Watchers

Basic Example

Computed properties allow us to declaratively compute derived values. However, there are cases where we need to perform "side effects" in reaction to state changes - for example, mutating the DOM, or changing another piece of state based on the result of an async operation.

With Composition API, we can use the `watch` function to trigger a callback whenever a piece of reactive state changes:

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

const question = ref('')
const answer = ref('Questions usually contain a question mark. ;-)')
const loading = ref(false)

// watch works directly on a ref
watch(question, async (newQuestion, oldQuestion) => {
  if (newQuestion.includes('?')) {
    loading.value = true
    answer.value = 'Thinking...'
    try {
      const res = await fetch('https://yesno.wtf/api')
      answer.value = (await res.json()).answer
    } catch (error) {
      answer.value = 'Error! Could not reach the API. ' + error
    } finally {
      loading.value = false
    }
  }
})
</script>

<template>
  <p>
    Ask a yes/no question:
    <input v-model="question" :disabled="loading" />
  </p>
```

Try it in the Playground

Watch Source Types

`watch`'s first argument can be different types of reactive "sources": it can be a ref (including computed refs), a reactive object, a getter function, or an array of multiple sources:

```js
const x = ref(0)
const y = ref(0)

// single ref
watch(x, (newX) => {
  console.log(`x is ${newX}`)
})

// getter
watch(
  () => x.value + y.value,
  (sum) => {
    console.log(`sum of x + y is: ${sum}`)
  }
)

// array of multiple sources
watch([x, () => y.value], ([newX, newY]) => {
  console.log(`x is ${newX} and y is ${newY}`)
})
```

Do note that you can't watch a property of a reactive object like this:

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

// this won't work because we are passing a number to watch()
watch(obj.count, (count) => {
  console.log(`Count is: ${count}`)
})
```

Instead, use a getter:

```js
// instead, use a getter:
watch(
  () => obj.count,
  (count) => {
    console.log(`Count is: ${count}`)
```

Deep Watchers

When you call `watch()` directly on a reactive object, it will implicitly create a deep watcher - the callback will be triggered on all nested mutations:

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

watch(obj, (newValue, oldValue) => {
  // fires on nested property mutations
  // Note: `newValue` will be equal to `oldValue` here
  // because they both point to the same object!
})

obj.count++
```

This should be differentiated with a getter that returns a reactive object - in the latter case, the callback will only fire if the getter returns a different object:

```js
watch(
  () => state.someObject,
  () => {
    // fires only when state.someObject is replaced
  }
)
```

You can, however, force the second case into a deep watcher by explicitly using the `deep` option:

```js
watch(
  () => state.someObject,
  (newValue, oldValue) => {
    // Note: `newValue` will be equal to `oldValue` here
    // *unless* state.someObject has been replaced
  },
  { deep: true }
)
```

In Vue 3.5+, the `deep` option can also be a number indicating the max traversal depth - i.e. how many levels should Vue traverse an object's nested properties.

⚠ Use with Caution

beware of the performance implications.

Eager Watchers

`watch` is lazy by default: the callback won't be called until the watched source has changed. But in some cases we may want the same callback logic to be run eagerly - for example, we may want to fetch some initial data, and then re-fetch the data whenever relevant state changes.

We can force a watcher's callback to be executed immediately by passing the `immediate: true` option:

```js
watch(
  source,
  (newValue, oldValue) => {
    // executed immediately, then again when `source` changes
  },
  { immediate: true }
)
```

Once Watchers

- Only supported in 3.4+

Watcher's callback will execute whenever the watched source changes. If you want the callback to trigger only once when the source changes, use the `once: true` option.

```js
watch(
  source,
  (newValue, oldValue) => {
    // when `source` changes, triggers only once
  },
  { once: true }
)
```

`watchEffect()`

whenever the `todoId` ref changes:

```js
const todoId = ref(1)
const data = ref(null)

watch(
  todoId,
  async () => {
    const response = await fetch(
      `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
    )
    data.value = await response.json()
  },
  { immediate: true }
)
```

In particular, notice how the watcher uses `todoId` twice, once as the source and then again inside the callback.

This can be simplified with `watchEffect()`. `watchEffect()` allows us to track the callback's reactive dependencies automatically. The watcher above can be rewritten as:

