

And the custom element usage:

```
<my-element selected index="1"></my-element>
```

In the component, `selected` will be cast to `true` (boolean) and `index` will be cast to `1` (number).

## Events

Events emitted via `this.$emit` or setup `emit` are dispatched as native [CustomEvents](#) on the custom element. Additional event arguments (payload) will be exposed as an array on the CustomEvent object as its `detail` property.

## Slots

Hello from the modal!

Inside the component, slots can be rendered using the `<slot/>` element as usual. However, when consuming the resulting element, it only accepts [native slots syntax](#):

- [Scoped slots](#) are not supported.
- When passing named slots, use the `slot` attribute instead of the `v-slot` directive:

```
<my-element>
  <div slot="named">hello</div>
</my-element>
```

## Provide / Inject

The [Provide / Inject API](#) and its [Composition API equivalent](#) also work between Vue-defined custom elements. However, note that this works **only between custom elements**. i.e. a Vue-defined custom element won't be able to inject properties provided by a non-custom-element Vue component.

## SFC as Custom Element

`defineCustomElement` also works with Vue Single-File Components (SFCs). However, with the default tooling setup, the `<style>` inside the SFCs will still be extracted and merged into a single CSS file during production build. When using an SFC as a custom element, it is often desirable to inject the `<style>` tags into the custom element's shadow root instead.

The official SFC toolings support importing SFCs in "custom element mode" (requires `@vitejs/plugin-vue@^1.4.0` or `vue-loader@^16.5.0`). An SFC loaded in custom element mode inlines its `<style>` tags as strings of CSS and exposes them under the component's `styles` option. This will be picked up by `defineCustomElement` and injected into the element's shadow root when instantiated.

To opt-in to this mode, simply end your component file name with `.ce.vue`:

```
import { defineCustomElement } from 'vue'
import Example from './Example.ce.vue'

console.log(Example.styles) // ["/* inlined css */"]

// convert into custom element constructor
const ExampleElement = defineCustomElement(Example)
```

```
// register
customElements.define('my-example', ExampleElement)
```

If you wish to customize what files should be imported in custom element mode (for example, treating *all* SFCs as custom elements), you can pass the `customElement` option to the respective build plugins:

- [@vitejs/plugin-vue](#)
- [vue-loader](#)

## Tips for a Vue Custom Elements Library

Hello from the modal!

When building custom elements with Vue, the elements will rely on Vue's runtime. There is a ~16kb baseline size cost depending on how many features are being used. This means it is not ideal to use Vue if you are shipping a single custom element - you may want to use vanilla JavaScript, [petite-vue](#), or frameworks that specialize in small runtime size. However, the base size is more than justifiable if you are shipping a collection of custom elements with complex logic, as Vue will allow each component to be authored with much less code. The more elements you are shipping together, the better the trade-off.

If the custom elements will be used in an application that is also using Vue, you can choose to externalize Vue from the built bundle so that the elements will be using the same copy of Vue from the host application.

It is recommended to export the individual element constructors to give your users the flexibility to import them on-demand and register them with desired tag names. You can also export a convenience function to automatically register all elements. Here's an example entry point of a Vue custom element library:

```
import { defineCustomElement } from 'vue'
import Foo from './MyFoo.ce.vue'
import Bar from './MyBar.ce.vue'

const MyFoo = defineCustomElement(Foo)
const MyBar = defineCustomElement(Bar)

// export individual elements
export { MyFoo, MyBar }

export function register() {
  customElements.define('my-foo', MyFoo)
  customElements.define('my-bar', MyBar)
}
```

If you have many components, you can also leverage build tool features such as Vite's [glob import](#) or webpack's `require.context` to load all components from a directory.

## Web Components vs. Vue Components

Some developers believe that framework-proprietary component models should be avoided, and that exclusively using Custom Elements makes an application "future-proof". Here we will try to explain why we believe that this is an overly simplistic take on the problem.

There is indeed a certain level of feature overlap between Custom Elements and Vue Components: they both allow us to define reusable components with data passing, event emitting, and lifecycle management. However, Web Components APIs are relatively low-level and bare-bones. To build an actual application, we need quite a few additional capabilities which the platform does not cover:

- A declarative and efficient templating system;
- A reactive state management system that facilitates cross-component logic extraction and reuse;
- A performant way to render the components on the server and hydrate them on the client (SSR), which is important for SEO and [Web Vitals metrics such as LCP](#). Native custom elements SSR typically involves simulating the DOM in Node.js and then serializing the mutated DOM, while Vue SSR compiles into string concatenation whenever possible, which is much more efficient.

Vue's component model is designed with these needs in mind as a coherent system.

With a competent engineering team, you could probably build the equivalent on top of native Custom Elements - but this also means you are taking on the long-term maintenance burden of an in-house framework, while losing out on the ecosystem and community benefits of a mature framework like Vue.

There are also frameworks built using Custom Elements as the basis of their component model, but they all inevitably have to introduce their proprietary solutions to the problems listed above. Using these frameworks entails buying into their technical decisions on how to solve these problems - which, despite what may be advertised, doesn't automatically insulate you from potential future chucks.

There are also some areas where we find custom elements to be limiting:

- Eager slot evaluation hinders component composition. Vue's [scoped slots](#) are a powerful mechanism for component composition, which can't be supported by custom elements due to native slots' eager nature. Eager slots also mean the receiving component cannot control when or whether to render a piece of slot content.
- Shipping custom elements with shadow DOM scoped CSS today requires embedding the CSS inside JavaScript so that they can be injected into shadow roots at runtime. They also result in duplicated styles in markup in SSR scenarios. There are [platform features](#) being worked on in this area - but as of now they are not yet universally supported, and there are still production performance / SSR concerns to be addressed. In the meanwhile, Vue SFCs provide [CSS scoping mechanisms](#) that support extracting the styles into plain CSS files.

Vue will always stay up to date with the latest standards in the web platform, and we will happily leverage whatever the platform provides if it makes our job easier. However, our goal is to provide solutions that work well and work today. That means we have to incorporate new platform features with a critical mindset - and that involves filling the gaps where the standards fall short while that is still the case.

[Go to TOC](#)

# Introduction

:::info You are reading the documentation for Vue 3!

- Vue 2 documentation has been moved to [v2.vuejs.org](https://v2.vuejs.org).
- Upgrading from Vue 2? Check out the [Migration Guide](#). :::



Hello from the modal!

Learn Vue with video tutorials on [VueMastery.com](https://VueMastery.com)



## What is Vue?

Vue (pronounced /vju:/, like **view**) is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS, and JavaScript and provides a declarative and component-based programming model that helps you efficiently develop user interfaces, be they simple or complex.

Here is a minimal example:

```
import { createApp } from 'vue'

createApp({
  data() {
    return {
      count: 0
    }
  }
}).mount('#app')

<div id="app">
  <button @click="count++">
    Count is: {{ count }}
  </button>
</div>
```

### Result

Count is: {{ count }}

The above example demonstrates the two core features of Vue:

- **Declarative Rendering:** Vue extends standard HTML with a template syntax that allows us to declaratively describe HTML output based on JavaScript state.
- **Reactivity:** Vue automatically tracks JavaScript state changes and efficiently updates the DOM when changes happen.

You may already have questions - don't worry. We will cover every little detail in the rest of the documentation. For now, please read along so you can have a high-level understanding of what Vue offers.

:::tip Prerequisites The rest of the documentation assumes basic familiarity with 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 and then come back! You can check your knowledge level with [this JavaScript overview](#). Prior experience with other frameworks helps, but is not required. :::

## The Progressive Framework

Hello from the modal!

Vue is a framework and ecosystem that covers most of the common features needed in frontend development. But the web is extremely diverse - the things we build on the web may vary drastically in form and scale. With that in mind, Vue is designed to be flexible and incrementally adoptable. Depending on your use case, Vue can be used in different ways:

- Enhancing static HTML without a build step
- Embedding as Web Components on any page
- Single-Page Application (SPA)
- Fullstack / Server-Side Rendering (SSR)
- Jamstack / Static Site Generation (SSG)
- Targeting desktop, mobile, WebGL, and even the terminal

If you find these concepts intimidating, don't worry! The tutorial and guide only require basic HTML and JavaScript knowledge, and you should be able to follow along without being an expert in any of these.

If you are an experienced developer interested in how to best integrate Vue into your stack, or you are curious about what these terms mean, we discuss them in more detail in [Ways of Using Vue](#).

Despite the flexibility, the core knowledge about how Vue works is shared across all these use cases. Even if you are just a beginner now, the knowledge gained along the way will stay useful as you grow to tackle more ambitious goals in the future. If you are a veteran, you can pick the optimal way to leverage Vue based on the problems you are trying to solve, while retaining the same productivity. This is why we call Vue "The Progressive Framework": it's a framework that can grow with you and adapt to your needs.

## Single-File Components

In most build-tool-enabled Vue projects, we author Vue components using an HTML-like file format called **Single-File Component** (also known as `*.vue` files, abbreviated as **SFC**). A Vue SFC, as the name suggests, encapsulates the component's logic (JavaScript), template (HTML), and styles (CSS) in a single file. Here's the previous example, written in SFC format:

```
<script>
export default {
  data() {
    return {
      count: 0
    }
  }
}
```

```
</script>

<template>
  <button @click="count++">Count is: {{ count }}</button>
</template>

<style scoped>
button {
  font-weight: bold;
}
</style>
```

SFC is a defining feature of Vue and is often the preferred way to author Vue components **if** your use case warrants a build setup. You can learn more about the [how and why of SFC](#) in its dedicated section - but for now, just know that Vue will handle all the build tools setup for you.

## API Styles

Vue components can be authored in two different API styles: **Options API** and **Composition API**.

### Options API

With Options API, we define a component's logic using an object of options such as `data`, `methods`, and `mounted`. Properties defined by options are exposed on `this` inside functions, which points to the component instance:

```
<script>
export default {
  // Properties returned from data() become reactive state
  // and will be exposed on `this`.
  data() {
    return {
      count: 0
    }
  },
  // Methods are functions that mutate state and trigger updates.
  // They can be bound as event listeners in templates.
  methods: {
    increment() {
      this.count++
    }
  },
  // Lifecycle hooks are called at different stages
  // of a component's lifecycle.
  // This function will be called when the component is mounted.
  mounted() {
    console.log(`The initial count is ${this.count}.`)
  }
}
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

[Try it in the Playground](#)

## Composition API

With Composition API, we define a component's logic using imported API functions. In SFCs, Composition API is typically used with `<script setup>`. The `setup` attribute is a hint that makes Vue perform compile-time transforms that allow us to use Composition API with less boilerplate. For example, imports and top-level variables / functions declared in `<script setup>` are directly usable in the template.

Here is the same component, with the exact same template, but using Composition API and `<script setup>` instead:

```
<script setup>
import { ref, onMounted } from 'vue'

// reactive state
const count = ref(0)

// functions that mutate state and trigger updates
function increment() {
  count.value++
}

// lifecycle hooks
onMounted(() => {
  console.log(`The initial count is ${count.value}.`)
})
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

[Try it in the Playground](#)

## Which to Choose?

Both API styles are fully capable of covering common use cases. They are different interfaces powered by the exact same underlying system. In fact, the Options API is implemented on top of the Composition API! The fundamental concepts and knowledge about Vue are shared across the two styles.

The Options API is centered around the concept of a "component instance" (`this` as seen in the example), which typically aligns better with a class-based mental model for users coming from OOP language backgrounds. It is also more beginner-friendly by abstracting away the reactivity details and enforcing code organization via option groups.

The Composition API is centered around declaring reactive state variables directly in a function scope and composing state from multiple functions together to handle complexity. It is more free-form and requires an understanding of how reactivity works in Vue to be used effectively. In return, its flexibility enables more powerful patterns for organizing and reusing logic.

You can learn more about the comparison between the two styles and the potential benefits of Composition API in the [Composition API FAQ](#).

If you are new to Vue, here's our general recommendation:

- For learning purposes, go with the style that looks easier to understand to you. Again, most of the core concepts are shared between the two styles. You can always pick up the other style later.
- For production use:
  - Go with Options API if you are not using build tools, or plan to use Vue primarily in low-complexity scenarios, e.g. progressive enhancement.
  - Go with Composition API + Single-File Components if you plan to build full applications with Vue.  
Hello from the Modular!

You don't have to commit to only one style during the learning phase. The rest of the documentation will provide code samples in both styles where applicable, and you can toggle between them at any time using the **API Preference switches** at the top of the left sidebar.

## Still Got Questions?

Check out our [FAQ](#).

## Pick Your Learning Path

Different developers have different learning styles. Feel free to pick a learning path that suits your preference - although we do recommend going over all of the content, if possible!

[Try the Tutorial](#)

For those who prefer learning things hands-on.

[Read the Guide](#)

The guide walks you through every aspect of the framework in full detail.

[Check out the Examples](#)

Explore examples of core features and common UI tasks.

---

[Go to TOC](#)

# Quick Start

## Try Vue Online

- To quickly get a taste of Vue, you can try it directly in our [Playground](#).
- If you prefer a plain HTML setup without any build steps, you can use this [JSFiddle](#) as your starting point.
- If you are already familiar with Node.js and the concept of build tools, you can also try a complete build setup right within your browser on [StackBlitz](#).

## Creating a Vue Application

:::tip Prerequisites

- Familiarity with the command line
- Install [Node.js](#) version 16.0 or higher :::

In this section we will introduce how to scaffold a Vue [Single Page Application](#) on your local machine. The created project will be using a build setup based on [Vite](#) and allow us to use Vue [Single-File Components](#) (SFCs).

Make sure you have an up-to-date version of [Node.js](#) installed, then run the following command in your command line (without the `>` sign):

```
> npm init vue@latest
```

This command will install and execute [create-vue](#), the official Vue project scaffolding tool. You will be presented with prompts for several optional features such as TypeScript and testing support:

- ✓ Project name: ... <your-project-name>
- ✓ Add TypeScript? ... [No](#) / Yes
- ✓ Add JSX Support? ... [No](#) / Yes
- ✓ Add Vue Router for Single Page Application development? ... [No](#) / Yes
- ✓ Add Pinia for state management? ... [No](#) / Yes
- ✓ Add Vitest for Unit testing? ... [No](#) / Yes
- ✓ Add Cypress for both Unit and End-to-End testing? ... [No](#) / Yes
- ✓ Add ESLint for code quality? ... [No](#) / Yes
- ✓ Add Prettier for code formatting? ... [No](#) / Yes

```
Scaffolding project in ./<your-project-name>...
```

```
Done.
```

If you are unsure about an option, simply choose [No](#) by hitting enter for now. Once the project is created, follow the instructions to install dependencies and start the dev server:

```
> cd <your-project-name>
> npm install
> npm run dev
```

You should now have your first Vue project running! Note that the example components in the generated project are written using the [Composition API](#) and `<script setup>`, rather than the [Options API](#). Here are some additional tips:

- The recommended IDE setup is [Visual Studio Code + Volar extension](#). If you use other editors, check out the [IDE support section](#). Hello from the modal!
- More tooling details, including integration with backend frameworks, are discussed in the [Tooling Guide](#).
- To learn more about the underlying build tool Vite, check out the [Vite docs](#).
- If you choose to use TypeScript, check out the [TypeScript Usage Guide](#).

