State management avec Vuex

Préparation du TP

Il vous faut installer la dernière version de Node afin d'avoir accès au gestionnaire de paquet NPM dans votre terminal.

Installez Yarn :

```
npm install --global yarn
```

Puis rendez-vous à l'url ci-dessous et forkez/téléchargez le dépôt Git :

https://github.com/gabcaron/vuex

Tapez la commande suivante dans votre dépôt local pour installer toutes les dépendances :

```
yarn install
```

Après l'installation, vous pourrez lancer votre projet en local avec la commande suivante :

```
yarn serve
```

Principe de state management

Lorsque l'on travaille avec des "single page applications", le concept de *state management* (gestion d'état) est un sujet couramment abordé.

Le objectif de ceci est le suivant : s'assurer que l'application utilise les bonnes données à tout moment.

Il faut voir le **state** comme un instantané du data store à un moment donné. De multiples states sont utilisés à travers une application, ce qui explique la complexité inhérente à la gestion de ces éléments. Après tout, chaque composant utilisé contient sa propre propriété *data*, ce qui signifie qu'il gère son propre state. Une fois que l'on ajoute la complexité du passage de données entre composants, cela devient effectivement très compliqué.

Dans les discussions autour de la gestion du state apparaît l'idée d'une "Single Source of Truth" (aussi abrégée "SSoT"). Le concept fondamental de cette idée est d'éviter la duplication inutile de données dans l'application.

Bien que cela puisse paraître anodin à première vue, le fait de stocker plusieurs copies des mêmes données dans une application est souvent la cause de bugs, car cela permet l'existence de différents jeux des "mêmes" données en même temps. Au lieu de n'avoir à mettre à jour qu'un seul jeu de données, l'application devra gérer plusieurs jeux de données, ce qui est problématique, car :

- du point de vue du code, cela s'avère bien plus difficile à gérer, car il existe une dépendance qu'il est facile d'oublier à mesure que le système se complexifie
- des bugs apparaissent en grand nombre suite à cette fragmentation

C'est pour cette raison que le concept de "SSoT" est essentiel au succès de tout système de state management.

Vuex

Vuex est un gestionnaire d'état et une bibliothèque pour les applications Vue.js. En d'autres termes, l'unique but de Vuex est de nous aider à créer un store centralisé qui nous permettra d'avoir cette "SSoT" pour récupérer nos datas.

Pour installer Vuex à notre projet (Vue CLI), il suffit d'exécuter la commande suivante :

```
vue add vuex
```

Allez voir ce que cette ligne magique a ajouté à votre code.

src/main.js

```
import Vue from 'vue'
import App from './App.vue'
import store from './store'

Vue.config.productionTip = false

new Vue({
store,
render: h => h(App)
}).$mount('#app')
```

En regardant le fichier *main.js*, vous remarquerez que le plugin a introduit le *store* en tant que nouvelle configuration sur notre instance de Vue. De la même manière que nous configurerions une propriété *data* pour mettre en place un data store, vous pouvez considérer cette propriété *store* comme le data store global que nous allons configurer avec Vuex.

src/store/index.js

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

export default new Vuex.Store({
  state: {

  },
  mutations: {

  },
  actions: {

  }
})
```

La méthode *Vue.use(Vuex)* permet d'ajouter une fonctionnalité globale à notre instance de Vue.

Ensuite, comme pour la mise en place d'une nouvelle instance de Vue avec *new Vue()*, Vuex est installé de la même manière avec *new Vuex.Store()*. Vous pouvez également lui passer des options de configuration, exactement comme pour une instance de Vue.

Récupérer des données depuis Vuex

Dans Vuex, notre data store est défini en tant que *state* dans *src/store/index.js*. Vous pouvez comparer cela à la propriété *data* que nous utilisons depuis plusieurs TP.

> A vous de jouer, migrez les données du *shoppingCart*, de *restaurantName* et celles de *simpleMenu* vers Vuex.

Pour récupérer les données du store, Vuex met à notre disposition un moyen d'associer le state à des variables que nous pouvons appeler dans nos composants : *mapState*. Cette méthode nous permet de demander à Vuex les propriétés de state de notre choix, et il les ajoutera à nos propriétés calculées.

