Application & Component Instances

#Creating an Application Instance

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

```
const app = Vue.createApp({ /* options */ })
```

The application instance is used to register 'globals' that can then be used by components within that application. We'll discuss that in detail later in the guide but as a quick example:

```
const app = Vue.createApp({})
app.component('SearchInput', SearchInputComponent)
app.directive('focus', FocusDirective)
app.use(LocalePlugin)
```

Most of the methods exposed by the application instance return that same instance, allowing for chaining:

```
Vue.createApp({})
  .component('SearchInput', SearchInputComponent)
  .directive('focus', FocusDirective)
  .use(LocalePlugin)
```

#The Root Component

The options passed to `createApp` are used to configure the root component. That component is used as the starting point for rendering when we mount the application.
An application needs to be mounted into a DOM element. For example, if we want to mount a Vue application into `<div id="app"></div>`, we should pass `#app`:

```
const RootComponent = { /* options */ }
const app = Vue.createApp(RootComponent)
const vm = app.mount('#app')
```

Unlike most of the application methods, `mount` does not return the application. Instead it returns the root component instance.
Although not strictly associated with the MVVM pattern , Vue's design was partly inspired by it. As a convention, we often use the variable `vm` (short for ViewModel) to refer to a component instance.
While all the examples on this page 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:

```
Root Component
└── TodoList
    ├── TodoItem
    │   ├── DeleteTodoButton
    │   └── EditTodoButton
    └── TodoListFooter
        ├── ClearTodosButton
        └── TodoListStatistics
```

Each component will have its own component instance, `vm`. For some components, such as `TodoItem`, there will likely be multiple instances rendered at any one time. All of the component instances in this application will share the same application instance.
We'll talk about the component system in detail later. For now, just be aware that the root component isn't really any different from any other component. The configuration options are the same, as is the behavior of the corresponding component instance.

#Component Instance Properties

Earlier in the guide we met `data` properties. Properties defined in `data` are exposed via the component instance:

```
const app = Vue.createApp({
  data() {
    return { count: 4 }
  }
})

const vm = app.mount('#app')
console.log(vm.count) // => 4
```

There are various other component options that add user-defined properties to the component instance, such as `methods`, `props`, `computed`, `inject` and `setup`. We'll discuss each of these in depth later in the guide. All of the properties of the component instance, no matter how they are defined, will be accessible in the component's template.

Vue also exposes some built-in properties via the component instance, such as `$attrs` and `$emit`. These properties all have a `$` prefix to avoid conflicting with user-defined property names.

#Lifecycle Hooks

Each 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.

For example, the created hook can be used to run code after an instance is created:

```
Vue.createApp({
  data() {
    return { count: 1 }
  },
  created() {
    // `this` points to the vm instance
    console.log('count is: ' + this.count) // => "count is: 1"
  }
})
```

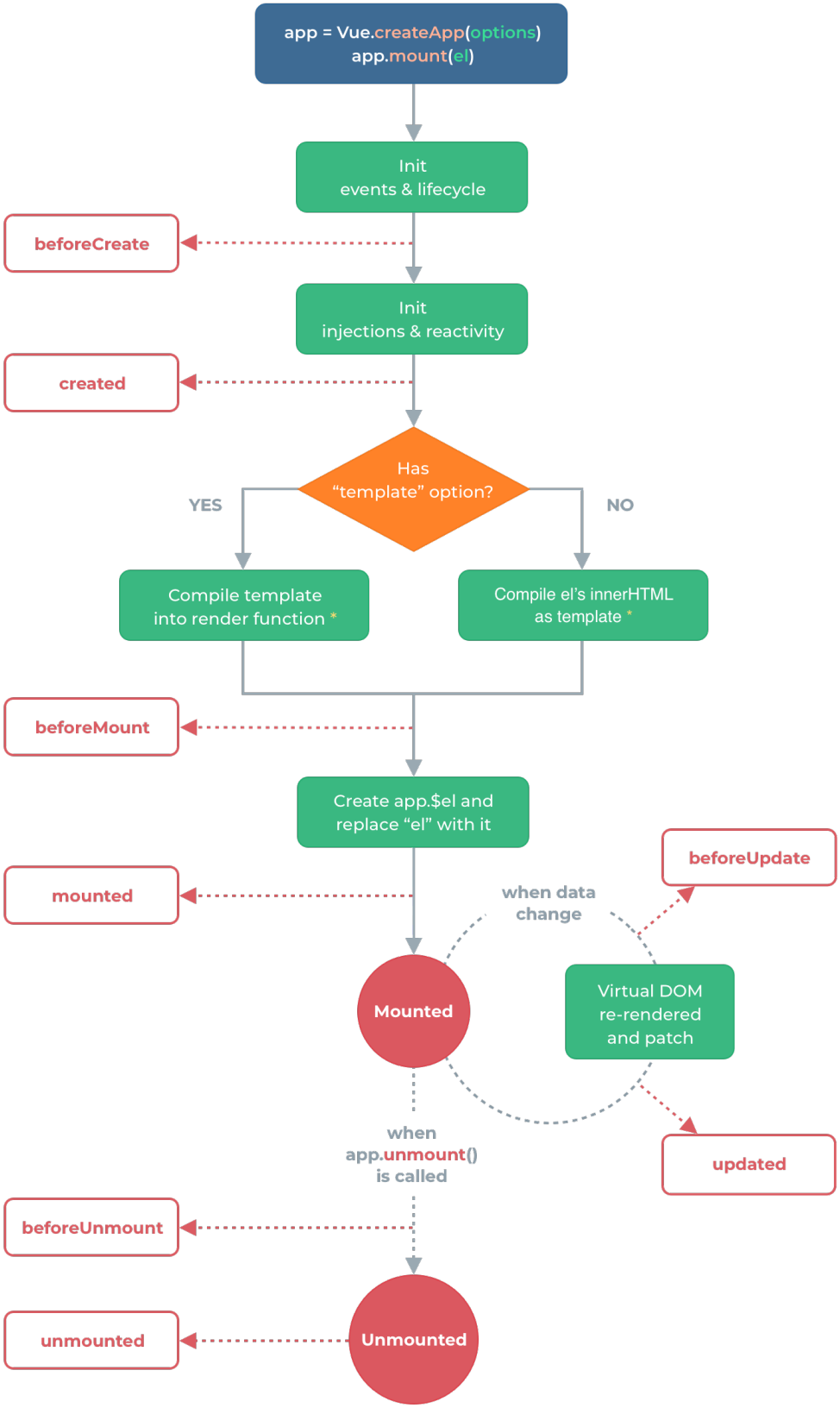
There are also other hooks which will be called at different stages of the instance's lifecycle, such as mounted, updated, and unmounted. All lifecycle hooks are called with their `this` context pointing to the current active instance invoking it.

TIP

Don't use arrow functions on an options property or callback, such as `created: () => console.log(this.a)` or `vm.$watch('a', newValue => this.myMethod())`. Since an arrow function doesn't have a `this`, `this` will be treated as any other variable and lexically looked up through parent scopes until found, often resulting in errors such as `Uncaught TypeError: Cannot read property of undefined` or `Uncaught TypeError: this.myMethod is not a function`.

#Lifecycle Diagram

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



* Template compilation is performed ahead-of-time if using a build step, e.g., with single-file components.

Template Syntax

Vue.js uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying component instance's data. All Vue.js templates are valid HTML that can be parsed by spec-compliant browsers and HTML parsers.
Under the hood, Vue compiles the templates into Virtual DOM render functions. Combined with the reactivity system, Vue is able to intelligently figure out the minimal number of components to re-render and apply the minimal amount of DOM manipulations when the app state changes.
If you are familiar with Virtual DOM concepts and prefer the raw power of JavaScript, you can also directly write render functions instead of templates, with optional JSX support.

#Interpolations

#Text

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

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

The mustache tag will be replaced with the value of the `msg` property from the corresponding component instance. It will also be updated whenever the `msg` property changes.
You can also perform one-time interpolations that do not update on data change by using the v-once directive, but keep in mind this will also affect any other bindings on the same node:

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

#Raw HTML

The double mustaches interprets the data as plain text, not HTML. In order to output real HTML, you will need to use the `v-html` directive:

```
<p>Using mustaches: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

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.
TIP
Dynamically rendering arbitrary HTML on your website can be very dangerous because it can easily lead to XSS vulnerabilities. Only use HTML interpolation on trusted content and never on user-provided content

#Attributes

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

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

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

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

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

#Using JavaScript Expressions

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

```
{{ number + 1 }} {{ ok ? 'YES' : 'NO' }} {{ message.split('').reverse().join('')
}}

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

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

```
<!-- this is a statement, not an expression: -->
{{ var a = 1 }}

<!-- flow control won't work either, use ternary expressions -->
{{ if (ok) { return message } }}
```

#Directives

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

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

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

#Arguments

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

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

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

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

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

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

#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 Expression Constraints" section below.
-->
<a v-bind:[attributeName]="url"> ... </a>
```

Here `attributeName` will be dynamically evaluated as a JavaScript expression, and its evaluated value will be used as the final value for the argument. For example, if your 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>
```

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

#Modifiers

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

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

You'll see other examples of modifiers later, for `v-on` and for `v-model`, when we explore those features.

#Shorthands

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

#`v-bind` Shorthand

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

<!-- shorthand -->
<a :href="url"> ... </a>

<!-- shorthand with dynamic argument -->
<a :[key]="url"> ... </a>
```

#`v-on` Shorthand

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

<!-- shorthand -->
<a @click="doSomething"> ... </a>

<!-- shorthand with dynamic argument -->
<a @[event]="doSomething"> ... </a>
```

They may look a bit different from normal HTML, but `:` and `@` are valid characters for attribute names and all Vue-supported browsers can parse it correctly. In addition, they do not appear in the final rendered markup. The shorthand syntax is totally optional, but you will likely appreciate it when you learn more about its usage later.
From the next page on, we'll use the shorthand in our examples, as that's the most common usage for Vue developers.

#Caveats

#Dynamic Argument Value Constraints

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

#Dynamic Argument Expression Constraints

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

```
<!-- This will trigger a compiler warning. -->
<a v-bind:['foo' + bar]="value"> ... </a>
```

We recommend replacing any complex expressions with a computed property, one of the most fundamental pieces of Vue, which we'll 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:

```
<!--
This will be converted to v-bind:[someattr] in in-DOM templates.
Unless you have a "someattr" property in your instance, your code won't work.
-->
<a v-bind:[someAttr]="value"> ... </a>
```

#JavaScript Expressions

Template expressions are sandboxed and only have access to a whitelist of globals such as `Math` and `Date`. You should not attempt to access user defined globals in template expressions.

Data Properties and Methods

#Data Properties

The `data` option for a component is a function. Vue calls this function as part of creating a new component instance. It should return an object, which Vue will then wrap in its reactivity system and store on the component instance as `$data`. For convenience, any top-level properties of that object are also exposed directly via the component instance:

```
const app = Vue.createApp({
  data() {
    return { count: 4 }
  }
})

const vm = app.mount('#app')

console.log(vm.$data.count) // => 4
console.log(vm.count)       // => 4

// Assigning a value to vm.count will also update $data.count
vm.count = 5
console.log(vm.$data.count) // => 5

// ... and vice-versa
vm.$data.count = 6
console.log(vm.count) // => 6
```

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 the component instance without including it in `data`. However, because this property isn't backed by the reactive `$data` object, it won't automatically be tracked by Vue's reactivity system.
Vue uses a `$` 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.

#Methods

To add methods to a component instance we use the `methods` option. This should be an object containing the desired methods:

```
const app = Vue.createApp({
  data() {
    return { count: 4 }
  },
  methods: {
    increment() {
```

```
    // `this` will refer to the component instance
    this.count++
  }
 }
})

const vm = app.mount('#app')

console.log(vm.count) // => 4

vm.increment()

console.log(vm.count) // => 5
```

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.

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">Up vote</button>
```

In the example above, the method `increment` will be called when the `<button>` is clicked.

It is also possible to call a method directly from a template. As we'll see shortly, it's usually better to use a computed property instead. However, using a method can be useful in scenarios where computed properties aren't a viable option. You can call a method anywhere that a template supports JavaScript expressions:

```
<span :title="toTitleDate(date)">
 {{ formatDate(date) }}
</span>
```

If the methods `toTitleDate` or `formatDate` access any reactive data then it will be tracked as a rendering dependency, just as if it had been used in the template directly.

Methods called from a template should not have any side effects, such as changing data or triggering asynchronous processes. If you find yourself tempted to do that you should probably use a lifecycle hook instead.

#Debouncing and Throttling

Vue doesn't include built-in support for debouncing or throttling but it can be implemented using libraries such as Lodash.

In cases where a component is only used once, the debouncing can be applied directly within `methods`:

```
<script src="https://unpkg.com/lodash@4.17.20/lodash.min.js"></script>
<script>
 Vue.createApp({
  methods: {
   // Debouncing with Lodash
   click: _.debounce(function() {
    // ... respond to click ...
   }, 500)
  }
 }).mount('#app')
</script>
```

However, this approach is potentially problematic for components that are reused because they'll all share the same debounced function. To keep the component instances independent from each other, we can add the debounced function in the `created` lifecycle hook:

```
app.component('save-button', {
  created() {
    // Debouncing with Lodash
    this.debouncedClick = _.debounce(this.click, 500)
  },
  unmounted() {
    // Cancel the timer when the component is removed
    this.debouncedClick.cancel()
  },
  methods: {
    click() {
      // ... respond to click ...
    }
  },
  template: `
    <button @click="debouncedClick">
      Save
    </button>
  `
})
```

Computed Properties and Watchers

#Computed Properties

In-template expressions are very convenient, but they are meant for simple operations. Putting too much logic in your templates can make them bloated and hard to maintain. For example, if we have an object with a nested array:

```
Vue.createApp({
  data() {
    return {
      author: {
        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

```
<div id="computed-basics">
  <p>Has published books:</p>
  <span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
</div>
```

At this point, the template is no longer simple and declarative. You have to look at it for a second before realizing that it performs a calculation depending on `author.books`. The problem is made worse when you want to include this calculation in your template more than once.
That's why for complex logic that includes reactive data, you should use a computed property.

#Basic Example

```
<div id="computed-basics">
  <p>Has published books:</p>
  <span>{{ publishedBooksMessage }}</span>
</div>
```

```
Vue.createApp({
  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 vm instance
      return this.author.books.length > 0 ? 'Yes' : 'No'
    }
  }
}).mount('#computed-basics')
```

Here we have declared a computed property `publishedBooksMessage`.
Try to change the value of `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 `vm.publishedBooksMessage` depends on `vm.author.books`, so it will update any bindings that depend on `vm.publishedBooksMessage` when `vm.author.books` changes. And the best part is that we've created this dependency relationship declaratively: the computed getter function has no side effects, which makes it easier to test and understand.

