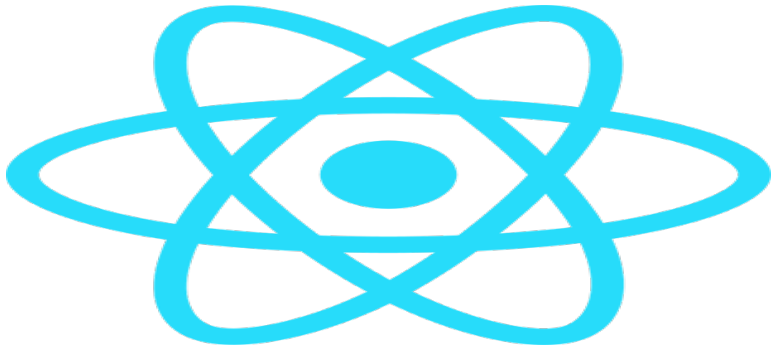
React : Redux

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. Slice
   - Reducers
   - Actions
4. Provider
5. Sélecteur
6. Dispatch
7. TP
8. Bonnes pratiques

React.js

Redux

- Gestionnaire d'état pour les applications **React.js**
- Céveloppé principalement par **Dan Abramov** et **Andrew Clark** en 2015 et présenté lors de la conférence **React Europe** en 2015 (ce qui a aidé à sa popularisation)
- Inspiré de la programmation fonctionnelle et du modèle Flux (modèle architectural conçu par **Facebook** pour gérer les données dans les applications **React**)

React.js

Redux : composants

- **State** : l'état global de l'application à un moment donné. C'est un simple objet **JavaScript** (immuable) contenant toutes les informations que l'application doit gérer.
- **Store** : le conteneur qui contient l'état global de l'application. Il esr créé en utilisant la fonction `createStore()` (ou `configureStore()`) de **Redux** et permet de
  - accéder à l'état via `store.getState()`,
  - modifier l'état via `dispatch(action)` et
  - couter les changements via `subscribe(listener)`.
- **Action** : un objet **JavaScript** décrivant ce qui doit se passer dans l'application et contenant au minimum
  - une clé type qui décrit le type d'événement/action et
  - des données supplémentaires sous forme de payload (charges utiles).
- **Reducer** : une fonction qui prend l'état actuel et une action, et renvoie un nouvel état.
- **Dispatch** : une méthode du store qui permet d'envoyer une action au **reducer**. Lorsqu'une action est envoyée via `dispatch()`, elle déclenche la logique dans le **reducer** pour calculer le nouvel état.

React.js

Rappel

Les **hooks** de **React** permettent aussi de gérer l'état d'une application.

React.js

Rappel

Les **hooks** de **React** permettent aussi de gérer l'état d'une application.

Question

Dans quel cas choisir l'un ou l'autre ?

	Redux	**React Hooks**
Complexité	Plus complexe à apprendre et configurer	Simple et natif de React, moins de configuration
État global	Excellent pour un état global partagé dans toute l'application	Gère l'état global, mais devient difficile à gérer à grande échelle
Performance	Optimisé pour les grandes applications	Bon pour les petites/moyennes applications
Écosystème	Riche avec outils comme **redux-devtools**	Moins d'outils disponibles
Scalabilité	Très scalable, utilisé dans des projets d'envergure	Scalabilité limitée par rapport à **Redux**

React

Commençons par créer un nouveau projet

```
npx create-react-app react-redux
```

React

Commençons par créer un nouveau projet

```
npx create-react-app react-redux
```

Pour lancer le projet

```
npm start
```

React.js

Pour installer `Redux`, exécutez la commande

```
npm install @reduxjs/toolkit react-redux
```

React.js

Pour installer `Redux`, exécutez la commande

```
npm install @reduxjs/toolkit react-redux
```


Explication

- **@reduxjs/toolkit** : package officiel de **Redux Toolkit** simplifiant l'utilisation de **Redux** en incluant des outils comme `createSlice`, `configureStore`...
- **react-redux** : binding officiel qui connecte **Redux** à **React**, permettantaux composants **React** d'accéder au store **Redux** via le `Provider` et les**hooks** comme `useSelector` et `useDispatch`.

React.js

Exemple

Nous voudrions gérer l'état d'un

- compteur (avec 4 actions : `increment`, `decrement`, `incrementAvecPas` et `decrementAvecPas`)
- panier (avec 3 actions : `addItem`, `removeItem` et `clearCart`

Structure de l'application

```
src/
|
|---- stores/
|      |---- store.js          // commun pour tous les états
|      |---- counter-slice.js // gère l'état du compteur
|      |---- cart-slice.js    // gère l'état du panier
|
|---- components
|      |---- Counter.js
|      |---- Cart.js
|
|---- index.js
```

Structure de l'application

```
src/
|
|---- stores/
|      |---- store.js          // commun pour tous les états
|      |---- counter-slice.js // gère l'état du compteur
|      |---- cart-slice.js    // gère l'état du panier
|
|---- components
|      |---- Counter.js
|      |---- Cart.js
|
|---- index.js
```


Explication

- `store.js` : contient la création et la configuration du store.
- `*-slice.js` : contient l'état initial, les actions et les reducers.