When you are ready to ship your app to production, run the following:

```
> npm run build
```

This will create a production-ready build of your app in the project's `./dist` directory. Check out the [Production Deployment Guide](#) to learn more about shipping your app to production.

[Next Steps >](#)

## Using Vue from CDN

You can use Vue directly from a CDN via a script tag:

```
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
```

Here we are using [unpkg](#), but you can also use any CDN that serves npm packages, for example [jsdelivr](#) or [cdnjs](#). Of course, you can also download this file and serve it yourself.

When using Vue from a CDN, there is no "build step" involved. This makes the setup a lot simpler, and is suitable for enhancing static HTML or integrating with a backend framework. However, you won't be able to use the Single-File Component (SFC) syntax.

## Using the Global Build

The above link loads the *global build* of Vue, where all top-level APIs are exposed as properties on the global `Vue` object. Here is a full example using the global build:

```
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>

<div id="app">{{ message }}</div>

<script>
  const { createApp } = Vue
  createApp({
    data() {
```

```

        return {
          message: 'Hello Vue!'
        }
      }).mount('#app')
</script>

```

[JSFiddle demo](#)

## Using the ES Module Build

Throughout the rest of the documentation we will be primarily using [ES modules](#) syntax. Most modern browsers now support ES modules natively, so we can use Vue from a CDN via native ES modules like this:

```

<div id="app">{{ message }}</div>

<script type="module">
  import { createApp } from 'https://unpkg.com/vue@3/dist/vue.esm-browser.js'

  createApp({
    data() {
      return {
        message: 'Hello Vue!'
      }
    }
  }).mount('#app')
</script>

```

Notice that we are using `<script type="module">`, and the imported CDN URL is pointing to the [ES modules build](#) of Vue instead.

[JSFiddle demo](#)

## Enabling Import maps

In the above example, we are importing from the full CDN URL, but in the rest of the documentation you will see code like this:

```
import { createApp } from 'vue'
```

We can teach the browser where to locate the `vue` import by using [Import Maps](#):

```

<script type="importmap">
{
  "imports": {
    "vue": "https://unpkg.com/vue@3/dist/vue.esm-browser.js"
  }
}
</script>

<div id="app">{{ message }}</div>

<script type="module">
  import { createApp } from 'vue'
  createApp({

```

```

data() {
  return {
    message: 'Hello Vue!'
  }
}).mount('#app')
</script>

```

### JSFiddle demo

You can also add entries for other dependencies to the import map - but make sure they point to the ES modules version of the library you intend to use.

Hello from the modal!

:::tip Import Maps Browser Support Import maps are supported by default in Chromium-based browsers, so we recommend using Chrome or Edge during the learning process.

If using Firefox, it is only supported in version 102+ and currently needs to be enabled via the `dom.importMaps.enabled` option in `about:config`.

If your preferred browser does not support import maps yet, you can polyfill it with [es-module-shims](#). :::

:::warning Notes on Production Use The examples so far are using the development build of Vue - if you intend to use Vue from a CDN in production, make sure to check out the [Production Deployment Guide](#). :::

## Splitting Up the Modules

As we dive deeper into the guide, we may need to split our code into separate JavaScript files so that they are easier to manage. For example:

```

<!-- index.html -->
<script type="module">
  import { createApp } from 'vue'
  import MyComponent from './my-component.js'

  createApp(MyComponent).mount('#app')
</script>

```

```

// my-component.js
export default {
  data() {
    return { count: 0 }
  },
  template: `<div>count is {{ count }}</div>`
}

```

If you directly open the above `index.html` in your browser, you will find that it throws an error because ES modules cannot work over the `file://` protocol. In order for this to work, you need to serve your `index.html` over the `http://` protocol, with a local HTTP server.

To start a local HTTP server, first install [Node.js](#) and then run `npx serve` from the command line in the same directory where your HTML file is. You can also use any other HTTP server that can serve static files with the correct MIME types.

You may have noticed that the imported component's template is inlined as a JavaScript string. If you are using VSCode, you can install the [es6-string-html](#) extension and prefix the strings with a `/*html*/` comment to get syntax highlighting for them.

## Using Composition API without a Build Step

Many of the examples for Composition API will be using the `<script setup>` syntax. If you intend to use Composition API without a build step, consult the usage of the [setup\(\) option](#).

## Next Steps

Hello from the modal!

If you skipped the [Introduction](#), we strongly recommend reading it before moving on to the rest of the documentation.

[Continue with the Guide](#)

The guide walks you through every aspect of the framework in full detail.

[Try the Tutorial](#)

For those who prefer learning things hands-on.

[Check out the Examples](#)

Explore examples of core features and common UI tasks.

---

[Go to TOC](#)

# Composables

:::tip This section assumes basic knowledge of Composition API. If you have been learning Vue with Options API only, you can set the API Preference to Composition API (using the toggle at the top of the left sidebar) and re-read the [Reactivity Fundamentals](#) and [Lifecycle Hooks](#) chapters. :::

## What is a "Composable"?

Hello from the modal!

In the context of Vue applications, a "composable" is a function that leverages Vue's Composition API to encapsulate and reuse **stateful logic**.

When building frontend applications, we often need to reuse logic for common tasks. For example, we may need to format dates in many places, so we extract a reusable function for that. This formatter function encapsulates **stateless logic**: it takes some input and immediately returns expected output. There are many libraries out there for reusing stateless logic - for example [lodash](#) and [date-fns](#), which you may have heard of.

By contrast, stateful logic involves managing state that changes over time. A simple example would be tracking the current position of the mouse on a page. In real world scenarios, it could also be more complex logic such as touch gestures or connection status to a database.

## Mouse Tracker Example

If we were to implement the mouse tracking functionality using the Composition API directly inside a component, it would look like this:

```
<script setup>
import { ref, onMounted, onUnmounted } from 'vue'

const x = ref(0)
const y = ref(0)

function update(event) {
  x.value = event.pageX
  y.value = event.pageY
}

onMounted(() => window.addEventListener('mousemove', update))
onUnmounted(() => window.removeEventListener('mousemove', update))
</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>
```

But what if we want to reuse the same logic in multiple components? We can extract the logic into an external file, as a composable function:

```
// mouse.js
import { ref, onMounted, onUnmounted } from 'vue'

// by convention, composable function names start with "use"
```

```

export function useMouse() {
  // state encapsulated and managed by the composable
  const x = ref(0)
  const y = ref(0)

  // a composable can update its managed state over time.
  function update(event) {
    x.value = event.pageX
    y.value = event.pageY
  }

  // a composable can also hook into its owner component's
  // lifecycle to setup and teardown side effects.
  onMounted(() => window.addEventListener('mousemove', update))
  onUnmounted(() => window.removeEventListener('mousemove', update))

  // expose managed state as return value
  return { x, y }
}

```

And this is how it can be used in components:

```

<script setup>
import { useMouse } from './mouse.js'

const { x, y } = useMouse()
</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>

```

Mouse position is at: {{ x }}, {{ y }}

[Try it in the Playground](#)

As we can see, the core logic remains identical - all we had to do was move it into an external function and return the state that should be exposed. Just like inside a component, you can use the full range of [Composition API functions](#) in composables. The same `useMouse()` functionality can now be used in any component.

The cooler part about composables though, is that you can also nest them: one composable function can call one or more other composable functions. This enables us to compose complex logic using small, isolated units, similar to how we compose an entire application using components. In fact, this is why we decided to call the collection of APIs that make this pattern possible Composition API.

For example, we can extract the logic of adding and removing a DOM event listener into its own composable:

```

// event.js
import { onMounted, onUnmounted } from 'vue'

export function useEventListener(target, event, callback) {
  // if you want, you can also make this
  // support selector strings as target
  onMounted(() => target.addEventListener(event, callback))
  onUnmounted(() => target.removeEventListener(event, callback))
}

```

And now our `useMouse()` composable can be simplified to:

```
// mouse.js
import { ref } from 'vue'
import { useEventListener } from './event'

export function useMouse() {
  const x = ref(0)
  const y = ref(0)

  useEventListener(window, 'mousemove', (event) => {
    x.value = event.pageX
    y.value = event.pageY
  })

  return { x, y }
}
```

:::tip Each component instance calling `useMouse()` will create its own copies of `x` and `y` state so they won't interfere with one another. If you want to manage shared state between components, read the [State Management](#) chapter. :::

## Async State Example

The `useMouse()` composable doesn't take any arguments, so let's take a look at another example that makes use of one. When doing async data fetching, we often need to handle different states: loading, success, and error:

```
<script setup>
import { ref } from 'vue'

const data = ref(null)
const error = ref(null)

fetch('...')
  .then((res) => res.json())
  .then((json) => (data.value = json))
  .catch((err) => (error.value = err))
</script>

<template>
  <div v-if="error">Oops! Error encountered: {{ error.message }}</div>
  <div v-else-if="data">
    Data loaded:
    <pre>{{ data }}</pre>
  </div>
  <div v-else>Loading...</div>
</template>
```

It would be tedious to have to repeat this pattern in every component that needs to fetch data. Let's extract it into a composable:

```
// fetch.js
import { ref } from 'vue'

export function useFetch(url) {
```

```

const data = ref(null)
const error = ref(null)

fetch(url)
  .then((res) => res.json())
  .then((json) => (data.value = json))
  .catch((err) => (error.value = err))

return { data, error }
}

```

Now in our component we can just do:

Hello from the modal!

```

<script setup>
import { useFetch } from './fetch.js'

const { data, error } = useFetch('...')
</script>

```

`useFetch()` takes a static URL string as input - so it performs the fetch only once and is then done. What if we want it to re-fetch whenever the URL changes? We can achieve that by also accepting refs as an argument:

```

// fetch.js
import { ref, isRef, unref, watchEffect } from 'vue'

export function useFetch(url) {
  const data = ref(null)
  const error = ref(null)

  function doFetch() {
    // reset state before fetching..
    data.value = null
    error.value = null
    // unref() unwraps potential refs
    fetch(unref(url))
      .then((res) => res.json())
      .then((json) => (data.value = json))
      .catch((err) => (error.value = err))
  }

  if (isRef(url)) {
    // setup reactive re-fetch if input URL is a ref
    watchEffect(doFetch)
  } else {
    // otherwise, just fetch once
    // and avoid the overhead of a watcher
    doFetch()
  }

  return { data, error }
}

```

This version of `useFetch()` now accepts both static URL strings and refs of URL strings. When it detects that the URL is a dynamic ref using `isRef()`, it sets up a reactive effect using `watchEffect()`. The effect will run immediately and will also track the URL ref as a dependency. Whenever the URL ref changes, the data will be reset and fetched again.

Here's the updated version of `useFetch()`, with an artificial delay and randomized error for demo purposes.

## Conventions and Best Practices

### Naming

It is a convention to name composable functions with camelCase names that start with "use".

### Input Arguments

Hello from the modal!

A composable can accept `ref` arguments even if it doesn't rely on them for reactivity. If you are writing a composable that may be used by other developers, it's a good idea to handle the case of input arguments being `refs` instead of raw values. The `unref()` utility function will come in handy for this purpose:

```
import { unref } from 'vue'

function useFeature(maybeRef) {
  // if maybeRef is indeed a ref, its .value will be returned
  // otherwise, maybeRef is returned as-is
  const value = unref(maybeRef)
}
```

If your composable creates reactive effects when the input is a `ref`, make sure to either explicitly watch the `ref` with `watch()`, or call `unref()` inside a `watchEffect()` so that it is properly tracked.

### Return Values

You have probably noticed that we have been exclusively using `ref()` instead of `reactive()` in composables. The recommended convention is for composables to always return a plain, non-reactive object containing multiple `refs`. This allows it to be destructured in components while retaining reactivity:

```
// x and y are refs
const { x, y } = useMouse()
```

Returning a reactive object from a composable will cause such destructures to lose the reactivity connection to the state inside the composable, while the `refs` will retain that connection.

If you prefer to use returned state from composables as object properties, you can wrap the returned object with `reactive()` so that the `refs` are unwrapped. For example:

```
const mouse = reactive(useMouse())
// mouse.x is linked to original ref
console.log(mouse.x)
```

```
Mouse position is at: {{ mouse.x }}, {{ mouse.y }}
```

### Side Effects

It is OK to perform side effects (e.g. adding DOM event listeners or fetching data) in composables, but pay attention to the following rules:

- If you are working on an application that uses [Server-Side Rendering \(SSR\)](#), make sure to perform DOM-specific side effects in post-mount lifecycle hooks, e.g. `onMounted()`. These hooks are only called in the browser, so you can be sure that code inside them has access to the DOM.
- Remember to clean up side effects in `onUnmounted()`. For example, if a composable sets up a DOM event listener, it should remove that listener in `onUnmounted()` as we have seen in the `useMouse()` example. It can be a good idea to use a composable that automatically does this for you, like the `useEventListener()` example.

## Usage Restrictions

Hello from the modal!

Composables should only be called **synchronously** in `<script setup>` or the `setup()` hook. In some cases, you can also call them in lifecycle hooks like `onMounted()`.

These are the contexts where Vue is able to determine the current active component instance. Access to an active component instance is necessary so that:

1. Lifecycle hooks can be registered to it.
2. Computed properties and watchers can be linked to it, so that they can be disposed when the instance is unmounted to prevent memory leaks.

:::tip `<script setup>` is the only place where you can call composables **after** using `await`. The compiler automatically restores the active instance context for you after the async operation. :::

## Extracting Composables for Code Organization

Composables can be extracted not only for reuse, but also for code organization. As the complexity of your components grow, you may end up with components that are too large to navigate and reason about. Composition API gives you the full flexibility to organize your component code into smaller functions based on logical concerns:

```
<script setup>
import { useFeatureA } from './featureA.js'
import { useFeatureB } from './featureB.js'
import { useFeatureC } from './featureC.js'

const { foo, bar } = useFeatureA()
const { baz } = useFeatureB(foo)
const { qux } = useFeatureC(baz)
</script>
```

To some extent, you can think of these extracted composables as component-scoped services that can talk to one another.

