Composables

> ⓘ TIP
>
> This section assumes basic knowledge of Composition API. If you have been learning Vue with Options API only, you can set the API Preference to Composition API (using the toggle at the top of the left sidebar) and re-read the Reactivity Fundamentals and Lifecycle Hooks chapters.

What is a "Composable"?

In the context of Vue applications, a "composable" is a function that leverages Vue's Composition API to encapsulate and reuse **stateful logic**.

When building frontend applications, we often need to reuse logic for common tasks. For example, we may need to format dates in many places, so we extract a reusable function for that. This formatter function encapsulates **stateless logic**: it takes some input and immediately returns expected output. There are many libraries out there for reusing stateless logic - for example lodash and date-fns, which you may have heard of.

By contrast, stateful logic involves managing state that changes over time. A simple example would be tracking the current position of the mouse on a page. In real-world scenarios, it could also be more complex logic such as touch gestures or connection status to a database.

Mouse Tracker Example

If we were to implement the mouse tracking functionality using the Composition API directly inside a component, it would look like this:

MouseComponent.vue

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

function update(event) {
  x.value = event.pageX
  y.value = event.pageY
}

onMounted(() => window.addEventListener('mousemove', update))
onUnmounted(() => window.removeEventListener('mousemove', update))
</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>
```

But what if we want to reuse the same logic in multiple components? We can extract the logic into an external file, as a composable function:

JS mouse.js

```js
import { ref, onMounted, onUnmounted } from 'vue'

// by convention, composable function names start with "use"
export function useMouse() {
  // state encapsulated and managed by the composable
  const x = ref(0)
  const y = ref(0)

  // a composable can update its managed state over time.
  function update(event) {
    x.value = event.pageX
    y.value = event.pageY
  }

  // a composable can also hook into its owner component's
  // lifecycle to setup and teardown side effects.
  onMounted(() => window.addEventListener('mousemove', update))
  onUnmounted(() => window.removeEventListener('mousemove', update))

  // expose managed state as return value
  return { x, y }
}
```

And this is how it can be used in components:

MouseComponent.vue

```vue
<script setup>
import { useMouse } from './mouse.js'

const { x, y } = useMouse()
</script>
```

Mouse position is at: 0, 0

Try it in the Playground

As we can see, the core logic remains identical - all we had to do was move it into an external function and return the state that should be exposed. Just like inside a component, you can use the full range of Composition API functions in composables. The same `useMouse()` functionality can now be used in any component.

The cooler part about composables though, is that you can also nest them: one composable function can call one or more other composable functions. This enables us to compose complex logic using small, isolated units, similar to how we compose an entire application using components. In fact, this is why we decided to call the collection of APIs that make this pattern possible Composition API.

For example, we can extract the logic of adding and removing a DOM event listener into its own composable:

JS event.js

```js
import { onMounted, onUnmounted } from 'vue'

export function useEventListener(target, event, callback) {
  // if you want, you can also make this
  // support selector strings as target
  onMounted(() => target.addEventListener(event, callback))
  onUnmounted(() => target.removeEventListener(event, callback))
}
```

And now our `useMouse()` composable can be simplified to:

JS mouse.js

```js
import { ref } from 'vue'
import { useEventListener } from './event'

export function useMouse() {
  const x = ref(0)
  const y = ref(0)

  useEventListener(window, 'mousemove', (event) => {
    x.value = event.pageX
    y.value = event.pageY
  })

  return { x, y }
}
```

Each component instance calling `useMouse()` will create its own copies of `x` and `y` state so they won't interfere with one another. If you want to manage shared state between components, read the State Management chapter.

Async State Example

The `useMouse()` composable doesn't take any arguments, so let's take a look at another example that makes use of one. When doing async data fetching, we often need to handle different states: loading, success, and error:

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

const data = ref(null)
const error = ref(null)

fetch('...')
  .then((res) => res.json())
  .then((json) => (data.value = json))
  .catch((err) => (error.value = err))
</script>

<template>
  <div v-if="error">Oops! Error encountered: {{ error.message }}</div>
  <div v-else-if="data">
    Data loaded:
    <pre>{{ data }}</pre>
  </div>
  <div v-else>Loading...</div>
</template>
```