React.js

Store

- Créé avec `configureStore`
- Contenant la liste des reducers

React.js

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

```
import { configureStore } from '@reduxjs/toolkit'

export const store = configureStore({
    reducer: {
    },
});

export default store;
```

React.js

Dans `stores/counter-slice.js`**, commençons par déclarer l'état initial**

```
const initialState = {
    count: 0,
};
```

React.js

Dans `stores/counter-slice.js`**, commençons par déclarer l'état initial**

```
const initialState = {
    count: 0,
};
```

Importons ensuite `createSlice`

```
import { createSlice } from '@reduxjs/toolkit';
```

React.js

Utilisons `createSlice` pour spécifier le nom

```
const counterSlice = createSlice({
    name: 'counter',
});
```

React.js

Utilisons `createSlice` pour spécifier le nom

```
const counterSlice = createSlice({
    name: 'counter',
});
```


Explication

Le nom du slice (ici `counter`) est utilisé pour identifier le slice dans le store **Redux**. Il sera aussi utilisé pour générer automatiquement des noms d'actions comme `count/increment`.

React.js

Utilisons `createSlice` pour spécifier le nom

```
const counterSlice = createSlice({
    name: 'counter',
});
```


Explication

Le nom du slice (ici `counter`) est utilisé pour identifier le slice dans le store **Redux**. Il sera aussi utilisé pour générer automatiquement des noms d'actions comme `count/increment`.

Et l'état initial

```
const counterSlice = createSlice({
    name: 'counter',
    initialState,
});
```

React.js

Et les reducers

```
const counterSlice = createSlice({
    name: 'counter',
    initialState,
    reducers: {
        increment: (state) => {
            state.count += 1;
        },
        decrement: (state) => {
            state.count -= 1;
        },
        incrementAvecPas: (state, action) => {
            state.count += action.payload;
        },
        decrementAvecPas: (state, action) => {
            state.count -= action.payload;
        },
    },
});
```

React.js

Exportons enfin les actions et le reducer

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = {
    count: 0,
};

const counterSlice = createSlice({
    name: 'counter',
    initialState,
    reducers: {
        increment: (state) => {
            state.count += 1;
        },
        decrement: (state) => {
            state.count -= 1;
        },
        incrementAvecPas: (state, action) => {
            state.count += action.payload;
        },
        decrementAvecPas: (state, action) => {
            state.count -= action.payload;
        },
    },
});

export const { increment, decrement, incrementAvecPas, decrementAvecPas } = counterSlice.
  actions;

export default counterSlice.reducer;
```

React.js

Dans `store.js`**, importons puis déclarons** `counterReducer` **dans** `configureStore()`

```
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from './counter-slice'

export const store = configureStore({
    reducer: {
        counter: counterReducer,
    },
});

export default store;
```

React.js

Dans `index.js`, enveloppons notre application dans un `Provider` pour que `Redux` soit disponible partout dans l'application

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { Provider } from 'react-redux';
import store from './stores/store';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
      <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);

reportWebVitals();
```

React.js

Dans le composant `Count.js`, commençons par utiliser le hook `useSelector` pour sélectionner le `count` de notre store

```
import { useSelector } from 'react-redux';

const Counter = () => {

    const count = useSelector((state) => state.counter.count);

    return (
        <div>
            <h1>Compteur : {count}</h1>
        </div>
    );
};

export default Counter;
```

React.js

Importons puis préparons le hook `useDispatch` pour pouvoir utiliser les actions

```
import { useSelector, useDispatch } from 'react-redux';

const Counter = () => {

    const count = useSelector((state) => state.counter.count);
    const dispatch = useDispatch();

    return (
        <div>
            <h1>Compteur : {count}</h1>
        </div>
    );
};

export default Counter;
```

React.js

Ajoutons un état `step` pour les actions avec payload (comme `incrementAvecPas` et `decrementAvecPas`)

```
import { useSelector, useDispatch } from 'react-redux';
import { useState } from 'react';

const Counter = () => {

    const count = useSelector((state) => state.counter.count);
    const dispatch = useDispatch();
    const [step, setStep] = useState(1);

    return (
        <div>
            <h1>Compteur : {count}</h1>
        </div>
    );
};

export default Counter;
```