## Using Composables in Options API

If you are using Options API, composables must be called inside `setup()`, and the returned bindings must be returned from `setup()` so that they are exposed to `this` and the template:

```

import { useMouse } from './mouse.js'
import { useFetch } from './fetch.js'

export default {
  setup() {
    const { x, y } = useMouse()
    const { data, error } = useFetch('...')
    return { x, y, data, error }
  },
  mounted() {
    // setup() exposed properties can be accessed on `this`
    console.log(this.x)
  }
  // ...other options
}

```

## Comparisons with Other Techniques

### vs. Mixins

Users coming from Vue 2 may be familiar with the [mixins](#) option, which also allows us to extract component logic into reusable units. There are three primary drawbacks to mixins:

- 1. Unclear source of properties:** when using many mixins, it becomes unclear which instance property is injected by which mixin, making it difficult to trace the implementation and understand the component's behavior. This is also why we recommend using the refs + destructure pattern for composables: it makes the property source clear in consuming components.
- 2. Namespace collisions:** multiple mixins from different authors can potentially register the same property keys, causing namespace collisions. With composables, you can rename the destructured variables if there are conflicting keys from different composables.
- 3. Implicit cross-mixin communication:** multiple mixins that need to interact with one another have to rely on shared property keys, making them implicitly coupled. With composables, values returned from one composable can be passed into another as arguments, just like normal functions.

For the above reasons, we no longer recommend using mixins in Vue 3. The feature is kept only for migration and familiarity reasons.

### vs. Renderless Components

In the component slots chapter, we discussed the [Renderless Component](#) pattern based on scoped slots. We even implemented the same mouse tracking demo using renderless components.

The main advantage of composables over renderless components is that composables do not incur the extra component instance overhead. When used across an entire application, the amount of extra component instances created by the renderless component pattern can become a noticeable performance overhead.

The recommendation is to use composables when reusing pure logic, and use components when reusing both logic and visual layout.

## vs. React Hooks

If you have experience with React, you may notice that this looks very similar to custom React hooks. Composition API was in part inspired by React hooks, and Vue composables are indeed similar to React hooks in terms of logic composition capabilities. However, Vue composables are based on Vue's fine-grained reactivity system, which is fundamentally different from React hooks' execution model. This is discussed in more detail in the [Composition API FAQ](#).

## Further Reading

Hello from the modal!

- [Reactivity In Depth](#): for a low-level understanding of how Vue's reactivity system works.
- [State Management](#): for patterns of managing state shared by multiple components.
- [Testing Composables](#): tips on unit testing composables.
- [VueUse](#): an ever-growing collection of Vue composables. The source code is also a great learning resource.

---

[Go to TOC](#)

# Custom Directives

## Introduction

In addition to the default set of directives shipped in core (like `v-model` or `v-show`), Vue also allows you to register your own custom directives.

We have introduced two forms of code reuse in Vue: [components](#) and [composables](#). Components are the main building blocks, while composable are focused on reusing stateful logic. Custom directives, on the other hand, are mainly intended for reusing logic that involves low-level DOM access on plain elements.

A custom directive is defined as an object containing lifecycle hooks similar to those of a component. The hooks receive the element the directive is bound to. Here is an example of a directive that focuses an input when the element is inserted into the DOM by Vue:

```
<script setup>
// enables v-focus in templates
const vFocus = {
  mounted: (el) => el.focus()
}
</script>

<template>
  <input v-focus />
</template>
```

```
const focus = {
  mounted: (el) => el.focus()
}

export default {
  directives: {
    // enables v-focus in template
    focus
  }
}
```

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

This should be focused

Assuming you haven't clicked elsewhere on the page, the input above should be auto-focused. This directive is more useful than the `autofocus` attribute because it works not just on page load - it also works when the element is dynamically inserted by Vue.

In `<script setup>`, any camelCase variable that starts with the `v` prefix can be used as a custom directive. In the example above, `vFocus` can be used in the template as `v-focus`.

If not using `<script setup>`, custom directives can be registered using the `directives` option:

```
export default {
  setup() {
    /*...*/
  },
  directives: {
    // enables v-focus in template
    focus: {
      /* ... */
    }
  }
}
```

Similar to components, custom directives from the module are registered so that they can be used in templates. In the example above, we are using local registration via the `directives` option.

It is also common to globally register custom directives at the app level:

```
const app = createApp({})

// make v-focus usable in all components
app.directive('focus', {
  /* ... */
})
```

:::tip Custom directives should only be used when the desired functionality can only be achieved via direct DOM manipulation. Prefer declarative templating using built-in directives such as `v-bind` when possible because they are more efficient and server-rendering friendly. :::

## Directive Hooks

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

```
const myDirective = {
  // called before bound element's attributes
  // or event listeners are applied
  created(el, binding, vnode, prevVnode) {
    // see below for details on arguments
  },
  // called right before the element is inserted into the DOM.
  beforeMount(el, binding, vnode, prevVnode) {},
  // called when the bound element's parent component
  // and all its children are mounted.
  mounted(el, binding, vnode, prevVnode) {},
  // called before the parent component is updated
  beforeUpdate(el, binding, vnode, prevVnode) {},
  // called after the parent component and
  // all of its children have updated
  updated(el, binding, vnode, prevVnode) {},
  // called before the parent component is unmounted
  beforeUnmount(el, binding, vnode, prevVnode) {},
  // called when the parent component is unmounted
  unmounted(el, binding, vnode, prevVnode) {}
}
```

## 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.
  - `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 `beforeUpdate` and `updated`. It is available whether or not the value has changed.
  - `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 }`.
  - `instance` : The instance of the component where the directive is used.
  - `dir` : the directive definition object.
- `vnode` : the underlying VNode representing the bound element.
- `prevNode` : the VNode representing the bound element from the previous render. Only available in the `beforeUpdate` and `updated` hooks.

As an example, consider the following directive usage:

```
<div v-example:foo.bar="baz">
```

The `binding` argument would be an object in the shape of:

```
{
  arg: 'foo',
  modifiers: { bar: true },
  value: /* value of `baz` */,
  oldValue: /* value of `baz` from previous update */
}
```

Similar to built-in directives, custom directive arguments can be dynamic. For example:

```
<div v-example:[arg]="value"></div>
```

Here the directive argument will be reactively updated based on `arg` property in our component state.

:::tip Note 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`. :::

## Function Shorthand

It's common for a custom directive to have the same behavior for `mounted` and `updated`, with no need for the other hooks. In such cases we can define the directive as a function:

```
<div v-color="color"></div>

app.directive('color', (el, binding) => {
  // this will be called for both `mounted` and `updated`
  el.style.color = binding.value
})
```

## Object Literals

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

Hello from the modal!

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

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

## Usage on Components

When used on components, custom directives will always apply to a component's root node, similar to [Fallthrough Attributes](#).

```
<MyComponent v-demo="test" />

<!-- template of MyComponent -->
<div> <!-- v-demo directive will be applied here -->
  <span>My component content</span>
</div>
```

Note that components can potentially have more than one root node. When applied to a multi-root component, a directive will be ignored and a warning will be thrown. Unlike attributes, directives can't be passed to a different element with `v-bind="$attrs"`. In general, it is **not** recommended to use custom directives on components.

# Plugins

## Introduction

Plugins are self-contained code that usually add app-level functionality to Vue. This is how we install a plugin:

```
import { createApp } from 'vue'
const app = createApp({})
app.use(myPlugin, {
  /* optional options */
})
```

A plugin is defined as either an object that exposes an `install()` method, or simply a function that acts as the `install` function itself. The `install` function receives the `app instance` along with additional options passed to `app.use()`, if any:

```
const myPlugin = {
  install(app, options) {
    // configure the app
  }
}
```

There is no strictly defined scope for a plugin, but common scenarios where plugins are useful include:

1. Register one or more global components or custom directives with `app.component()` and `app.directive()`.
2. Make a resource `injectable` throughout the app by calling `app.provide()`.
3. Add some global instance properties or methods by attaching them to `app.config.globalProperties`.
4. A library that needs to perform some combination of the above (e.g. `vue-router`).

## Writing a Plugin

In order to better understand how to create your own Vue.js plugins, we will create a very simplified version of a plugin that displays `i18n` (short for `Internationalization`) strings.

Let's begin by setting up the plugin object. It is recommended to create it in a separate file and export it, as shown below to keep the logic contained and separate.

```
// plugins/i18n.js
export default {
  install: (app, options) => {
    // Plugin code goes here
  }
}
```

We want to create a translation function. This function will receive a dot-delimited `key` string, which we will use to look up the translated string in the user-provided options. This is the intended usage in templates:

```
<h1>{{ $translate('greetings.hello') }}</h1>
```

Since this function should be globally available in all templates, we will make it so by attaching it to `app.-config.globalProperties` in our plugin:

```
// plugins/i18n.js
export default {
  install: (app, options) => {
    // inject a globally available $translate() method
    app.config.globalProperties.$translate = (key) => {
      // retrieve a nested property in `options`
      // using `key` as the path
      return key.split('.').reduce((o, i) => {
        if (o) return o[i]
      }, options)
    }
  }
}
```

Our `$translate` function will take a string such as `greetings.hello`, look inside the user provided configuration and return the translated value.

The object containing the translated keys should be passed to the plugin during installation via additional parameters to `app.use()`:

```
import i18nPlugin from './plugins/i18n'

app.use(i18nPlugin, {
  greetings: {
    hello: 'Bonjour!'
  }
})
```

Now, our initial expression `$translate('greetings.hello')` will be replaced by `Bonjour!` at runtime.

See also: [Augmenting Global Properties](#)

:::tip Use global properties scarcely, since it can quickly become confusing if too many global properties injected by different plugins are used throughout an app. :::

## Provide / Inject with Plugins

Plugins also allow us to use `inject` to provide a function or attribute to the plugin's users. For example, we can allow the application to have access to the `options` parameter to be able to use the translations object.

```
// plugins/i18n.js
export default {
  install: (app, options) => {
    app.config.globalProperties.$translate = (key) => {
      return key.split('.').reduce((o, i) => {
```

```
        if (o) return o[i]
    }, options)
}

app.provide('i18n', options)
}
```

Plugin users will now be able to inject the plugin options into their components using the `i18n` key:

```
<script setup>
import { inject } from 'vue'

const i18n = inject('i18n')

console.log(i18n.greetings.hello)
</script>
```

```
export default {
  inject: ['i18n'],
  created() {
    console.log(this.i18n.greetings.hello)
  }
}
```

# Routing

## Client-Side vs. Server-Side Routing

Routing on the server side means the server sending a response based on the URL path that the user is visiting. When we click on a link in a traditional server-rendered web app, the browser receives an HTML response from the server and reloads the entire page with the new HTML.

Hello from the modal!

In a [Single-Page Application](#) (SPA), however, the client-side JavaScript can intercept the navigation, dynamically fetch new data, and update the current page without full page reloads. This typically results in a more snappy user experience, especially for use cases that are more like actual "applications", where the user is expected to perform many interactions over a long period of time.

In such SPAs, the "routing" is done on the client side, in the browser. A client-side router is responsible for managing the application's rendered view using browser APIs such as [History API](#) or the [hashchange event](#).

## Official Router

[Watch a Free Video Course on Vue School](#)

Vue is well-suited for building SPAs. For most SPAs, it's recommended to use the officially-supported [Vue Router library](#). For more details, see Vue Router's [documentation](#).

## Simple Routing from Scratch

If you only need very simple routing and do not wish to involve a full-featured router library, you can do so with [Dynamic Components](#) and update the current component state by listening to browser [hashchange events](#) or using the [History API](#).

Here's a bare-bone example:

```
<script setup>
import { ref, computed } from 'vue'
import Home from './Home.vue'
import About from './About.vue'
import NotFound from './NotFound.vue'

const routes = {
  '/': Home,
  '/about': About
}

const currentPath = ref(window.location.hash)

window.addEventListener('hashchange', () => {
  currentPath.value = window.location.hash
})
```

```

const currentView = computed(() => {
  return routes[currentPath.value.slice(1) || '/') || NotFound
})
</script>

<template>
  <a href="/">Home</a> |
  <a href="#/about">About</a> |
  <a href="#/non-existent-path">Broken Link</a>
  <component :is="currentView" />
</template>

```

[Try it in the Playground](#)

Hello from the modal!

```

<script>
import Home from './Home.vue'
import About from './About.vue'
import NotFound from './NotFound.vue'

const routes = {
  '/': Home,
  '/about': About
}

export default {
  data() {
    return {
      currentPath: window.location.hash
    }
  },
  computed: {
    currentView() {
      return routes[this.currentPath.slice(1) || '/') || NotFound
    }
  },
  mounted() {
    window.addEventListener('hashchange', () => {
      this.currentPath = window.location.hash
    })
  }
}
</script>

<template>
  <a href="/">Home</a> |
  <a href="#/about">About</a> |
  <a href="#/non-existent-path">Broken Link</a>
  <component :is="currentView" />
</template>

```

[Try it in the Playground](#)[Go to TOC](#)

# Single-File Components

## Introduction

Vue Single-File Components (a.k.a. `*.vue` files, abbreviated as **SFC**) is a special file format that allows us to encapsulate the template, logic, **and** styling of a Vue component in a single file. Here's an example SFC:

```
<script>
export default {
  data() {
    return {
      greeting: 'Hello World!'
    }
  }
}</script>

<template>
  <p class="greeting">{{ greeting }}</p>
</template>

<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

As we can see, Vue SFC is a natural extension of the classic trio of HTML, CSS and JavaScript. The `<template>`, `<script>`, and `<style>` blocks encapsulate and colocate the view, logic and styling of a component in the same file. The full syntax is defined in the [SFC Syntax Specification](#).

## Why SFC

While SFCs require a build step, there are numerous benefits in return:

- Author modularized components using familiar HTML, CSS and JavaScript syntax
- [Colocation of inherently coupled concerns](#)
- Pre-compiled templates without runtime compilation cost
- [Component-scoped CSS](#)
- [More ergonomic syntax when working with Composition API](#)
- More compile-time optimizations by cross-analyzing template and script
- [IDE support](#) with auto-completion and type-checking for template expressions
- Out-of-the-box Hot-Module Replacement (HMR) support

SFC is a defining feature of Vue as a framework, and is the recommended approach for using Vue in the following scenarios:

- Single-Page Applications (SPA)
- Static Site Generation (SSG)

- Any non-trivial frontend where a build step can be justified for better development experience (DX).

That said, we do realize there are scenarios where SFCs can feel like overkill. This is why Vue can still be used via plain JavaScript without a build step. If you are just looking for enhancing largely static HTML with light interactions, you can also check out [petite-vue](#), a 6 kB subset of Vue optimized for progressive enhancement.

## How It Works

Vue SFC is a framework-specific file format and must be pre-compiled by [@vue/compiler-sfc](#) into standard JavaScript and CSS. A compiled SFC is a standard JavaScript (ES) module - which means with proper build setup you can import an SFC like a module:

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

export default {
  components: {
    MyComponent
  }
}
```

`<style>` tags inside SFCs are typically injected as native `<style>` tags during development to support hot updates. For production they can be extracted and merged into a single CSS file.

You can play with SFCs and explore how they are compiled in the [Vue SFC Playground](#).

