Vue Composables

Pattern for organizing and encapsulating logics in Vue

Table of contents

What is Composable Function?

A function that leverages Vue's Composition API to encapsulate and reuse stateful logic.

LEGO bricks for components.

Composables vs Utility Functions vs Mixins

Utility Functions

General-Purpose: not specific to component or component lifecycle.

Stateless: stateless and don’t hold or manage any internal state.

Low-Level: often lower-level and can be used for a wide range of tasks, from simple calculations to more complex operations.

Mixins (Vue 2)

Object-Based: JavaScript objects merged into a component's options (e.g., data, methods, computed).

Global Scope: Unexpected interactions and naming conflicts if not used carefully.

Composables (Vue 3)

Functions: Return objects, easy to destructure and use.

Scoped Logic: Single Responsibility Principle, focuses on one aspect of behavior.

Reactivity: Use Vue’s reactivity system (e.g., ref, reactive, computed) to manage state and trigger component updates when data changes.

Explicit Use: Explicitly imported and used within a component’s setup function.

Mouse Tracker Example

Mouse position is at: 0, 0

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

```
<script setup>
import { ref, onMounted, onUnmounted } from '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 reuse the same logic in multiple components? We can extract the logic into an external file, as a composable function:

```
// mouse.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 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:

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

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

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

Mouse position is at: 0, 0

As we can see, the core logic remains exactly the same - all we had to do was moving it into an external function and return the state that should be exposed. The same `useMouse()` functionality can now be used in any component.

Furthermore: We can extract the logic of adding and cleaning up a DOM event listener

```
// event.js
import { onMounted, onUnmounted } from "vue";

export function useEventListener(target, event, callback) {
  onMounted(() => target.addEventListener(event, callback));
  onUnmounted(() => target.removeEventListener(event, callback));
}
```

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

```
// mouse.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 };
}
```

One Thing at a Time

Just the same as writing JavaScript Functions

- Extract duplicated logics into composable functions
- Have meaningful names
- Consistent naming conventions - `` `useXX` ``
- Keep functions small and simple
- "Do one thing and do it well"

NDS

Until now we still use mixins (handleInactiveUser.js & datePickerComponent.js) to extract component logic into reusable units in NDS micro UIs.

There are two primary drawbacks for mixins:

1. Unclear source of properties: 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 the destructured variables in case there are conclicting keys from different composables.

Learn More

Vue JS Official Docs · Anthony Fu - Composable Vue · Vue School - What is a Vue.js Composable?