#Computed Caching vs Methods

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

```
<p>{{ calculateBooksMessage() }}</p>
// in component
methods: {
  calculateBooksMessage() {
    return this.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 the `publishedBooksMessage` computed property will immediately return the previously computed result without having to run the function again.

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

```
computed: {
  now() {
    return Date.now()
  }
}
```

In comparison, a method invocation will always run the function whenever a re-render happens. Why do we need caching? Imagine we have an expensive computed property `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` instead.

#Computed Setter

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

```
// ...
computed: {
  fullName: {
    // getter
    get() {
      return this.firstName + ' ' + this.lastName
    },
    // setter
    set(newValue) {
      const names = newValue.split(' ')
      this.firstName = names[0]
      this.lastName = names[names.length - 1]
    }
  }
}
// ...
```

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

#Watchers

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

For example:

```
<div id="watch-example">
  <p>
    Ask a yes/no question:
    <input v-model="question" />
  </p>
  <p>{{ answer }}</p>
</div>


<!-- Since there is already a rich ecosystem of ajax libraries    -->
<!-- and collections of general-purpose utility methods, Vue core -->
<!-- is able to remain small by not reinventing them. This also   -->
<!-- gives you the freedom to use what you're familiar with.      -->
<script src="https://cdn.jsdelivr.net/npm/axios@0.12.0/dist/axios.min.js"></script>
<script>
  const watchExampleVM = Vue.createApp({
    data() {
      return {
        question: '',
        answer: 'Questions usually contain a question mark. ;-)'
      }
    },
    watch: {
      // whenever question changes, this function will run
      question(newQuestion, oldQuestion) {
        if (newQuestion.indexOf('?') > -1) {
          this.getAnswer()
        }
      }
    },
    methods: {
      getAnswer() {
        this.answer = 'Thinking...'
        axios
          .get('https://yesno.wtf/api')
          .then(response => {
            this.answer = response.data.answer
          })
          .catch(error => {
            this.answer = 'Error! Could not reach the API. ' + error
```

```
        })
      }
    }
  }).mount('#watch-example')
</script>
```

In this case, using the `watch` option allows us to perform an asynchronous operation (accessing an API) and sets a condition for performing this operation. None of that would be possible with a computed property.

In addition to the `watch` option, you can also use the imperative vm.$watch API.

#Computed vs Watched Property

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

```
<div id="demo">{{ fullName }}</div>
const vm = Vue.createApp({
  data() {
    return {
      firstName: 'Foo',
      lastName: 'Bar',
      fullName: 'Foo Bar'
    }
  },
  watch: {
    firstName(val) {
      this.fullName = val + ' ' + this.lastName
    },
    lastName(val) {
      this.fullName = this.firstName + ' ' + val
    }
  }
}).mount('#demo')
```

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

```
const vm = Vue.createApp({
  data() {
    return {
      firstName: 'Foo',
      lastName: 'Bar'
    }
  },
  computed: {
    fullName() {
      return this.firstName + ' ' + this.lastName
    }
  }
}).mount('#demo')
```

Much better, isn't it?

Class and Style Bindings

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

#Binding HTML Classes

#Object Syntax

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 template:

```
<div
  class="static"
  :class="{ active: isActive, 'text-danger': hasError }"
></div>
```

And the following data:

```
data() {
  return {
    isActive: true,
    hasError: false
  }
}
```

It will render:

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

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

The bound object doesn't have to be inline:

```
<div :class="classObject"></div>
```

```
data() {
  return {
    classObject: {
      active: true,
      'text-danger': false
    }
  }
}
```

This will render the same result. We can also bind to a computed property that returns an object. This is a common and powerful pattern:

```
<div :class="classObject"></div>
```

```
data() {
  return {
    isActive: true,
    error: null
  }
},
computed: {
  classObject() {
    return {
      active: this.isActive && !this.error,
      'text-danger': this.error && this.error.type === 'fatal'
    }
  }
}
```

#Array Syntax

We can pass an array to `:class` to apply a list of classes:

```
<div :class="[activeClass, errorClass]"></div>
```

```
data() {
  return {
    activeClass: 'active',
    errorClass: 'text-danger'
  }
}
```

Which will render:

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

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

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

This will always apply `errorClass`, but will only apply `activeClass` when `isActive` is truthy.
However, this can be a bit verbose if you have multiple conditional classes. That's why it's also possible to use the object syntax inside array syntax:

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

#With Components

This section assumes knowledge of Vue Components. Feel free to skip it and come back later.
When you use the `class` attribute on a custom component with a single root element, those classes will be added to this element. Existing classes on this element will not be overwritten.
For example, if you declare this component:

```
const app = Vue.createApp({})

app.component('my-component', {
  template: `<p class="foo bar">Hi!</p>`
})
```

Then add some classes when using it:

```
<div id="app">
  <my-component class="baz boo"></my-component>
</div>
```

The rendered HTML will be:
```
<p class="foo bar baz boo">Hi</p>
```

The same is true for class bindings:
```
<my-component :class="{ active: isActive }"></my-component>
```
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 component will receive this class. You can do this using `$attrs` component property:
```
<div id="app">
  <my-component class="baz"></my-component>
</div>

const app = Vue.createApp({})

app.component('my-component', {
  template: `
    <p :class="$attrs.class">Hi!</p>
    <span>This is a child component</span>
  `
})
```

You can learn more about component attribute inheritance in Non-Prop Attributes section.

#Binding Inline Styles

#Object Syntax

The object syntax for `:style` is pretty straightforward - it looks almost like CSS, except it's a JavaScript object. You can use either camelCase or kebab-case (use quotes with kebab-case) for the CSS property names:
```
<div :style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>

data() {
  return {
    activeColor: 'red',
    fontSize: 30
  }
}
```
It is often a good idea to bind to a style object directly so that the template is cleaner:
```
<div :style="styleObject"></div>

data() {
  return {
    styleObject: {
      color: 'red',
      fontSize: '13px'
    }
  }
}
```

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

#Array Syntax

The array syntax for `:style` allows you to apply multiple style objects to the same element:

```
<div :style="[baseStyles, overridingStyles]"></div>
```

#Auto-prefixing

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

#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.

Conditional Rendering

#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>
```

1

It is also possible to add an "else block" with `v-else`:

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

<h1 v-else>Oh no 😢</h1>
```

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

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>
```

1

#v-else

You can use the `v-else` directive to indicate an "else block" for `v-if`:

```
<div v-if="Math.random() > 0.5">
  Now you see me
</div>
<div v-else>
```

```
  Now you don't
</div>
```

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

#v-else-if

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-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>
1
```

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` 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

Note

Using `v-if` and `v-for` together is not recommended. See the style guide for further information.

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.

List Rendering

#Mapping an Array to Elements with `v-for`

We can use the `v-for` directive to render a list of items based on an array. The `v-for` directive requires a special syntax in the form of `item in items`, where `items` is the source data array and `item` is an alias for the array element being iterated on:

```
<ul id="array-rendering">
  <li v-for="item in items">
    {{ item.message }}
  </li>
</ul>


Vue.createApp({
  data() {
    return {
      items: [{ message: 'Foo' }, { message: 'Bar' }]
    }
  }
}).mount('#array-rendering')
```

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

```
<ul id="array-with-index">
  <li v-for="(item, index) in items">
    {{ parentMessage }} - {{ index }} - {{ item.message }}
  </li>
</ul>


Vue.createApp({
  data() {
    return {
      parentMessage: 'Parent',
      items: [{ message: 'Foo' }, { message: 'Bar' }]
    }
  }
}).mount('#array-with-index')
```

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

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

#v-for with an Object

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

```
<ul id="v-for-object" class="demo">
  <li v-for="value in myObject">
    {{ value }}
  </li>
</ul>


Vue.createApp({
  data() {
    return {
      myObject: {
        title: 'How to do lists in Vue',
        author: 'Jane Doe',
        publishedAt: '2016-04-10'
      }
    }
  }
}).mount('#v-for-object')
```

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

```
<li v-for="(value, name) in myObject">
  {{ name }}: {{ value }}
</li>
```

And another for the index:

```
<li v-for="(value, name, index) in myObject">
  {{ index }}. {{ name }}: {{ value }}
</li>
```

Note

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

#Maintaining State

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

This 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>
```

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

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

Note

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

For detailed usage of the `key` attribute, please see the `key` API documentation.

#Array Change Detection

#Mutation Methods

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

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

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

#Replacing an Array

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

```
example1.items = example1.items.filter(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:

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


data() {
  return {
    numbers: [ 1, 2, 3, 4, 5 ]
  }
},
```

```
computed: {
  evenNumbers() {
    return this.numbers.filter(number => number % 2 === 0)
  }
}
```

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

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


data() {
  return {
    sets: [[ 1, 2, 3, 4, 5 ], [6, 7, 8, 9, 10]]
  }
},
methods: {
  even(numbers) {
    return numbers.filter(number => number % 2 === 0)
  }
}
```

#`v-for` with a Range

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

```
<div id="range" class="demo">
  <span v-for="n in 10">{{ n }} </span>
</div>
```

#v-for on a <template>

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

```
<ul>
  <template v-for="item in items">
    <li>{{ item.msg }}</li>
    <li class="divider" role="presentation"></li>
  </template>
</ul>
```

#`v-for` with `v-if`

TIP

Note that it's not recommended to use `v-if` and `v-for` together. 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 }}
</li>
```

This can be fixed by moving `v-for` to a wrapping `<template>` tag:

```
<template v-for="todo in todos">
  <li v-if="!todo.isComplete">
    {{ todo }}
  </li>
</template>
```

#`v-for` with a Component

This section assumes knowledge of Components. Feel free to skip it and come back later.

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

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

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

```
<my-component
  v-for="(item, index) in items"
  :item="item"
  :index="index"
  :key="item.id"
></my-component>
```

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

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

```
<div id="todo-list-example">
  <form v-on:submit.prevent="addNewTodo">
    <label for="new-todo">Add a todo</label>
    <input
      v-model="newTodoText"
      id="new-todo"
      placeholder="E.g. Feed the cat"
    />
    <button>Add</button>
  </form>
  <ul>
    <todo-item
      v-for="(todo, index) in todos"
      :key="todo.id"
      :title="todo.title"
      @remove="todos.splice(index, 1)"
    ></todo-item>
  </ul>
</div>
```

```
const app = Vue.createApp({
  data() {
    return {
      newTodoText: '',
      todos: [
        {
          id: 1,
          title: 'Do the dishes'
        },
        {
          id: 2,
          title: 'Take out the trash'
        },
        {
          id: 3,
          title: 'Mow the lawn'
        }
      ],
      nextTodoId: 4
    }
  },
  methods: {
    addNewTodo() {
      this.todos.push({
        id: this.nextTodoId++,
        title: this.newTodoText
      })
      this.newTodoText = ''
    }
  }
})

app.component('todo-item', {
  template: `
    <li>
      {{ title }}
      <button @click="$emit('remove')">Remove</button>
    </li>
  `,
  props: ['title']
})

app.mount('#todo-list-example')
```

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="methodName"` or with the shortcut, `@click="methodName"`
For example:

```
<div id="basic-event">
  <button @click="counter += 1">Add 1</button>
  <p>The button above has been clicked {{ counter }} times.</p>
</div>


Vue.createApp({
  data() {
    return {
      counter: 1
    }
  }
}).mount('#basic-event')
```

#Method Event Handlers

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

```
<div id="event-with-method">
  <!-- `greet` is the name of a method defined below -->
  <button @click="greet">Greet</button>
</div>


Vue.createApp({
  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)
      }
    }
  }
```

```
}).mount('#event-with-method')
```

#Methods in Inline Handlers

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

```
<div id="inline-handler">
  <button @click="say('hi')">Say hi</button>
  <button @click="say('what')">Say what</button>
</div>


Vue.createApp({
  methods: {
    say(message) {
      alert(message)
    }
  }
}).mount('#inline-handler')
```

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

```
<button @click="warn('Form cannot be submitted yet.', $event)">
  Submit
</button>


// ...
methods: {
  warn(message, event) {
    // now we have access to the native event
    if (event) {
      event.preventDefault()
    }
    alert(message)
  }
}
```

#Multiple Event Handlers

You can have multiple methods in an event handler separated by a comma operator like this:

```
<!-- both one() and two() will execute on button click -->
<button @click="one($event), two($event)">
  Submit
</button>


// ...
methods: {
  one(event) {
```

```
    // first handler logic...
  },
  two(event) {
    // second handler logic...
  }
}
```

#Event Modifiers

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

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

- .stop
- .prevent
- .capture
- .self
- .once
- .passive

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

<!-- 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>

<!-- 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 all clicks while `@click.self.prevent` will only prevent clicks on the element itself.

```
<!-- the click event will be triggered at most once -->
<a @click.once="doThis"></a>
```

Unlike the other modifiers, which are exclusive to native DOM events, the `.once` modifier can also be used on component events. If you haven't read about components yet, don't worry about this for now.

Vue also offers the `.passive` modifier, corresponding to `addEventListener`'s `passive` option.

```
<!-- the scroll event's default behavior (scrolling) will happen -->
<!-- immediately, instead of waiting for `onScroll` to complete  -->
<!-- in case it contains `event.preventDefault()`                -->
<div @scroll.passive="onScroll">...</div>
```

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

TIP

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

#Key Modifiers

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

```
<!-- only call `vm.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

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.

#Why Listeners in HTML?

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

1. It's easier to locate the handler function implementations within your JS code by skimming the HTML template.

2. Since you don't have to manually attach event listeners in JS, your ViewModel code can be pure logic and DOM-free. This makes it easier to test.

3. When a ViewModel is destroyed, all event listeners are automatically removed. You don't need to worry about cleaning it up yourself.

Form Input Bindings

#Basic Usage

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

Note

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

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

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

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 cater for these updates as well, use `input` event instead.

#Text

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

#Multiline text

```
<span>Multiline message is:</span>
<p style="white-space: pre-line;">{{ message }}</p>
<br />
<textarea v-model="message" placeholder="add multiple lines"></textarea>
```

Interpolation on textareas 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>
```

Multiple checkboxes, bound to the same Array:

```
<div id="v-model-multiple-checkboxes">
  <input type="checkbox" id="jack" value="Jack" v-model="checkedNames" />
```

```
  <label for="jack">Jack</label>
  <input type="checkbox" id="john" value="John" v-model="checkedNames" />
  <label for="john">John</label>
  <input type="checkbox" id="mike" value="Mike" v-model="checkedNames" />
  <label for="mike">Mike</label>
  <br />
  <span>Checked names: {{ checkedNames }}</span>
</div>


Vue.createApp({
  data() {
    return {
      checkedNames: []
    }
  }
}).mount('#v-model-multiple-checkboxes')
```

#Radio

```
<div id="v-model-radiobutton">
  <input type="radio" id="one" value="One" v-model="picked" />
  <label for="one">One</label>
  <br />
  <input type="radio" id="two" value="Two" v-model="picked" />
  <label for="two">Two</label>
  <br />
  <span>Picked: {{ picked }}</span>
</div>


Vue.createApp({
  data() {
    return {
      picked: ''
    }
  }
}).mount('#v-model-radiobutton')
```

#Select

Single select:

```
<div id="v-model-select" class="demo">
  <select v-model="selected">
    <option disabled value="">Please select one</option>
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <span>Selected: {{ selected }}</span>
</div>
```

Vue 3

```
Vue.createApp({
  data() {
    return {
      selected: ''
    }
  }
}).mount('#v-model-select')
```

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):

```
<select v-model="selected" multiple>
  <option>A</option>
  <option>B</option>
  <option>C</option>
</select>
<br />
<span>Selected: {{ selected }}</span>
```

Dynamic options rendered with `v-for`:

```
<div id="v-model-select-dynamic" class="demo">
  <select v-model="selected">
    <option v-for="option in options" :value="option.value">
      {{ option.text }}
    </option>
  </select>
  <span>Selected: {{ selected }}</span>
</div>
```

```
Vue.createApp({
  data() {
    return {
      selected: 'A',
      options: [
        { text: 'One', value: 'A' },
        { text: 'Two', value: 'B' },
        { text: 'Three', value: 'C' }
      ]
    }
  }
}).mount('#v-model-select-dynamic')
```

#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" />
// when checked:
vm.toggle === 'yes'
// when unchecked:
vm.toggle === 'no'
```

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" v-bind:value="a" />


// when checked:
vm.pick === vm.a
```

#Select Options

```
<select v-model="selected">
  <!-- inline object literal -->
  <option :value="{ number: 123 }">123</option>
</select>

// when selected:
typeof vm.selected // => 'object'
vm.selected.number // => 123
```

#Modifiers

#.lazy

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

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

#.number

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

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

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

#.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 custom inputs in the Components guide.

Components Basics

#Base Example

Here's an example of a Vue component:

```
// Create a Vue application
const app = Vue.createApp({})

// Define a new global component called button-counter
app.component('button-counter', {
  data() {
    return {
      count: 0
    }
  },
  template: `
    <button @click="count++">
      You clicked me {{ count }} times.
    </button>`
})
```

INFO

We're showing you a simple example here, but in a typical Vue application we use Single File Components instead of a string template. You can find more information about them in this section.

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

```
<div id="components-demo">
  <button-counter></button-counter>
</div>
app.mount('#components-demo')
```

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

#Reusing Components

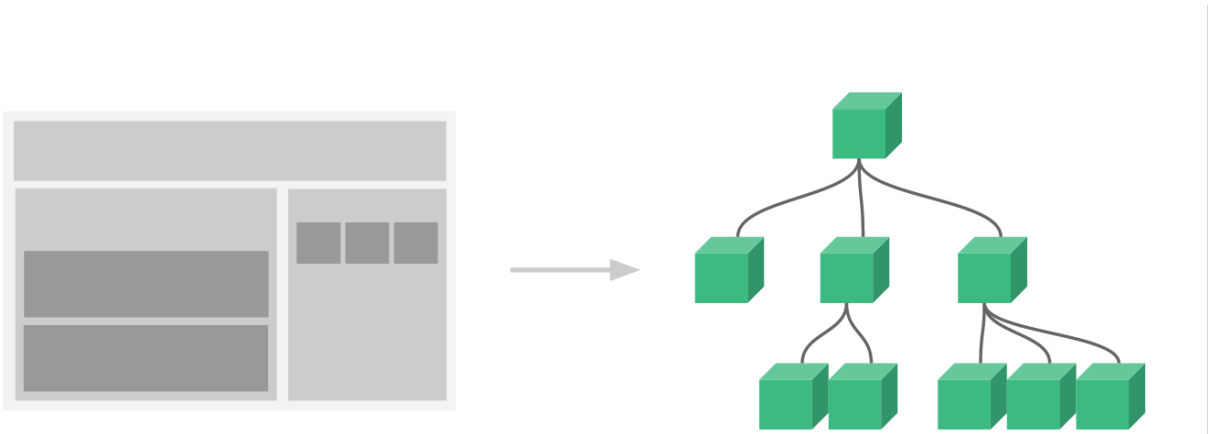
Components can be reused as many times as you want:

```
<div id="components-demo">
  <button-counter></button-counter>
  <button-counter></button-counter>
  <button-counter></button-counter>
</div>
```

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.