In actual projects, we typically integrate the SFC compiler with a build tool such as [Vite](#) or [Vue CLI](#) (which is based on [webpack](#)), and Vue provides official scaffolding tools to get you started with SFCs as fast as possible. Check out more details in the [SFC Tooling](#) section.

## What About Separation of Concerns?

Some users coming from a traditional web development background may have the concern that SFCs are mixing different concerns in the same place - which HTML/CSS/JS were supposed to separate!

To answer this question, it is important for us to agree that **separation of concerns is not equal to the separation of file types**. The ultimate goal of engineering principles is to improve the maintainability of codebases. Separation of concerns, when applied dogmatically as separation of file types, does not help us reach that goal in the context of increasingly complex frontend applications.

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 colocating them actually makes the component more cohesive and maintainable.

Note 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 using [Src Imports](#).

---

[Go to TOC](#)

# Server-Side Rendering (SSR)

## Overview

### What is SSR?

Vue.js is a framework for building client-side applications. By default, Vue components produce and manipulate DOM in the browser as output. However, it is also possible to render the same components into HTML strings on the server, send them directly to the browser, and finally "hydrate" the static markup into a fully interactive app on the client.

A server-rendered Vue.js app can also be considered "isomorphic" or "universal", in the sense that the majority of your app's code runs on both the server **and** the client.

### Why SSR?

Compared to a client-side Single-Page Application (SPA), the advantage of SSR primarily lies in:

- **Faster time-to-content:** this is more prominent on slow internet or slow devices. Server-rendered markup doesn't need to wait until all JavaScript has been downloaded and executed to be displayed, so your user will see a fully-rendered page sooner. In addition, data fetching is done on the server-side for the initial visit, which likely has a faster connection to your database than the client. This generally results in improved [Core Web Vitals](#) metrics, better user experience, and can be critical for applications where time-to-content is directly associated with conversion rate.
- **Unified mental model:** you get to use the same language and the same declarative, component-oriented mental model for developing your entire app, instead of jumping back and forth between a backend templating system and a frontend framework.
- **Better SEO:** the search engine crawlers will directly see the fully rendered page.

:::tip As of now, Google and Bing can index synchronous JavaScript applications just fine. Synchronous being the key word there. If your app starts with a loading spinner, then fetches content via Ajax, the crawler will not wait for you to finish. This means if you have content fetched asynchronously on pages where SEO is important, SSR might be necessary. :::

There are also some trade-offs to consider when using SSR:

- Development constraints. Browser-specific code can only be used inside certain lifecycle hooks; some external libraries may need special treatment to be able to run in a server-rendered app.
- More involved build setup and deployment requirements. Unlike a fully static SPA that can be deployed on any static file server, a server-rendered app requires an environment where a Node.js server can run.
- More server-side load. Rendering a full app in Node.js is going to be more CPU-intensive than just serving static files, so if you expect high traffic, be prepared for corresponding server load and wisely employ caching strategies.

Before using SSR for your app, the first question you should ask is whether you actually need it. It mostly depends on how important time-to-content is for your app. For example, if you are building an internal dashboard where an extra few hundred milliseconds on initial load doesn't matter that much, SSR would be an overkill. However, in cases where time-to-content is absolutely critical, SSR can help you achieve the best possible initial load performance.

## SSR vs. SSG

**Static Site Generation (SSG)**, also referred to as pre-rendering, is another popular technique for building fast websites. If the data needed to ~~server-render a page~~ is the same for every user, then instead of rendering the page every time a request comes in, we can render it only once, ahead of time, during the build process. Pre-rendered pages are generated and served as static HTML files.

SSG retains the same performance characteristics of SSR apps: it provides great time-to-content performance. At the same time, it is cheaper and easier to deploy than SSR apps because the output is static HTML and assets. The keyword here is **static**: SSG can only be applied to pages consuming static data, i.e. data that is known at build time and does not change between deploys. Every time the data changes, a new deployment is needed.

If you're only investigating SSR to improve the SEO of a handful of marketing pages (e.g. `/`, `/about`, `/contact`, etc.), then you probably want SSG instead of SSR. SSG is also great for content-based websites such as documentation sites or blogs. In fact, this website you are reading right now is statically generated using [VitePress](#), a Vue-powered static site generator.

# Basic Tutorial

## Rendering an App

Let's take a look at the most bare-bones example of Vue SSR in action.

1. Create a new directory and `cd` into it
2. Run `npm init -y`
3. Add `"type": "module"` in `package.json` so that Node.js runs in [ES modules mode](#).
4. Run `npm install vue`
5. Create an `example.js` file:

```
// this runs in Node.js on the server.
import { createSSRApp } from 'vue'
// Vue's server-rendering API is exposed under `vue/server-renderer`.
import { renderToString } from 'vue/server-renderer'

const app = createSSRApp({
  data: () => ({ count: 1 }),
  template: `<button @click="count++">{{ count }}</button>`
})

renderToString(app).then((html) => {
  console.log(html)
})
```

Then run:

```
> node example.js
```

It should print the following to the command line:

```
<button>1</button>
```

`renderToString()` takes a Vue app instance and returns a Promise that resolves to the rendered HTML of the app. It is also possible to stream rendering using the [Node.js Stream API](#) or [Web Streams API](#). Check out the [SSR API Reference](#) for full details.

We can then move the Vue SSR code into a server request handler, which wraps the application markup with the full page HTML. We will be using `express` for the next steps:

- Run `npm install express`
- Create the following `server.js` file:

```
import express from 'express'
import { createSSRAppl } from 'vue'
import { renderToString } from 'vue/server-renderer'

const server = express()

server.get('/', (req, res) => {
  const app = createSSRAppl({
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })

  renderToString(app).then((html) => {
    res.send(`
      <!DOCTYPE html>
      <html>
        <head>
          <title>Vue SSR Example</title>
        </head>
        <body>
          <div id="app">${html}</div>
        </body>
      </html>
    `)
  })
}

server.listen(3000, () => {
  console.log('ready')
})
```

Finally, run `node server.js` and visit `http://localhost:3000`. You should see the page working with the button.

[Try it on StackBlitz](#)

## Client Hydration

If you click the button, you'll notice the number doesn't change. The HTML is completely static on the client since we are not loading Vue in the browser.

To make the client-side app interactive, Vue needs to perform the **hydration** step. During hydration, it creates the same Vue application that was run on the server, matches each component to the DOM nodes it should control, and attaches DOM event listeners.

To mount an app in hydration mode, we need to use `createSSRApp()` instead of `createApp()` :

Hello from the modal!

```
// this runs in the browser.
import { createSSRApp } from 'vue'

const app = createSSRApp({
  // ...same app as on server
})

// mounting an SSR app on the client assumes
// the HTML was pre-rendered and will perform
// hydration instead of mounting new DOM nodes.
app.mount('#app')
```

## Code Structure

Notice how we need to reuse the same app implementation as on the server. This is where we need to start thinking about code structure in an SSR app - how do we share the same application code between the server and the client?

Here we will demonstrate the most bare-bones setup. First, let's split the app creation logic into a dedicated file, `app.js` :

```
// app.js (shared between server and client)
import { createSSRApp } from 'vue'

export function createApp() {
  return createSSRApp({
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })
}
```

This file and its dependencies are shared between the server and the client - we call them **universal code**. There are a number of things you need to pay attention to when writing universal code, as we will [discuss below](#).

Our client entry imports the universal code, creates the app, and performs the mount:

```
// client.js
import { createApp } from './app.js'

createApp().mount('#app')
```

And the server uses the same app creation logic in the request handler:

```
// server.js (irrelevant code omitted)
import { createApp } from './app.js'

server.get('/', (req, res) => {
  const app = createApp()
  renderToString(app).then(html => {
    // ...
  })
})
```

In addition, in order to load the client files in the browser, we also need to:

1. Serve client files by adding `server.use(express.static('.'))` in `server.js`.
2. Load the client entry by adding `<script type="module" src="/client.js"></script>` to the HTML shell.
3. Support usage like `import * from 'vue'` in the browser by adding an [Import Map](#) to the HTML shell.

[Try the completed example on StackBlitz](#). The button is now interactive!

## Higher Level Solutions

Moving from the example to a production-ready SSR app involves a lot more. We will need to:

- Support Vue SFCs and other build step requirements. In fact, we will need to coordinate two builds for the same app: one for the client, and one for the server.
- ::::tip Vue components are compiled differently when used for SSR - templates are compiled into string concatenations instead of Virtual DOM render functions for more efficient rendering performance. ::::
- In the server request handler, render the HTML with the correct client-side asset links and optimal resource hints. We may also need to switch between SSR and SSG mode, or even mix both in the same app.
  - Manage routing, data fetching, and state management stores in a universal manner.

A complete implementation would be quite complex and depends on the build toolchain you have chosen to work with. Therefore, we highly recommend going with a higher-level, opinionated solution that abstracts away the complexity for you. Below we will introduce a few recommended SSR solutions in the Vue ecosystem.

## Nuxt

[Nuxt](#) is a higher-level framework built on top of the Vue ecosystem which provides a streamlined development experience for writing universal Vue applications. Better yet, you can also use it as a static site generator! We highly recommend giving it a try.

## Quasar

[Quasar](#) is a complete Vue-based solution that allows you to target SPA, SSR, PWA, mobile app, desktop app, and browser extension all using one codebase. It not only handles the build setup, but also provides a full collection of Material Design compliant UI components.

## Vite SSR

Vite provides built-in [support for Vue server-side rendering](#), but it is intentionally low-level. If you wish to go directly with Vite, check out [vite-plugin-ssr](#), a community plugin that abstracts away many challenging details for you.

You can also find an example Vue + Vite SSR project using manual setup [here](#), which can serve as a base to build upon. Note this is only recommended if you are experienced with SSR / build tools and really want to have complete control over the higher-level architecture.

## Writing SSR-friendly Code

Hello from the modal!

Regardless of your build setup or higher-level framework choice, there are some principles that apply in all Vue SSR applications.

### Reactivity on the Server

During SSR, each request URL maps to a desired state of our application. There is no user interaction and no DOM updates, so reactivity is unnecessary on the server. By default, reactivity is disabled during SSR for better performance.

### Component Lifecycle Hooks

Since there are no dynamic updates, lifecycle hooks such as `mounted` `onMounted` or `updated` `onUpdated` will **NOT** be called during SSR and will only be executed on the client. The only hooks that are called during SSR are `beforeCreate` and `created`.

You should avoid code that produces side effects that need cleanup in `beforeCreate` and `created` `setup()` or the root scope of `<script setup>`. An example of such side effects is setting up timers with `setInterval`. In client-side only code we may setup a timer and then tear it down in `beforeUnmount` `onBeforeUnmount` or `unmounted` `onUnmounted`. However, because the unmount hooks will never be called during SSR, the timers will stay around forever. To avoid this, move your side-effect code into `mounted` `onMounted` instead.

### Access to Platform-Specific APIs

Universal code cannot assume access to platform-specific APIs, so if your code directly uses browser-only globals like `window` or `document`, they will throw errors when executed in Node.js, and vice-versa.

For tasks that are shared between server and client but with different platform APIs, it's recommended to wrap the platform-specific implementations inside a universal API, or use libraries that do this for you. For example, you can use `node-fetch` to use the same fetch API on both server and client.

For browser-only APIs, the common approach is to lazily access them inside client-only lifecycle hooks such as `mounted` `onMounted`.

Note that if a third-party library is not written with universal usage in mind, it could be tricky to integrate it into a server-rendered app. You *might* be able to get it working by mocking some of the globals, but it would be hacky and may interfere with the environment detection code of other libraries.

## Cross-Request State Pollution

In the State Management chapter, we introduced a [simple state management pattern using Reactivity APIs](#). In an SSR context, this pattern requires some additional adjustments.

The pattern declares shared state in a JavaScript module's root scope. This makes them **singletons** - i.e. there is only one instance of the reactive object throughout the entire lifecycle of our application. This works as expected in a pure client-side Vue application, since the modules in our application are initialized fresh for each browser page visit.

However, in an SSR context, the application modules are typically initialized only once on the server, when the server boots up. The same module instances will be reused across multiple server requests, and so will our singleton state objects. If we mutate the shared singleton state with data specific to one user, it can be accidentally leaked to a request from another user. We call this **cross-request state pollution**.

We can technically re-initialize all the JavaScript modules on each request, just like we do in browsers. However, initializing JavaScript modules can be costly, so this would significantly affect server performance.

The recommended solution is to create a new instance of the entire application - including the router and global stores - on each request. Then, instead of directly importing it in our components, we provide the shared state using [app-level provide](#) and inject it in components that need it:

```
// app.js (shared between server and client)
import { createSSRApp } from 'vue'
import { createStore } from './store.js'

// called on each request
export function createApp() {
  const app = createSSRApp(/* ... */)
  // create new instance of store per request
  const store = createStore(/* ... */)
  // provide store at the app level
  app.provide('store', store)
  // also expose store for hydration purposes
  return { app, store }
}
```

State Management libraries like Pinia are designed with this in mind. Consult [Pinia's SSR guide](#) for more details.

## Hydration Mismatch

If the DOM structure of the pre-rendered HTML does not match the expected output of the client-side app, there will be a hydration mismatch error. Hydration mismatch is most commonly introduced by the following causes:

1. The template contains invalid HTML nesting structure, and the rendered HTML got "corrected" by the browser's native HTML parsing behavior. For example, a common gotcha is that `<div>` cannot be placed inside `<p>`:

```
<p><div>hi</div></p>
```

If we produce this in our server-rendered HTML, the browser will terminate the first `<p>` when `<div>` is encountered and parse it into the following DOM structure:

```
<p></p>
<div>hi</div>
<p></p>
```

2. The data used during render contains randomly generated values. Since the same application will run twice - once on the server, and once on the client - the random values are not guaranteed to be the same between the two runs. There are two ways to avoid random-value-induced mismatches:

1. Use `v-if` + `onMounted` to render the part that depends on random values only on the client. Your framework may also have built-in features to make this easier, for example the `<ClientOnly>` component in VitePress.
2. Use a random number generator library that supports generating with seeds, and guarantee the server run and the client run are using the same seed (e.g. by including the seed in serialized state and retrieving it on the client).
3. The server and the client are in different time zones. Sometimes, we may want to convert a timestamp into the user's local time. However, the timezone during the server run and the timezone during the client run are not always the same, and we may not reliably know the user's timezone during the server run. In such cases, the local time conversion should also be performed as a client-only operation.

When Vue encounters a hydration mismatch, it will attempt to automatically recover and adjust the pre-rendered DOM to match the client-side state. This will lead to some rendering performance loss due to incorrect nodes being discarded and new nodes being mounted, but in most cases, the app should continue to work as expected. That said, it is still best to eliminate hydration mismatches during development.