It would be tedious to have to repeat this pattern in every component that needs to fetch data. Let's extract it into a composable:

JS fetch.js

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

export function useFetch(url) {
  const data = ref(null)
  const error = ref(null)

  fetch(url)
    .then((res) => res.json())
    .then((json) => (data.value = json))
    .catch((err) => (error.value = err))
```

```
}
```

Now in our component we can just do:

```vue
<script setup>
import { useFetch } from './fetch.js'

const { data, error } = useFetch('...')
</script>
```

Accepting Reactive State

`useFetch()` takes a static URL string as input - so it performs the fetch only once and is then done. What if we want it to re-fetch whenever the URL changes? In order to achieve this, we need to pass reactive state into the composable function, and let the composable create watchers that perform actions using the passed state.

For example, `useFetch()` should be able to accept a ref:

```js
const url = ref('/initial-url')

const { data, error } = useFetch(url)

// this should trigger a re-fetch
url.value = '/new-url'
```

Or, accept a getter function:

```js
// re-fetch when props.id changes
const { data, error } = useFetch(() => `/posts/${props.id}`)
```

We can refactor our existing implementation with the `watchEffect()` and `toValue()` APIs:

JS fetch.js

```js
import { ref, watchEffect, toValue } from 'vue'

export function useFetch(url) {
  const data = ref(null)
  const error = ref(null)

  const fetchData = () => {
    // reset state before fetching..
    data.value = null
    error.value = null
```

```
      .then((json) => (data.value = json))
      .catch((err) => (error.value = err))
  }

  watchEffect(() => {
    fetchData()
  })

  return { data, error }
}
```

`toValue()` is an API added in 3.3. It is designed to normalize refs or getters into values. If the argument is a ref, it returns the ref's value; if the argument is a function, it will call the function and return its return value. Otherwise, it returns the argument as-is. It works similarly to `unref()`, but with special treatment for functions.

Notice that `toValue(url)` is called **inside** the `watchEffect` callback. This ensures that any reactive dependencies accessed during the `toValue()` normalization are tracked by the watcher.

This version of `useFetch()` now accepts static URL strings, refs, and getters, making it much more flexible. The watch effect will run immediately, and will track any dependencies accessed during `toValue(url)`. If no dependencies are tracked (e.g. url is already a string), the effect runs only once; otherwise, it will re-run whenever a tracked dependency changes.

Here's the updated version of `useFetch()`, with an artificial delay and randomized error for demo purposes.

Conventions and Best Practices

Naming

It is a convention to name composable functions with camelCase names that start with "use".

Input Arguments

A composable can accept ref or getter arguments even if it doesn't rely on them for reactivity. If you are writing a composable that may be used by other developers, it's a good idea to handle the case of input arguments being refs or getters instead of raw values. The `toValue()` utility function will come in handy for this purpose:

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

```
  // its normalized value will be returned.
  // Otherwise, it is returned as-is.
  const value = toValue(maybeRefOrGetter)
}
```

If your composable creates reactive effects when the input is a ref or a getter, make sure to either explicitly watch the ref / getter with `watch()`, or call `toValue()` inside a `watchEffect()` so that it is properly tracked.

The useFetch() implementation discussed earlier provides a concrete example of a composable that accepts refs, getters and plain values as input argument.

Return Values

You have probably noticed that we have been exclusively using `ref()` instead of `reactive()` in composables. The recommended convention is for composables to always return a plain, non-reactive object containing multiple refs. This allows it to be destructured in components while retaining reactivity:

```js
// x and y are refs
const { x, y } = useMouse()
```

Returning a reactive object from a composable will cause such destructures to lose the reactivity connection to the state inside the composable, while the refs will retain that connection.

If you prefer to use returned state from composables as object properties, you can wrap the returned object with `reactive()` so that the refs are unwrapped. For example:

```js
const mouse = reactive(useMouse())
// mouse.x is linked to original ref
console.log(mouse.x)
```

