

Vue.js : Pinia

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

`elmouelhi.achref@gmail.com`



Plan

- 1 Introduction
- 2 Store
- 3 State
- 4 Getters
- 5 Actions
- 6 Encapsulation
- 7 Solution avec API Composition

Pinia

- Gestionnaire d'état pour les applications **Vue.js** 2 et 3
- Créé par **Eduardo San Martin Morote** : créateur de **Vue Router**
- Compatible avec **Vue.js Devtools**
- Supportant **TypeScript**
- Plus simple que **Vuex** : considéré comme **Vuex 5** [documentation officielle de **Vuex**]
- Bénéficiant des avantages de **Composition API**
- Reconnu par **Evan You** comme le gestionnaire d'état officiel pour les applications **Vue.js** (**Vuex** n'est pas obsolète, vous pouvez continuer à l'utiliser) [Déclaration de **Evan You** en décembre 2021]

Comparaison : Vuex vs Pinia

Critère	Vuex	Pinia
API	Basée sur une structure modulaire classique (state, mutations, actions, getters)	Basée sur Composition API, plus intuitive avec Vue 3
Verbosité	Assez verbeux (mutations nécessaires)	Plus concis (pas de mutations obligatoires)
TypeScript	Support limité, nécessite plus de configuration	Support natif et simplifié
DevTools	Intégré à Vue Devtools	Intégré avec affichage clair des stores
Apprentissage	Plus complexe pour les débutants	Courbe d'apprentissage plus douce
Modularité	Modules à configurer manuellement	Chaque store est une fonction indépendante
État centralisé	Oui	Oui, mais plus flexible et léger
Support officiel	Toujours maintenu mais non recommandé pour les nouveaux projets	Recommandé par Evan You (créateur de Vue) comme successeur de Vuex

Pour installer Pinia, exécutez la commande

```
npm install pinia
```

Vue.js

Dans `main.js`, il faut importer `createPinia`

```
import { createPinia } from 'pinia'
```

© Achref EL MOUELHI ©

Vue.js

Dans `main.js`, **il faut importer** `createPinia`

```
import { createPinia } from 'pinia'
```

Ensuite créer un objet Pinia

```
const pinia = createPinia()
```

Vue.js

Dans `main.js`, il faut importer `createPinia`

```
import { createPinia } from 'pinia'
```

Ensuite créer un objet Pinia

```
const pinia = createPinia()
```

Et enfin déclarer son utilisation

```
app.use(pinia)
```


Exemple

Nous voudrions

- permettre à l'utilisateur d'ajouter des produits dans son panier
- afficher le nombre de produits dans le panier dans le composant `MenuNav`

Store

- Un conteneur centralisé qui gère l'état (les données) et la logique de l'application **Vue.js**
- `= { state, getters, actions }`

Dans `produit.store.js` (à créer dans `src/stores`), commençons par importer `defineStore` afin de préparer notre store

```
import { defineStore } from 'pinia'
```

© Achref EL MOUËL

Dans `produit.store.js` (à créer dans `src/stores`), commençons par importer `defineStore` afin de préparer notre store

```
import { defineStore } from 'pinia'
```

Ensuite, définissons un store `produits`

```
export const useProduitStore = defineStore('produits', {  
  
  })
```

State

- Équivalent de `data` dans les composants
- Les données (variables réactives) stockées à l'intérieur du store.

EDans le store `produits`, ajoutons un **state** `lignesCommandes`

```
export const useProduitStore = defineStore('produits', {  
  state() {  
    return {  
      lignesCommandes: []  
    }  
  }  
})
```

Dans le template de `Produit.vue`, ajoutons un bouton qui permettra d'ajouter le produit avec la quantité réservée dans le panier

```
<template>
  <ul>
    <li> {{ produit.nom }} </li>
    <li v-movement="{color: 'white', bgColor: 'pink'}">
      Quantité en stock : {{ produit.quantite }}
    </li>
    <li>
      Quantité réservée :
      <button @click="decrementer">-</button>
      {{ counter.valeur }}
      <button @click="incrementer">+</button>
      <button @click="ajouter(produit)">Ajouter</button>
    </li>
    <li>
      <PrixComponent :prix="produit.prix" />
    </li>
  </ul>
</template>
```