## Custom Directives

Since most custom directives involve direct DOM manipulation, they are ignored during SSR. However, if you want to specify how a custom directive should be rendered (i.e. what attributes it should add to the rendered element), you can use the `getSSRProps` directive hook:

```
const myDirective = {
  mounted(el, binding) {
    // client-side implementation:
    // directly update the DOM
    el.id = binding.value
  },
  getSSRProps(binding) {
    // server-side implementation:
    // return the props to be rendered.
    // getSSRProps only receives the directive binding.
  }
}
```

```

    return {
      id: binding.value
    }
}

```

## Teleports

Teleports require special handling during SSR. If the rendered app contains Teleports, the teleported content will not be part of the rendered string. An easier solution is to conditionally render the Teleport on mount.

If you do need to hydrate teleport content, the node exposed under the `teleports` property of the ssr context object:

```

const ctx = []
const html = await renderToString(app, ctx)

console.log(ctx.teleports) // { '#teleported': 'teleported content' }

```

You need to inject the teleport markup into the correct location in your final page HTML similar to how you need to inject the main app markup.

:::tip Avoid targeting `body` when using Teleports and SSR together - usually, `<body>` will contain other server-rendered content which makes it impossible for Teleports to determine the correct starting location for hydration.

Instead, prefer a dedicated container, e.g. `<div id="teleported"></div>` which contains only teleported content. :::

# State Management

## What is State Management?

Technically, every Vue component instance already "manages" its own reactive state. Take a simple counter component as an example:

```
<script setup>
import { ref } from 'vue'

// state
const count = ref(0)

// actions
function increment() {
  count.value++
}
</script>

<!-- view -->
<template>{{ count }}</template>
```

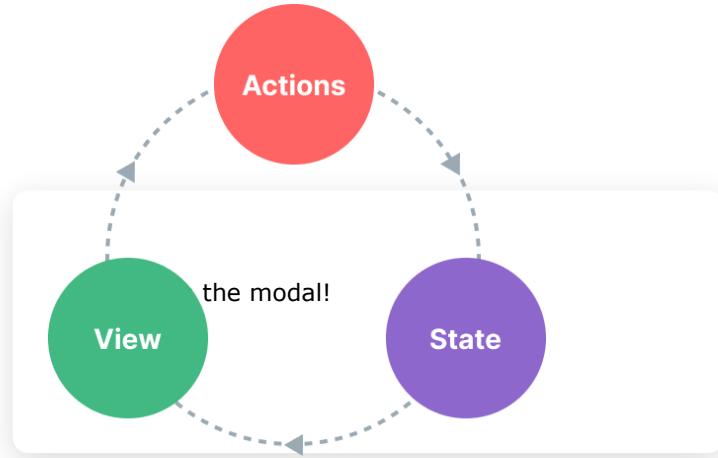
```
<script>
export default {
  // state
  data() {
    return {
      count: 0
    }
  },
  // actions
  methods: {
    increment() {
      this.count++
    }
  }
}
</script>

<!-- view -->
<template>{{ count }}</template>
```

It is a self-contained unit with the following parts:

- The **state**, the source of truth that drives our app;
- The **view**, a declarative mapping of the **state**;
- The **actions**, the possible ways the state could change in reaction to user inputs from the **view**.

This is a simple representation of the concept of "one-way data flow":



However, the simplicity starts to break down when we have **multiple components that share a common state**:

1. Multiple views may depend on the same piece of state.
2. Actions from different views may need to mutate the same piece of state.

For case one, a possible workaround is by "lifting" the shared state up to a common ancestor component, and then pass it down as props. However, this quickly gets tedious in component trees with deep hierarchies, leading to another problem known as [Prop Drilling](#).

For case two, we often find ourselves resorting to solutions such as reaching for direct parent / child instances via template refs, or trying to mutate and synchronize multiple copies of the state via emitted events. Both of these patterns are brittle and quickly lead to unmaintainable code.

A simpler and more straightforward solution is to extract the shared state out of the components, and manage it in a global singleton. With this, our component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!

## Simple State Management with Reactivity API

In Options API, reactive data is declared using the `data()` option. Internally, the object returned by `data()` is made reactive via the `reactive()` function, which is also available as a public API.

If you have a piece of state that should be shared by multiple instances, you can use `reactive()` to create a reactive object, and then import it into multiple components:

```
// store.js
import { reactive } from 'vue'

export const store = reactive({
  count: 0
})
```

```
<!-- ComponentA.vue -->
<script setup>
import { store } from './store.js'
</script>

<template>From A: {{ store.count }}</template>
```

```
<!-- ComponentB.vue -->
<script setup>
import { store } from './store.js'
</script>

<template>From B: {{ store.count }}</template>
```

```
<!-- ComponentA.vue -->
<script>
import { store } from './store.js'

export default {
  data() {
    return {
      store
    }
  }
}
</script>

<template>From A: {{ store.count }}</template>
```

```
<!-- ComponentB.vue -->
<script>
import { store } from './store.js'

export default {
  data() {
    return {
      store
    }
  }
}
</script>

<template>From B: {{ store.count }}</template>
```

Now whenever the `store` object is mutated, both `<ComponentA>` and `<ComponentB>` will update their views automatically - we have a single source of truth now.

However, this also means any component importing `store` can mutate it however they want:

```
<template>
  <button @click="store.count++">
    From B: {{ store.count }}
  </button>
</template>
```

While this works in simple cases, global state that can be arbitrarily mutated by any component is not going to be very maintainable in the long run. To ensure the state-mutating logic is centralized like the state itself, it is recommended to define methods on the store with names that express the intention of the actions:

```
// store.js
import { reactive } from 'vue'

export const store = reactive({
  count: 0,
  increment() {
    this.count++
  }
})

<template>
  <button @click="store.increment()">
    From B: {{ store.count }}
  </button>
</template>
```

[Try it in the Playground](#)[Try it in the Playground](#)

:::tip Note the click handler uses `store.increment()` with the parenthesis - this is necessary to call the method with the proper `this` context since it's not a component method. :::

Although here we are using a single reactive object as a store, you can also share reactive state created using other [Reactivity APIs](#) such as `ref()` or `computed()`, or even return global state from a [Composable](#):

```
import { ref } from 'vue'

// global state, created in module scope
const globalCount = ref(1)

export function useCount() {
  // local state, created per-component
  const localCount = ref(1)

  return {
    globalCount,
    localCount
  }
}
```

The fact that Vue's reactivity system is decoupled from the component model makes it extremely flexible.

## SSR Considerations

If you are building an application that leverages [Server-Side Rendering \(SSR\)](#), the above pattern can lead to issues due to the store being a singleton shared across multiple requests. This is discussed in [more details](#) in the SSR guide.

## Pinia

While our hand-rolled state management solution will suffice in simple scenarios, there are many more things to consider in large-scale production applications:

- Stronger conventions for team collaboration
- Integrating with the Vue DevTools, including timeline, in-component inspection, and time-travel debugging
- Hot Module Replacement
- Server-Side Rendering support

[Pinia](#) is a state management library that implements all of the above. It is maintained by the Vue core team, and works with both Vue 2 and Vue 3.

Existing users may be familiar with [Vuex](#), the previous official state management library for Vue. With Pinia serving the same role in the ecosystem, Vuex is now in maintenance mode. It still works, but will no longer receive new features. It is recommended to use Pinia for new applications.

Pinia started out as an exploration of what the next iteration of Vuex could look like, incorporating many ideas from core team discussions for Vuex 5. Eventually, we realized that Pinia already implements most of what we wanted in Vuex 5, and decided to make it the new recommendation instead.

Compared to Vuex, Pinia provides a simpler API with less ceremony, offers Composition-API-style APIs, and most importantly, has solid type inference support when used with TypeScript.

# Testing

## Why Test?

Automated tests help you and your team build complex Vue applications quickly and confidently by preventing regressions and encouraging you to break apart your application into testable functions, modules, classes, and components. As with any application, your new Vue app can break in many ways, and it's important that you can catch these issues and [fix them before releasing](#).

In this guide, we'll cover basic terminology and provide our recommendations on which tools to choose for your Vue 3 application.

There is one Vue-specific section covering composables. See [Testing Composables](#) below for more details.

## When to Test

Start testing early! We recommend you begin writing tests as soon as you can. The longer you wait to add tests to your application, the more dependencies your application will have, and the harder it will be to start.

## Testing Types

When designing your Vue application's testing strategy, you should leverage the following testing types:

- **Unit:** Checks that inputs to a given function, class, or composable are producing the expected output or side effects.
- **Component:** Checks that your component mounts, renders, can be interacted with, and behaves as expected. These tests import more code than unit tests, are more complex, and require more time to execute.
- **End-to-end:** Checks features that span multiple pages and make real network requests against your production-built Vue application. These tests often involve standing up a database or other backend.

Each testing type plays a role in your application's testing strategy and each will protect you against different types of issues.

## Overview

We will briefly discuss what each of these are, how they can be implemented for Vue applications, and provide some general recommendations.

# Unit Testing

Unit tests are written to verify that small, isolated units of code are working as expected. A unit test usually covers a single function, class, composable, or module. Unit tests focus on logical correctness and only concern themselves with a small portion of the application's overall functionality. They may mock large parts of your application's environment (e.g. initial state, complex classes, 3rd party modules, and network requests).

In general, unit tests will catch issues with a function's business logic and logical correctness.

Hello from the modal!

Take for example this `increment` function:

```
// helpers.js
export function increment (current, max = 10) {
  if (current < max) {
    return current + 1
  }
  return current
}
```

Because it's very self-contained, it'll be easy to invoke the `increment` function and assert that it returns what it's supposed to, so we'll write a Unit Test.

If any of these assertions fail, it's clear that the issue is contained within the `increment` function.

```
// helpers.spec.js
import { increment } from './helpers'

describe('increment', () => {
  test('increments the current number by 1', () => {
    expect(increment(0, 10)).toBe(1)
  })

  test('does not increment the current number over the max', () => {
    expect(increment(10, 10)).toBe(10)
  })

  test('has a default max of 10', () => {
    expect(increment(10)).toBe(10)
  })
})
```

As mentioned previously, unit testing is typically applied to self-contained business logic, components, classes, modules, or functions that do not involve UI rendering, network requests, or other environmental concerns.

These are typically plain JavaScript / TypeScript modules unrelated to Vue. In general, writing unit tests for business logic in Vue applications does not differ significantly from applications using other frameworks.

There are two instances where you DO unit test Vue-specific features:

1. Composables
2. Components

## Composables

One category of functions specific to Vue applications are [Composables](#), which may require special handling during tests. See [Testing Composables](#) below for more details.

## Unit Testing Components

A component can be tested in two ways:

### 1. Whitebox: Unit Testing

Hello from the modal!

Tests that are "Whitebox tests" are aware of the implementation details and dependencies of a component. They are focused on **isolating** the component under test. These tests will usually involve mocking some, if not all of your component's children, as well as setting up plugin state and dependencies (e.g. Vuex).

### 2. Blackbox: Component Testing

Tests that are "Blackbox tests" are unaware of the implementation details of a component. These tests mock as little as possible to test the integration of your component and the entire system. They usually render all child components and are considered more of an "integration test". See the [Component Testing recommendations](#) below.

## Recommendation

- [Vitest](#)

Since the official setup created by `create-vue` is based on [Vite](#), we recommend using a unit testing framework that can leverage the same configuration and transform pipeline directly from Vite. [Vitest](#) is a unit testing framework designed specifically for this purpose, created and maintained by Vue / Vite team members. It integrates with Vite-based projects with minimal effort, and is blazing fast.

## Other Options

- [Peeky](#) is another fast unit test runner with first-class Vite integration. It is also created by a Vue core team member and offers a GUI-based testing interface.
- [Jest](#) is a popular unit testing framework, and can be made to work with Vite via the [vite-jest](#) package. However, we only recommend Jest if you have an existing Jest test suite that needs to be migrated over to a Vite-based project, as Vitest offers a more seamless integration and better performance.

## Component Testing

In Vue applications, components are the main building blocks of the UI. Components are therefore the natural unit of isolation when it comes to validating your application's behavior. From a granularity perspective, component testing sits somewhere above unit testing and can be considered a form of integration testing. Much of your Vue Application should be covered by a component test and we recommend that each Vue component has its own spec file.

Component tests should catch issues relating to your component's props, events, slots that it provides, styles, classes, lifecycle hooks, and more.

Component tests should not mock child components, but instead test the interactions between your component and its children by interacting with the components as a user would. For example, a component test should click on an element like a user would instead of programmatically interacting with the component.

Component tests should focus on the component's public interfaces rather than internal implementation details. For most components, the public interface is limited to: events emitted, props, and slots. When testing, remember to **test what a component does, not how it does it.**

Hello from the modal!

## DO

- For **Visual** logic: assert correct render output based on inputted props and slots.
- For **Behavioral** logic: assert correct render updates or emitted events in response to user input events.

In the below example, we demonstrate a Stepper component that has a DOM element labeled "increment" and can be clicked. We pass a prop called `max` that prevents the Stepper from being incremented past `2`, so if we click the button 3 times, the UI should still say `2`.

We know nothing about the implementation of Stepper, only that the "input" is the `max` prop and the "output" is the state of the DOM as the user will see it.

```
const { getByText } = render(Stepper, {
  props: {
    max: 1
  }
}

getByText('0') // Implicit assertion that "0" is within the component

const button = getByText('increment')

// Dispatch a click event to our increment button.
await fireEvent.click(button)

getByText('1')

await fireEvent.click(button)
```

```
const valueSelector = '[data-testid=stepper-value]'
const buttonSelector = '[data-testid=increment]'

const wrapper = mount(Stepper, {
  props: {
    max: 1
  }
}

expect(wrapper.find(valueSelector).text()).toContain('0')
await wrapper.find(buttonSelector).trigger('click')
expect(wrapper.find(valueSelector).text()).toContain('1')
```

```

const valueSelector = '[data-testid=stepper-value]'
const buttonSelector = '[data-testid=increment]'

mount(Stepper, {
  props: {
    max: 1
  }
})

cy.get(valueSelector).should('be.visible').and('contain.text', '0')
  .get(buttonSelector).click()
  .get(valueSelector).should('contain.text', '1')

```

Hello from the modal!

- **DON'T**

Don't assert the private state of a component instance or test the private methods of a component. Testing implementation details makes the tests brittle, as they are more likely to break and require updates when the implementation changes.

The component's ultimate job is rendering the correct DOM output, so tests focusing on the DOM output provide the same level of correctness assurance (if not more) while being more robust and resilient to change.

Don't rely exclusively on snapshot tests. Asserting HTML strings does not describe correctness. Write tests with intentionality.

If a method needs to be tested thoroughly, consider extracting it into a standalone utility function and write a dedicated unit test for it. If it cannot be extracted cleanly, it may be tested as a part of a component, integration, or end-to-end test that covers it.

