PAU CASESNOVES
Centre Integrat de Formació Professional

Desenvolupament d'aplicacions web

Desenvolupament web a l'entorn client

UT 8.3: Vue Estat

Joan Pons Tugores

Índex de continguts

Estat: Pinia

Les aplicacions normalment tenen tres parts:

- L'estat: les propietats que determinen les dades que utilitza l'aplicació.
- La vista: mostra l'estat a l'usuari.
- Les accions: permeten modificar l'estat, el valor de les dades de l'aplicació.

Els exemples que hem vist fins ara eren molt simples i ens bastava amb les dades de la instància arrel per fer funcionar l'aplicació. En una aplicació real hi haurà molts més components i molts d'ells hauran de modificar els valors de les dades.

Quan l'estructura es complica una mica, passar les dades a través de propietats dels components no és una opció vàlida.

En aquests casos *Vue* proposa extreure l'estat a un objecte global i únic que permeti gestionar-lo centralitzadament. D'aquesta manera no ens hem de preocupar de passar les dades a cada component sinó que tots els components de l'aplicació accedeixen a aquest objecte global, anomenat *store*.

*Vue h*o fa amb la llibreria *Pinia.js*.

Això no vol dir que cada component no pugui mantenir el seu estat particular amb *data.* L'*store* s'hauria d'utilitzar per a l'estat general de l'aplicació.

Els *store* són reactius, és a dir, els canvis que sofreixin les propietats de l'store es reflectiran en els components que mostren aquestes propietats.

Instal·lació de *Pinia*

La millor manera és amb *npm*, executant la següent comanda al directori del projecte:

```
npm install pinia
```

Utilització de *Pinia*

Al main.js hem de crear una instància de *Pinia*, el que serà el magatzem principal, i la passarem a l'aplicació.

```
import { createApp } from 'vue'
import { createPinia } from 'pinia' //importam la funció createPinia.
import App from './App.vue'

import './assets/main.css'

const pinia=createPinia() //Cream la instància
const app = createApp(App)

app.use(pinia) //La passam a l'apicació

app.mount('#app')
```

Definició d'un *store*

Per poder utilitzar *Pinia* el primer que haurem de fer serà definir un magatzem, un *store*.

Pinia permet separar l'estat global de l'aplicació dins diferents *stores*. Això permet modularitzar l'estat, dividir l'estat en parts més manejables.

Per a cada *store* crearem un fitxer *js* apart. Normalment els posarem dins la carpeta *src/stores*

Dins aquest fitxer importarem la funció *defineStore* de pinia. Aquesta funció té dos arguments:

- L'identificador, hauria de ser únic.
- Una objecte amb la definició de l'store.

El resultat de l'store s'ha d'assignar a una constant i exportar-la per fer-la accessible des dels components de l'aplicació. El nom d'aquesta constant comença amb *use* després l'identificador de l'store i finalment *Store*.

Per exemple si la nostra aplicació ha de mantenir el nombre de vegades que s'ha pitjat qualsevol botó de l'aplicació podríem definir el següent store:

```
import { defineStore } from 'pinia'
export const useCounterStore = defineStore('counter', {
  state: () => ({ count: 0 })
})
```

Parts de l'*store*

state

Propietats que formen part de l'estat del magatzem. Pinia ho defineix com una funció que torna l'estat inicial del magatzem.

```
import { defineStore } from 'pinia'
export const useCounterStore = defineStore('counter', {
  state: () => ({
            count: 0
  })
```

Cada dada que volem incloure a l'estat de l'aplicació es crearà dins el magatzem, per exemple, *count* comptarà les vegades que s'ha fet click a qualsevol botó de l'aplicació.

Les dades de *l'store* són reactives, això vol dir que quan canvien les dades s'actualitzen els components que les utilitzen.

No es poden afegir dades a l'estat de l'aplicació, s'han de definir al moment de crear l'*store*.

Podem llegir i modificar les dades del magatzem directament des de qualsevol component de l'aplicació.

Per exemple, si volem mostrar dins App.vue el valor de la variable *count* del magatzem haurem d'importar la funció definida al magatzem, useCounterStore() i assignar el seu valor a una variable.