#Organizing Components

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



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

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

```
const app = Vue.createApp({})

app.component('my-component-name', {
  // ... options ...
```

```
})
```

Globally registered components can be used in the template of `app` instance created afterwards - and even inside all subcomponents of that root instance's component tree.

That's all you need to know about registration for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Component Registration.

#Passing Data to Child Components with Props

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

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

```
const app = Vue.createApp({})

app.component('blog-post', {
  props: ['title'],
  template: `<h4>{{ title }}</h4>`
})

app.mount('#blog-post-demo')
```

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

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

```
<div id="blog-post-demo" class="demo">
  <blog-post title="My journey with Vue"></blog-post>
  <blog-post title="Blogging with Vue"></blog-post>
  <blog-post title="Why Vue is so fun"></blog-post>
</div>
```

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

```
const App = {
  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 app = Vue.createApp(App)
```

```
app.component('blog-post', {
  props: ['title'],
  template: `<h4>{{ title }}</h4>`
})

app.mount('#blog-posts-demo')
```

Then want to render a component for each one:

```
<div id="blog-posts-demo">
  <blog-post
    v-for="post in posts"
    :key="post.id"
    :title="post.title"
  ></blog-post>
</div>
```

Above, you'll see that we can use `v-bind` to dynamically pass props. This is especially useful when you don't know the exact content you're going to render ahead of time.

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 Child Components Events

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

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

```
const App = {
  data() {
    return {
      posts: [
        /* ... */
      ],
      postFontSize: 1
    }
  }
}
```

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

```
<div id="blog-posts-events-demo">
  <div v-bind:style="{ fontSize: postFontSize + 'em' }">
    <blog-post v-for="post in posts" :key="post.id" :title="post.title"></blog-post>
  </div>
</div>
```

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

```
app.component('blog-post', {
  props: ['title'],
  template: `
    <div class="blog-post">
      <h4>{{ title }}</h4>
      <button>
```

```
      Enlarge text
    </button>
  </div>
`
})
```

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

```
<button>
  Enlarge text
</button>
```

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

```
<blog-post ... @enlarge-text="postFontSize += 0.1"></blog-post>
```

Then the child component can emit an event on itself by calling the built-in `$emit` method, passing the name of the event:

```
<button @click="$emit('enlarge-text')">
  Enlarge text
</button>
```

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

We can list emitted events in the component's `emits` option.

```
app.component('blog-post', {
  props: ['title'],
  emits: ['enlarge-text']
})
```

This will allow you to check all the events component emits and optionally validate them

#Emitting a Value With an Event

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

```
<button @click="$emit('enlarge-text', 0.1)">
  Enlarge text
</button>
```

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

```
<blog-post ... @enlarge-text="postFontSize += $event"></blog-post>
```

Or, if the event handler is a method:

```
<blog-post ... @enlarge-text="onEnlargeText"></blog-post>
```

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

```
methods: {
  onEnlargeText(enlargeAmount) {
    this.postFontSize += enlargeAmount
  }
}
```

#Using `v-model` on Components

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

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

does the same thing as:

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

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

```
<custom-input
  :model-value="searchText"
  @update:model-value="searchText = $event"
></custom-input>
```

WARNING

Please note we used `model-value` with kebab-case here because we are working with in-DOM template. You can find a detailed explanation on kebab-cased vs camelCased attributes in the DOM Template Parsing Caveats section

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

- Bind the `value` attribute to a `modelValue` prop
- On `input`, emit an `update:modelValue` event with the new value

Here's that in action:

```
app.component('custom-input', {
  props: ['modelValue'],
  template: `
    <input
      :value="modelValue"
      @input="$emit('update:modelValue', $event.target.value)"
    >
  `
})
```

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

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

Another way of creating the `v-model` capability within a custom component is to use the ability of `computed` properties' to define a getter and setter.

In the following example, we refactor the `custom-input` component using a computed property.

Keep in mind, the `get` method should return the `modelValue` property, or whichever property is being using for binding, and the `set` method should fire off the corresponding `$emit` for that property.

```
app.component('custom-input', {
  props: ['modelValue'],
  template: `
    <input v-model="value">
  `,
  computed: {
    value: {
      get() {
        return this.modelValue
      },
```

```
      set(value) {
        this.$emit('update:modelValue', value)
      }
    }
  }
})
```

That's all you need to know about custom component events for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Custom Events.

#Content Distribution with Slots

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

```
<alert-box>
  Something bad happened.
</alert-box>
```

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

```
app.component('alert-box', {
  template: `
    <div class="demo-alert-box">
      <strong>Error!</strong>
      <slot></slot>
    </div>
  `
})
```

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

That's all you need to know about slots for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Slots.

#Dynamic Components

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

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

```
<!-- Component changes when currentTabComponent changes -->
<component :is="currentTabComponent"></component>
```

In the example above, `currentTabComponent` can contain either:

- the name of a registered component, or
- a component's options object

See this sandbox to experiment with the full code, or this version for an example binding to a component's options object, instead of its registered name.

Keep in mind that this attribute can be used with regular HTML elements, however they will be treated as components, which means all attributes will be bound as DOM attributes. For some properties such as `value` to work as you would expect, you will need to bind them using the `.prop` modifier.

That's all you need to know about dynamic components for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Dynamic & Async Components.

#DOM Template Parsing Caveats

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

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

```
<table>
  <blog-post-row></blog-post-row>
</table>
```

The custom component `<blog-post-row>` will be hoisted out as invalid content, causing errors in the eventual rendered output. Fortunately, we can use `v-is` special directive as a workaround:

```
<table>
  <tr v-is="'blog-post-row'"></tr>
</table>
```

WARNING

`v-is` value should be a JavaScript string literal:

```
<!-- Incorrect, nothing will be rendered -->
<tr v-is="blog-post-row"></tr>

<!-- Correct -->
<tr v-is="'blog-post-row'"></tr>
```

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

```
// camelCase in JavaScript

app.component('blog-post', {
  props: ['postTitle'],
  template: `
    <h3>{{ postTitle }}</h3>
  `
})
<!-- kebab-case in HTML -->

<blog-post post-title="hello!"></blog-post>
```

It should be noted that these limitations do not apply if you are using string templates from one of the following sources:

- String templates (e.g. `template: '...'`)
- Single-file (`.vue`) components
- <script type="text/x-template">

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

Once you feel comfortable with the knowledge you've just digested, we recommend coming back to read the full guide on Dynamic & Async Components, as well as the other pages in the Components In-Depth section of the sidebar.

Component Registration

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

#Component Names

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

```
const app = Vue.createApp({...})

app.component('my-component-name', {
  /* ... */
})
```

The component's name is the first argument of `app.component`. In the example above, the component's name is "my-component-name".

The name you give a component may depend on where you intend to use it. When using a component directly in the DOM (as opposed to in a string template or single-file component), we strongly recommend following the W3C rules for custom tag names:

1. All lowercase
2. Contains a hyphen (i.e., has multiple words connected with the hyphen symbol)

By doing so, this will help you avoid conflicts with current and future HTML elements.

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

#Name Casing

When defining components in a string template or a single-file component, you have two options when defining component names:

#With kebab-case

```
app.component('my-component-name', {
  /* ... */
})
```

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

#With PascalCase

```
app.component('MyComponentName', {
  /* ... */
})
```

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

#Global Registration

So far, we've only created components using `app.component`:

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

These components are globally registered for the application. That means they can be used in the template of any component instance within this application:

```
const app = Vue.createApp({})
```

```
app.component('component-a', {
  /* ... */
})
app.component('component-b', {
  /* ... */
})
app.component('component-c', {
  /* ... */
})

app.mount('#app')
```

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

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

#Local Registration

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

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

```
const ComponentA = {
  /* ... */
}
const ComponentB = {
  /* ... */
}
const ComponentC = {
  /* ... */
}
```

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

```
const app = Vue.createApp({
  components: {
    'component-a': ComponentA,
    'component-b': ComponentB
  }
})
```

For each property in the `components` object, the key will be the name of the custom element, while the value will contain the options object for the component.
Note that locally registered components are not also available in subcomponents. For example, if you wanted `ComponentA` to be available in `ComponentB`, you'd have to use:

```
const ComponentA = {
  /* ... */
}

const ComponentB = {
  components: {
    'component-a': ComponentA
  }
  // ...
}
```

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

```
import ComponentA from './ComponentA.vue'

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

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

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

#Module Systems

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

#Local Registration in a Module System

If you're still here, then it's likely you're using a module system, such as with Babel and Webpack. In these cases, we recommend creating a `components` directory, with each component in its own file.
Then you'll need to import each component you'd like to use, before you locally register it. For example, in a hypothetical `ComponentB.js` or `ComponentB.vue` file:

```
import ComponentA from './ComponentA'
import ComponentC from './ComponentC'

export default {
  components: {
    ComponentA,
    ComponentC
```

```
  }
  // ...
}
```

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

Props

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

#Prop Types

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

```
props: ['title', 'likes', 'isPublished', 'commentIds', 'author']
```

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

```
props: {
  title: String,
  likes: Number,
  isPublished: Boolean,
  commentIds: Array,
  author: Object,
  callback: Function,
  contactsPromise: Promise // or any other constructor
}
```

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

#Passing Static or Dynamic Props

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

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

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

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

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

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

#Passing a Number

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

```
<blog-post :likes="42"></blog-post>

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

#Passing a Boolean

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

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

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

#Passing an Array

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

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

#Passing an Object

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

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

#Passing the Properties of an Object

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

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

The following template:

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

Will be equivalent to:

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

#One-Way Data Flow

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

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

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

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

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

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

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

Note

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

#Prop Validation

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

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

```
app.component('my-component', {
  props: {
    // Basic type check (`null` and `undefined` values will pass any type validation)
    propA: Number,
    // Multiple possible types
    propB: [String, Number],
    // Required string
    propC: {
      type: String,
      required: true
    },
```

```
    // Number with a default value
    propD: {
      type: Number,
      default: 100
    },
    // Object with a default value
    propE: {
      type: Object,
      // Object or array defaults must be returned from
      // a factory function
      default: function() {
        return { message: 'hello' }
      }
    },
    // Custom validator function
    propF: {
      validator: function(value) {
        // The value must match one of these strings
        return ['success', 'warning', 'danger'].indexOf(value) !== -1
      }
    },
    // 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: function() {
        return 'Default function'
      }
    }
  }
})
```

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

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.

#Type Checks

The `type` can be one of the following native constructors:

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

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

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

You could use:

```
app.component('blog-post', {
  props: {
    author: Person
  }
})
```

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

#Prop Casing (camelCase vs kebab-case)

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

```
const app = Vue.createApp({})

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

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

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

Non-Prop Attributes

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

A component non-prop attribute is an attribute or event listener that is passed to a component, but does not have a corresponding property defined in props or emits. Common examples of this include `class`, `style`, and `id` attributes. You can access those attributes via `$attrs` property.

#Attribute Inheritance

When a component returns a single root node, non-prop attributes will automatically be added to the root node's attributes. For example, in the instance of a date-picker component:

```
app.component('date-picker', {
  template: `
    <div class="date-picker">
      <input type="datetime" />
    </div>
  `
})
```

In the event we need to define the status of the date-picker component via a `data-status` property, it will be applied to the root node (i.e., `div.date-picker`).

```
<!-- Date-picker component with a non-prop attribute -->
<date-picker data-status="activated"></date-picker>

<!-- Rendered date-picker component -->
<div class="date-picker" data-status="activated">
  <input type="datetime" />
</div>
```

Same rule applies to the event listeners:

```
<date-picker @change="submitChange"></date-picker>


app.component('date-picker', {
  created() {
    console.log(this.$attrs) // { onChange: () => {}  }
  }
})
```

This might be helpful when we have an HTML element with `change` event as a root element of `date-picker`.

```
app.component('date-picker', {
  template: `
    <select>
      <option value="1">Yesterday</option>
      <option value="2">Today</option>
      <option value="3">Tomorrow</option>
    </select>
  `
})
```

In this case, `change` event listener is passed from the parent component to the child and it will be triggered on native `<select>` `change` event. We won't need to emit an event from the `date-picker` explicitly:

```
<div id="date-picker" class="demo">
  <date-picker @change="showChange"></date-picker>
</div>

const app = Vue.createApp({
  methods: {
    showChange(event) {
      console.log(event.target.value) // will log a value of the selected option
    }
  }
})
```

#Disabling Attribute Inheritance

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

The common scenario for disabling an 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 control to apply to other elements attributes to use the component's `$attrs` property, which includes all attributes not included to component `props` and `emits` properties (e.g., `class`, `style`, `v-on` listeners, etc.).

Using our date-picker component example from the previous section, in the event we need to apply all non-prop attributes to the `input` element rather than the root `div` element, this can be accomplished by using the `v-bind` shortcut.

```
app.component('date-picker', {
  inheritAttrs: false,
  template: `
    <div class="date-picker">
      <input type="datetime" v-bind="$attrs" />
    </div>
  `
})
```

With this new configuration, our `data-status` attribute will be applied to our `input` element!

```
<!-- Date-picker component with a non-prop attribute -->
<date-picker data-status="activated"></date-picker>

<!-- Rendered date-picker component -->
<div class="date-picker">
  <input type="datetime" data-status="activated" />
</div>
```

#Attribute Inheritance on Multiple Root Nodes

Unlike single root node components, 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.

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


// This will raise a warning
app.component('custom-layout', {
  template: `
    <header>...</header>
    <main>...</main>
    <footer>...</footer>
  `
})

// No warnings, $attrs are passed to <main> element
app.component('custom-layout', {
```

```
  template: `
    <header>...</header>
    <main v-bind="$attrs">...</main>
    <footer>...</footer>
  `
})
```

Custom Events

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

#Event Names

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

```
this.$emit('my-event')
```

```
<my-component @my-event="doSomething"></my-component>
```

If we're emitting a camelCased event name:

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

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

```
<!-- Won't work -->
<my-component @my-event="doSomething"></my-component>
```

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

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

#Defining Custom Events

Watch a free video about Defining Custom Events on Vue School

Emitted events can be defined on the component via the `emits` option.

```
app.component('custom-form', {
  emits: ['in-focus', 'submit']
})
```

When a native event (e.g., `click`) is defined in the `emits` option, the component event will be used instead of a native event listener.

TIP

It is recommended to define all emitted events in order to better document how a component should work.

#Validate Emitted Events

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 `$emit` call and returns a boolean to indicate whether the event is valid or not.

```
app.component('custom-form', {
  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() {
      this.$emit('submit', { email, password })
    }
  }
})
```

#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`:

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

In this case, child component will expect a `title` prop and emits `update:title` event to sync:

```
const app = Vue.createApp({})

app.component('my-component', {
  props: {
    title: String
  },
  template: `
    <input
      type="text"
      :value="title"
      @input="$emit('update:title', $event.target.value)">
  `
})
<my-component v-model:title="bookTitle"></my-component>
```

1

#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:

```
<user-name
  v-model:first-name="firstName"
  v-model:last-name="lastName"
></user-name>


const app = Vue.createApp({})

app.component('user-name', {
  props: {
    firstName: String,
    lastName: String
  },
  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)">
  `
})
```

#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, however, you might also want to add your own custom modifiers.

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

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.

Notice that when the component's `created` lifecycle hook triggers, the `modelModifiers` prop contains `capitalize` and its value is `true` - due to it being set on the `v-model` binding `v-model.capitalize="bar"`.

```
<my-component v-model.capitalize="bar"></my-component>


app.component('my-component', {
  props: {
    modelValue: String,
    modelModifiers: {
      default: () => ({})
    }
```

```
  },
  template: `
    <input type="text"
      :value="modelValue"
      @input="$emit('update:modelValue', $event.target.value)">
  `,
  created() {
    console.log(this.modelModifiers) // { capitalize: true }
  }
})
```

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.

```
<div id="app">
  <my-component v-model.capitalize="myText"></my-component>
  {{ myText }}
</div>


const app = Vue.createApp({
  data() {
    return {
      myText: ''
    }
  }
})

app.component('my-component', {
  props: {
    modelValue: String,
    modelModifiers: {
      default: () => ({})
    }
  },
  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)
    }
  },
  template: `<input
    type="text"
    :value="modelValue"
    @input="emitValue">`
```

```
})

app.mount('#app')
```

For `v-model` bindings with arguments, the generated prop name will be `arg + "Modifiers"`:

```
<my-component v-model:foo.capitalize="bar"></my-component>
```

```
app.component('my-component', {
  props: ['foo', 'fooModifiers'],
  template: `
    <input type="text"
      :value="foo"
      @input="$emit('update:foo', $event.target.value)">
  `,
  created() {
    console.log(this.fooModifiers) // { capitalize: true }
  }
})
```

Slots

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

#Slot Content

Vue implements a content distribution API inspired by the Web Components spec draft, using the `<slot>` element to serve as distribution outlets for content.