## Recommendation

- [Vitest](#) for components or composables that render headlessly (e.g. the `useFavicon` function in [VueUse](#)). Components and DOM can be tested using [@testing-library/vue](#).
- [Cypress Component Testing](#) for components whose expected behavior depends on properly rendering styles or triggering native DOM events. Can be used with Testing Library via [@testing-library/cypress](#).

The main differences between Vitest and browser-based runners are speed and execution context. In short, browser-based runners, like Cypress, can catch issues that node-based runners, like Vitest, cannot (e.g. style issues, real native DOM events, cookies, local storage, and network failures), but browser-based runners are *orders of magnitude slower than Vitest* because they do open a browser, compile your stylesheets, and more. Cypress is a browser-based runner that supports component testing. Please read [Vitest's comparison page](#) for the latest information comparing Vitest and Cypress.

## Mounting Libraries

Component testing often involves mounting the component being tested in isolation, triggering simulated user input events, and asserting on the rendered DOM output. There are dedicated utility libraries that make these tasks simpler.

- `@testing-library/vue` is a Vue testing library focused on testing components without relying on implementation details. Built with accessibility in mind, its approach also makes refactoring a breeze. Its guiding principle is that the more tests resemble the way software is used, the more confidence they can provide.
- `@vue/test-utils` is the official low-level component testing library that was written to provide users access to Vue specific APIs. It's also the lower-level library `@testing-library/vue` is built on top of.

We recommend using `@testing-library/vue` for testing components in applications, as its focus aligns better with the testing priorities of applications. Use `@vue/test-utils` only if you are building advanced components that require testing Vue-specific internals.

## Other Options

- [Nightwatch](#) is an E2E test runner with Vue Component Testing support. ([Example Project](#) in Nightwatch v2)

## E2E Testing

While unit tests provide developers with some degree of confidence, unit and component tests are limited in their abilities to provide holistic coverage of an application when deployed to production. As a result, end-to-end (E2E) tests provide coverage on what is arguably the most important aspect of an application: what happens when users actually use your applications.

End-to-end tests focus on multi-page application behavior that makes network requests against your production-built Vue application. They often involve standing up a database or other backend and may even be run against a live staging environment.

End-to-end tests will often catch issues with your router, state management library, top-level components (e.g. an App or Layout), public assets, or any request handling. As stated above, they catch critical issues that may be impossible to catch with unit tests or component tests.

End-to-end tests do not import any of your Vue application's code, but instead rely completely on testing your application by navigating through entire pages in a real browser.

End-to-end tests validate many of the layers in your application. They can either target your locally built application, or even a live Staging environment. Testing against your Staging environment not only includes your frontend code and static server, but all associated backend services and infrastructure.

The more your tests resemble the way your software is used, the more confidence they can give you.  
- [Kent C. Dodds](#) - Author of the Testing Library

By testing how user actions impact your application, E2E tests are often the key to higher confidence in whether an application is functioning properly or not.

## Choosing an E2E Testing Solution

While end-to-end (E2E) testing on the web has gained a negative reputation for unreliable (flaky) tests and slowing down development processes, modern E2E tools have made strides forward to create more reliable, interactive, and useful tests. When choosing an E2E testing framework, the following sections provide some guidance on things to keep in mind when choosing a testing framework for your application.

### Cross-browser testing

One of the primary benefits that end-to-end (E2E) testing is known for is its ability to test your application across multiple browsers. While it ~~Hello from the modal!~~ may seem desirable to have 100% cross-browser coverage, it is important to note that cross browser testing has diminishing returns on a team's resources due the additional time and machine power required to run them consistently. As a result, it is important to be mindful of this trade-off when choosing the amount of cross-browser testing your application needs.

### Faster feedback loops

One of the primary problems with end-to-end (E2E) tests and development is that running the entire suite takes a long time. Typically, this is only done in continuous integration and deployment (CI/CD) pipelines. Modern E2E testing frameworks have helped to solve this by adding features like parallelization, which allows for CI/CD pipelines to often run magnitudes faster than before. In addition, when developing locally, the ability to selectively run a single test for the page you are working on while also providing hot reloading of tests can help to boost a developer's workflow and productivity.

### First-class debugging experience

While developers have traditionally relied on scanning logs in a terminal window to help determine what went wrong in a test, modern end-to-end (E2E) test frameworks allow developers to leverage tools that they are already familiar with, e.g. browser developer tools.

### Visibility in headless mode

When end-to-end (E2E) tests are run in continuous integration / deployment pipelines, they are often run in headless browsers (i.e., no visible browser is opened for the user to watch). A critical feature of modern E2E testing frameworks is the ability to see snapshots and/or videos of the application during testing, providing some insight into why errors are happening. Historically, it was tedious to maintain these integrations.

## Recommendation

- [Cypress](#)

Overall, we believe Cypress provides the most complete E2E solution with features like an informative graphical interface, excellent debuggability, built-in assertions and stubs, flake-resistance, parallelization, and snapshots. As mentioned above, it also provides support for [Component Testing](#). However, it only supports Chromium-based browsers and Firefox.

## Other Options

- [Playwright](#) is also a great E2E testing solution with a wider range of browser support (mainly WebKit). See [Why Playwright](#) for more details.
- [Nightwatch v2](#) is an E2E testing solution based on [Selenium WebDriver](#). This gives it the widest browser support range.

## Recipes

### Adding Vitest to a Project

In a Vite-based Vue project, run:

```
> npm install -D vitest happy-dom @testing-library/vue
```

Next, update the Vite configuration to add the `test` option block:

```
// vite.config.js
import { defineConfig } from 'vite'

export default defineConfig({
  // ...
  test: {
    // enable jest-like global test APIs
    globals: true,
    // simulate DOM with happy-dom
    // (requires installing happy-dom as a peer dependency)
    environment: 'happy-dom'
  }
})
```

:::tip If you are using TypeScript, add `vitest/globals` to the `types` field in your `tsconfig.json`.

```
// tsconfig.json
{
  "compilerOptions": {
    "types": ["vitest/globals"]
  }
}
```

...

Then create a file ending in `*.test.js` in your project. You can place all test files in a test directory in project root, or in test directories next to your source files. Vitest will automatically search for them using the naming convention.

```
// MyComponent.test.js
import { render } from '@testing-library/vue'
import MyComponent from './MyComponent.vue'

test('it should work', () => {
  const { getByText } = render(MyComponent, {
```

```

    props: {
      /* ... */
    }
  })

// assert output
getByText('...')
})

```

Finally, update `package.json` to add the test script and run it:

```
{
  // ...
  "scripts": {
    "test": "vitest"
  }
}
```

```
> npm test
```

## Testing Composables

This section assumes you have read the [Composables](#) section.

When it comes to testing composable, we can divide them into two categories: composable that do not rely on a host component instance, and composable that do.

A composable depends on a host component instance when it uses the following APIs:

- Lifecycle hooks
- Provide / Inject

If a composable only uses Reactivity APIs, then it can be tested by directly invoking it and asserting its returned state / methods:

```
// counter.js
import { ref } from 'vue'

export function useCounter() {
  const count = ref(0)
  const increment = () => count.value++

  return {
    count,
    increment
  }
}
```

```
// counter.test.js
import { useCounter } from './counter.js'

test('useCounter', () => {
  const { count, increment } = useCounter()
  expect(count.value).toBe(0)
```

```

    increment()
    expect(count.value).toBe(1)
})

```

A composable that relies on lifecycle hooks or Provide / Inject needs to be wrapped in a host component to be tested. We can create a helper like the following:

```

// test-utils.js
import { createApp } from 'vue'

export function withSetup(composable) {
  let result
  const app = createApp({
    setup() {
      result = composable()
      // suppress missing template warning
      return () => {}
    }
  })
  app.mount(document.createElement('div'))
  // return the result and the app instance
  // for testing provide / unmount
  return [result, app]
}

```

```

import { withSetup } from './test-utils'
import { useFoo } from './foo'

test('useFoo', () => {
  const [result, app] = withSetup(() => useFoo(123))
  // mock provide for testing injections
  app.provide(...)
  // run assertions
  expect(result.foo.value).toBe(1)
  // trigger onUnmounted hook if needed
  app.unmount()
})

```

For more complex composables, it could also be easier to test it by writing tests against the wrapper component using [Component Testing](#) techniques.

# Tooling

## Try It Online

You don't need to install anything on your machine to try out Vue SFCs - there are online playgrounds that allow you to do so right in the browser:

- [Vue SFC Playground](#) Hello from the modal!
  - Always deployed from latest commit
  - Designed for inspecting component compilation results
- [Vue + Vite on StackBlitz](#)
  - IDE-like environment running actual Vite dev server in the browser
  - Closest to local setup

It is also recommended to use these online playgrounds to provide reproductions when reporting bugs.

## Project Scaffolding

### Vite

[Vite](#) is a lightweight and fast build tool with first-class Vue SFC support. It is created by Evan You, who is also the author of Vue!

To get started with Vite + Vue, simply run:

```
$ npm init vue@latest
```

This command will install and execute [create-vue](#), the official Vue project scaffolding tool.

- To learn more about Vite, check out the [Vite docs](#).
- To configure Vue-specific behavior in a Vite project, for example passing options to the Vue compiler, check out the docs for [@vitejs/plugin-vue](#).

Both online playgrounds mentioned above also support downloading files as a Vite project.

### Vue CLI

[Vue CLI](#) is the official webpack-based toolchain for Vue. It is now in maintenance mode and we recommend starting new projects with Vite unless you rely on specific webpack-only features. Vite will provide superior developer experience in most cases.

For information on migrating from Vue CLI to Vite:

- [Vue CLI -> Vite Migration Guide from VueSchool.io](#)
- [Tools / Plugins that help with auto migration](#)

## Note on In-Browser Template Compilation

When using Vue without a build step, component templates are written either directly in the page's HTML or as inlined JavaScript strings. In such cases, Vue needs to ship the template compiler to the browser in order to perform on-the-fly template compilation. On the other hand, the compiler would be unnecessary if we pre-compile the templates with a build step. To reduce client bundle size, Vue provides [different "builds"](#) optimized for different use cases.

- Build files that start with `vue.runtime.*` are **runtime-only builds**: they do not include the compiler. When using these builds, all templates must be pre-compiled via a build step.  
Hello from the modal!
- Build files that do not include `.runtime` are **full builds**: they include the compiler and support compiling templates directly in the browser. However, they will increase the payload by ~14kb.

Our default tooling setups use the runtime-only build since all templates in SFCs are pre-compiled. If, for some reason, you need in-browser template compilation even with a build step, you can do so by configuring the build tool to alias `vue` to `vue/dist/vue.esm-bundler.js` instead.

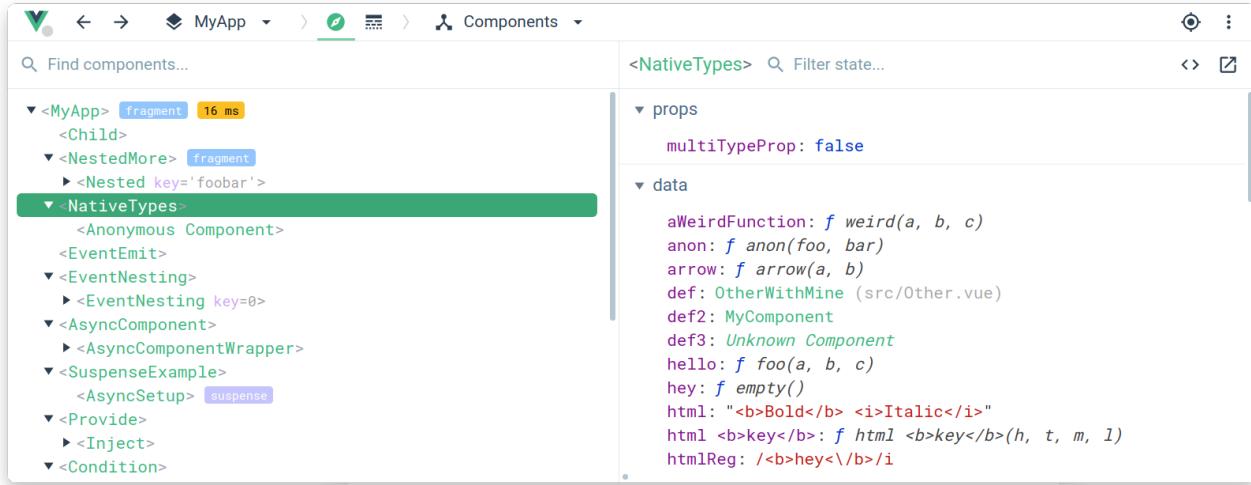
If you are looking for a lighter-weight alternative for no-build-step usage, check out [petite-vue](#).

## IDE Support

- The recommended IDE setup is [VSCode](#) + the [Volar](#) extension. Volar provides syntax highlighting, TypeScript support, and intellisense for template expressions and component props.  
:::tip Volar replaces [Vetur](#), our previous official VSCode extension for Vue 2. If you have Vetur currently installed, make sure to disable it in Vue 3 projects. :::
- [WebStorm](#) also provides great built-in support for Vue SFCs.
- Other IDEs that support the [Language Service Protocol](#) (LSP) can also leverage Volar's core functionalities via LSP:
  - Sublime Text support via [LSP-Volar](#).
  - vim / Neovim support via [coc-volar](#).
  - emacs support via [lsp-mode](#)

## Browser Devtools

The Vue browser devtools extension allows you to explore a Vue app's component tree, inspect the state of individual components, track state management events, and profile performance.



devtools screenshot

- [Documentation](#)
- [Chrome Extension](#)
- [Firefox Addon](#)
- [Standalone Electron app](#)

## TypeScript

Main article: [Using Vue with TypeScript](#).

- [Volar](#) provides type checking for SFCs using `<script lang="ts">` blocks, including template expressions and cross-component props validation.
- Use `vue-tsc` for performing the same type checking from the command line, or for generating `d.ts` files for SFCs.

## Testing

Main article: [Testing Guide](#).

- [Cypress](#) is recommended for E2E tests. It can also be used for component testing for Vue SFCs via the [Cypress Component Test Runner](#).
- [Vitest](#) is a test runner created by Vue / Vite team members that focuses on speed. It is specifically designed for Vite-based applications to provide the same instant feedback loop for unit / component testing.
- [Jest](#) can be made to work with Vite via [vite-jest](#). However, this is only recommended if you have existing Jest-based test suites that you need to migrate over to a Vite-based setup, as Vitest provides similar functionalities with a much more efficient integration.

# Linting

The Vue team maintains [eslint-plugin-vue](#), an [ESLint](#) plugin that supports SFC-specific linting rules.

