Props

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

Props Declaration

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

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

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

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

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

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

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

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

```
  title: String,
  likes: Number
})
```

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

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

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

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

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

More details: Typing Component Props TS

Reactive Props Destructure 3.5+

Vue's reactivity system tracks state usage based on property access. E.g. when you access `props.foo` in a computed getter or a watcher, the `foo` prop gets tracked as a dependency.

So, given the following code:

```js
const { foo } = defineProps(['foo'])

watchEffect(() => {
  // runs only once before 3.5
  // re-runs when the "foo" prop changes in 3.5+
```

In version 3.4 and below, `foo` is an actual constant and will never change. In version 3.5 and above, Vue's compiler automatically prepends `props.` when code in the same `<script setup>` block accesses variables destructured from `defineProps`. Therefore the code above becomes equivalent to the following:

```js
const props = defineProps(['foo'])

watchEffect(() => {
  // `foo` transformed to `props.foo` by the compiler
  console.log(props.foo)
})
```

In addition, you can use JavaScript's native default value syntax to declare default values for the props. This is particularly useful when using the type-based props declaration:

```ts
const { foo = 'hello' } = defineProps<{ foo?: string }>()
```

If you prefer to have more visual distinction between destructured props and normal variables in your IDE, Vue's VSCode extension provides a setting to enable inlay-hints for destructured props.

Passing Destructured Props into Functions

When we pass a destructured prop into a function, e.g.:

```js
const { foo } = defineProps(['foo'])

watch(foo, /* ... */)
```

This will not work as expected because it is equivalent to `watch(props.foo, ...)` - we are passing a value instead of a reactive data source to `watch`. In fact, Vue's compiler will catch such cases and throw a warning.

Similar to how we can watch a normal prop with `watch(() => props.foo, ...)`, we can watch a destructured prop also by wrapping it in a getter:

```js
watch(() => foo, /* ... */)
```

In addition, this is the recommended approach when we need to pass a destructured prop into an external function while retaining reactivity:

The external function can call the getter (or normalize it with toValue) when it needs to track changes of the provided prop, e.g. in a computed or watcher getter.

Prop Passing Details

Prop Name Casing

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

```js
defineProps({
  greetingMessage: String
})
```

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

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

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

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

Static vs. Dynamic Props

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

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

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

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

Passing Different Value Types

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

Number

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

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

Boolean

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

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

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

Array

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

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

Object

```template
<!-- Even though the object is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string.             -->
<BlogPost
  :author="{
```

```
  }"
 />

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

Binding Multiple Properties Using an Object

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

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

The following template:

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

Will be equivalent to:

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

One-Way Data Flow

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

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

```js
const props = defineProps(['foo'])

// ❌ warning, props are readonly!
props.foo = 'bar'
```

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:

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

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

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:

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

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

Mutating Object / Array Props

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

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

Prop Validation

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

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

```
  //  (`null` and `undefined` values will allow any type)
  propA: Number,
  // Multiple possible types
  propB: [String, Number],
  // Required string
  propC: {
    type: String,
    required: true
  },
  // Required but nullable string
  propD: {
    type: [String, null],
    required: true
  },
  // Number with a default value
  propE: {
    type: Number,
    default: 100
  },
  // Object with a default value
  propF: {
    type: Object,
    // Object or array defaults must be returned from
    // a factory function. The function receives the raw
    // props received by the component as the argument.
    default(rawProps) {
      return { message: 'hello' }
    }
  },
  // Custom validator function
  // full props passed as 2nd argument in 3.4+
  propG: {
    validator(value, props) {
      // The value must match one of these strings
      return ['success', 'warning', 'danger'].includes(value)
    }
  },
  // Function with a default value
  propH: {
    type: Function,
    // Unlike object or array default, this is not a factory
    // function - this is a function to serve as a default value
    default() {
      return 'Default function'
    }
  }
})
```

ⓘ TIP

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

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

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

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

Runtime Type Checks

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

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

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

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

You could use it as a prop's type:

```
})
```

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

Nullable Type

If the type is required but nullable, you can use the array syntax that includes `null`:

```js
defineProps({
  id: {
    type: [String, null],
    required: true
  }
})
```

Note that if `type` is just `null` without using the array syntax, it will allow any type.

Boolean Casting

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

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

The component can be used like this:

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

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

When a prop is declared to allow multiple types, the casting rules for `Boolean` will also be applied. However, there is an edge when both `String` and `Boolean` are allowed - the Boolean casting rule only applies if Boolean appears before String:

```
  disabled: [Boolean, Number]
})

// disabled will be casted to true
defineProps({
  disabled: [Boolean, String]
})

// disabled will be casted to true
defineProps({
  disabled: [Number, Boolean]
})

// disabled will be parsed as an empty string (disabled="")
defineProps({
  disabled: [String, Boolean]
})
```

Edit this page on GitHub

Previous
Registration

Next
Events