Vue.js

Dans le script de `Produit.vue`, commençons par importer le store

```
import { useProduitStore } from '../stores/produit.store.js'  
  
const produitStore = useProduitStore();
```

© Achref EL MOUELHI ©

Vue.js

Dans le script de `Produit.vue`, commençons par importer le store

```
import { useProduitStore } from '../stores/produit.store.js'

const produitStore = useProduitStore();
```

Implémentons ensuite la méthode `ajouter` qui permettra d'ajouter l'objet souhaité dans le store

```
const ajouter = (p) => {
  produitStore.lignesCommandes.push(
    {
      'quantiteReservee': counter.valeur,
      'produit': p
    })
}
```

Vue.js

Dans le script de `Produit.vue`, commençons par importer le store

```
import { useProduitStore } from '../stores/produit.store.js'

const produitStore = useProduitStore();
```

Implémentons ensuite la méthode `ajouter` qui permettra d'ajouter l'objet souhaité dans le store

```
const ajouter = (p) => {
  produitStore.lignesCommandes.push(
    {
      'quantiteReservee': counter.valeur,
      'produit': p
    })
}
```

Vérifiez avec **DevTools** que le code fonctionne correctement.

Getters

- Équivalent de `computed` dans les composants
- Permettent de recalculer certaines propriétés après chaque modification du `state`
- Chaque getter prend comme paramètre le `state`

Vue.js

Définissons deux getters dans notre store : un premier pour le nombre de produits différents et un deuxième pour la quantité totale

```
import { defineStore } from 'pinia'

export const useProduitStore = defineStore('produits', {
  state() {
    return {
      lignesCommandes: []
    }
  },
  getters: {
    nombreProduits(state) {
      return state.lignesCommandes.length
    },
    quantiteTotale(state) {
      return state.lignesCommandes
        .map(elt => elt.quantiteReservee)
        .reduce((prev, current) => prev + current, 0)
    }
  }
})
```

Dans `script.js` de `MenuNav.vue`, commençons par importer le store

```
<script setup>

import { useProduitStore } from '../stores/produit.store.js'

const produitStore = useProduitStore();

</script>
```

Vue.js

Dans `template.js` de `MenuNav.vue`, utilisons les getters pour afficher le nombre de produits et la quantité totale

```
<template>
  <nav>
    <router-link to="/">Home</router-link> |
    <router-link to="/about">About</router-link> |
    <router-link to="/compteur">Compteur</router-link> |
    <router-link to="/calculatrice">Calculatrice</router-link> |
    <router-link to="/primeur">Primeur</router-link> |
    <router-link to="/dynamic">Dynamic</router-link> |
    <router-link to="/personne">Personne</router-link>
  </nav>
  <span>Nombre de produits {{ produitStore.nombreProduits }}</span> |
  <span>Quantité totale {{ produitStore.quantiteTotale }}</span>
</template>
```

Vue.js

Dans `template.js` de `MenuNav.vue`, utilisons les getters pour afficher le nombre de produits et la quantité totale

```
<template>
  <nav>
    <router-link to="/">Home</router-link> |
    <router-link to="/about">About</router-link> |
    <router-link to="/compteur">Compteur</router-link> |
    <router-link to="/calculatrice">Calculatrice</router-link> |
    <router-link to="/primeur">Primeur</router-link> |
    <router-link to="/dynamic">Dynamic</router-link> |
    <router-link to="/personne">Personne</router-link>
  </nav>
  <span>Nombre de produits {{ produitStore.nombreProduits }}</span> |
  <span>Quantité totale {{ produitStore.quantiteTotale }}</span>
</template>
```

Vérifiez avec **DevTools** que le code fonctionne correctement.