Users previously using Vue CLI may be used to having linters configured via webpack loaders. However when using a Vite-based build setup, our general recommendation is:

1. `npm install -D eslint eslint-plugin-vue`, then follow [eslint-plugin-vue's configuration guide](#).
2. Setup ESLint IDE extensions, for example [ESLint for VSCode](#), so you get linter feedback right in your editor during development. This also avoids unnecessary linting cost when starting the dev server.
3. Run ESLint as part of the production build command, so you get full linter feedback before shipping to production.
4. (Optional) Setup tools like [lint-staged](#) to automatically lint modified files on git commit.

# Formatting

- The [Volar](#) VSCode extension provides formatting for Vue SFCs out of the box.
- Alternatively, [Prettier](#) provides built-in Vue SFC formatting support.

# SFC Custom Block Integrations

Custom blocks are compiled into imports to the same Vue file with different request queries. It is up to the underlying build tool to handle these import requests.

- If using Vite, a custom Vite plugin should be used to transform matched custom blocks into executable JavaScript. [Example](#)
- If using Vue CLI or plain webpack, a webpack loader should be configured to transform the matched blocks. [Example](#)

# Lower-Level Packages

## `@vue/compiler-sfc` {#vue-compiler-sfc}

- [Docs](#)

This package is part of the Vue core monorepo and is always published with the same version as the main `vue` package. It is included as a dependency of the main `vue` package and proxied under `vue/compiler-sfc` so you don't need to install it individually.

The package itself provides lower-level utilities for processing Vue SFCs and is only meant for tooling authors that need to support Vue SFCs in custom tools.

:::tip Always prefer using this package via the `vue/compiler-sfc` deep import since this ensures its version is in sync with the Vue runtime. :::

## @vitejs/plugin-vue `{#vitejs-plugin-vue}`

- [Docs](#)

Official plugin that provides Vue SFC support in Vite.

## vue-loader `{#vue-loader}`

- [Docs](#)

The official loader that provides Vue SFC support in webpack. If you are using Vue CLI, also see [docs on modifying vue-loader options in Vue CLI](#).

## Other Online Playgrounds

- [VueUse Playground](#)
- [Vue + Vite on Repl.it](#)
- [Vue on CodeSandbox](#)
- [Vue on Codepen](#)
- [Vue on Components.studio](#)
- [Vue on WebComponents.dev](#)

---

[Go to TOC](#)

# TypeScript with Composition API

This page assumes you've already read the overview on [Using Vue with TypeScript](#).

## Typing Component Props

Hello from the modal!

### Using `<script setup> {#using-script-setup}`

When using `<script setup>`, the `defineProps()` macro supports inferring the props types based on its argument:

```
<script setup lang="ts">
const props = defineProps<{
  foo: { type: String, required: true },
  bar: Number
})

props.foo // string
props.bar // number | undefined
</script>
```

This is called "runtime declaration", because the argument passed to `defineProps()` will be used as the runtime `props` option.

However, it is usually more straightforward to define props with pure types via a generic type argument:

```
<script setup lang="ts">
const props = defineProps<{
  foo: string
  bar?: number
}>()
</script>
```

This is called "type-based declaration". The compiler will try to do its best to infer the equivalent runtime options based on the type argument. In this case, our second example compiles into the exact same runtime options as the first example.

You can use either type-based declaration OR runtime declaration, but you cannot use both at the same time.

We can also move the props types into a separate interface:

```
<script setup lang="ts">
interface Props {
  foo: string
  bar?: number
}
```

```
const props = defineProps<Props>()
</script>
```

## Syntax Limitations

In order to generate the correct runtime code, the generic argument for `defineProps()` must be one of the following:

- An object literal type:

```
defineProps<{ /* ... */ }>()
```

- A reference to an interface or object literal type **in the same file**:

```
interface Props {/* ... */}

defineProps<Props>()
```

The interface or object literal type can contain references to types imported from other files, however, the generic argument itself passed to `defineProps` **cannot** be an imported type:

```
import { Props } from './other-file'

// NOT supported
defineProps<Props>()
```

This is because Vue components are compiled in isolation and the compiler currently does not crawl imported files in order to analyze the source type. This limitation could be removed in a future release.

## Props Default Values

When using type-based declaration, we lose the ability to declare default values for the props. This can be resolved by the `withDefaults` compiler macro:

```
export interface Props {
  msg?: string
  labels?: string[]
}

const props = withDefaults(defineProps<Props>(), {
  msg: 'hello',
  labels: () => ['one', 'two']
})
```

This will be compiled to equivalent runtime props `default` options. In addition, the `withDefaults` helper provides type checks for the default values, and ensures the returned `props` type has the optional flags removed for properties that do have default values declared.

Alternatively, you can use the currently experimental [Reactivity Transform](#):

```
<script setup lang="ts">
interface Props {
  name: string
```

```

    count?: number
}

// reactive destructure for defineProps()
// default value is compiled to equivalent runtime option
const { name, count = 100 } = defineProps<Props>()
</script>

```

This behavior currently requires [explicit opt-in](#).

## Without `<script setup>` [\[#without-script-setup\]](#)

Hello from the modal!  
 If not using `<script setup>`, it is necessary to use `defineComponent()` to enable props type inference.  
 The type of the props object passed to `setup()` is inferred from the `props` option.

```

import { defineComponent } from 'vue'

export default defineComponent({
  props: {
    message: String
  },
  setup(props) {
    props.message // <-- type: string
  }
})

```

## Typing Component Emits

In `<script setup>`, the `emit` function can also be typed using either runtime declaration OR type declaration:

```

<script setup lang="ts">
// runtime
const emit = defineEmits(['change', 'update'])

// type-based
const emit = defineEmits<{
  (e: 'change', id: number): void
  (e: 'update', value: string): void
}>()
</script>

```

The type argument should be a type literal with [Call Signatures](#). The type literal will be used as the type of the returned `emit` function. As we can see, the type declaration gives us much finer-grained control over the type constraints of emitted events.

When not using `<script setup>`, `defineComponent()` is able to infer the allowed events for the `emit` function exposed on the setup context:

```

import { defineComponent } from 'vue'

export default defineComponent({
  emits: ['change'],
  setup(props, { emit }) {

```

```
    emit('change') // <-- type check / auto-completion
  }
}
```

## Typing `ref()` {#typing-ref}

Refs infer the type from the initial value:

```
import { ref } from 'vue'

// inferred type: Ref<number>
const year = ref(2020)

// => TS Error: Type 'string' is not assignable to type 'number'.
year.value = '2020'
```

Sometimes we may need to specify complex types for a ref's inner value. We can do that by using the `Ref` type:

```
import { ref } from 'vue'
import type { Ref } from 'vue'

const year: Ref<string | number> = ref('2020')

year.value = 2020 // ok!
```

Or, by passing a generic argument when calling `ref()` to override the default inference:

```
// resulting type: Ref<string | number>
const year = ref<string | number>('2020')

year.value = 2020 // ok!
```

If you specify a generic type argument but omit the initial value, the resulting type will be a union type that includes `undefined`:

```
// inferred type: Ref<number | undefined>
const n = ref<number>()
```

## Typing `reactive()` {#typing-reactive}

`reactive()` also implicitly infers the type from its argument:

```
import { reactive } from 'vue'

// inferred type: { title: string }
const book = reactive({ title: 'Vue 3 Guide' })
```

To explicitly type a `reactive` property, we can use interfaces:

```
import { reactive } from 'vue'

interface Book {
  title: string
```

```

    year?: number
}

const book: Book = reactive({ title: 'Vue 3 Guide' })

```

:::tip It's not recommended to use the generic argument of `reactive()` because the returned type, which handles nested ref unwrapping, is different from the generic argument type. :::

## Typing `computed()` {#typing-computed}

`computed()` infers its type based on the return value:

```

import { ref, computed } from 'vue'

const count = ref(0)

// inferred type: ComputedRef<number>
const double = computed(() => count.value * 2)

// => TS Error: Property 'split' does not exist on type 'number'
const result = double.value.split('')

```

You can also specify an explicit type via a generic argument:

```

const double = computed<number>(() => {
  // type error if this doesn't return a number
})

```

## Typing Event Handlers

When dealing with native DOM events, it might be useful to type the argument we pass to the handler correctly. Let's take a look at this example:

```

<script setup lang="ts">
function handleChange(event) {
  // `event` implicitly has `any` type
  console.log(event.target.value)
}
</script>

<template>
  <input type="text" @change="handleChange" />
</template>

```

Without type annotation, the `event` argument will implicitly have a type of `any`. This will also result in a TS error if `"strict": true` or `"noImplicitAny": true` are used in `tsconfig.json`. It is therefore recommended to explicitly annotate the argument of event handlers. In addition, you may need to explicitly cast properties on `event`:

```

function handleChange(event: Event) {
  console.log((event.target as HTMLInputElement).value)
}

```

## Typing Provide / Inject

Provide and inject are usually performed in separate components. To properly type injected values, Vue provides an `InjectionKey` interface, which is a generic type that extends `Symbol`. It can be used to sync the type of the injected value between the provider and the consumer:

```
import { provide, inject } from 'vue'
import type { InjectionKey } from 'vue'

const key = Symbol() as InjectionKey<string>

provide(key, 'foo') // providing non-string value will result in error
const foo = inject(key) // type of foo: string | undefined
```

It's recommended to place the injection key in a separate file so that it can be imported in multiple components.

When using string injection keys, the type of the injected value will be `unknown`, and needs to be explicitly declared via a generic type argument:

```
const foo = inject<string>('foo') // type: string | undefined
```

Notice the injected value can still be `undefined`, because there is no guarantee that a provider will provide this value at runtime.

The `undefined` type can be removed by providing a default value:

```
const foo = inject<string>('foo', 'bar') // type: string
```

If you are sure that the value is always provided, you can also force cast the value:

```
const foo = inject('foo') as string
```

## Typing Template Refs

Template refs should be created with an explicit generic type argument and an initial value of `null`:

```
<script setup lang="ts">
import { ref, onMounted } from 'vue'

const el = ref<HTMLInputElement | null>(null)

onMounted(() => {
  el.value?.focus()
})
</script>

<template>
  <input ref="el" />
</template>
```

Note that for strict type safety, it is necessary to use optional chaining or type guards when accessing `el.value`. This is because the initial ref value is `null` until the component is mounted, and it can also be set to `null` if the referenced element is unmounted by `v-if`.

## Typing Component Template Refs

Sometimes you might need to annotate a template ref for a child component in order to call its public method. For example, we have a `MyModal` child component with a method that opens the modal:

```
<!-- MyModal.vue -->
<script setup lang="ts">
import { ref } from 'vue'

const isContentShown = ref(false)
const open = () => (isContentShown.value = true)

defineExpose({
  open
})
</script>
```

In order to get the instance type of `MyModal`, we need to first get its type via `typeof`, then use TypeScript's built-in `InstanceType` utility to extract its instance type:

```
<!-- App.vue -->
<script setup lang="ts">
import MyModal from './MyModal.vue'

const modal = ref<InstanceType<typeof MyModal> | null>(null)

const openModal = () => {
  modal.value?.open()
}
</script>
```

Note if you want to use this technique in TypeScript files instead of Vue SFCs, you need to enable Volar's [Takeover Mode](#).

# TypeScript with Options API

This page assumes you've already read the overview on [Using Vue with TypeScript](#).

:::tip While Vue does support TypeScript usage with Options API, it is recommended to use Vue with TypeScript via Composition API as it offers simpler, more efficient and more robust type inference. :::  
Hello from the modal!

## Typing Component Props

Type inference for props in Options API requires wrapping the component with `defineComponent()`. With it, Vue is able to infer the types for the props based on the `props` option, taking additional options such as `required: true` and `default` into account:

```
import { defineComponent } from 'vue'

export default defineComponent({
  // type inference enabled
  props: {
    name: String,
    id: [Number, String],
    msg: { type: String, required: true },
    metadata: null
  },
  mounted() {
    this.name // type: string | undefined
    this.id // type: number | string | undefined
    this.msg // type: string
    this.metadata // type: any
  }
})
```

However, the runtime `props` options only support using constructor functions as a prop's type - there is no way to specify complex types such as objects with nested properties or function call signatures.

To annotate complex props types, we can use the `PropType` utility type:

```
import { defineComponent } from 'vue'
import type { PropType } from 'vue'

interface Book {
  title: string
  author: string
  year: number
}

export default defineComponent({
  props: {
    book: {
      // provide more specific type to `Object`
      type: Object as PropType<Book>,
      required: true
    }
  }
})
```

```

    },
    // can also annotate functions
    callback: Function as PropType<(id: number) => void>
},
mounted() {
  this.book.title // string
  this.book.year // number

  // TS Error: argument of type 'string' is not
  // assignable to parameter of type 'number'
  this.callback?.('123')
}
})

```

## Caveats

If your TypeScript version is less than 4.7, you have to be careful when using function values for `validator` and `default` prop options - make sure to use arrow functions:

```

import { defineComponent } from 'vue'
import type { PropType } from 'vue'

interface Book {
  title: string
  year?: number
}

export default defineComponent({
  props: {
    bookA: {
      type: Object as PropType<Book>,
      // Make sure to use arrow functions if your TypeScript version is less than
      4.7
      default: () => ({
        title: 'Arrow Function Expression'
      }),
      validator: (book: Book) => !!book.title
    }
  }
})

```

This prevents TypeScript from having to infer the type of `this` inside these functions, which, unfortunately, can cause the type inference to fail. It was a previous [design limitation](#), and now has been improved in [TypeScript 4.7](#).

## Typing Component Emits

We can declare the expected payload type for an emitted event using the object syntax of the `emits` option. Also, all non-declared emitted events will throw a type error when called:

```

import { defineComponent } from 'vue'

export default defineComponent({
  emits: {
    addBook(payload: { bookName: string }) {
      // perform runtime validation
      return payload.bookName.length > 0
    }
  }
})

```

```

        }
    },
    methods: {
        onSubmit() {
            this.$emit('addBook', {
                bookName: 123 // Type error!
            })

            this.$emit('non-declared-event') // Type error!
        }
    }
)

```

Hello from the modal!

## Typing Computed Properties

A computed property infers its type based on its return value:

```

import { defineComponent } from 'vue'

export default defineComponent({
    data() {
        return {
            message: 'Hello!'
        }
    },
    computed: {
        greeting() {
            return this.message + '!'
        }
    },
    mounted() {
        this.greeting // type: string
    }
})

```

In some cases, you may want to explicitly annotate the type of a computed property to ensure its implementation is correct:

```

import { defineComponent } from 'vue'

export default defineComponent({
    data() {
        return {
            message: 'Hello!'
        }
    },
    computed: {
        // explicitly annotate return type
        greeting(): string {
            return this.message + '!'
        },
        // annotating a writable computed property
        greetingUppercased: {
            get(): string {
                return this.greeting.toUpperCase()
            },
            set(newValue: string) {
                this.message = newValue.toUpperCase()
            }
        }
    }
})

```

```

        }
    }
})

```

Explicit annotations may also be required in some edge cases where TypeScript fails to infer the type of a computed property due to circular inference loops.