A partir d'aquest moment, aquesta variable serà un objecte amb les propietats de finides a l'*state* de l'*store*.

Per accedir-hi simplement utilitzam la notació per punts variable.propietat:

```
<script setup>
import {useCounterStore} from "@/stores/counter";
let counterStore = useCounterStore();
</script>
```

```
<template>
  <div>
      <img alt="Vue logo" class="logo" src="./assets/logo.svg" width="125" height="125"/>
  </div>
  <div>
    <h1>Comptador global</h1>
    <h2>{{ counterStore.count }}</h2>
  </div>
</template>
```

Des dels components també és pot modificar directament el valor de les propietats de l'estat:

```
<script setup>
import {ref} from "vue";
import {useCounterStore} from "@/stores/counter";

const counterStore = useCounterStore();
const comptador = ref(0) //Particular del component
function incrementar() {
  comptador.value++
  counterStore.count++
}
defineProps({nomBoto: String})
</script>
<template>
    <button @click="incrementar">{{ nomBoto }}: {{ comptador }} - {{counterStore.count}}</button>
</template>
```

Ara cada vegada que es pitgi un botó, una instància d'aquest component s'incrementarà el comptador particular de la instància del component, però també el comptador global de l'store.

getters

Podem pensar en els *getters* com les propietats computades de l'*store*. El valor que torna un getter només es calcula quan alguna de les seves dependències canvia.

Els getters no poden ser asíncrons, no es poden utilitzar per exemple per cridar a una API.

```
import { defineStore } from 'pinia'
export const useCounterStore = defineStore('counter', {
  state: () => ({
            count: 0
  }),
  getters: {
    doubleCount: (state) => state.count * 2
  }
})
```

El primer paràmetre del getter sempre és l'estat. A l'exemple anterior definim un getter que torna el doble de les vegades que s'ha clicat el botó.

Amb la notació tradicional el getter seria:

```
  doubleCount: function(state){
          return  state.count * 2
       }
```

Només es recalcula el resultat del getter quan canvia el valor de *count*.

En principi no és possible passar arguments als getters. Però podem fer que el getter torni una funció que si que rebi un argument. Per exemple, si a l'estat de l'store tenim

```
state: () => ({
    illes: [{id:'071', nom:'Mallorca'},{id:'072', nom:'Menorca'},
            {id:'073', nom:'Eivissa'},{id:'074', nom:'Formentera'}]
  }),
```

podem voler un getter que ens torni una illa a partir del seu *id*:

```
getters: {
    getIllaById: (state) => (id) => {
            return state.illes.find(illa => illa.id === id)
    }
  }
```

Per accedir al *getter* des d'un component ho podem fer de dues maneres:

- Com a propietats. En aquest cas el getter és manté dins el sistema de reactivitat de *Vue*.

```
<h2>Doble de clicks</h2>
<p>{{ counterStore.doubleCount }}</p>
```

- Com a mètode. Es sol utilitzar quan hem de passar un argument al *getter*, per exemple per recuperar una illa per identificador. En aquest cas el getter es calcula cada vegada que es crida.

  Des del component es cridaria com:

```
<h2>Getter, todo per id</h2>
<p>{{ counterStore.getIllaById('071')}}</p>
```

actions

Les accions vendrien a ser mètodes que normalment s'utilitzen per implementar la lògica de negoci de l'aplicació.

Tenen accés a tot l'*store* amb *this*.

Poden ser *asíncrones*.

```
actions: {
        randomizeCounter() {
                this.count = Math.round(100 * Math.random())
        },
},
```

Normalment s'hauria d'utilitzar una acció quan el seu codi sigui asíncron. Un cas molt usual és fer una crida a una API.

Per posar un exemple senzill, posarem un retard al botó:

```
actions:{
        incrementaAmbRetard(){
                setTimeout(()=>{this.count++},1000)}
        }
}
```

Les accions es criden com qualsevol funció o mètode.

```
counterStore.incrementaAmbRetard()
```

Si volem que el valor de l'increment ens arribi com a argument podem afegir un segon paràmetre a l'acció:

```
actions:{
        incrementa(quantitat){
                this.count+=quantitat;
        }
 }
```

La cridarem com:

```
counterStore.incrementa(10)
```