This allows you to compose components like this:

```
<todo-button>
  Add todo
</todo-button>
```

Then in the template for `<todo-button>`, you might have:

```
<!-- todo-button component template -->
<button class="btn-primary">
  <slot></slot>
</button>
```

When the component renders, `<slot></slot>` will be replaced by "Add Todo".

```
<!-- rendered HTML -->
<button class="btn-primary">
  Add todo
</button>
```

Strings are just the beginning though! Slots can also contain any template code, including HTML:

```
<todo-button>
  <!-- Add a Font Awesome icon -->
```

```
  <i class="fas fa-plus"></i>
  Add todo
</todo-button>
```

Or even other components:

```
<todo-button>
  <!-- Use a component to add an icon -->
  <font-awesome-icon name="plus"></font-awesome-icon>
  Add todo
</todo-button>
```

If `<todo-button>`'s template did not contain a `<slot>` element, any content provided between its opening and closing tag would be discarded.

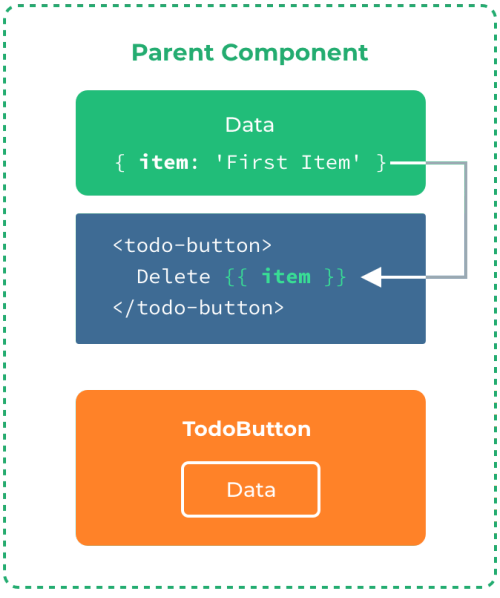
```
<!-- todo-button component template -->

<button class="btn-primary">
  Create a new item
</button>


<todo-button>
  <!-- Following text won't be rendered -->
  Add todo
</todo-button>
```

#Render Scope

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

```
<todo-button>
  Delete a {{ item.name }}
</todo-button>
```

That slot has access to the same instance properties (i.e. the same "scope") as the rest of the



template.

The slot does not have access to `<todo-button>`'s scope. For example, trying to access `action` would not work:

```
<todo-button action="delete">
  Clicking here will {{ action }} an item
  <!--
  The `action` will be undefined, because this content is passed
  _to_ <todo-button>, rather than defined _inside_ the
  <todo-button> component.
  -->
</todo-button>
```

As a rule, remember that:

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

#Fallback Content

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

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

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

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

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

```
<submit-button></submit-button>
```

will render the fallback content, "Submit":

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

But if we provide content:

```
<submit-button>
  Save
</submit-button>
```

Then the provided content will be rendered instead:

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

#Named Slots

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

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

```
  </header>
  <main>
    <!-- We want main content here -->
  </main>
  <footer>
    <!-- We want footer content here -->
  </footer>
</div>
```

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

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

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

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

  <template v-slot:default>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </template>

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

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

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

```
  </main>
  <footer>
    <p>Here's some contact info</p>
  </footer>
</div>
```

Note that `v-slot` can only be added to a `<template>` (with one exception)

#Scoped Slots

Sometimes, it's useful for slot content to have access to data only available in the child component. It's a common case when a component is used to render an array of items, and we want to be able to customize the way each item is rendered.

For example, we have a component, containing a list of todo-items.

```
app.component('todo-list', {
  data() {
    return {
      items: ['Feed a cat', 'Buy milk']
    }
  },
  template: `
    <ul>
      <li v-for="(item, index) in items">
        {{ item }}
      </li>
    </ul>
  `
})
```

We might want to replace the slot to customize it on parent component:

```
<todo-list>
  <i class="fas fa-check"></i>
  <span class="green">{{ item }}</span>
</todo-list>
```

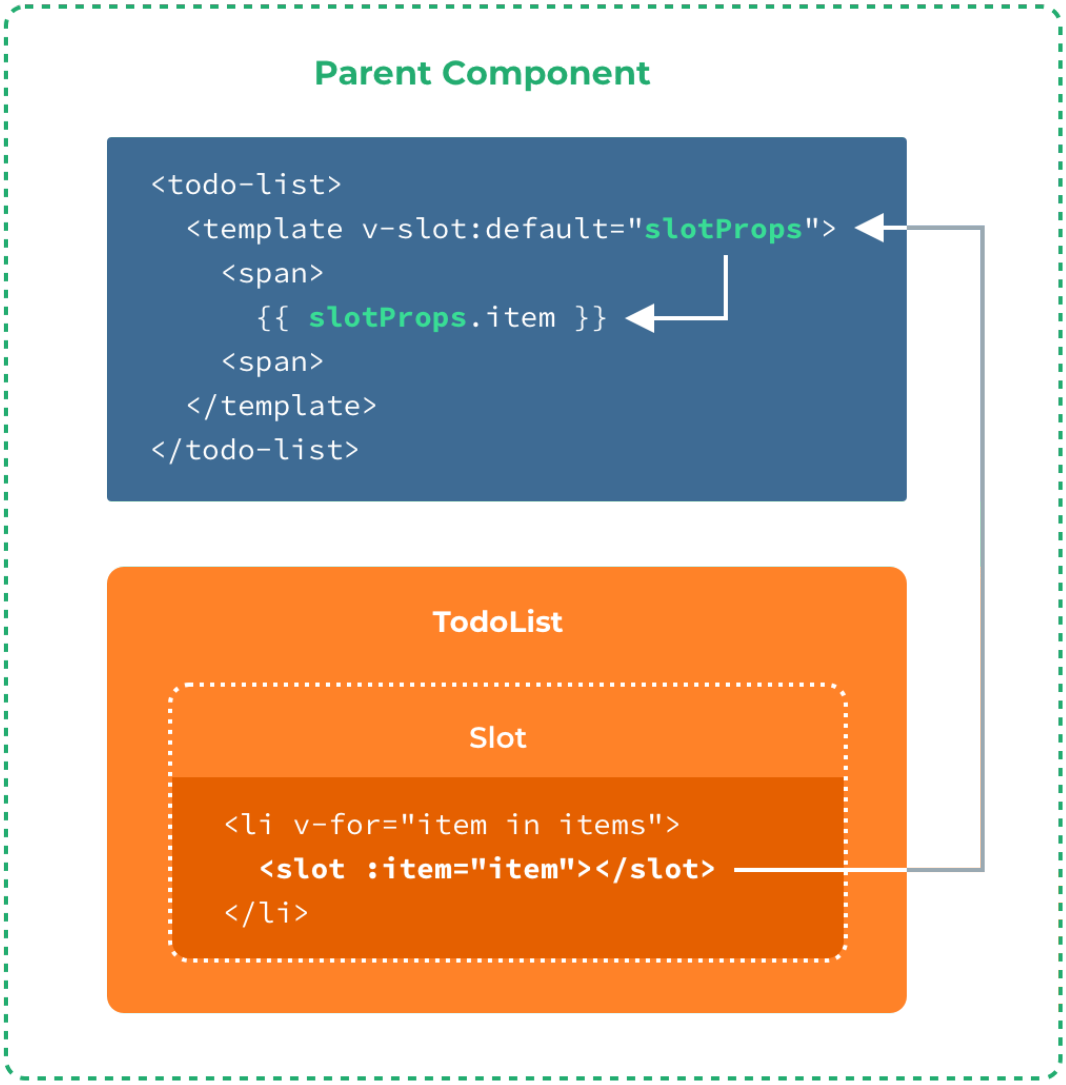
That won't work, however, because only the `<todo-list>` component has access to the `item` and we are providing the slot content from its parent.

To make `item` available to the slot content provided by the parent, we can add a `<slot>` element and bind it as an attribute:

```
<ul>
  <li v-for="( item, index ) in items">
    <slot :item="item"></slot>
  </li>
</ul>
```

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

```
<todo-list>
  <template v-slot:default="slotProps">
    <i class="fas fa-check"></i>
    <span class="green">{{ slotProps.item }}</span>
  </template>
</todo-list>
```



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

#Abbreviated Syntax for Lone Default Slots

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

```
<todo-list v-slot:default="slotProps">
  <i class="fas fa-check"></i>
  <span class="green">{{ slotProps.item }}</span>
</todo-list>
```

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

```
<todo-list v-slot="slotProps">
  <i class="fas fa-check"></i>
  <span class="green">{{ slotProps.item }}</span>
</todo-list>
```

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

```
<!-- INVALID, will result in warning -->
<todo-list v-slot="slotProps">
  <todo-list v-slot:default="slotProps">
    <i class="fas fa-check"></i>
    <span class="green">{{ slotProps.item }}</span>
  </todo-list>
  <template v-slot:other="otherSlotProps">
    slotProps is NOT available here
  </template>
</todo-list>
```

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

```
<todo-list>
  <template v-slot:default="slotProps">
    <i class="fas fa-check"></i>
    <span class="green">{{ slotProps.item }}</span>
  </template>

  <template v-slot:other="otherSlotProps">
    ...
  </template>
</todo-list>
```

#Destructuring Slot Props

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

```
function (slotProps) {
  // ... slot content ...
}
```

That means the value of `v-slot` can actually accept any valid JavaScript expression that can appear in the argument position of a function definition. So you can also use ES2015 destructuring to pull out specific slot props, like so:

```
<todo-list v-slot="{ item }">
  <i class="fas fa-check"></i>
  <span class="green">{{ item }}</span>
</todo-list>
```

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

```
<todo-list v-slot="{ item: todo }">
  <i class="fas fa-check"></i>
  <span class="green">{{ todo }}</span>
</todo-list>
```

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

```
<todo-list v-slot="{ item = 'Placeholder' }">
  <i class="fas fa-check"></i>
  <span class="green">{{ item }}</span>
</todo-list>
```

#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>
</base-layout>
```

#Named Slots Shorthand

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

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

  <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>
</base-layout>
```

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

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

<todo-list #="{ item }">
  <i class="fas fa-check"></i>
  <span class="green">{{ item }}</span>
</todo-list>
```

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

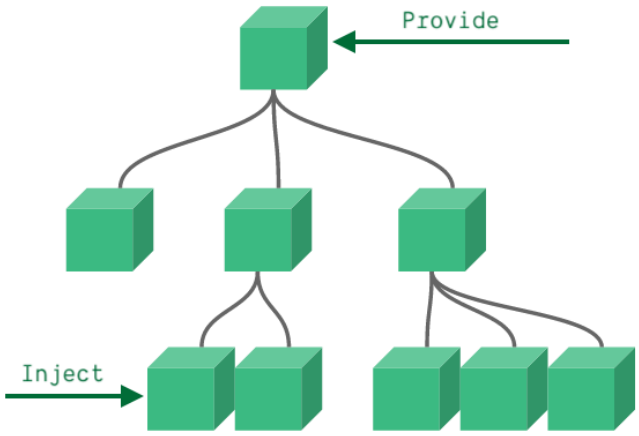
```
<todo-list #default="{ item }">
  <i class="fas fa-check"></i>
  <span class="green">{{ item }}</span>
</todo-list>
```

Provide / inject

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

Usually, when we need to pass data from the parent to child component, we use props. Imagine the structure where you have some deeply nested components and you only need something from the parent component in the deep nested child. In this case, you still need to pass the prop down the whole component chain which might be annoying.

For such cases, we can use the `provide` and `inject` pair. Parent components can serve as dependency provider for all its children, regardless how deep the component hierarchy is. This feature works on two parts: parent component has a `provide` option to provide data and child component has an `inject` option to start using this data.



For example, if we have a hierarchy like this:

```
Root
└── TodoList
    ├── TodoItem
    └── TodoListFooter
        ├── ClearTodosButton
        └── TodoListStatistics
```

If we want to pass the length of todo-items directly to `TodoListStatistics`, we would pass the prop down the hierarchy: `TodoList` -> `TodoListFooter` -> `TodoListStatistics`. With provide/inject approach, we can do this directly:

```
const app = Vue.createApp({})

app.component('todo-list', {
  data() {
    return {
      todos: ['Feed a cat', 'Buy tickets']
    }
  },
  provide: {
    user: 'John Doe'
  },
  template: `
```

```
    <div>
      {{ todos.length }}
      <!-- rest of the template -->
    </div>
  `
})

app.component('todo-list-statistics', {
  inject: ['user'],
  created() {
    console.log(`Injected property: ${this.user}`) // > Injected property: John Doe
  }
})
```

However, this won't work if we try to provide some component instance property here:

```
app.component('todo-list', {
  data() {
    return {
      todos: ['Feed a cat', 'Buy tickets']
    }
  },
  provide: {
    todoLength: this.todos.length // this will result in error 'Cannot read property 'length' of undefined`
  },
  template: `
    ...
  `
})
```

To access component instance properties, we need to convert `provide` to be a function returning an object

```
app.component('todo-list', {
  data() {
    return {
      todos: ['Feed a cat', 'Buy tickets']
    }
  },
  provide() {
    return {
      todoLength: this.todos.length
    }
  },
  template: `
    ...
  `
})
```

This allows us to more safely keep developing that component, without fear that we might change/remove something that a child component is relying on. The interface between these components remains clearly defined, just as with props.

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

- parent components don’t need to know which descendants use the properties it provides
- child components don’t need to know where injected properties are coming from

#Working with reactivity

In the example above, if we change the list of `todos`, this change won't be reflected in the injected `todoLength` property. This is because `provide/inject` bindings are not reactive by default. We can change this behavior by passing a `ref` property or `reactive` object to `provide`. In our case, if we wanted to react to changes in the ancestor component, we would need to assign a Composition API `computed` property to our provided `todoLength`:

```
app.component('todo-list', {
  // ...
  provide() {
    return {
      todoLength: Vue.computed(() => this.todos.length)
    }
  }
})
```

In this, any change to `todos.length` will be reflected correctly in the components, where `todoLength` is injected. Read more about `reactive` provide/inject in the Composition API section

Dynamic & Async Components

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

#Dynamic Components with `keep-alive`

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

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

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

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

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

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

Now the Posts tab maintains its state (the selected post) even when it's not rendered.

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

#Async Components

In large applications, we may need to divide the app into smaller chunks and only load a component from the server when it's needed. To make that possible, Vue has a `defineAsyncComponent` method:

```
const app = Vue.createApp({})

const AsyncComp = Vue.defineAsyncComponent(
  () =>
    new Promise((resolve, reject) => {
      resolve({
        template: '<div>I am async!</div>'
      })
    })
)

app.component('async-example', AsyncComp)
```

As you can see, this method accepts a factory function returning a `Promise`. 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.

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

```
import { defineAsyncComponent } from 'vue'

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

app.component('async-component', AsyncComp)
```

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

```
import { createApp, defineAsyncComponent } from 'vue'

createApp({
  // ...
  components: {
    AsyncComponent: defineAsyncComponent(() =>
      import('./components/AsyncComponent.vue')
    )
  }
})
```

#Using with Suspense

Async components are suspensible by default. This means 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.
You can check the list of available options in the API Reference

Template refs

This page assumes you've already read the Components Basics. Read that first if you are new to components.
Despite the existence of props and events, sometimes you might still need to directly access a child component in JavaScript. To achieve this you can assign a reference ID to the child component or HTML element using the `ref` attribute. For example:

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

This may be useful when you want to, for example, programmatically focus this input on component mount:

```
const app = Vue.createApp({})

app.component('base-input', {
  template: `
    <input ref="input" />
  `,
  methods: {
    focusInput() {
      this.$refs.input.focus()
    }
  },
  mounted() {
    this.focusInput()
  }
})
```

Also, you can add another `ref` to the component itself and use it to trigger `focusInput` event from the parent component:

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


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

WARNING

`$refs` are only populated after the component has been rendered. It is only meant as an escape hatch for direct child manipulation - you should avoid accessing `$refs` from within templates or computed properties.

Handling Edge Cases

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

Note

All the features on this page document the handling of edge cases, meaning unusual situations that sometimes require bending Vue's rules a little. Note however, that they all have disadvantages or situations where they could be dangerous. These are noted in each case, so keep them in mind when deciding to use each feature.

#Controlling Updates

Thanks to Vue's Reactivity system, it always knows when to update (if you use it correctly). There are edge cases, however, when you might want to force an update, despite the fact that no reactive data has changed. Then there are other cases when you might want to prevent unnecessary updates.

#Forcing an Update

If you find yourself needing to force an update in Vue, in 99.99% of cases, you've made a mistake somewhere. For example, you may be relying on state that isn't tracked by Vue's reactivity system, e.g. with `data` property added after component creation.

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

#Cheap Static Components with `v-once`

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

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

TIP

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

Transitions & Animation

TBD

Transitions & Animation

Mixins

#Basics

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

Example:

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

// define an app that uses this mixin
const app = Vue.createApp({
  mixins: [myMixin]
})

app.mount('#mixins-basic') // => "hello from mixin!"
```

#Option Merging

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

For example, data objects undergo a recursive merge, with the component's data taking priority in cases of conflicts.

```
const myMixin = {
  data() {
    return {
      message: 'hello',
      foo: 'abc'
    }
  }
}

const app = Vue.createApp({
  mixins: [myMixin],
  data() {
    return {
      message: 'goodbye',
      bar: 'def'
    }
  },
```

```
  created() {
    console.log(this.$data) // => { message: "goodbye", foo: "abc", bar: "def" }
  }
})
```

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

```
const myMixin = {
  created() {
    console.log('mixin hook called')
  }
}

const app = Vue.createApp({
  mixins: [myMixin],
  created() {
    console.log('component hook called')
  }
})

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

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

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

const app = Vue.createApp({
  mixins: [myMixin],
  methods: {
    bar() {
      console.log('bar')
    },
    conflicting() {
      console.log('from self')
    }
  }
})

const vm = app.mount('#mixins-basic')
```

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

#Global Mixin

You can also apply a mixin globally for a Vue application:

```
const app = Vue.createApp({
  myOption: 'hello!'
})

// inject a handler for `myOption` custom option
app.mixin({
  created() {
    const myOption = this.$options.myOption
    if (myOption) {
      console.log(myOption)
    }
  }
})

app.mount('#mixins-global') // => "hello!"
```

Use with caution! Once you apply a mixin globally, it will affect every component instance created afterwards in the given app (for example, child components):

```
const app = Vue.createApp({
  myOption: 'hello!'
})

// inject a handler for `myOption` custom option
app.mixin({
  created() {
    const myOption = this.$options.myOption
    if (myOption) {
      console.log(myOption)
    }
  }
})

// add myOption also to child component
app.component('test-component', {
  myOption: 'hello from component!'
})

app.mount('#mixins-global')

// => "hello!"
// => "hello from component!"
```

In most cases, you should only use it for custom option handling like demonstrated in the example above. It's also a good idea to ship them as Plugins to avoid duplicate application.