```template
Mouse position is at: {{ mouse.x }}, {{ mouse.y }}
```

Side Effects

It is OK to perform side effects (e.g. adding DOM event listeners or fetching data) in composables, but pay attention to the following rules:

- If you are working on an application that uses Server-Side Rendering (SSR), make sure to perform DOM-specific side effects in post-mount lifecycle hooks, e.g. `onMounted()`.

- Remember to clean up side effects in `onUnmounted()`. For example, if a composable sets up a DOM event listener, it should remove that listener in `onUnmounted()` as we have seen in the `useMouse()` example. It can be a good idea to use a composable that automatically does this for you, like the `useEventListener()` example.

Usage Restrictions

Composables should only be called in `<script setup>` or the `setup()` hook. They should also be called **synchronously** in these contexts. In some cases, you can also call them in lifecycle hooks like `onMounted()`.

These restrictions are important because these are the contexts where Vue is able to determine the current active component instance. Access to an active component instance is necessary so that:

1. Lifecycle hooks can be registered to it.
2. Computed properties and watchers can be linked to it, so that they can be disposed when the instance is unmounted to prevent memory leaks.

> ⓘ TIP
>
> `<script setup>` is the only place where you can call composables **after** using `await`. The compiler automatically restores the active instance context for you after the async operation.

Extracting Composables for Code Organization

Composables can be extracted not only for reuse, but also for code organization. As the complexity of your components grows, you may end up with components that are too large to navigate and reason about. Composition API gives you the full flexibility to organize your component code into smaller functions based on logical concerns:

```vue
<script setup>
import { useFeatureA } from './featureA.js'
import { useFeatureB } from './featureB.js'
import { useFeatureC } from './featureC.js'

const { foo, bar } = useFeatureA()
const { baz } = useFeatureB(foo)
```

To some extent, you can think of these extracted composables as component-scoped services that can talk to one another.

Using Composables in Options API

If you are using Options API, composables must be called inside `setup()`, and the returned bindings must be returned from `setup()` so that they are exposed to `this` and the template:

```js
import { useMouse } from './mouse.js'
import { useFetch } from './fetch.js'

export default {
  setup() {
    const { x, y } = useMouse()
    const { data, error } = useFetch('...')
    return { x, y, data, error }
  },
  mounted() {
    // setup() exposed properties can be accessed on `this`
    console.log(this.x)
  }
  // ...other options
}
```

Comparisons with Other Techniques

vs. Mixins

Users coming from Vue 2 may be familiar with the mixins option, which also allows us to extract component logic into reusable units. There are three primary drawbacks to mixins:

1. **Unclear source of properties**: when using many mixins, it becomes unclear which instance property is injected by which mixin, making it difficult to trace the implementation and understand the component's behavior. This is also why we recommend using the refs + destructure pattern for composables: it makes the property source clear in consuming components.

2. **Namespace collisions**: multiple mixins from different authors can potentially register the same property keys, causing namespace collisions. With composables, you can rename

another have to rely on shared property keys, making them implicitly coupled. With composables, values returned from one composable can be passed into another as arguments, just like normal functions.

For the above reasons, we no longer recommend using mixins in Vue 3. The feature is kept only for migration and familiarity reasons.

vs. Renderless Components

In the component slots chapter, we discussed the Renderless Component pattern based on scoped slots. We even implemented the same mouse tracking demo using renderless components.

The main advantage of composables over renderless components is that composables do not incur the extra component instance overhead. When used across an entire application, the amount of extra component instances created by the renderless component pattern can become a noticeable performance overhead.

The recommendation is to use composables when reusing pure logic, and use components when reusing both logic and visual layout.

vs. React Hooks

If you have experience with React, you may notice that this looks very similar to custom React hooks. Composition API was in part inspired by React hooks, and Vue composables are indeed similar to React hooks in terms of logic composition capabilities. However, Vue composables are based on Vue's fine-grained reactivity system, which is fundamentally different from React hooks' execution model. This is discussed in more detail in the Composition API FAQ.

Further Reading

- Reactivity In Depth: for a low-level understanding of how Vue's reactivity system works.
- State Management: for patterns of managing state shared by multiple components.
- Testing Composables: tips on unit testing composables.
- VueUse: an ever-growing collection of Vue composables. The source code is also a great learning resource.

Async Components

Custom Directives