Exercice

- Créer un composant `PanierView.vue` dans `views`
- Associer une route à ce composant et ajoutez le au menu
- Utiliser le store pour lister tous les produits y présents ainsi que leurs quantités

Vue.js

Solution

```
<template>
<ul>
  <li v-for="elt in produitStore.lignesCommandes">
    {{ elt.produit.nom }} : {{ elt.quantiteReservee }}
  </li>
</ul>
</template>

<script setup>

import { useProduitStore } from './stores/produit.store.js'

const produitStore = useProduitStore();

</script>
```

Vue.js

Solution

```
<template>
<ul>
  <li v-for="elt in produitStore.lignesCommandes">
    {{ elt.produit.nom }} : {{ elt.quantiteReservee }}
  </li>
</ul>
</template>

<script setup>

import { useProduitStore } from './stores/produit.store.js'

const produitStore = useProduitStore();

</script>
```

Vérifiez avec **DevTools** que le code fonctionne correctement.

Actions

- Équivalent de `methods` dans les composants
- Permettent d'exécuter une méthode (qui s'applique souvent sur le store)

© Achref EL M...

Actions

- Équivalent de `methods` dans les composants
- Permettent d'exécuter une méthode (qui s'applique souvent sur le store)

Exemple

Nous voudrions permettre à l'utilisateur de supprimer un produit déjà ajouté dans le store

Définissons l'action qui permettra la suppression d'un produit selon son nom

```
import { defineStore } from 'pinia'

export const useProduitStore = defineStore('produits', {
  state() {
    return {
      lignesCommandes: []
    }
  },
  getters: {
    nombreProduits(state) {
      return state.lignesCommandes.length
    },
    quantiteTotale(state) {
      return state.lignesCommandes
        .map(elt => elt.quantiteReservee)
        .reduce((prev, current) => prev + current, 0)
    }
  },
  actions: {
    supprimerLigneCommande(ind) {
      this.lignesCommandes.splice(ind, 1);
    }
  }
})
```

Dans template de `PanierView.vue`, on ajoute un bouton qui appelle cette méthode

```
<template>
  <ul>
    <li v-for="(elt, ind) in produitStore.lignesCommandes">
      {{ elt.produit.nom }} : {{ elt.quantiteReservee }}
      <button
        @click="produitStore.supprimerLigneCommande(ind)">
        Supprimer
      </button>
    </li>
  </ul>
</template>
```

Dans `template` de `PanierView.vue`, on ajoute un bouton qui appelle cette méthode

```
<template>
  <ul>
    <li v-for="(elt, ind) in produitStore.lignesCommandes">
      {{ elt.produit.nom }} : {{ elt.quantiteReservee }}
      <button
        @click="produitStore.supprimerLigneCommande(ind)">
        Supprimer
      </button>
    </li>
  </ul>
</template>
```

Vérifiez avec **DevTools** que le code fonctionne correctement.

Exercice

Dans le composant `PanierView.vue`, ajouter deux boutons autour de la quantité réservée affichée :

- + pour incrémenter la quantité réservée sans dépasser la quantité en stock.
- – pour décrémenter la quantité tout en restant positive ou nulle.

Après avoir exploré les différents éléments **Pinia**, restructurons le `store` en

- évitant l'accès direct aux objets du `state`
- ajoutant un getter pour les objets du `state`
- ajoutant les actions permettant la manipulation des objets du `state`

© Achref EL M...

Après avoir exploré les différents éléments **Pinia**, restructurons le `store` en

- évitant l'accès direct aux objets du `state`
- ajoutant un getter pour les objets du `state`
- ajoutant les actions permettant la manipulation des objets du `state`

Remarques

- Préfixer les objets du `state` par `_` ne rend pas le champ inaccessible, ce n'est qu'une convention.
- Aucune syntaxe ne permet de rendre un objet du `state` privé avec **Pinia**.