React.js

Importons les actions afin de les utiliser avec `dispatch`

```
import { useSelector, useDispatch } from 'react-redux';
import { useState } from 'react';
import { increment, decrement, incrementAvecPas, decrementAvecPas }
   from '../stores/counter-slice';

const Counter = () => {

    const count = useSelector((state) => state.counter.count);
    const dispatch = useDispatch();
    const [step, setStep] = useState(1);

    return (
        <div>
            <h1>Compteur : {count}</h1>
        </div>
    );
};

export default Counter;
```

Et enfin, les boutons pour dispatcher les actions

```
return (
    <div>
        <h1>Compteur : {count}</h1>
        <button onClick={() => dispatch(increment())}>Incrémenter</
          button>
        <button onClick={() => dispatch(decrement())}>Décrémenter</
          button>

        <div>
            <input
                type="number"
                value={step}
                onChange={(e) => setStep(Number(e.target.value))}
            />
            <button onClick={() => dispatch(incrementAvecPas(step))}>
                Incrémenter avec pas ({step})
            </button>
            <button onClick={() => dispatch(decrementAvecPas(step))}>
                Décrémenter avec pas ({step})
            </button>
        </div>
    </div>
);
```

React.js

Remarques

- **kebab-case** est généralement utilisé pour les noms de fichiers (modules), en particulier dans les projets **Node.js**, car il est simple, lisible et largement accepté dans l'écosystème **JavaScript**.
- **camelCase** est souvent utilisé pour nommer des variables, objets, propriétés (attributs), méthodes et fonctions.
- Choisir entre **camelCase** et **kebab-case** pour renommer les fichiers contenant les slices (`counter-slice.js` ou `counterSlice.js`) revient aux pratiques souvent adoptées dans vos projets.

React

Considérons le code du composant `Produit.js` utilisé dans les chapitres précédents

```
import { useRef, useState } from "react"

export default function Produit({ elt, onSendQuantite }) {
    const quantite = useRef(0)
    const [disabled, setDisabled] = useState(false)

    const ajouter = () => {
        onSendQuantite(quantite.current.value)
        setDisabled(true)
    }
    return (
        <li>
            {elt.nom} {elt.quantite} {elt.prix}
            <input type="number" ref={quantite}/>
            <button onClick={ajouter} disabled={disabled}>
                Ajouter
            </button>
        </li>
    )
}
```

React

Et le code du composant `Primeur.js`

```
import { useState } from 'react'
import Produit from './Produit'

export default function Primeur() {
    let produits = [
        { nom: "banane", prix: 3, quantite: 10 },
        { nom: "fraise", prix: 10, quantite: 20 },
        { nom: "poivron", prix: 5, quantite: 10 }
    ]
    let [total, setTotal] = useState(0)

    const calculerTotal = (qte, ind) => {
        setTotal(tot => tot + qte * produits[ind].prix)
    }
    return (
        <div>
            <h2>Primeur : {total}</h2>
            <ul>
                {
                    produits.map((elt, ind) =>
                        <Produit
                        key={ind} elt={elt}
                        onSendQuantite={(qte) => calculerTotal(qte, ind)} />
                    )
                }
            </ul>
        </div>
    )
}
```

React

Exercice : **primeur-produit**

- Créez les fichiers suivants et
  - `src/components/Cart.js`
  - `src/stores/cart-slice.js`
- Utilisez **Redux** pour
  - afficher le contenu du panier dans `Cart.js`
  - supprimer un produit du panier
  - vider le panier
  - déplacer le total de `Primeur.js` vers `Cart.js`

React

Correction (première partie de `cart-slice.js`)

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = {
  items: [], // liste des produits dans le panier
  total: 0   // total du panier
};
```

Correction (deuxième partie de `cart-slice.js`)

```
const cartSlice = createSlice({
  name: 'cart',
  initialState,
  reducers: {
    addItem(state, action) {
      const newItem = action.payload;
      const existingItem = state.items.find(item => item.nom === newItem.nom);
      if (existingItem) {
        existingItem.quantite += newItem.quantite;
      } else {
        state.items.push(newItem);
      }
      state.total += newItem.prix * newItem.quantite;
    },
    removeItem(state, action) {
      const nom = action.payload;
      const itemToRemove = state.items.find(item => item.nom === nom);
      if (itemToRemove) {
        state.total -= itemToRemove.prix * itemToRemove.quantite;
        state.items = state.items.filter(item => item.nom !== nom);
      }
    },
    clearCart(state) {
      state.items = [];
      state.total = 0;
    }
  }
});

export const { addItem, removeItem, clearCart } = cartSlice.actions;
export default cartSlice.reducer;
```

Correction (`store.js`)

```
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from './counter-slice'
import cartReducer from './cart-slice'

export const store = configureStore({
    reducer: {
        counter: counterReducer,
        cart: cartReducer,
    },
});

export default store;
```

Correction (`Cart.js`)