#Custom Option Merge Strategies

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

```
const app = Vue.createApp({})

app.config.optionMergeStrategies.customOption = (toVal, fromVal) => {
  // return mergedVal
}
```

The merge strategy receives the value of that option defined on the parent and child instances as the first and second arguments, respectively. Let's try to check what do we have in these parameters when we use a mixin:

```
const app = Vue.createApp({
  custom: 'hello!'
})

app.config.optionMergeStrategies.custom = (toVal, fromVal) => {
  console.log(fromVal, toVal)
  // => "goodbye!", undefined
  // => "hello", "goodbye!"
  return fromVal || toVal
}

app.mixin({
  custom: 'goodbye!',
  created() {
    console.log(this.$options.custom) // => "hello!"
  }
})
```

As you can see, in the console we have `toVal` and `fromVal` printed first from the mixin and then from the `app`. We always return `fromVal` if it exists, that's why `this.$options.custom` is set to `hello!` in the end. Let's try to change a strategy to always return a value from the child instance:

```
const app = Vue.createApp({
  custom: 'hello!'
})

app.config.optionMergeStrategies.custom = (toVal, fromVal) => toVal || fromVal

app.mixin({
  custom: 'goodbye!',
  created() {
    console.log(this.$options.custom) // => "goodbye!"
  }
})
```

#Precautions

In Vue 2, mixins were the primary tool to abstract parts of component logic into reusable chunks. However, they have a few issues:

- Mixins are conflict-prone: Since properties from each feature are merged into the same component, you still have to know about every other feature to avoid property name conflicts and for debugging.

- Reusability is limited: we cannot pass any parameters to the mixin to change its logic which reduces their flexibility in terms of abstracting logic

To address these issues, we added a new way to organize code by logical concerns: the Composition API.

Custom Directives

#Intro

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. Note that in Vue, the primary form of code reuse and abstraction is components - however, there may be cases where you need some low-level DOM access on plain elements, and this is where custom directives would still be useful. An example would be focusing on an input element, like this one:

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

```
const app = Vue.createApp({})
// Register a global custom directive called `v-focus`
app.directive('focus', {
  // When the bound element is mounted into the DOM...
  mounted(el) {
    // Focus the element
    el.focus()
  }
})
```

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

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

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

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

#Hook Functions

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

- `beforeMount`: called when the directive is first bound to the element and before parent component is mounted. This is where you can do one-time setup work.
- `mounted`: called when the bound element's parent component is mounted.
- `beforeUpdate`: called before the containing component's VNode is updated

Note
We'll cover VNodes in more detail later, when we discuss render functions.

- `updated`: called after the containing component's VNode and the VNodes of its children have updated.
- `beforeUnmount`: called before the bound element's parent component is unmounted
- `unmounted`: called only once, when the directive is unbound from the element and the parent component is unmounted.

You can check the arguments passed into these hooks (i.e. `el`, `binding`, `vnode`, and `prevVnode`) in Custom Directive API

#Dynamic Directive Arguments

Directive arguments can be dynamic. For example, in `v-mydirective:[argument]="value"`, the `argument` can be updated based on data properties in our component instance! This makes our custom directives flexible for use throughout our application.

Let's say you want to make a custom directive that allows you to pin elements to your page using fixed positioning. We could create a custom directive where the value updates the vertical positioning in pixels, like this:

```
<div id="dynamic-arguments-example" class="demo">
  <p>Scroll down the page</p>
  <p v-pin="200">Stick me 200px from the top of the page</p>
</div>


const app = Vue.createApp({})

app.directive('pin', {
  mounted(el, binding) {
    el.style.position = 'fixed'
    // binding.value is the value we pass to directive - in this case, it's 200
    el.style.top = binding.value + 'px'
  }
})

app.mount('#dynamic-arguments-example')
```

This would pin the element 200px from the top of the page. But what happens if we run into a scenario when we need to pin the element from the left, instead of the top? Here's where a dynamic argument that can be updated per component instance comes in very handy:

```
<div id="dynamicexample">
  <h3>Scroll down inside this section ↓</h3>
  <p v-pin:[direction]="200">I am pinned onto the page at 200px to the left.</p>
</div>

const app = Vue.createApp({
  data() {
    return {
      direction: 'right'
    }
  }
})

app.directive('pin', {
  mounted(el, binding) {
    el.style.position = 'fixed'
    // binding.arg is an argument we pass to directive
    const s = binding.arg || 'top'
    el.style[s] = binding.value + 'px'
  }
})

app.mount('#dynamic-arguments-example')
```

Our custom directive is now flexible enough to support a few different use cases. To make it even more dynamic, we can also allow to modify a bound value. Let's create an additional property `pinPadding` and bind it to the `<input type="range">`

```
<div id="dynamicexample">
  <h2>Scroll down the page</h2>
  <input type="range" min="0" max="500" v-model="pinPadding">
  <p v-pin:[direction]="pinPadding">Stick me {{ pinPadding + 'px' }} from the {{ direction }} of the
page</p>
</div>

const app = Vue.createApp({
  data() {
    return {
      direction: 'right',
      pinPadding: 200
    }
  }
})
```

Now let's extend our directive logic to recalculate the distance to pin on component update:

```
app.directive('pin', {
  mounted(el, binding) {
    el.style.position = 'fixed'
    const s = binding.arg || 'top'
    el.style[s] = binding.value + 'px'
  },
  updated(el, binding) {
    const s = binding.arg || 'top'
    el.style[s] = binding.value + 'px'
  }
})
```

#Function Shorthand

In previous example, you may want the same behavior on `mounted` and `updated`, but don't care about the other hooks. You can do it by passing the callback to directive:

```
app.directive('pin', (el, binding) => {
  el.style.position = 'fixed'
  const s = binding.arg || 'top'
  el.style[s] = binding.value + 'px'
})
```

#Object Literals

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

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

```
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 directive will always apply to component's root node, similarly to non-prop attributes.

```
<my-component v-demo="test"></my-component>
```

1

```
app.component('my-component', {
  template: `
    <div> // v-demo directive will be applied here
      <span>My component content</span>
    </div>
  `
})
```

Unlike attributes, directives can't be passed to a different element with `v-bind="$attrs"`.

With fragments support, components can potentially have more than one root nodes. When applied to a multi-root component, directive will be ignored and the warning will be thrown.

Teleport

Learn how to use teleport with a free lesson on Vue School

Vue encourages us to build our UIs by encapsulating UI and related behavior into components. We can nest them inside one another to build a tree that makes up an application UI.

However, sometimes a part of a component's template belongs to this component logically, while from a technical point of view, it would be preferable to move this part of the template somewhere else in the DOM, outside of the Vue app.

A common scenario for this is creating a component that includes a full-screen modal. In most cases, you'd want the modal's logic to live within the component, but the positioning of the modal quickly becomes difficult to solve through CSS, or requires a change in component composition.

Consider the following HTML structure.

```
<body>
  <div style="position: relative;">
    <h3>Tooltips with Vue 3 Teleport</h3>
    <div>
      <modal-button></modal-button>
    </div>
  </div>
</body>
```

Let's take a look at `modal-button`.

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

```
const app = Vue.createApp({});

app.component('modal-button', {
  template: `
    <button @click="modalOpen = true">
      Open full screen modal!
    </button>

    <div v-if="modalOpen" class="modal">
      <div>
        I'm a modal!
        <button @click="modalOpen = false">
          Close
        </button>
      </div>
    </div>
  `,
  data() {
    return {
      modalOpen: false
    }
  }
})
```

When using this component inside the initial HTML structure, we can see a problem - the modal is being rendered inside the deeply nested `div` and the `position: absolute` of the modal takes the parent relatively positioned `div` as reference.

Teleport provides a clean way to allow us to control under which parent in our DOM we want a piece of HTML to be rendered, without having to resort to global state or splitting this into two components.

Let's modify our `modal-button` to use `<teleport>` and tell Vue "teleport this HTML to the "body" tag".

```
app.component('modal-button', {
  template: `
    <button @click="modalOpen = true">
      Open full screen modal! (With teleport!)
    </button>

    <teleport to="body">
      <div v-if="modalOpen" class="modal">
        <div>
          I'm a teleported modal!
          (My parent is "body")
          <button @click="modalOpen = false">
            Close
          </button>
        </div>
      </div>
    </teleport>
  `,
  data() {
    return {
      modalOpen: false
    }
  }
})
```

As a result, once we click the button to open the modal, Vue will correctly render the modal's content as a child of the `body` tag.

#Using with Vue components

If `<teleport>` contains a Vue component, it will remain a logical child component of the `<teleport>`'s parent:

```
const app = Vue.createApp({
  template: `
    <h1>Root instance</h1>
    <parent-component />
  `
})

app.component('parent-component', {
  template: `
```

```
    <h2>This is a parent component</h2>
    <teleport to="#endofbody">
      <child-component name="John" />
    </teleport>
  `
})

app.component('child-component', {
  props: ['name'],
  template: `
    <div>Hello, {{ name }}</div>
  `
})
```

In this case, even when `child-component` is rendered in the different place, it will remain a child of `parent-component` and will receive a `name` prop from it.

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.

#Using multiple teleports on the same target

A common use case scenario would be a reusable `<Modal>` component of which there might be multiple instances 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.

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

<!-- result-->
<div id="modals">
  <div>A</div>
  <div>B</div>
</div>
```

You can check `<teleport>` component options in the API reference.

Render Functions

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.

Let's dive into an example where a `render()` function would be practical. Say we want to generate anchored headings:

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

Anchored headings are used very frequently, we should create a component:

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

The component must generate a heading based on the `level` prop, and we quickly arrive at this:

```
const app = Vue.createApp({})

app.component('anchored-heading', {
  template: `
    <h1 v-if="level === 1">
      <slot></slot>
    </h1>
    <h2 v-else-if="level === 2">
      <slot></slot>
    </h2>
    <h3 v-else-if="level === 3">
      <slot></slot>
    </h3>
    <h4 v-else-if="level === 4">
      <slot></slot>
    </h4>
    <h5 v-else-if="level === 5">
      <slot></slot>
    </h5>
    <h6 v-else-if="level === 6">
      <slot></slot>
    </h6>
  `,
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})
```

This template doesn't feel great. It's not only verbose, but we're duplicating `<slot></slot>` for every heading level. And when we add the anchor element, we have to again duplicate it in every `v-if/v-else-if` branch.

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

```
const app = Vue.createApp({})

app.component('anchored-heading', {
  render() {
    const { h } = Vue

    return h(
      'h' + this.level, // tag name
      {}, // props/attributes
      this.$slots.default() // array of children
    )
  },
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})
```

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

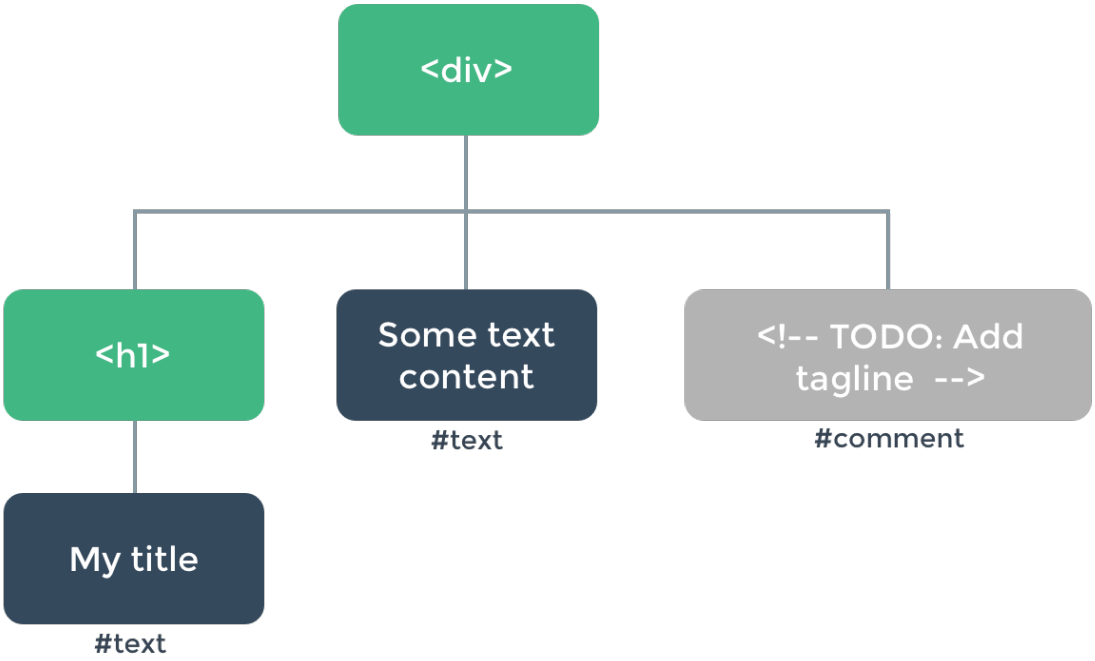
#The DOM tree

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

```
<div>
  <h1>My title</h1>
  Some text content
  <!-- TODO: Add tagline -->
</div>
```

When a browser reads this code, it builds a tree of "DOM nodes" to help it keep track of everything.

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



Every element is a node. Every piece of text is a node. Even comments are nodes! Each node can have children (i.e. each node can contain other nodes).

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

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

Or in a render function:

```
render() {
  return Vue.h('h1', {}, this.blogTitle)
}
```

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

#The Virtual DOM tree

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

```
return Vue.h('h1', {}, this.blogTitle)
```

What is the `h()` function returning? It's not exactly a real DOM element. It returns a plain object which contains information describing to Vue what kind of node it should render on the page, including descriptions of any child nodes. We call this node description a "virtual node", usually abbreviated to VNode. "Virtual DOM" is what we call the entire tree of VNodes, built by a tree of Vue components.