```
export default new Vuex.Store({
    state: {
        month: 3,
        day: 2,
        year: 2022
    }
})
```

```
<template>
    <p>La date stockée dans Vuex est le {{ day }}-{{ uniqueMonth }}-{{ customYear }}.<p>
</template>

<script>
import { mapState } from 'vuex'

export default {
    computed: {
        ...mapState({
            customYear: 'year',
            uniqueMonth: 'month',
            day: 'day'
        })
    }
}
</script>
```

Comme vous le constatez, nous sommes en train d'intégrer notre date manuellement. Si cette date était référencée à plusieurs endroits dans l'application, cela serait assez difficile à maintenir ...

Lorsque votre store est généré, il laisse de côté une des propriétés qui est très utile dans une multitude de scénarios : **les getters**. Ceux-ci sont l'équivalent de la propriété *computed* que vous connaissez déjà.

Dans l'exemple précédent, voilà la manière avec laquelle nous définirions notre date formattée :

```
export default new Vuex.Store({
    state: {
        month: 3,
        day: 2,
        year: 2022
    },
    getters: {
        formattedDate: state => {
            return `${state.day}-${state.month}-${state.year}`
        }
    }
})
```

Comme vous le constatez, chaque **getter** est une fonction qui reçoit le *state* comme argument et retourne une valeur à laquelle nous pourrons accéder plus tard.

Ensuite, nous pouvons simplifier notre code encore davantage dans le composant, grâce à *mapGetters*, qui fonctionne comme *mapState* :

```
<template>
    <p>La date stockée dans Vuex est le {{ formattedDate }}<p>
</template>

<script>
import { mapGetters } from 'vuex'

export default {
    computed: {
        ...mapGetters(['formattedDate'])
    }
}
</script>
```

> Allez-y, migrez *copyright* vers les getters.

Modifier vos données dans Vuex

Comme vous avez pu le voir précédemment, votre store contient une propriété appelée **mutation**. Comme son nom l'indique, elle contiendra un responsable des modifications du *state*.

Par exemple, pour incrémenter un compteur, nous pouvons définir une mutation comme celle-ci :

```
export default new Vuex.Store({
    state: {
        count: 0
    },
    mutations: {
        INCREASE_COUNT(state) {
            state.count += 1
        }
    },
    actions: {

    }
})
```

A noter : La convention de nommage utilisée pour les mutations consiste à n'utiliser que des majuscules et des underscores pour séparer les mots. Grâce à celle-ci, nous savons qu'il ne s'agit pas de fonctions normales.

Par défaut, la mutation reçoit le *state* en premier argument. Dans l'exemple précédent, nous n'incrémentons le *state.count* que de 1.

Cependant, si nous voulons incrémenter le *count* de manière dynamique, nous pouvons utiliser un deuxième argument : **payload** :

```
export default new Vuex.Store({
    state: {
        count: 0
    },
    mutations: {
        INCREASE_COUNT(state, payload) {
            state.count += Number(payload)
        }
    },
    actions: {

    }
})
```

Ici, nous autorisons le passage en paramètre de l'incrémentation souhaitée. De plus, nous convertissons la valeur en nombre pour éviter toute concaténation potentielle par accident.

Plutôt que d'appeler la mutation comme une fonction normale (ce qu'elle n'est pas), nous utilisons une action spéciale : **commit**, qui permet d'acter la mutation :

```
this.$store.commit('INCREASE_COUNT', 5)
```

Quand une mutation est actée, l'action *commit* prend deux paramètre, le nom de la mutation et le payload (qui lui est facultatif).

> A vous de jouer, créez une mutation qui met à jour le *state* du *shoppingCart*.

Comme vous pouvez le remarquer, il nous reste les actions. Celles-ci sont similaires à la propriété *methods* dans une instance de Vue.

Dans la continuité de notre exemple de compteur, nous pourrions définir une action comme ci-dessous :