## Typing Event Handlers

When dealing with native DOM events, it might be useful to type the argument we pass to the handler correctly. Let's take a look at this example:

```

<script lang="ts">
import { defineComponent } from 'vue'

export default defineComponent({
  methods: {
    handleChange(event) {
      // `event` implicitly has `any` type
      console.log(event.target.value)
    }
  }
)</script>

<template>
  <input type="text" @change="handleChange" />
</template>

```

Without type annotation, the `event` argument will implicitly have a type of `any`. This will also result in a TS error if `"strict": true` or `"noImplicitAny": true` are used in `tsconfig.json`. It is therefore recommended to explicitly annotate the argument of event handlers. In addition, you may need to explicitly cast properties on `event`:

```

import { defineComponent } from 'vue'

export default defineComponent({
  methods: {
    handleChange(event: Event) {
      console.log((event.target as HTMLInputElement).value)
    }
  }
})

```

## Augmenting Global Properties

Some plugins install globally available properties to all component instances via `app.config.globalProperties`. For example, we may install `this.$http` for data-fetching or `this.$translate` for internationalization. To make this play well with TypeScript, Vue exposes a `ComponentCustomProperties` interface designed to be augmented via [TypeScript module augmentation](#):

```

import axios from 'axios'

declare module 'vue' {
  interface ComponentCustomProperties {
    $http: typeof axios
    $translate: (key: string) => string
  }
}

```

See also:

- [TypeScript unit tests for component type extensions](#)  
Hello from the modal!

## Type Augmentation Placement

We can put this type augmentation in a `.ts` file, or in a project-wide `*.d.ts` file. Either way, make sure it is included in `tsconfig.json`. For library / plugin authors, this file should be specified in the `types` property in `package.json`.

In order to take advantage of module augmentation, you will need to ensure the augmentation is placed in a [TypeScript module](#). That is to say, the file needs to contain at least one top-level `import` or `export`, even if it is just `export {}`. If the augmentation is placed outside of a module, it will overwrite the original types rather than augmenting them!

```

// Does not work, overwrites the original types.
declare module 'vue' {
  interface ComponentCustomProperties {
    $translate: (key: string) => string
  }
}

```

```

// Works correctly
export {}

declare module 'vue' {
  interface ComponentCustomProperties {
    $translate: (key: string) => string
  }
}

```

## Augmenting Custom Options

Some plugins, for example `vue-router`, provide support for custom component options such as `beforeRouteEnter`:

```

import { defineComponent } from 'vue'

export default defineComponent({
  beforeRouteEnter(to, from, next) {
    // ...
  }
})

```

Without proper type augmentation, the arguments of this hook will implicitly have `any` type. We can augment the `ComponentCustomOptions` interface to support these custom options:

```
import { Route } from 'vue-router'

declare module 'vue' {
  interface ComponentCustomOptions {
    beforeRouteEnter?(to: Route, from: Route, next: () => void): void
  }
}
```

Now the `beforeRouteEnter` option has been properly typed. Note this is just an example - well-typed libraries like `vue-router` should automatically perform these augmentations in their own type definitions.

The placement of this augmentation is subject the [same restrictions](#) as global property augmentations.

See also:

- [TypeScript unit tests for component type extensions](#)

# Using Vue with TypeScript

A type system like TypeScript can detect many common errors via static analysis at build time. This reduces the chance of runtime errors in production, and also allows us to more confidently refactor code in large-scale applications. TypeScript also improves developer ergonomics via type-based auto-completion in IDEs.

Vue is written in TypeScript itself and provides first-class TypeScript support. All official Vue packages come with bundled type declarations that should work out-of-the-box.

Hello from the modal!

## Project Setup

`create-vue`, the official project scaffolding tool, offers the options to scaffold a Vite-powered, TypeScript-ready Vue project.

## Overview

With a Vite-based setup, the dev server and the bundler are transpilation-only and do not perform any type-checking. This ensures the Vite dev server stays blazing fast even when using TypeScript.

- During development, we recommend relying on a good [IDE setup](#) for instant feedback on type errors.
- If using SFCs, use the `vue-tsc` utility for command line type checking and type declaration generation. `vue-tsc` is a wrapper around `tsc`, TypeScript's own command line interface. It works largely the same as `tsc` except that it supports Vue SFCs in addition to TypeScript files. You can run `vue-tsc` in watch mode in parallel to the Vite dev server, or use a Vite plugin like [vite-plugin-checker](#) which runs the checks in a separate worker thread.
- Vue CLI also provides TypeScript support, but is no longer recommended. See [notes below](#).

## IDE Support

- [Visual Studio Code](#) (VSCode) is strongly recommended for its great out-of-the-box support for TypeScript.
  - [Volar](#) is the official VSCode extension that provides TypeScript support inside Vue SFCs, along with many other great features.

::::tip Volar replaces [Vetur](#), our previous official VSCode extension for Vue 2. If you have Vetur currently installed, make sure to disable it in Vue 3 projects. ::::

    - [TypeScript Vue Plugin](#) is also needed to get type support for `*.vue` imports in TS files.
- [WebStorm](#) also provides out-of-the-box support for both TypeScript and Vue. Other JetBrains IDEs support them too, either out of the box or via [a free plugin](#).

## Configuring `tsconfig.json` [\[#configuring-tsconfig-json\]](#)

Projects scaffolded via `create-vue` include pre-configured `tsconfig.json`. The base config is abstracted in the `@vue/tsconfig` package. Inside the project, we use [Project References](#) to ensure correct types for code running in different environments (e.g. app code and test code should have different global variables).

When configuring `tsconfig.json` manually, some notable options include:

- `compilerOptions.isolatedModules` is set to `true` because Vite uses [esbuild](#) for transpiling TypeScript and is subject to single-file transpile limitations.  
Hello from the modal!
- If you're using Options API, you need to set `compilerOptions.strict` to `true` (or at least enable `compilerOptions.noImplicitThis`, which is a part of the `strict` flag) to leverage type checking of `this` in component options. Otherwise `this` will be treated as `any`.
- If you have configured resolver aliases in your build tool, for example the `@/*` alias configured by default in a `create-vue` project, you need to also configure it for TypeScript via `compilerOptions.paths`.

See also:

- [Official TypeScript compiler options docs](#)
- [esbuild TypeScript compilation caveats](#)

## Volar Takeover Mode

This section only applies for VSCode + Volar.

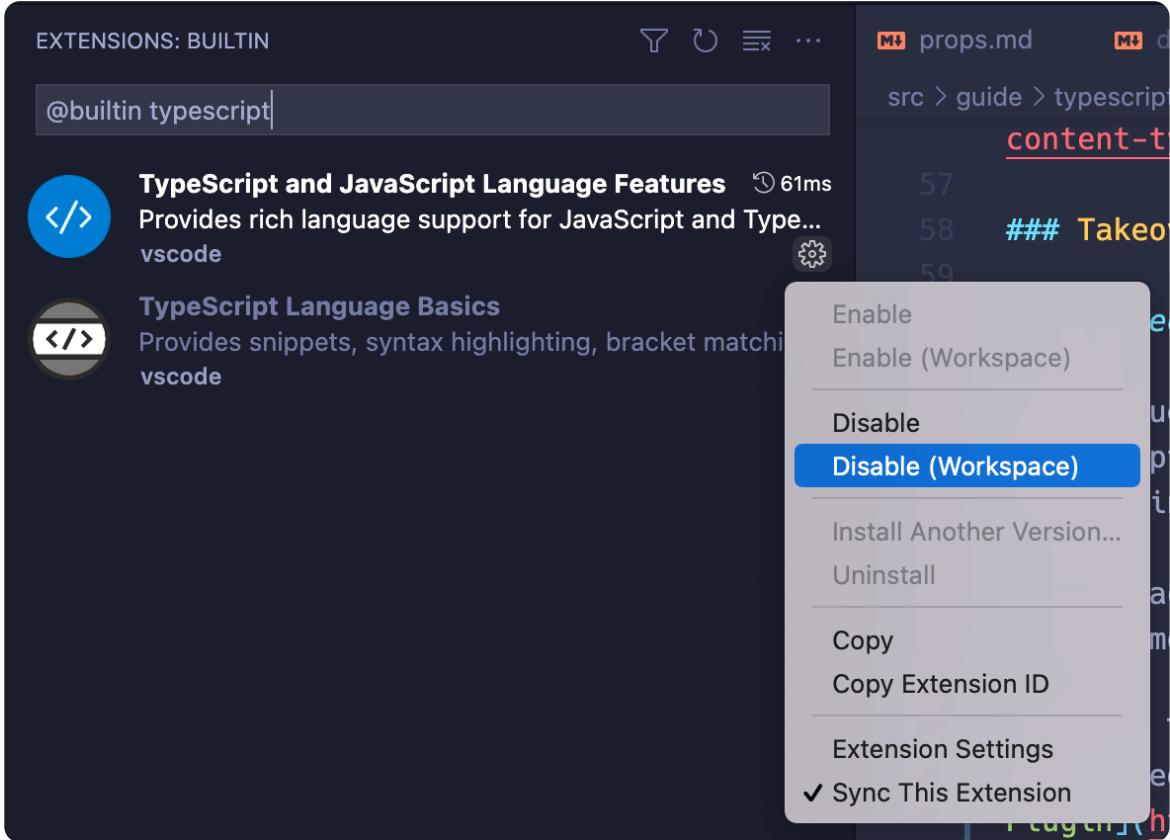
To get Vue SFCs and TypeScript working together, Volar creates a separate TS language service instance patched with Vue-specific support, and uses it in Vue SFCs. At the same time, plain TS files are still handled by VSCode's built-in TS language service, which is why we need [TypeScript Vue Plugin](#) to support Vue SFC imports in TS files. This default setup works, but for each project we are running two TS language service instances: one from Volar, one from VSCode's built-in service. This is a bit inefficient and can lead to performance issues in large projects.

Volar provides a feature called "Takeover Mode" to improve performance. In takeover mode, Volar provides support for both Vue and TS files using a single TS language service instance.

To enable Takeover Mode, you need to disable VSCode's built-in TS language service in **your project's workspace only** by following these steps:

1. In your project workspace, bring up the command palette with `Ctrl + Shift + P` (macOS: `Cmd + Shift + P`).
2. Type `built` and select "Extensions: Show Built-in Extensions".
3. Type `typescript` in the extension search box (do not remove `@builtin` prefix).

4. Click the little gear icon of "TypeScript and JavaScript Language Features", and select "Disable (Workspace)".
5. Reload the workspace. Takeover mode will be enabled when you open a Vue or TS file.



## Note on Vue CLI and `ts-loader` {#note-on-vue-cli-and-ts-loader}

In webpack-based setups such as Vue CLI, it is common to perform type checking as part of the module transform pipeline, for example with `ts-loader`. This, however, isn't a clean solution because the type system needs knowledge of the entire module graph to perform type checks. Individual module's transform step simply is not the right place for the task. It leads to the following problems:

- `ts-loader` can only type check post-transform code. This doesn't align with the errors we see in IDEs or from `vue-tsc`, which map directly back to the source code.
- Type checking can be slow. When it is performed in the same thread / process with code transformations, it significantly affects the build speed of the entire application.
- We already have type checking running right in our IDE in a separate process, so the cost of dev experience slow down simply isn't a good trade-off.

If you are currently using Vue 3 + TypeScript via Vue CLI, we strongly recommend migrating over to Vite. We are also working on CLI options to enable transpile-only TS support, so that you can switch to `vue-tsc` for type checking.

## General Usage Notes

### `defineComponent()` **{#definecomponent}**

To let TypeScript properly infer types inside component options, we need to define components with `defineComponent()`:

```
import { defineComponent } from 'vue'

export default defineComponent({
  // type inference enabled
  props: {
    name: String,
    msg: { type: String, required: true }
  },
  data() {
    return {
      count: 1
    }
  },
  mounted() {
    this.name // type: string | undefined
    this.msg // type: string
    this.count // type: number
  }
})
```

`defineComponent()` also supports inferring the props passed to `setup()` when using Composition API without `<script setup>`:

```
import { defineComponent } from 'vue'

export default defineComponent({
  // type inference enabled
  props: {
    message: String
  },
  setup(props) {
    props.message // type: string | undefined
  }
})
```

See also:

- [Note on webpack Treeshaking](#)
- [type tests for `defineComponent`](#)

:::tip `defineComponent()` also enables type inference for components defined in plain JavaScript. :::

## Usage in Single-File Components

To use TypeScript in SFCs, add the `lang="ts"` attribute to `<script>` tags. When `lang="ts"` is present, all template expressions also enjoy stricter type checking.

```
<script lang="ts">
import { defineComponent } from 'vue'

export default defineComponent({
  data() {
    return {
      count: 1
    }
  }
})
</script>

<template>
  <!-- type checking and auto-completion enabled -->
  {{ count.toFixed(2) }}
</template>
```

`lang="ts"` can also be used with `<script setup>`:

```
<script setup lang="ts">
// TypeScript enabled
import { ref } from 'vue'

const count = ref(1)
</script>

<template>
  <!-- type checking and auto-completion enabled -->
  {{ count.toFixed(2) }}
</template>
```

## TypeScript in Templates

The `<template>` also supports TypeScript in binding expressions when `<script lang="ts">` or `<script setup lang="ts">` is used. This is useful in cases where you need to perform type casting in template expressions.

Here's a contrived example:

```
<script setup lang="ts">
let x: string | number = 1
</script>

<template>
  <!-- error because x could be a string -->
  {{ x.toFixed(2) }}
</template>
```

This can be worked around with an inline type cast:

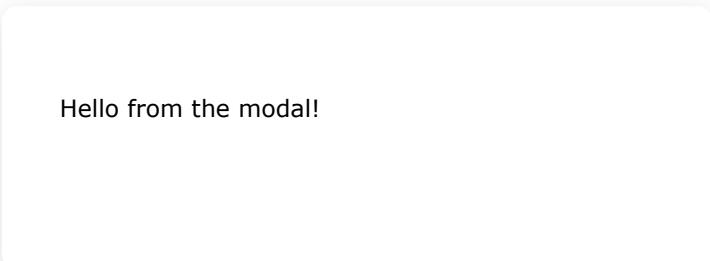
```
<script setup lang="ts">
let x: string | number = 1
</script>

<template>
  {{ (x as number).toFixed(2) }}
</template>
```

:::tip If using Vue CLI or a webpack-based setup, TypeScript in template expressions requires `vue-loader@^16.8.0`. :::

## API-Specific Recipes

- [TS with Composition API](#)
- [TS with Options API](#)



Hello from the modal!

## Colophon

This book is created by using the following sources:

- Vue - English
- GitHub source: [vuejs/docs/src/guide](https://vuejs.org/guide)
- Created: 2022-11-27
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/> Hello from the modal!
- By: @shinokada
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>