State Management

What is State Management?

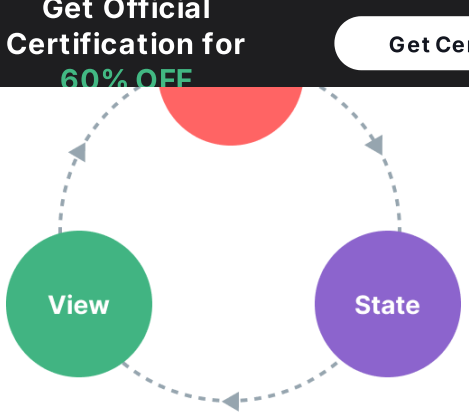
Technically, every Vue component instance already "manages" its own reactive state. Take a simple counter component as an example:

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

// state
const count = ref(0)

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

<!-- view -->
<template>{{ count }}</template>
```

It is a self-contained unit with the following parts:

- The **state**, the source of truth that drives our app;
- The **view**, a declarative mapping of the **state**;
- The **actions**, the possible ways the state could change in reaction to user inputs from the **view**.

This is a simple representation of the concept of "one-way data flow":



However, the simplicity starts to break down when we have **multiple components that share a common state**:

1. Multiple views may depend on the same piece of state.
2. Actions from different views may need to mutate the same piece of state.

For case one, a possible workaround is by "lifting" the shared state up to a common ancestor component, and then pass it down as props. However, this quickly gets tedious in component trees with deep hierarchies, leading to another problem known as Prop Drilling.

For case two, we often find ourselves resorting to solutions such as reaching for direct parent / child instances via template refs, or trying to mutate and synchronize multiple copies of the state via emitted events. Both of these patterns are brittle and quickly lead to unmaintainable code.

A simpler and more straightforward solution is to extract the shared state out of the components, and manage it in a global singleton. With this, our component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!

Simple State Management with Reactivity API

If you have a piece of state that should be shared by multiple instances, you can use `reactive()` to create a reactive object, and then import it into multiple components:

JS store.js

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

export const store = reactive({
  count: 0
})
```

ComponentA.vue

```
</script>

<template>From A: {{ store.count }}</template>
```

ComponentB.vue

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

<template>From B: {{ store.count }}</template>
```

Now whenever the `store` object is mutated, both `<ComponentA>` and `<ComponentB>` will update their views automatically - we have a single source of truth now.

However, this also means any component importing `store` can mutate it however they want:

```template
<template>
  <button @click="store.count++">
    From B: {{ store.count }}
  </button>
</template>
```

While this works in simple cases, global state that can be arbitrarily mutated by any component is not going to be very maintainable in the long run. To ensure the state-mutating logic is centralized like the state itself, it is recommended to define methods on the store with names that express the intention of the actions:

store.js

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

export const store = reactive({
  count: 0,
  increment() {
    this.count++
  }
})
```

```template
<template>
  <button @click="store.increment()">
    From B: {{ store.count }}
  </button>
</template>
```

Try it in the Playground

TIP

Note the click handler uses `store.increment()` with parentheses - this is necessary to call the method with the proper `this` context since it's not a component method.

Although here we are using a single reactive object as a store, you can also share reactive state created using other Reactivity APIs such as `ref()` or `computed()`, or even return global state from a Composable:

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

// global state, created in module scope
const globalCount = ref(1)

export function useCount() {
  // local state, created per-component
  const localCount = ref(1)

  return {
    globalCount,
    localCount
  }
}
```

The fact that Vue's reactivity system is decoupled from the component model makes it extremely flexible.

SSR Considerations

If you are building an application that leverages Server-Side Rendering (SSR), the above pattern can lead to issues due to the store being a singleton shared across multiple requests. This is discussed in more details in the SSR guide.

Pinia

While our hand-rolled state management solution will suffice in simple scenarios, there are many more things to consider in large-scale production applications:

- Stronger conventions for team collaboration

- Hot Module Replacement
- Server-Side Rendering support

Pinia is a state management library that implements all of the above. It is maintained by the Vue core team, and works with both Vue 2 and Vue 3.

Existing users may be familiar with Vuex, the previous official state management library for Vue. With Pinia serving the same role in the ecosystem, Vuex is now in maintenance mode. It still works, but will no longer receive new features. It is recommended to use Pinia for new applications.

Pinia started out as an exploration of what the next iteration of Vuex could look like, incorporating many ideas from core team discussions for Vuex 5. Eventually, we realized that Pinia already implements most of what we wanted in Vuex 5, and decided to make it the new recommendation instead.

Compared to Vuex, Pinia provides a simpler API with less ceremony, offers Composition-API-style APIs, and most importantly, has solid type inference support when used with TypeScript.

Edit this page on GitHub

Previous
Routing

Next
Testing