#`h()` Arguments

The `h()` function is a utility to create VNodes. It could perhaps more accurately be named `createVNode()`, but it's called `h()` due to frequent use and for brevity. It accepts three arguments:

```
// @returns {VNode}
h(
```

```
  // {String | Object | Function } tag
  // An HTML tag name, a component or an async component.
  // Using function returning null would render a comment.
  //
  // Required.
  'div',

  // {Object} props
  // An object corresponding to the attributes, props and events
  // we would use in a template.
  //
  // Optional.
  {},

  // {String | Array | Object} children
  // Children VNodes, built using `h()`,
  // or using strings to get 'text VNodes' or
  // an object with slots.
  //
  // Optional.
  [
    'Some text comes first.',
    h('h1', 'A headline'),
    h(MyComponent, {
      someProp: 'foobar'
    })
  ]
)
```

#Complete Example

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

```
const app = Vue.createApp({})

/** Recursively get text from children nodes */
function getChildrenTextContent(children) {
  return children
    .map(node => {
      return typeof node.children === 'string'
        ? node.children
        : Array.isArray(node.children)
        ? getChildrenTextContent(node.children)
        : ''
    })
    .join('')
}

app.component('anchored-heading', {
  render() {
    // create kebab-case id from the text contents of the children
```

```
    const headingId = getChildrenTextContent(this.$slots.default())
      .toLowerCase()
      .replace(/\W+/g, '-') // replace non-word characters with dash
      .replace(/(^-|-$)/g, '') // remove leading and trailing dashes

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

#Constraints

#VNodes Must Be Unique

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

```
render() {
  const myParagraphVNode = Vue.h('p', 'hi')
  return Vue.h('div', [
    // Yikes - duplicate VNodes!
    myParagraphVNode, myParagraphVNode
  ])
}
```

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

```
render() {
  return Vue.h('div',
    Array.apply(null, { length: 20 }).map(() => {
      return Vue.h('p', 'hi')
    })
  )
}
```

#Replacing Template Features with Plain JavaScript

#`v-if` and `v-for`

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

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

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

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

#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:

```
props: ['modelValue'],
render() {
  return Vue.h(SomeComponent, {
    modelValue: this.modelValue,
    'onUpdate:modelValue': value => this.$emit('update:modelValue', value)
  })
}
```

#v-on

We have to provide a proper prop name for the event handler, e.g., to handle `click` events, the prop name would be `onClick`.

```
render() {
  return Vue.h('div', {
    onClick: $event => console.log('clicked', $event.target)
  })
}
```

#Event Modifiers

For the `.passive`, `.capture`, and `.once` event modifiers, they can be concatenated after event name using camel case.

For example:

```
render() {
  return Vue.h('input', {
    onClickCapture: this.doThisInCapturingMode,
    onKeyupOnce: this.doThisOnce,
    onMouseoverOnceCapture: this.doThisOnceInCapturingMode,
  })
```

}

For all other event and key modifiers, no special API is necessary, because we can use event methods in the handler:

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

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

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

#Slots

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

```
render() {
  // `<div><slot></slot></div>`
  return Vue.h('div', {}, this.$slots.default())
}
```

```
props: ['message'],
render() {
```

```
  // `<div><slot :text="message"></slot></div>`
  return Vue.h('div', {}, this.$slots.default({
    text: this.message
  }))
}
```

To pass slots to a child component using render functions:

```
render() {
  // `<div><child v-slot="props"><span>{{ props.text }}</span></child></div>`
  return Vue.h('div', [
    Vue.h(
      Vue.resolveComponent('child'),
      {},
      // pass `slots` as the children object
      // in the form of { name: props => VNode | Array<VNode> }
      {
        default: (props) => Vue.h('span', props.text)
      }
    )
  ])
}
```

#JSX

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

```
Vue.h(
  Vue.resolveComponent('anchored-heading'),
  {
    level: 1
  },
  {
    default: () => [Vue.h('span', 'Hello'), ' world!']
  }
)
```

Especially when the template version is so concise in comparison:

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

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

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

const app = createApp({
  render() {
    return (
      <AnchoredHeading level={1}>
        <span>Hello</span> world!
      </AnchoredHeading>
    )
  }
})
```

```
app.mount('#demo')
```

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

Plugins

Plugins are self-contained code that usually add global-level functionality to Vue. It is either an `object` that exposes an `install()` method, or a `function`.

There is no strictly defined scope for a plugin, but common scenarios where plugins are useful include:

1. Add some global methods or properties, e.g. vue-custom-element.
2. Add one or more global assets: directives/filters/transitions etc. (e.g. vue-touch).
3. Add some component options by global mixin (e.g. vue-router).
4. Add some global instance methods by attaching them to `config.globalProperties`.
5. A library that provides an API of its own, while at the same time injecting some combination of the above (e.g. vue-router).

#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` ready strings.

Whenever this plugin is added to an application, the `install` method will be called if it is an object. If it is a `function`, the function itself will be called. In both cases, it will receive two parameters - the `app` object resulting from Vue's `createApp`, and the options passed in by the user.

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 make a function to translate keys available to the whole application, so we will expose it using `app.config.globalProperties`.

This function will receive a `key` string, which we will use to look up the translated string in the user-provided options.

```
// 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)
    }
```

```
  }
}
```

We will assume that our users will pass in an object containing the translated keys in the `options` parameter when they use the plugin. Our `$translate` function will take a string such as `greetings.hello`, look inside the user provided configuration and return the translated value - in this case, `Bonjour!`

Ex:

```
greetings: {
  hello: 'Bonjour!',
}
```

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['i18n']` into their components and access the object.

Additionally, since we have access to the `app` object, all other capabilities like using `mixin` and `directive` are available to the plugin. To learn more about `createApp` and the application instance, check out the Application API documentation.

```
// 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)

    app.directive('my-directive', {
      mounted (el, binding, vnode, oldVnode) {
        // some logic ...
      }
      ...
    })
```

```
    app.mixin({
      created() {
        // some logic ...
      }
      ...
    })
  }
}
```

#Using a Plugin

After a Vue app has been initialized with `createApp()`, you can add a plugin to your application by calling the `use()` method.

We will use the `i18nPlugin` we created in the Writing a Plugin section for demo purposes.

The `use()` method takes two parameters. The first one is the plugin to be installed, in this case `i18nPlugin`.

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

The second parameter is optional, and depends on each particular plugin. In the case of the demo `i18nPlugin`, it is an object with the translated strings.

INFO

If you are using third party plugins such as `Vuex` or `Vue Router`, always check the documentation to know what that particular plugin expects to receive as a second parameter.

```
import { createApp } from 'vue'
import Root from './App.vue'
import i18nPlugin from './plugins/i18n'

const app = createApp(Root)
const i18nStrings = {
  greetings: {
    hi: 'Hallo!'
  }
}

app.use(i18nPlugin, i18nStrings)
app.mount('#app')
```

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

Advanced Guides

Reactivity in Depth

Now it's time to take a deep dive! One of Vue's most distinct features is the unobtrusive reactivity system. Models are proxied 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?

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.

If you put the number two in the first cell, and the number 3 in the second and asked for the SUM, the spreadsheet would give it to you. No surprises there. But if you update that first number, the SUM automagically updates too.

JavaScript doesn't usually work like this -- If we were to write something comparable in JavaScript:

```
var val1 = 2
var val2 = 3
var sum = val1 + val2

// sum
// 5

val1 = 3

// sum
// 5
```

If we update the first value, the sum is not adjusted.

So how would we do this in JavaScript?

- Detect when there's a change in one of the values
- Track the function that changes it
- Trigger the function so it can update the final value

#How Vue Tracks These Changes

When you pass a plain JavaScript object to an application or component instance as its `data` option, Vue will walk through all of its properties and convert them to Proxies using a handler with getters and setters. This is an ES6-only feature, but we offer a version of Vue 3 that uses the older `Object.defineProperty` to support IE browsers. Both have the same surface API, but the Proxy version is slimmer and offers improved performance.

That was rather quick and requires some knowledge of Proxies to understand! So let's dive in a bit. There's a lot of literature on Proxies, but what you really need to know is that a Proxy is an object that encases another object or function and allows you to intercept it.

We use it like this: new Proxy(target, handler)

```
const dinner = {
  meal: 'tacos'
```

```
}

const handler = {
  get(target, prop) {
    return target[prop]
  }
}

const proxy = new Proxy(dinner, handler)
console.log(proxy.meal)

// tacos
```

Ok, so far, we're just wrapping that object and returning it. Cool, but not that useful yet. But watch this, we can also intercept this object while we wrap it in the Proxy. This interception is called a trap.

```
const dinner = {
  meal: 'tacos'
}

const handler = {
  get(target, prop) {
    console.log('intercepted!')
    return target[prop]
  }
}

const proxy = new Proxy(dinner, handler)
console.log(proxy.meal)

// intercepted!
// tacos
```

Beyond a console log, we could do anything here we wish. We could even not return the real value if we wanted to. This is what makes Proxies so powerful for creating APIs.

Furthermore, there's another feature Proxies offer us. Rather than just returning the value like this: `target[prop]`, we could take this a step further and use a feature called `Reflect`, which allows us to do proper `this` binding. It looks like this:

```
const dinner = {
  meal: 'tacos'
}

const handler = {
  get(target, prop, receiver) {
    return Reflect.get(...arguments)
  }
}

const proxy = new Proxy(dinner, handler)
console.log(proxy.meal)
```

// tacos

We mentioned before that in order to have an API that updates a final value when something changes, we're going to have to set new values when something changes. We do this in the handler, in a function called `track`, where we pass in the `target` and `key`.

```
const dinner = {
  meal: 'tacos'
}

const handler = {
  get(target, prop, receiver) {
    track(target, prop)
    return Reflect.get(...arguments)
  }
}

const proxy = new Proxy(dinner, handler)
console.log(proxy.meal)
```

// tacos

Finally, we also set new values when something changes. For this, we're going to set the changes on our new proxy, by triggering those changes:

```
const dinner = {
  meal: 'tacos'
}

const handler = {
  get(target, prop, receiver) {
    track(target, prop)
    return Reflect.get(...arguments)
  },
  set(target, key, value, receiver) {
    trigger(target, key)
    return Reflect.set(...arguments)
  }
}

const proxy = new Proxy(dinner, handler)
console.log(proxy.meal)
```

// tacos

Remember this list from a few paragraphs ago? Now we have some answers to how Vue handles these changes:

- ~~Detect when there's a change in one of the values~~: we no longer have to do this, as Proxies allow us to intercept it
- Track the function that changes it: We do this in a getter within the proxy, called `effect`
- Trigger the function so it can update the final value: We do in a setter within the proxy, called `trigger`

The proxied object is invisible to the user, but under the hood they enable Vue to perform dependency-tracking and change-notification when properties are accessed or modified. As of

Vue 3, our reactivity is now available in a separate package. One caveat is that browser consoles format differently when converted data objects are logged, so you may want to install vue-devtools for a more inspection-friendly interface.

#Proxied Objects

Vue internally tracks all objects that have been made reactive, so it always returns the same proxy for the same object.

When a nested object is accessed from a reactive proxy, that object is also converted into a proxy before being returned:

```
const handler = {
  get(target, prop, receiver) {
    track(target, prop)
    const value = Reflect.get(...arguments)
    if (isObject(value)) {
      return reactive(value)
    } else {
      return value
    }
  }
  // ...
}
```

#Proxy vs. original identity

The use of Proxy does introduce a new caveat to be aware with: the proxied object is not equal to the original object in terms of identity comparison (`===`). For example:

```
const obj = {}
const wrapped = new Proxy(obj, handlers)

console.log(obj === wrapped) // false
```

The original and the wrapped version will behave the same in most cases, but be aware that they will fail operations that rely on strong identity comparisons, such as `.filter()` or `.map()`. This caveat is unlikely to come up when using the options API, because all reactive state is accessed from `this` and guaranteed to already be proxies.

However, when using the composition API to explicitly create reactive objects, the best practice is to never hold a reference to the original raw object and only work with the reactive version:

```
const obj = reactive({
  count: 0
}) // no reference to original
```

#Watchers

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

When you pass an object to a component instance as data, Vue converts it to a proxy. This proxy enables Vue to perform dependency-tracking and change-notification when properties are accessed or modified. Each property is considered a dependency.

After the first render, a component would have tracked a list of dependencies — the properties it accessed during the render. Conversely, the component becomes a subscriber to each of these properties. When a proxy intercepts a set operation, the property will notify all of its subscribed components to re-render.

Reactivity Fundamentals

#Declaring Reactive State

To create a reactive state from a JavaScript object, we can use a `reactive` method:

```
import { reactive } from 'vue'

// reactive state
const state = reactive({
  count: 0
})
```

`reactive` is the equivalent of the `Vue.observable()` API in Vue 2.x, renamed to avoid confusion with RxJS observables. Here, the returned state is a reactive object. The reactive conversion is "deep" - it affects all nested properties of the passed object.

The essential use case for reactive state in Vue is that we can use it during render. Thanks to dependency tracking, the view automatically updates when reactive state changes.

This is the very essence of Vue's reactivity system. When you return an object from `data()` in a component, it is internally made reactive by `reactive()`. The template is compiled into a render function that makes use of these reactive properties.

You can learn more about `reactive` in the Basic Reactivity API's section

#Creating Standalone Reactive Values as `refs`

Imagine the case where we have a standalone primitive value (for example, a string) and we want to make it reactive. Of course, we could make an object with a single property equal to our string, and pass it to `reactive`. Vue has a method that will do the same for us - it's a `ref`:

```
import { ref } from 'vue'

const count = ref(0)
```

`ref` will return a reactive and mutable object that serves as a reactive reference to the internal value it is holding - that's where the name comes from. This object contains the only one property named `value`:

```
import { ref } from 'vue'

const count = ref(0)
console.log(count.value) // 0

count.value++
console.log(count.value) // 1
```

#Ref Unwrapping

When a ref is returned as a property on the render context (the object returned from setup()) and accessed in the template, it automatically unwraps to the inner value. There is no need to append `.value` in the template:

```
<template>
  <div>
    <span>{{ count }}</span>
    <button @click="count ++">Increment count</button>
  </div>
</template>
```

```
<script>
  import { ref } from 'vue'
  export default {
    setup() {
      const count = ref(0)
      return {
        count
      }
    }
  }
</script>
```

#Access in Reactive Objects

When a `ref` is accessed or mutated as a property of a reactive object, it automatically unwraps to the inner value 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
console.log(count.value) // 1
```

Ref unwrapping only happens when nested inside a reactive `Object`. There is no unwrapping performed when the ref is accessed from an `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)
```

#Destructuring Reactive State

When we want to use a few properties of the large reactive object, it could be tempting to use ES6 destructuring to get properties we want:

```
import { reactive } from 'vue'

const book = reactive({
  author: 'Vue Team',
  year: '2020',
  title: 'Vue 3 Guide',
```

```
  description: 'You are reading this book right now ;)',
  price: 'free'
})

let { author, title } = book
```

Unfortunately, with such a destructuring the reactivity for both properties would be lost. For such a case, we need to convert our reactive object to a set of refs. These refs will retain the reactive connection to the source object:

```
import { reactive, toRefs } from 'vue'

const book = reactive({
  author: 'Vue Team',
  year: '2020',
  title: 'Vue 3 Guide',
  description: 'You are reading this book right now ;)',
  price: 'free'
})

let { author, title } = toRefs(book)

title.value = 'Vue 3 Detailed Guide' // we need to use .value as title is a ref now
console.log(book.title) // 'Vue 3 Detailed Guide'
```

You can learn more about `refs` in the Refs API section

#Prevent Mutating Reactive Objects with `readonly`

Sometimes we want to track changes of the reactive object (`ref` or `reactive`) but we also want prevent changing it from a certain place of the application. For example, when we have a provided reactive object, we want to prevent mutating it where it's injected. To do so, we can create a readonly proxy to the original object:

```
import { reactive, readonly } from 'vue'

const original = reactive({ count: 0 })

const copy = readonly(original)

// mutating original will trigger watchers relying on the copy
original.count++

// mutating the copy will fail and result in a warning
copy.count++ // warning: "Set operation on key 'count' failed: target is readonly."
```

Computed and Watch

This section uses single-file component syntax for code examples

#Computed values

Sometimes we need state that depends on other state - in Vue this is handled with component computed properties. To directly create a computed value, we can use the `computed` method: it takes a getter function and returns an immutable reactive ref object for the returned value from the getter.

```
const count = ref(1)
const plusOne = computed(() => count.value + 1)

console.log(plusOne.value) // 2

plusOne.value++ // error
```

Alternatively, it can take an object with `get` and `set` functions to create a writable ref object.

```
const count = ref(1)
const plusOne = computed({
  get: () => count.value + 1,
  set: val => {
    count.value = val - 1
  }
})

plusOne.value = 1
console.log(count.value) // 0
```

#watchEffect

To apply and automatically re-apply a side effect based on reactive state, we can use the `watchEffect` method. It runs a function immediately while reactively tracking its dependencies and re-runs it whenever the dependencies are changed.

```
const count = ref(0)

watchEffect(() => console.log(count.value))
// -> logs 0

setTimeout(() => {
  count.value++
  // -> logs 1
}, 100)
```

#Stopping the Watcher

When `watchEffect` is called during a component's setup() function or lifecycle hooks, the watcher is linked to the component's lifecycle and will be automatically stopped when the component is unmounted.