Commençons par définir les objets du `state` comme privé en les préfixant par `_` (Convention ES 6)

```
export const useProduitStore = defineStore('produits', {  
  state() {  
    return {  
      _lignesCommandes: []  
    }  
  },  
  
  // + le code précédent  
})
```

Et les getters

```
getters: {  
  lignesCommandes(state) {  
    return state._lignesCommandes;  
  },  
  nombreProduits(state) {  
    return state.lignesCommandes.length  
  },  
  quantiteTotaleCommandee(state) {  
    return state.lignesCommandes  
      .map(lc => lc.quantiteReservee)  
      .reduce((p, c) => p + c, 0)  
  }  
},
```

Et enfin les actions

```
actions: {  
  ajouterLigneCommande(ligneCommande) {  
    this.lignesCommandes.push(ligneCommande);  
  },  
  supprimerProduit(ind) {  
    this.lignesCommandes.splice(ind, 1);  
  }  
}
```

Mettons à jour `ProduitComponent` **en utilisant** `ajouterLigneCommande`

```
const ajouter = () => {  
  emit('sendData', qteReservee.value)  
  submitted.value = true  
  produitStore.ajouterLigneCommande({  
    quantiteReservee: qteReservee.value,  
    produit: props.produit  
  })  
}
```

Et PanierComponent

```
<template>
  <h1>Panier</h1>
  <ul>
    <li v-for='elt in produitStore.lignesCommandes'>
      {{ elt.produit.nom }} -
      {{ elt.quantiteReservee }} éléments -
      {{ elt.produit.prix }} euros
      <button @click=supprimer(id)>
        supprimer
      </button>
    </li>
  </ul>
</template>

<script setup>
import { useProduitStore } from '../stores/produit.store.js';

const produitStore = useProduitStore()

const supprimer = (ind) => produitStore.supprimerLigneCommande(ind)
</script>
```

Solution avec **API Composition** : **Setup Stores**

- Les valeurs réactives (`ref` et `reactive`) remplacent les `state`
- Les `computed` remplacent les `getters`
- Les `function` remplacent les `actions`

© Achref

Solution avec **API Composition** : **Setup Stores**

- Les valeurs réactives (`ref` et `reactive`) remplacent les `state`
- Les `computed` remplacent les `getters`
- Les `function` remplacent les `actions`

Exemple

Réécrire l'exemple précédent avec **API Composition** (**Setup Stores**)

On commence par définir le `state` comme valeur réactive

```
import { defineStore } from 'pinia'
import { reactive } from 'vue'

export const useProduitStore = defineStore('produits', () => {

  let lignesCommandes = reactive([])

})
```

Remplaçons les `getters` par des `computed`

```
import { defineStore } from 'pinia'
import { reactive, computed } from 'vue'

export const useProduitStore = defineStore('produits', () => {

  let lignesCommandes = reactive([])

  const nombreProduits = computed(() => lignesCommandes.length)
  const quantiteTotale = computed(() =>
    lignesCommandes
      .map((elt) => elt.quantiteReservee)
      .reduce((prev, current) => prev + current, 0)
  )
})
```

Et les actions par des function

```
import { defineStore } from 'pinia'
import { reactive, computed } from 'vue'

export const useProduitStore = defineStore('produits', () => {

  let lignesCommandes = reactive([])

  const nombreProduits = computed(() => lignesCommandes.length)
  const quantiteTotale = computed(() =>
    lignesCommandes
      .map((elt) => elt.quantiteReservee)
      .reduce((prev, current) => prev + current, 0)
  )

  function supprimerLigneCommande(ind) {
    lignesCommandes.splice(ind, 1)
  }

})
```

Vue.js

Enfin, il faut penser à tout retourner

```
import { defineStore } from 'pinia'
import { reactive, computed } from 'vue'

export const useProduitStore = defineStore('produits', () => {

  let lignesCommandes = reactive([])

  const nombreProduits = computed(() => lignesCommandes.length)
  const quantiteTotale = computed(() =>
    lignesCommandes
      .map((elt) => elt.quantiteReservee)
      .reduce((prev, current) => prev + current, 0)
  )

  function supprimerLigneCommande(ind) {
    lignesCommandes.splice(ind, 1)
  }

  return {
    lignesCommandes, nombreProduits, quantiteTotale, removeLigneCommande
  }
})
```