Reactivity Fundamentals

> ⓘ API Preference
>
> This page and many other chapters later in the guide contain different content for the Options API and the Composition API. Your current preference is **Composition API**. You can toggle between the API styles using the "API Preference" switches at the top of the left sidebar.

Declaring Reactive State

`ref()`

In Composition API, the recommended way to declare reactive state is using the `ref()` function:

```js
import { ref } from 'vue'

const count = ref(0)
```

`ref()` takes the argument and returns it wrapped within a ref object with a `.value` property:

```js
const count = ref(0)

console.log(count) // { value: 0 }
console.log(count.value) // 0

count.value++
console.log(count.value) // 1
```

> See also: Typing Refs TS

```js
import { ref } from 'vue'

export default {
  // `setup` is a special hook dedicated for the Composition API.
  setup() {
    const count = ref(0)

    // expose the ref to the template
    return {
      count
    }
  }
}
```

```template
<div>{{ count }}</div>
```

Notice that we did **not** need to append `.value` when using the ref in the template. For convenience, refs are automatically unwrapped when used inside templates (with a few caveats).

You can also mutate a ref directly in event handlers:

```template
<button @click="count++">
  {{ count }}
</button>
```

For more complex logic, we can declare functions that mutate refs in the same scope and expose them as methods alongside the state:

```js
import { ref } from 'vue'

export default {
  setup() {
    const count = ref(0)

    function increment() {
      // .value is needed in JavaScript
      count.value++
    }

    // don't forget to expose the function as well.
    return {
      count,
      increment
    }
  }
}
```

```template
<button @click="increment">
  {{ count }}
</button>
```

Here's the example live on Codepen, without using any build tools.

`<script setup>`

Manually exposing state and methods via `setup()` can be verbose. Luckily, it can be avoided when using Single-File Components (SFCs). We can simplify the usage with `<script setup>`:

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

const count = ref(0)

function increment() {
  count.value++
}
</script>

<template>
  <button @click="increment">
    {{ count }}
  </button>
</template>
```

Try it in the Playground

Top-level imports, variables and functions declared in `<script setup>` are automatically usable in the template of the same component. Think of the template as a JavaScript function declared in the same scope - it naturally has access to everything declared alongside it.

> ⓘ TIP
>
> For the rest of the guide, we will be primarily using SFC + `<script setup>` syntax for the Composition API code examples, as that is the most common usage for Vue developers.
>
> If you are not using SFC, you can still use Composition API with the `setup()` option.

Why Refs?

When you use a ref in a template, and change the ref's value later, Vue automatically detects the change and updates the DOM accordingly. This is made possible with a dependency-tracking based reactivity system. When a component is rendered for the first time, Vue **tracks** every ref that was used during the render. Later on, when a ref is mutated, it will **trigger** a re-render for components that are tracking it.

In standard JavaScript, there is no way to detect the access or mutation of plain variables. However, we can intercept the get and set operations of an object's properties using getter and setter methods.

The `.value` property gives Vue the opportunity to detect when a ref has been accessed or mutated. Under the hood, Vue performs the tracking in its getter, and performs triggering in its setter. Conceptually, you can think of a ref as an object that looks like this:

```js
// pseudo code, not actual implementation
const myRef = {
  _value: 0,
  get value() {
    track()
    return this._value
  },
  set value(newValue) {
    this._value = newValue
    trigger()
  }
}
```

Another nice trait of refs is that unlike plain variables, you can pass refs into functions while retaining access to the latest value and the reactivity connection. This is particularly useful when refactoring complex logic into reusable code.

The reactivity system is discussed in more details in the Reactivity in Depth section.

Deep Reactivity

Refs can hold any value type, including deeply nested objects, arrays, or JavaScript built-in data structures like `Map`.

A ref will make its value deeply reactive. This means you can expect changes to be detected even when you mutate nested objects or arrays:

```js
import { ref } from 'vue'

const obj = ref({
  nested: { count: 0 },
  arr: ['foo', 'bar']
```

```
function mutateDeeply() {
  // these will work as expected.
  obj.value.nested.count++
  obj.value.arr.push('baz')
}
```

Non-primitive values are turned into reactive proxies via `reactive()`, which is discussed below.

It is also possible to opt-out of deep reactivity with shallow refs. For shallow refs, only `.value` access is tracked for reactivity. Shallow refs can be used for optimizing performance by avoiding the observation cost of large objects, or in cases where the inner state is managed by an external library.

Further reading:

- Reduce Reactivity Overhead for Large Immutable Structures
- Integration with External State Systems

DOM Update Timing

When you mutate reactive state, the DOM is updated automatically. However, it should be noted that the DOM updates are not applied synchronously. Instead, Vue buffers them until the "next tick" in the update cycle to ensure that each component updates only once no matter how many state changes you have made.

To wait for the DOM update to complete after a state change, you can use the nextTick() global API:

```js
import { nextTick } from 'vue'

async function increment() {
  count.value++
  await nextTick()
  // Now the DOM is updated
}
```

`reactive()`

There is another way to declare reactive state, with the `reactive()` API. Unlike a ref which wraps the inner value in a special object, `reactive()` makes an object itself reactive:

```
const state = reactive({ count: 0 })
```

> See also: Typing Reactive TS

Usage in template:

```template
<button @click="state.count++">
  {{ state.count }}
</button>
```

Reactive objects are JavaScript Proxies and behave just like normal objects. The difference is that Vue is able to intercept the access and mutation of all properties of a reactive object for reactivity tracking and triggering.

`reactive()` converts the object deeply: nested objects are also wrapped with `reactive()` when accessed. It is also called by `ref()` internally when the ref value is an object. Similar to shallow refs, there is also the `shallowReactive()` API for opting-out of deep reactivity.

Reactive Proxy vs. Original

It is important to note that the returned value from `reactive()` is a Proxy of the original object, which is not equal to the original object:

```js
const raw = {}
const proxy = reactive(raw)

// proxy is NOT equal to the original.
console.log(proxy === raw) // false
```

Only the proxy is reactive - mutating the original object will not trigger updates. Therefore, the best practice when working with Vue's reactivity system is to **exclusively use the proxied versions of your state**.

To ensure consistent access to the proxy, calling `reactive()` on the same object always returns the same proxy, and calling `reactive()` on an existing proxy also returns that same proxy:

```js
// calling reactive() on the same object returns the same proxy
console.log(reactive(raw) === proxy) // true

// calling reactive() on a proxy returns itself
console.log(reactive(proxy) === proxy) // true
```

```js
const proxy = reactive({})

const raw = {}
proxy.nested = raw

console.log(proxy.nested === raw) // false
```

Limitations of `reactive()`

The `reactive()` API has a few limitations:

1. **Limited value types:** it only works for object types (objects, arrays, and collection types such as `Map` and `Set`). It cannot hold primitive types such as `string`, `number` or `boolean`.

2. **Cannot replace entire object:** since Vue's reactivity tracking works over property access, we must always keep the same reference to the reactive object. This means we can't easily "replace" a reactive object because the reactivity connection to the first reference is lost:

   ```js
   let state = reactive({ count: 0 })

   // the above reference ({ count: 0 }) is no longer being tracked
   // (reactivity connection is lost!)
   state = reactive({ count: 1 })
   ```

3. **Not destructure-friendly:** when we destructure a reactive object's primitive type property into local variables, or when we pass that property into a function, we will lose the reactivity connection:

   ```js
   const state = reactive({ count: 0 })

   // count is disconnected from state.count when destructured.
   let { count } = state
   // does not affect original state
   count++

   // the function receives a plain number and
   // won't be able to track changes to state.count
   // we have to pass the entire object in to retain reactivity
   callSomeFunction(state.count)
   ```

Due to these limitations, we recommend using `ref()` as the primary API for declaring reactive state.

Additional Ref Unwrapping Details

As Reactive Object Property

A ref is automatically unwrapped when accessed or mutated as a property of a reactive object. In other words, it behaves like a normal property:

```js
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:

```js
const otherCount = ref(2)

state.count = otherCount
console.log(state.count) // 2
// original ref is now disconnected from state.count
console.log(count.value) // 1
```

Ref unwrapping only happens when nested inside a deep reactive object. It does not apply when it is accessed as a property of a shallow reactive object.

Caveat in Arrays and Collections

Unlike reactive objects, there is **no** unwrapping performed when the ref is accessed as an element of a reactive array or a native collection type like `Map`:

```js
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)
```

Caveat when Unwrapping in Templates

In the example below, `count` and `object` are top-level properties, but `object.id` is not:

```js
const count = ref(0)
const object = { id: ref(1) }
```

Therefore, this expression works as expected:

```template
{{ count + 1 }}
```

...while this one does **NOT**:

```template
{{ object.id + 1 }}
```

The rendered result will be `[object Object]1` because `object.id` is not unwrapped when evaluating the expression and remains a ref object. To fix this, we can destructure `id` into a top-level property:

```js
const { id } = object
```

```template
{{ id + 1 }}
```

Now the render result will be `2`.

Another thing to note is that a ref does get unwrapped if it is the final evaluated value of a text interpolation (i.e. a `{{ }}` tag), so the following will render `1`:

```template
{{ object.id }}
```

This is just a convenience feature of text interpolation and is equivalent to `{{ object.id.value }}`.

Edit this page on GitHub

Previous
Template Syntax

Next
Computed Properties