In other cases, it returns a stop handle which can be called to explicitly stop the watcher:

```
const stop = watchEffect(() => {
  /* ... */
})
```

```
// later
stop()
```

#Side Effect Invalidation

Sometimes the watched effect function will perform asynchronous side effects that need to be cleaned up when it is invalidated (i.e state changed before the effects can be completed). The effect function receives an `onInvalidate` function that can be used to register an invalidation callback. This invalidation callback is called when:

- the effect is about to re-run
- the watcher is stopped (i.e. when the component is unmounted if `watchEffect` is used inside `setup()` or lifecycle hooks)

```
watchEffect(onInvalidate => {
  const token = performAsyncOperation(id.value)
  onInvalidate(() => {
    // id has changed or watcher is stopped.
    // invalidate previously pending async operation
    token.cancel()
  })
})
```

We are registering the invalidation callback via a passed-in function instead of returning it from the callback because the return value is important for async error handling. It is very common for the effect function to be an async function when performing data fetching:

```
const data = ref(null)
watchEffect(async onInvalidate => {
  onInvalidate(() => {...}) // we register cleanup function before Promise resolves
  data.value = await fetchData(props.id)
})
```

An async function implicitly returns a Promise, but the cleanup function needs to be registered immediately before the Promise resolves. In addition, Vue relies on the returned Promise to automatically handle potential errors in the Promise chain.

#Effect Flush Timing

Vue's reactivity system buffers invalidated effects and flushes them asynchronously to avoid unnecessary duplicate invocation when there are many state mutations happening in the same "tick". Internally, a component's `update` function is also a watched effect. When a user effect is queued, it is by default invoked before all component `update` effects:

```
<template>
  <div>{{ count }}</div>
</template>

<script>
  export default {
    setup() {
      const count = ref(0)

      watchEffect(() => {
        console.log(count.value)
      })

      return {
```

```
      count
    }
  }
}
</script>
```

In this example:

- The count will be logged synchronously on initial run.
- When `count` is mutated, the callback will be called before the component has updated.

In cases where a watcher effect needs to be re-run after component updates, we can pass an additional `options` object with the `flush` option (default is `'pre'`):

```
// fire after component updates so you can access the updated DOM
// Note: this will also defer the initial run of the effect until the
// component's first render is finished.
watchEffect(
  () => {
    /* ... */
  },
  {
    flush: 'post'
  }
)
```

The `flush` option also accepts `'sync'`, which forces the effect to always trigger synchronously. This is however inefficient and should be rarely needed.

#Watcher Debugging

The `onTrack` and `onTrigger` options can be used to debug a watcher's behavior.

- `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 a debugger event which contains information on the dependency in question. It is recommended to place a `debugger` statement in these callbacks to interactively inspect the dependency:

```
watchEffect(
  () => {
    /* side effect */
  },
  {
    onTrigger(e) {
      debugger
    }
  }
)
```

`onTrack` and `onTrigger` only work in development mode.

#watch

The `watch` API is the exact equivalent of the component watch property. `watch` requires watching a specific data source and applies side effects in a separate callback function. It also is lazy by default - i.e. the callback is only called when the watched source has changed.

- Compared to watchEffect, `watch` allows us to:
  - Perform the side effect lazily;
  - Be more specific about what state should trigger the watcher to re-run;
  - Access both the previous and current value of the watched state.

#Watching a Single Source

A watcher data source can either be a getter function that returns a value, or directly a `ref`:

```
// watching a getter
const state = reactive({ count: 0 })
watch(
  () => state.count,
  (count, prevCount) => {
    /* ... */
  }
)

// directly watching a ref
const count = ref(0)
watch(count, (count, prevCount) => {
  /* ... */
})
```

#Watching Multiple Sources

A watcher can also watch multiple sources at the same time using an array:

```
watch([fooRef, barRef], ([foo, bar], [prevFoo, prevBar]) => {
  /* ... */
})
```

#Shared Behavior with `watchEffect`

`watch` shares behavior with `watchEffect` in terms of manual stoppage, side effect invalidation (with `onInvalidate` passed to the callback as the 3rd argument instead), flush timing and debugging.

Composition API
Introduction
#Why Composition API?

Note

Reaching this far in the documentation, you should already be familiar with both the basics of Vue and creating components.

Creating Vue components allows us to extract repeatable parts of the interface coupled with its functionality into reusable pieces of code. This alone can get our application pretty far in terms of maintainability and flexibility. However, our collective experience has proved that this alone might not be enough, especially when your application is getting really big – think several hundred components. When dealing with such large applications, sharing and reusing code becomes especially important.

Let's imagine that in our app, we have a view to show a list of repositories of a certain user. On top of that, we want to apply search and filter capabilities. Our component handling this view could look like this:

```
// src/components/UserRepositories.vue

export default {
  components: { RepositoriesFilters, RepositoriesSortBy, RepositoriesList },
  props: {
    user: {
      type: String,
      required: true
    }
  },
  data () {
    return {
      repositories: [], // 1
      filters: { ... }, // 3
      searchQuery: '' // 2
    }
  },
  computed: {
    filteredRepositories () { ... }, // 3
    repositoriesMatchingSearchQuery () { ... }, // 2
  },
  watch: {
    user: 'getUserRepositories' // 1
  },
  methods: {
    getUserRepositories () {
      // using `this.user` to fetch user repositories
    }, // 1
    updateFilters () { ... }, // 3
  },
  mounted () {
    this.getUserRepositories() // 1
  }
```

}

This component has several responsibilities:

1. Getting repositories from a presumedly external API for that user name and refreshing it whenever the user changes
2. Searching for repositories using a `searchQuery` string
3. Filtering repositories using a `filters` object

Organizing logics with component's options (`data`, `computed`, `methods`, `watch`) works in most cases. However, when our components get bigger, the list of logical concerns also grows. This can lead to components that are hard to read and understand, especially for people who didn't write them in the first place.

Example presenting a large component where its logical concerns are grouped by colors.

Such fragmentation is what makes it difficult to understand and maintain a complex component. The separation of options obscures the underlying logical concerns. In addition, when working on a single logical concern, we have to constantly "jump" around option blocks for the relevant code.

It would be much nicer if we could collocate code related to the same logical concern. And this is exactly what the Composition API enables us to do.

#Basics of Composition API

Now that we know the why we can get to the how. To start working with the Composition API we first need a place where we can actually use it. In a Vue component, we call this place the `setup`.

#`setup` Component Option

The new `setup` component option is executed before the component is created, once the `props` are resolved, and serves as the entry point for composition API's.

WARNING

Because the component instance is not yet created when `setup` is executed, there is no `this` inside a `setup` option. This means, with the exception of `props`, you won't be able to access any properties declared in the component – local state, computed properties or methods.

The `setup` option should be a function that accepts `props` and `context` which we will talk about later. Additionally, everything that we return from `setup` will be exposed to the rest of our component (computed properties, methods, lifecycle hooks and so on) as well as to the component's template.

Let's add `setup` to our component:

```
// src/components/UserRepositories.vue

export default {
  components: { RepositoriesFilters, RepositoriesSortBy, RepositoriesList },
  props: {
    user: {
      type: String,
      required: true
    }
  },
  setup(props) {
    console.log(props) // { user: '' }

    return {} // anything returned here will be available for the rest of the component
```

```
  }
  // the "rest" of the component
}
```

Now let's start with extracting the first logical concern (marked as "1" in the original snippet).

1. Getting repositories from a presumedly external API for that user name and refreshing it whenever the user changes

We will start with the most obvious parts:

- The list of repositories
- The function to update the list of repositories
- Returning both the list and the function so they are accessible by other component options

```
// src/components/UserRepositories.vue `setup` function
import { fetchUserRepositories } from '@/api/repositories'

// inside our component
setup (props) {
  let repositories = []
  const getUserRepositories = async () => {
    repositories = await fetchUserRepositories(props.user)
  }

  return {
    repositories,
    getUserRepositories // functions returned behave the same as methods
  }
}
```

This is our starting point, except it's not working yet because our `repositories` variable is not reactive. This means from a user's perspective, the repository list would remain empty. Let's fix that!

#Reactive Variables with `ref`

In Vue 3.0 we can make any variable reactive anywhere with a new `ref` function, like this:

```
import { ref } from 'vue'

const counter = ref(0)
```

`ref` takes the argument and returns it wrapped within an object with a `value` property, which can then be used to access or mutate the value of the reactive variable:

```
import { ref } from 'vue'

const counter = ref(0)

console.log(counter) // { value: 0 }
console.log(counter.value) // 0

counter.value++
console.log(counter.value) // 1
```

Wrapping values inside an object might seem unnecessary but is required to keep the behavior unified across different data types in JavaScript. That's because in JavaScript, primitive types like `Number` or `String` are passed by value, not by reference:

Having a wrapper object around any value allows us to safely pass it across our whole app without worrying about losing its reactivity somewhere along the way.

Note

In other words, `ref` creates a Reactive Reference to our value. The concept of working with References will be used often throughout the Composition API.

Back to our example, let's create a reactive `repositories` variable:

```
// src/components/UserRepositories.vue `setup` function
import { fetchUserRepositories } from '@/api/repositories'
import { ref } from 'vue'

// in our component
setup (props) {
  const repositories = ref([])
  const getUserRepositories = async () => {
    repositories.value = await fetchUserRepositories(props.user)
  }

  return {
    repositories,
    getUserRepositories
  }
}
```

Done! Now whenever we call `getUserRepositories`, `repositories` will be mutated and the view will be updated to reflect the change. Our component should now look like this:

```
// src/components/UserRepositories.vue
import { fetchUserRepositories } from '@/api/repositories'
import { ref } from 'vue'

export default {
  components: { RepositoriesFilters, RepositoriesSortBy, RepositoriesList },
  props: {
    user: {
      type: String,
      required: true
    }
  },
  setup (props) {
    const repositories = ref([])
    const getUserRepositories = async () => {
      repositories.value = await fetchUserRepositories(props.user)
    }

    return {
      repositories,
      getUserRepositories
    }
  },
```

```
  data () {
    return {
      filters: { ... }, // 3
      searchQuery: '' // 2
    }
  },
  computed: {
    filteredRepositories () { ... }, // 3
    repositoriesMatchingSearchQuery () { ... }, // 2
  },
  watch: {
    user: 'getUserRepositories' // 1
  },
  methods: {
    updateFilters () { ... }, // 3
  },
  mounted () {
    this.getUserRepositories() // 1
  }
}
```

We have moved several pieces of our first logical concern into the `setup` method, nicely put close to each other. What's left is calling `getUserRepositories` in the `mounted` hook and setting up a watcher to do that whenever the `user` prop changes.

We will start with the lifecycle hook.

#Lifecycle Hook Registration Inside `setup`

To make Composition API feature-complete compared to Options API, we also need a way to register lifecycle hooks inside `setup`. This is possible thanks to several new functions exported from Vue. Lifecycle hooks on composition API have the same name as for Options API but are prefixed with `on`: i.e. `mounted` would look like `onMounted`.

These functions accept a callback that will be executed when the hook is called by the component.

Let's add it to our `setup` function:

```
// src/components/UserRepositories.vue `setup` function
import { fetchUserRepositories } from '@/api/repositories'
import { ref, onMounted } from 'vue'

// in our component
setup (props) {
  const repositories = ref([])
  const getUserRepositories = async () => {
    repositories.value = await fetchUserRepositories(props.user)
  }

  onMounted(getUserRepositories) // on `mounted` call `getUserRepositories`

  return {
    repositories,
```

```
    getUserRepositories
  }
}
```

Now we need to react to the changes made to the `user` prop. For that we will use the standalone `watch` function.

#Reacting to Changes with `watch`

Just like how we set up a watcher on the `user` property inside our component using the `watch` option, we can do the same using the `watch` function imported from Vue. It accepts 3 arguments:

- A Reactive Reference or getter function that we want to watch
- A callback
- Optional configuration options

Here's a quick look at how it works.

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

const counter = ref(0)
watch(counter, (newValue, oldValue) => {
  console.log('The new counter value is: ' + counter.value)
})
```

Whenever `counter` is modified, for example `counter.value = 5`, the watch will trigger and execute the callback (second argument) which in this case will log `'The new counter value is: 5'` into our console.

Below is the Options API equivalent:

```
export default {
  data() {
    return {
      counter: 0
    }
  },
  watch: {
    counter(newValue, oldValue) {
      console.log('The new counter value is: ' + this.counter)
    }
  }
}
```

For more details on `watch`, refer to our in-depth guide.

Let's now apply it to our example:

```
// src/components/UserRepositories.vue `setup` function
import { fetchUserRepositories } from '@/api/repositories'
import { ref, onMounted, watch, toRefs } from 'vue'

// in our component
setup (props) {
  // using `toRefs` to create a Reactive Reference to the `user` property of props
  const { user } = toRefs(props)

  const repositories = ref([])
```

```
  const getUserRepositories = async () => {
    // update `props.user` to `user.value` to access the Reference value
    repositories.value = await fetchUserRepositories(user.value)
  }

  onMounted(getUserRepositories)

  // set a watcher on the Reactive Reference to user prop
  watch(user, getUserRepositories)

  return {
    repositories,
    getUserRepositories
  }
}
```

You probably have noticed the use of `toRefs` at the top of our `setup`. This is to ensure our watcher will react to changes made to the `user` prop.

With those changes in place, we've just moved the whole first logical concern into a single place. We can now do the same with the second concern – filtering based on `searchQuery`, this time with a computed property.

#Standalone `computed` properties

Similar to `ref` and `watch`, computed properties can also be created outside of a Vue component with the `computed` function imported from Vue. Let's get back to our counter example:

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

const counter = ref(0)
const twiceTheCounter = computed(() => counter.value * 2)

counter.value++
console.log(counter.value) // 1
console.log(twiceTheCounter.value) // 2
```

Here, the `computed` function returns a read-only Reactive Reference to the output of the getter-like callback passed as the first argument to `computed`. In order to access the value of the newly-created computed variable, we need to use the `.value` property just like with `ref`.

Let's move our search functionality into `setup`:

```
// src/components/UserRepositories.vue `setup` function
import { fetchUserRepositories } from '@/api/repositories'
import { ref, onMounted, watch, toRefs, computed } from 'vue'

// in our component
setup (props) {
  // using `toRefs` to create a Reactive Reference to the `user` property of props
  const { user } = toRefs(props)

  const repositories = ref([])
  const getUserRepositories = async () => {
```

```
    // update `props.user` to `user.value` to access the Reference value
    repositories.value = await fetchUserRepositories(user.value)
  }

  onMounted(getUserRepositories)

  // set a watcher on the Reactive Reference to user prop
  watch(user, getUserRepositories)

  const searchQuery = ref('')
  const repositoriesMatchingSearchQuery = computed(() => {
    return repositories.value.filter(
      repository => repository.name.includes(searchQuery.value)
    )
  })

  return {
    repositories,
    getUserRepositories,
    searchQuery,
    repositoriesMatchingSearchQuery
  }
}
```

We could do the same for other logical concerns but you might be already asking the question – Isn't this just moving the code to the `setup` option and making it extremely big? Well, that's true. That's why before moving on with the other responsibilities, we will first extract the above code into a standalone composition function. Let's start with creating `useUserRepositories`:

```
// src/composables/useUserRepositories.js

import { fetchUserRepositories } from '@/api/repositories'
import { ref, onMounted, watch } from 'vue'

export default function useUserRepositories(user) {
  const repositories = ref([])
  const getUserRepositories = async () => {
    repositories.value = await fetchUserRepositories(user.value)
  }

  onMounted(getUserRepositories)
  watch(user, getUserRepositories)

  return {
    repositories,
    getUserRepositories
  }
}
```

And then the searching functionality:

```
// src/composables/useRepositoryNameSearch.js
```

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

export default function useRepositoryNameSearch(repositories) {
  const searchQuery = ref('')
  const repositoriesMatchingSearchQuery = computed(() => {
    return repositories.value.filter(repository => {
      return repository.name.includes(searchQuery.value)
    })
  })

  return {
    searchQuery,
    repositoriesMatchingSearchQuery
  }
}
```

Now having those two functionalities in separate files, we can start using them in our component. Here's how this can be done:

```
// src/components/UserRepositories.vue
import useUserRepositories from '@/composables/useUserRepositories'
import useRepositoryNameSearch from '@/composables/useRepositoryNameSearch'
import { toRefs } from 'vue'

export default {
  components: { RepositoriesFilters, RepositoriesSortBy, RepositoriesList },
  props: {
    user: {
      type: String,
      required: true
    }
  },
  setup (props) {
    const { user } = toRefs(props)

    const { repositories, getUserRepositories } = useUserRepositories(user)

    const {
      searchQuery,
      repositoriesMatchingSearchQuery
    } = useRepositoryNameSearch(repositories)

    return {
      // Since we don't really care about the unfiltered repositories
      // we can expose the filtered results under the `repositories` name
      repositories: repositoriesMatchingSearchQuery,
      getUserRepositories,
      searchQuery,
    }
  },
```

```
  data () {
    return {
      filters: { ... }, // 3
    }
  },
  computed: {
    filteredRepositories () { ... }, // 3
  },
  methods: {
    updateFilters () { ... }, // 3
  }
}
```

At this point you probably already know the drill, so let's skip to the end and migrate the leftover filtering functionality. We don't really need to get into the implementation details as it's not the point of this guide.

