17.Manipuler des states complexes d'une application React avec Redux

17.1. Redux dans une app React

Dans la séance précédente nous avons découvert Redux. Il nous a permis de **mieux organiser la logique** des applications grâce à un système en trois parties : **state**, **actions** et **reducer**.

Nous avons également vu **le store** qui permet de lier ces éléments et de les faire fonctionner ensemble.

Pour utiliser React et Redux ensemble de manière efficace, on utilise une librairie tierce : React-Redux.

Pour utiliser React-Redux, il faut :

- Envoyer le **store** grâce au **Provider** qui englobe toute l'application.
- Utiliser **useDispatch** pour envoyer des actions depuis les composants.
- Utiliser **useSelector** pour extraire des morceaux de state et mettre à jour le composant en cas de changement de state.

17.2. State avec Immer

En Javascript, il existe 2 types de variables : les variables primitives (Booléen, Null, Undefined, Nombre, BigInt, Chaîne de caractères / String, Symbole ; qui sont non-mutables) et les objets prédéfinis (array, function, object, ... qui sont donc mutables).

Le contenu des objets prédéfinis peuvent donc être modifiés après leur initialisation. Par exemple, si nous définissons un objet :

```
const myObject = { test: true };
```

Nous pouvons très bien, ajouter ou supprimer des attributs par la suite :

```
myObject.otherAttribute = false;
```

Dans ce cas, l'objet garde la même référence en mémoire, il ne sera donc pas possible de savoir si l'objet a été modifié ou non après coup.

Il en est de même pour les tableaux, l'ajout de nouveaux éléments à un tableau ne permet pas de savoir s'il a modifié ou non car la référence est toujours la même.

Par défaut, JS ne propose de solution pour rendre les tableaux et objets immutables (bien que l'utilisation de proxy permette de contourner un peu le problème). De plus, certaines fonctions sont piégeuses car elles vont modifier le tableau. Nous pouvons citer :

- Ajout/suppression d'un élément : **push**, **shift**, **unshift**, **pop**, **delete**
- Ajout/suppression de plusieurs éléments : **splice**
- Modification de l'ordre : **reverse**, **sort**

Alors que les fonctions suivantes renvoient un nouveau tableau :

- Ajout d'éléments : **concat**
- Opération sur les éléments : **map**, **filter**, **reduce**
- Copie superficielle du tableau : **slice**

Pour éviter les problèmes de fonctions qui modifient ou non les tableaux ou objets, nous allons utiliser la librairie **immer** : https://immerjs.github.io/immer/

D'après les séances précédentes, Redux est basé sur 3 principes :

- Une unique source de vérité : il y a un seul état global dans votre application
- L'état global est accessible uniquement en lecture seule
- Les modifications de l'état global sont réalisées uniquement à l'aide de fonctions pures

Les 2 types de méthodes utilisées par Redux sont :

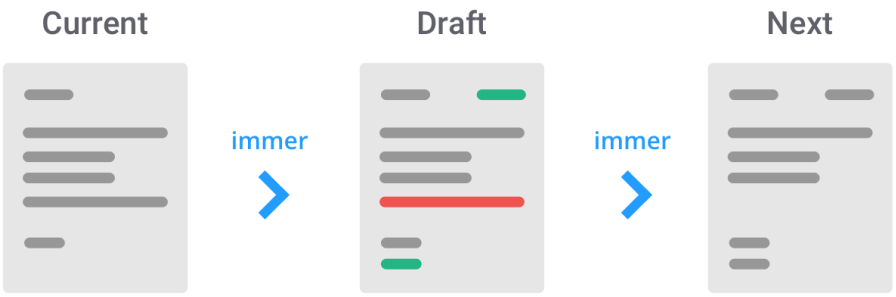
- Les **actions** : effectuent les appels externes (appels API, ...)
- Les **reducers** : mettent à jour l'état global à partir des nouvels données reçues dans les actions

Chaque méthode dans un reducer doit être une fonction "pure", c'est-à-dire, elle ne peut pas :

- **Modifier les arguments en paramètres**
- Utiliser des variables globales
- Modifier des variables globales
- Aucun appel externe (comme API)
- Appeler une autre fonction non "pure" (comme les dates ou des nombres aléatoires)

Pour résoudre le problème principal de modification des données, par inadvertance, contenues dans l'état global, il est fortement conseillé de se reposer sur des données immutables.

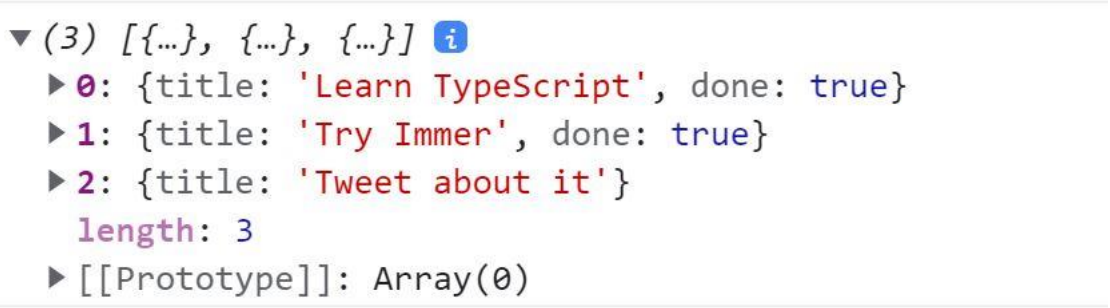
La librairie **Immer** va permettre de **modifier les données** comme si elles étaient de simple tableau ou objet mais sans impacter directement l'état global.



Un exemple pour la comparaison ;

```
const baseState = [
    {
        title: "Learn TypeScript",
        done: true
    },
    {
        title: "Try Immer",
        done: false
    }
]
```

Imaginez que nous avons l'état de base ci-dessus et que nous devions mettre à jour la deuxième tâche et ajouter une troisième. Cependant, nous ne voulons pas muter **baseState** d'origine, et nous voulons également éviter le clonage en profondeur (pour préserver la première tâche).



Sans Immer

Sans Immer, nous devrons copier chaque niveau de la structure d'état qui est affecté par notre changement :

```
const nextState = baseState.slice() // shallow clone the array
nextState[1] = {
    // replace element 1...
    ...nextState[1], // with a shallow clone of element 1
    done: true // ...combined with the desired update
}
nextState.push({title: "Tweet about it"})
```

Avec Immer

```
import produce from "immer"
const nextState = produce(baseState, draft => {
    draft[1].done = true
    draft.push({title: "Tweet about it"})
})
```

17.3. Les sélecteurs

- Un selector est une fonction que l'on passe à **useSelector** pour extraire un morceau du state.
- Il est possible d'utiliser les props dans les selectors, cela permet de faire des composants dynamiques.
- Pour simplifier les composants, on peut déclarer les selectors dans un fichier séparé ; si le composant utilise des props, il faut utiliser une fonction qui retourne le selector.
- Il ne faut jamais créer de référence dans les selectors, car React-Redux va croire qu'ils changent à chaque changement de state.

17.4. Les événements asynchrones

- Redux n'est pas capable de manipuler des événements asynchrones, mais on peut interagir avec dans une fonction asynchrone.
- Dans les composants React, on peut utiliser le hook **useStore** pour accéder au store et utiliser **getState** et **dispatch** dans notre action.
- Il est recommandé de déclarer les actions asynchrones dans des fonctions séparées (dans le fichier **store.js** , par exemple) ; il faut alors passer le store en paramètre.