```
import { useSelector, useDispatch } from 'react-redux';
import { removeItem, clearCart } from '../stores/cart-slice';

const Cart = () => {
    const items = useSelector((state) => state.cart.items);
    const total = useSelector((state) => state.cart.total);
    const dispatch = useDispatch();

    return (
        <div>
            <h2>Panier</h2>
            {items.length === 0 ? (
                <p>Votre panier est vide.</p>
            ) : (
                <ul>
                    {items.map((item) => (
                        <li key={item.nom}>
                            {item.nom} - Quantité : {item.quantite} - Prix : {item.prix}
                            <button onClick={() => dispatch(removeItem(item.nom))}>
                                Supprimer
                            </button>
                        </li>
                    ))}
                </ul>
            )}
            <h3>Total : {total}</h3>
            <button onClick={() => dispatch(clearCart())}>Vider le panier</button>
        </div>
    );
};

export default Cart;
```

Correction (`Produit.js`)

```
import { useRef, useState } from 'react';
import { useDispatch } from 'react-redux';
import { addItem } from '../stores/cart-slice';

export default function Produit({ elt }) {
    const quantite = useRef(0);
    const [disabled, setDisabled] = useState(false);
    const dispatch = useDispatch();

    const ajouter = () => {
        const quantityValue = parseInt(quantite.current.value, 10);
        if (quantityValue > 0) {
            dispatch(
                addItem({
                    nom: elt.nom,
                    prix: elt.prix,
                    quantite: quantityValue
                })
            );
            setDisabled(true);
        }
    };

    return (
        <li>
            {elt.nom} {elt.quantite} {elt.prix}
            <input type="number" ref={quantite} min="1" />
            <button onClick={ajouter} disabled={disabled}>
                Ajouter
            </button>
        </li>
    );
}
```

React

Correction (`Primeur.js`)

```
import Produit from './Produit';

export default function Primeur() {
    let produits = [
        { nom: 'banane', prix: 3, quantite: 10 },
        { nom: 'fraise', prix: 10, quantite: 20 },
        { nom: 'poivron', prix: 5, quantite: 10 }
    ];

    return (
        <div>
            <h2>Produits disponibles</h2>
            <ul>
                {produits.map((elt, ind) => (
                    <Produit key={ind} elt={elt} />
                ))}
            </ul>
        </div>
    );
}
```

React.js

Bonnes pratiques

- Séparez les actions, les réducteurs et la configuration du store dans des fichiers distincts pour une meilleure lisibilité (un fichier pour le store et un pour chaque slice).
- N'altérez jamais directement l'état dans les réducteurs. Utilisez des méthodes comme `Object.assign` ou l'opérateur spread (...) pour retourner un nouvel objet.
- Les actions ne devraient contenir que les informations nécessaires pour l'état. Pas de données supplémentaires ou inutiles.
- Ne stockez pas des données éphémères ou des états d'interface utilisateur dans **Redux** (comme les états de formulaire ou d'affichage de modales), utilisez plutôt le state local des composants **React**.
  - Les actions `increment` et `decrement` n'ont pas de payload, car elles n'en ont pas besoin.
  - Les actions `incrementAvecPas` et `decrementAvecPas` utilisent le payload pour passer le pas (`step`), ce qui est approprié et minimal.

React

Le code suivant respecte les règles de l'immuabilité et les bonnes pratiques de `Redux`

```
state.count += 1;
```

React

Le code suivant respecte les règles de l'immuabilité et les bonnes pratiques de `Redux`

```
state.count += 1;
```


Explication

- Dans **Redux** classique, on doit gérer manuellement l'immutabilité de l'état en utilisant des méthodes comme `Object.assign` ou l'opérateur spread (...).
- Cependant, **@reduxjs/toolkit** utilise la bibliothèque **Immer** sous le capot, qui permet de muter directement l'état dans les réducteurs.
- En effet, **Immer** capture les modifications et retourne un nouvel état immuable en arrière-plan.

React

Dans `package-lock.json`, on peut facilement remarquer la présence de `Immer` comme sous-dépendance de `@reduxjs/toolkit`

```
"node_modules/@reduxjs/toolkit": {
    "version": "2.2.7",
    "resolved": "https://registry.npmjs.org/@reduxjs/toolkit/-/toolkit-2.2.7.tgz",
    "license": "MIT",
    "dependencies": {
        "immer": "^10.0.3",
        "redux": "^5.0.1",
        "redux-thunk": "^3.1.0",
        "reselect": "^5.1.0"
    },
    "peerDependencies": {
        "react": "^16.9.0 || ^17.0.0 || ^18",
        "react-redux": "^7.2.1 || ^8.1.3 || ^9.0.0"
    },
    "peerDependenciesMeta": {
        "react": {
            "optional": true
        },
        "react-redux": {
            "optional": true
        }
    }
},
```