```
// src/components/UserRepositories.vue
import { toRefs } from 'vue'
import useUserRepositories from '@/composables/useUserRepositories'
import useRepositoryNameSearch from '@/composables/useRepositoryNameSearch'
import useRepositoryFilters from '@/composables/useRepositoryFilters'

export default {
  components: { RepositoriesFilters, RepositoriesSortBy, RepositoriesList },
  props: {
    user: {
      type: String,
      required: true
    }
  },
  setup(props) {
    const { user } = toRefs(props)

    const { repositories, getUserRepositories } = useUserRepositories(user)

    const {
      searchQuery,
      repositoriesMatchingSearchQuery
    } = useRepositoryNameSearch(repositories)

    const {
      filters,
      updateFilters,
      filteredRepositories
    } = useRepositoryFilters(repositoriesMatchingSearchQuery)

    return {
      // Since we don't really care about the unfiltered repositories
      // we can expose the end results under the `repositories` name
      repositories: filteredRepositories,
```

```
    getUserRepositories,
    searchQuery,
    filters,
    updateFilters
  }
 }
}
```

And we are done!

Keep in mind that we've only scratched the surface of Composition API and what it allows us to do. To learn more about it, refer to the in-depth guide.

Setup

This section uses single-file component syntax for code examples

This guide assumes that you have already read the Composition API Introduction and Reactivity Fundamentals. Read that first if you are new to Composition API.

#Arguments

When using the `setup` function, it will take two arguments:

1. props
2. context

Let's dive deeper into how each argument can be used.

#Props

The first argument in the `setup` function is the `props` argument. Just as you would expect in a standard component, `props` inside of a `setup` function are reactive and will be updated when new props are passed in.

```
// MyBook.vue

export default {
 props: {
  title: String
 },
 setup(props) {
  console.log(props.title)
 }
}
```

WARNING

However, because `props` are reactive, you cannot use ES6 destructuring because it will remove props reactivity.

If you need to destructure your props, you can do this by utilizing the toRefs inside of the `setup` function:

```
// MyBook.vue

import { toRefs } from 'vue'

setup(props) {
```

```
        const { title } = toRefs(props)

        console.log(title.value)
}
```

If `title` is an optional prop, it could be missing from `props`. In that case, `toRefs` won't create a ref for `title`. Instead you'd need to use `toRef`:

```
// MyBook.vue

import { toRef } from 'vue'

setup(props) {
        const title = toRef(props, 'title')

        console.log(title.value)
}
```

#Context

The second argument passed to the `setup` function is the `context`. The `context` is a normal JavaScript object that exposes three component properties:

```
// MyBook.vue

export default {
  setup(props, context) {
    // Attributes (Non-reactive object)
    console.log(context.attrs)

    // Slots (Non-reactive object)
    console.log(context.slots)

    // Emit Events (Method)
    console.log(context.emit)
  }
}
```

The `context` object is a normal JavaScript object, i.e., it is not reactive, this means you can safely use ES6 destructuring on `context`.

```
// MyBook.vue
export default {
  setup(props, { attrs, slots, emit }) {
    ...
  }
}
```

`attrs` and `slots` are stateful objects that are always updated when the component itself is updated. This means you should avoid destructuring them and always reference properties as `attrs.x` or `slots.x`. Also note that unlike `props`, `attrs` and `slots` are not reactive. If you intend to apply side effects based on `attrs` or `slots` changes, you should do so inside an `onUpdated` lifecycle hook.

#Accessing Component Properties

When `setup` is executed, the component instance has not been created yet. As a result, you will only be able to access the following properties:

- props
- attrs
- slots
- emit

In other words, you will not have access to the following component options:

- data
- computed
- methods

#Usage with Templates

If `setup` returns an object, the properties on the object can be accessed in the component's template, as well as the properties of the `props` passed into `setup`:

```
<!-- MyBook.vue -->
<template>
  <div>{{ collectionName }}: {{ readersNumber }} {{ book.title }}</div>
</template>

<script>
  import { ref, reactive } from 'vue'

  export default {
    props: {
      collectionName: String
    },
    setup(props) {
      const readersNumber = ref(0)
      const book = reactive({ title: 'Vue 3 Guide' })

      // expose to template
      return {
        readersNumber,
        book
      }
    }
  }
</script>
```

Note that refs returned from `setup` are automatically unwrapped when accessed in the template so you shouldn't use `.value` in templates.

#Usage with Render Functions

`setup` can also return a render function which can directly make use of the reactive state declared in the same scope:

```
// MyBook.vue

import { h, ref, reactive } from 'vue'
```

```
export default {
  setup() {
    const readersNumber = ref(0)
    const book = reactive({ title: 'Vue 3 Guide' })
    // Please note that we need to explicitly expose ref value here
    return () => h('div', [readersNumber.value, book.title])
  }
}
```

#Usage of `this`

Inside `setup()`, `this` won't be a reference to the current active instance Since `setup()` is called before other component options are resolved, `this` inside `setup()` will behave quite differently from `this` in other options. This might cause confusions when using `setup()` along other Options API.

Lifecycle Hooks

This guide assumes that you have already read the Composition API Introduction and Reactivity Fundamentals. Read that first if you are new to Composition API.

Watch a free video about Lifecycle Hooks on Vue Mastery

You can access a component's lifecycle hook by prefixing the lifecycle hook with "on".

The following table contains how the lifecycle hooks are invoked inside of setup():

Options API	Hook inside `setup`
`beforeCreate`	Not needed*
`created`	Not needed*
`beforeMount`	`onBeforeMount`
`mounted`	`onMounted`
`beforeUpdate`	`onBeforeUpdate`
`updated`	`onUpdated`
`beforeUnmount`	`onBeforeUnmount`
`unmounted`	`onUnmounted`
`errorCaptured`	`onErrorCaptured`
`renderTracked`	`onRenderTracked`
`renderTriggered`	`onRenderTriggered`

TIP

Because `setup` is run around the `beforeCreate` and `created` lifecycle hooks, you do not need to explicitly define them. In other words, any code that would be written inside those hooks should be written directly in the `setup` function.

These functions accept a callback that will be executed when the hook is called by the component:

```
// MyBook.vue

export default {
  setup() {
    // mounted
    onMounted(() => {
      console.log('Component is mounted!')
    })
  }
}
```

Provide / Inject

This guide assumes that you have already read Provide / Inject, Composition API Introduction, and Reactivity Fundamentals.

We can use provide / inject with the Composition API as well. Both can only be called during `setup()` with a current active instance.

#Scenario Background

Let's assume that we want to rewrite the following code, which contains a `MyMap` component that provides a `MyMarker` component with the user's location, using the Composition API.

```
<!-- src/components/MyMap.vue -->
<template>
  <MyMarker />
</template>

<script>
import MyMarker from './MyMarker.vue'

export default {
  components: {
    MyMarker
  },
  provide: {
    location: 'North Pole',
    geolocation: {
      longitude: 90,
      latitude: 135
    }
  }
}
```

```
</script>
<!-- src/components/MyMarker.vue -->
<script>
export default {
  inject: ['location', 'geolocation']
}
</script>
```

#Using Provide

When using `provide` in `setup()`, we start by explicitly importing the method from `vue`. This allows us to define each property with its own invocation of `provide`.

The `provide` function allows you to define the property through two parameters:

1. The property's name (`<String>` type)
2. The property's value

Using our `MyMap` component, our provided values can be refactored as the following:

```
<!-- src/components/MyMap.vue -->
<template>
  <MyMarker />
</template>

<script>
import { provide } from 'vue'
import MyMarker from './MyMarker.vue

export default {
  components: {
    MyMarker
  },
  setup() {
    provide('location', 'North Pole')
    provide('geolocation', {
      longitude: 90,
      latitude: 135
    })
  }
}
</script>
```

#Using Inject

When using `inject` in `setup()`, we also need to explicitly import it from `vue`. Once we do so, this allows us to invoke it to define how we want to expose it to our component.

The `inject` function takes two parameters:

1. The name of the property to inject
2. A default value (Optional)

Using our `MyMarker` component, we can refactor it with the following code:

```
<!-- src/components/MyMarker.vue -->
<script>
import { inject } from 'vue'

export default {
  setup() {
    const userLocation = inject('location', 'The Universe')
    const userGeolocation = inject('geolocation')

    return {
      userLocation,
      userGeolocation
    }
  }
}
</script>
```

#Reactivity

#Adding Reactivity

To add reactivity between provided and injected values, we can use a ref or reactive when providing a value.

Using our `MyMap` component, our code can be updated as follows:

```
<!-- src/components/MyMap.vue -->
<template>
  <MyMarker />
</template>

<script>
import { provide, reactive, ref } from 'vue'
import MyMarker from './MyMarker.vue'

export default {
  components: {
    MyMarker
  },
  setup() {
    const location = ref('North Pole')
    const geolocation = reactive({
      longitude: 90,
      latitude: 135
    })

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

Now, if anything changes in either property, the `MyMarker` component will automatically be updated as well!

#Mutating Reactive Properties

When using reactive provide / inject values, it is recommended to keep any mutations to reactive properties inside of the provider whenever possible.

For example, in the event we needed to change the user's location, we would ideally do this inside of our `MyMap` component.

```
<!-- src/components/MyMap.vue -->
<template>
  <MyMarker />
</template>

<script>
import { provide, reactive, ref } from 'vue'
import MyMarker from './MyMarker.vue

export default {
  components: {
    MyMarker
  },
  setup() {
    const location = ref('North Pole')
    const geolocation = reactive({
      longitude: 90,
      latitude: 135
    })

    provide('location', location)
    provide('geolocation', geolocation)

    return {
      location
    }
  },
  methods: {
    updateLocation() {
      this.location = 'South Pole'
    }
  }
}
</script>
```

However, there are times where we need to update the data inside of the component where the data is injected. In this scenario, we recommend providing a method that is responsible for mutating the reactive property.

```
<!-- src/components/MyMap.vue -->
<template>
  <MyMarker />
</template>
```

```
<script>
import { provide, reactive, ref } from 'vue'
import MyMarker from './MyMarker.vue

export default {
  components: {
    MyMarker
  },
  setup() {
    const location = ref('North Pole')
    const geolocation = reactive({
      longitude: 90,
      latitude: 135
    })

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

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

<!-- src/components/MyMarker.vue -->
<script>
import { inject } from 'vue'

export default {
  setup() {
    const userLocation = inject('location', 'The Universe')
    const userGeolocation = inject('geolocation')
    const updateUserLocation = inject('updateLocation')

    return {
      userLocation,
      userGeolocation,
      updateUserLocation
    }
  }
}
</script>
```

Finally, we recommend using `readonly` on provided property if you want to ensure that the data passed through `provide` cannot be mutated by the injected component.

```
<!-- src/components/MyMap.vue -->
```

```
<template>
  <MyMarker />
</template>

<script>
import { provide, reactive, readonly, ref } from 'vue'
import MyMarker from './MyMarker.vue

export default {
  components: {
    MyMarker
  },
  setup() {
    const location = ref('North Pole')
    const geolocation = reactive({
      longitude: 90,
      latitude: 135
    })

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

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

Template Refs

This section uses single-file component syntax for code examples

This guide assumes that you have already read the Composition API Introduction and Reactivity Fundamentals. Read that first if you are new to Composition API.

When using the Composition API, the concept of reactive refs and template refs are unified. In order to obtain a reference to an in-template element or component instance, we can declare a ref as usual and return it from setup():

```
<template>
  <div ref="root">This is a root element</div>
</template>

<script>
  import { ref, onMounted } from 'vue'

  export default {
    setup() {
```

```
    const root = ref(null)

    onMounted(() => {
      // the DOM element will be assigned to the ref after initial render
      console.log(root.value) // <div>This is a root element</div>
    })

    return {
      root
    }
  }
}
</script>
```

Here we are exposing `root` on the render context and binding it to the div as its ref via `ref="root"`. In the Virtual DOM patching algorithm, if a VNode's `ref` key corresponds to a ref on the render context, the VNode's corresponding element or component instance will be assigned to the value of that ref. This is performed during the Virtual DOM mount / patch process, so template refs will only get assigned values after the initial render.

Refs used as templates refs behave just like any other refs: they are reactive and can be passed into (or returned from) composition functions.

#Usage with JSX

```
export default {
  setup() {
    const root = ref(null)

    return () =>
      h('div', {
        ref: root
      })

    // with JSX
    return () => <div ref={root} />
  }
}
```

#Usage inside `v-for`

Composition API template refs do not have special handling when used inside `v-for`. Instead, use function refs to perform custom handling:

```
<template>
  <div v-for="(item, i) in list" :ref="el => { if (el) divs[i] = el }">
    {{ item }}
  </div>
</template>

<script>
  import { ref, reactive, onBeforeUpdate } from 'vue'

  export default {
```

```
  setup() {
    const list = reactive([1, 2, 3])
    const divs = ref([])

    // make sure to reset the refs before each update
    onBeforeUpdate(() => {
      divs.value = []
    })

    return {
      list,
      divs
    }
  }
}
</script>
```

Rendering Mechanisms and Optimizations

This page is not required reading in order to learn how to use Vue well, but it provides more information, should you be curious how rendering works under the hood.

#Virtual DOM

Now that we know how watchers are updating the components, you might ask how those changes eventually make it to the DOM! Perhaps you've heard of the Virtual DOM before, many frameworks including Vue use this paradigm to make sure our interfaces reflect the changes we're updating in JavaScript effectively

Here is the DOM
We make a copy in JavaScript called the Virtual DOM
We do this because touching the DOM with JavaScript is computationally expensive.
While performing updates in JavaScript is cheap,
Finding the required DOM nodes and updating them with JS is expensive
So we batch calls, and change the DOM all at once.
The virtual DOM in is a lightweight JavaScript object, created by this render function
It takes three arguments: the element, an object with data, props, attrs and more, and an array
The array is where we pass in the children, which have all these arguments too
Here's the text in the div
And it's child, the ul
Now in turn it's children, the lis
If we need to update the list items, we do so in javascript
And only then do we update the actual DOM
The Virtual DOM allows us to make performant update our UIs!

The Virtual DOM in is a lightweight JavaScript object, created by a render function. It takes three arguments: the element, an object with data, props, attrs and more, and an array. The array is where we pass in the children, which have all these arguments too, and then they can have children and so on, until we build a full tree of elements.

If we need to update the list items, we do so in JavaScript, using the reactivity we mentioned earlier. We then make all the changes to the JavaScript copy, the virtual DOM, and perform a diff between this and the actual DOM. Only then do we make our updates to just what has changed. The Virtual DOM allows us to make performant updates to our UIs!

Change Detection Caveats in Vue 2

This page applies only to Vue 2.x and below, and assumes you've already read the Reactivity Section. Please read that section first.

Due to limitations in JavaScript, there are types of changes that Vue cannot detect. However, there are ways to circumvent them to preserve reactivity.

#For Objects

Vue cannot detect property addition or deletion. Since Vue performs the getter/setter conversion process during instance initialization, a property must be present in the `data` object in order for Vue to convert it and make it reactive. For example:

```
var vm = new Vue({
  data: {
    a: 1
  }
})
// `vm.a` is now reactive

vm.b = 2
// `vm.b` is NOT reactive
```

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

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

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

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

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

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

#For Arrays

Vue cannot detect the following changes to an array:

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

For example:

```
var vm = new Vue({
  data: {
    items: ['a', 'b', 'c']
  }
})
vm.items[1] = 'x' // is NOT reactive
vm.items.length = 2 // is NOT reactive
```

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

```
// Vue.set
```

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


// Array.prototype.splice
vm.items.splice(indexOfItem, 1, newValue)
```

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

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

To deal with caveat 2, you can use `splice`:

```
vm.items.splice(newLength)
```

#Declaring Reactive Properties

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

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

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

There are technical reasons behind this restriction - it eliminates a class of edge cases in the dependency tracking system, and also makes component instances play nicer with type checking systems. But there is also an important consideration in terms of code maintainability: the `data` object is like the schema for your component's state. Declaring all reactive properties upfront makes the component code easier to understand when revisited later or read by another developer.

#Async Update Queue

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

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

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

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

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

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

Since `$nextTick()` returns a promise, you can achieve the same as the above using the new ES2017 async/await syntax:

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

本文档来源于官方文档，方便离线阅读或者打印。
有疑问或建议请加微信：zwdg789
2020.11.09