```js
watchEffect(async () => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
  )
  data.value = await response.json()
})
```

Here, the callback will run immediately, there's no need to specify `immediate: true`. During its execution, it will automatically track `todoId.value` as a dependency (similar to computed properties). Whenever `todoId.value` changes, the callback will be run again. With `watchEffect()`, we no longer need to pass `todoId` explicitly as the source value.

You can check out this example of `watchEffect()` and reactive data-fetching in action.

For examples like these, with only one dependency, the benefit of `watchEffect()` is relatively small. But for watchers that have multiple dependencies, using `watchEffect()` removes the burden of having to maintain the list of dependencies manually. In addition, if you need to watch several properties in a nested data structure, `watchEffect()` may prove more efficient than a deep watcher, as it will only track the properties that are used in the callback, rather than recursively tracking all of them.

ⓘ TIP

will be tracked.

`watch` vs. `watchEffect`

`watch` and `watchEffect` both allow us to reactively perform side effects. Their main difference is the way they track their reactive dependencies:

- `watch` only tracks the explicitly watched source. It won't track anything accessed inside the callback. In addition, the callback only triggers when the source has actually changed. `watch` separates dependency tracking from the side effect, giving us more precise control over when the callback should fire.
- `watchEffect`, on the other hand, combines dependency tracking and side effect into one phase. It automatically tracks every reactive property accessed during its synchronous execution. This is more convenient and typically results in terser code, but makes its reactive dependencies less explicit.

Side Effect Cleanup

Sometimes we may perform side effects, e.g. asynchronous requests, in a watcher:

```js
watch(id, (newId) => {
  fetch(`/api/${newId}`).then(() => {
    // callback logic
  })
})
```

But what if `id` changes before the request completes? When the previous request completes, it will still fire the callback with an ID value that is already stale. Ideally, we want to be able to cancel the stale request when `id` changes to a new value.

We can use the `onWatcherCleanup()` 3.5+ API to register a cleanup function that will be called when the watcher is invalidated and is about to re-run:

```js
import { watch, onWatcherCleanup } from 'vue'

watch(id, (newId) => {
  const controller = new AbortController()

  fetch(`/api/${newId}`, { signal: controller.signal }).then(() => {
    // callback logic
  })
```

```
    // abort stale request
    controller.abort()
  })
})
```

Note that `onWatcherCleanup` is only supported in Vue 3.5+ and must be called during the synchronous execution of a `watchEffect` effect function or `watch` callback function: you cannot call it after an `await` statement in an async function.

Alternatively, an `onCleanup` function is also passed to watcher callbacks as the 3rd argument, and to the `watchEffect` effect function as the first argument:

```js
watch(id, (newId, oldId, onCleanup) => {
  // ...
  onCleanup(() => {
    // cleanup logic
  })
})

watchEffect((onCleanup) => {
  // ...
  onCleanup(() => {
    // cleanup logic
  })
})
```

This works in versions before 3.5. In addition, `onCleanup` passed via function argument is bound to the watcher instance so it is not subject to the synchronous constraint of `onWatcherCleanup`.

Callback Flush Timing

When you mutate reactive state, it may trigger both Vue component updates and watcher callbacks created by you.

Similar to component updates, user-created watcher callbacks are batched to avoid duplicate invocations. For example, we probably don't want a watcher to fire a thousand times if we synchronously push a thousand items into an array being watched.

By default, a watcher's callback is called **after** parent component updates (if any), and **before** the owner component's DOM updates. This means if you attempt to access the owner component's own DOM inside a watcher callback, the DOM will be in a pre-update state.

If you want to access the owner component's DOM in a watcher callback **after** Vue has updated it, you need to specify the `flush: 'post'` option:

```js
watch(source, callback, {
  flush: 'post'
})

watchEffect(callback, {
  flush: 'post'
})
```

Post-flush `watchEffect()` also has a convenience alias, `watchPostEffect()`:

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

watchPostEffect(() => {
  /* executed after Vue updates */
})
```

Sync Watchers

It's also possible to create a watcher that fires synchronously, before any Vue-managed updates:

```js
watch(source, callback, {
  flush: 'sync'
})

watchEffect(callback, {
  flush: 'sync'
})
```

Sync `watchEffect()` also has a convenience alias, `watchSyncEffect()`:

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

watchSyncEffect(() => {
  /* executed synchronously upon reactive data change */
})
```

⚠ Use with Caution

on data sources that might be synchronously mutated many times, e.g. arrays.

Stopping a Watcher

Watchers declared synchronously inside `setup()` or `<script setup>` are bound to the owner component instance, and will be automatically stopped when the owner component is unmounted. In most cases, you don't need to worry about stopping the watcher yourself.

The key here is that the watcher must be created **synchronously**: if the watcher is created in an async callback, it won't be bound to the owner component and must be stopped manually to avoid memory leaks. Here's an example:

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

// this one will be automatically stopped
watchEffect(() => {})

// ...this one will not!
setTimeout(() => {
  watchEffect(() => {})
}, 100)
</script>
```

To manually stop a watcher, use the returned handle function. This works for both `watch` and `watchEffect`:

```js
const unwatch = watchEffect(() => {})

// ...later, when no longer needed
unwatch()
```

Note that there should be very few cases where you need to create watchers asynchronously, and synchronous creation should be preferred whenever possible. If you need to wait for some async data, you can make your watch logic conditional instead:

```js
// data to be loaded asynchronously
const data = ref(null)

watchEffect(() => {
  if (data.value) {
    // do something when data is loaded
```

Edit this page on GitHub

Previous
Form Input Bindings

Next
Template Refs