```
export default new Vuex.Store({
    state: {
        count: 0
    },
    mutations: {
        INCREASE_COUNT(state, amount = 1) {
            state.count += Number(amount)
        }
    },
    actions: {
        incrementCount(context, amount) {
            context.commit('INCREASE_COUNT', amount)
        }
    }
})
```

Comme vous pouvez le constater, une action ressemble à une fonction, avec un paramètre *context* et un *payload* optionnel comme pour les mutations. Ici : * *incrementCount* est le nom de l'action * le paramètre *context* nous donne accès aux mêmes méthodes et propriétés dans l'instance du store (exemple : commit, state, getters, ...) * *amount* est le payload, qui sera transmis à la mutation

Nous pourrions nous demander pourquoi ne pas appeler la mutation directement. Que se passerait-il si l'on veut décrémenter le *count* ?

```
export default new Vuex.Store({
    state: {
        count: 0
    },
    mutations: {
        INCREASE_COUNT(state, amount = 1) {
            state.count += Number(amount)
        },
        DECREASE_AMOUNT(state, amount = 1) {
            state.count -= Number(amount)
        }
    },
    actions: {
        updateCount({ commit }, amount) {
            if (amount >= 0) {
                commit('INCREASE_AMOUNT', amount)
            } else {
                commit('DECREASE_AMOUNT', amount)
            }
        }
    }
})
```

Vous vous demandez certainement pourquoi ne pas réunir les deux mutations en une seule *CHANGE_COUNT* ?

La réponse est simple : cela nous fournirait moins de détails lors du débuggage, ce qui rendrait celui-ci plus difficile. Par conséquent, gardez en tête que les mutations ne doivent garder qu'un seul usage. Laissez la logique aux actions.

Vous avez aussi probablement remarqué que le paramètre *context* a été échangé pour *{ commit }*. En effet, *context* peut être utilisé pour accéder à de nombreuses propriétés du store. J'ai donc utilisé ce que l'on appelle le **destructuring**, qui correspond à l'affectation par décomposition. Le **destructuring** sert à simplifier le code et le rendre plus lisible.

En plus de nous donner la liberté de déterminer la logique du moment où les mutations sont déclenchées, les*actions* sont **asynchrones**. Cela signifie que vous ne pouvez appeler des API et acter des mutations qu'en cas de réussite. Ou en fonction de ce que vous souhaitez réaliser.

Tout cela est intéressant (ps : j'espère), mais comment utiliser les actions dans les composants ?

Nous avons vu qu'il existe un terme spécial pour invoquer les mutations, acter ou *commit*. Il en existe donc un aussi pour les actions : **propager** (ou ***dispatch***). En d'autres termes, vous envoyez l'action pour exécuter une tâche. Si vous vouliez propager une action depuis votre composant, cela ressemblerait à ce qui suit :

```
<template>
    <div>
        <p>{{ count }}</p>
        <button @click="sendUpdateCountAction">Increment</button>
    </div>
</template>

<script>
import { mapState } from 'vuex'

export default {
    computed: {
        ...mapState(['count'])
    },
    methods: {
        sendUpdateCountAction() {
            this.$store.dispatch('updateCount')
        }
    }
}
</script>
```

Cependant, tout comme il existe *mapState* et *mapGetters*, il existe *mapActions* pour les actions :

```
<template>
    <div>
        <p>{{ count }}</p>
        <button @click="sendUpdateCountAction">Increment</button>
    </div>
</template>

<script>
import { mapState, mapActions } from 'vuex'

export default {
    computed: {
        ...mapState(['count'])
    },
    methods: {
        ...mapActions(['updateCount'])
    }
}
</script>
```

> A vous ! Migrez l'événement d'ajout au panier de *Home.vue* et *MenuItem.vue* vers Vuex. Créez une action qui peut être déclenchée depuis *MenuItem.vue* qui met à jour le panier dans *Home.vue*.