Les extensions Redux

Table des matières

I. Contexte **3**

II. Liaison avec React : React Redux **3**

III. Exercice : Appliquez la notion **9**

IV. Redux Toolkit **10**

V. Exercice : Appliquez la notion **13**

VI. Persistance locale avec Redux Persist **13**

VII. Exercice : Appliquez la notion **16**

VIII. Essentiel **17**

IX. Auto-évaluation **17**

A. Exercice final 17

B. Exercice : Défi 19

Solutions des exercices **20**

I. Contexte

Durée : 1 h 30

Environnement de travail : Visual Studio Code

Pré-requis : React, Redux

Contexte

La popularité de Redux a été sans précédent durant les dernières années. À ce titre, Redux est devenu beaucoup plus qu'un simple package : tout un écosystème s'est construit autour de l'outil principal.

Ces extensions ont de multiples objectifs, comme simplifier la syntaxe, permettre l'ajout d'actions asynchrones dans la boucle Flux ou encore gérer l'intégration avec React. Nous allons voir une sélection de ces extensions, que vous pouvez utiliser conjointement à Redux dans vos projets.

II. Liaison avec React : React Redux

Objectifs

- Comprendre que React Redux permet une meilleure intégration de Redux avec les composants React
- Installer React Redux
- Comprendre comment définir un container et créer la liaison entre le store et le composant
- Comprendre qu'il est maintenant nécessaire de modifier les imports pour prendre en compte la couche de container

Mise en situation

Il est bon de rappeler que Redux est un outil agnostique du framework JavaScript, avec lequel il est utilisé. Cette flexibilité se fait au prix d'une moins bonne intégration avec React. Il sera nécessaire de créer cette liaison entre les deux outils manuellement par le biais des méthodes disponibles sur le store, ou alternativement de passer par un outil qui viendra se placer entre le data store et les composants.

Nous avons déterminé qu'il existait deux manières pour un composant React d'interagir avec le store Redux : s'y abonner, ce qui lui confère un droit en lecture sur le state, et déclencher des actions, qui seront responsable de muter le state.

Quand nous avions déclaré notre store Redux, nous avions vu que celui-ci mettait à notre disposition la fonction :

```
1 store.subscribe(() => {});
```

Cette méthode permettait de déclencher la fonction anonyme passée en paramètre à l'initialisation, et à toutes les mutations qui pouvaient advenir sur le state.

La fonction :

```
1 store.dispatch({});
```

Permettait, de son côté, d'exécuter une action de mutation sur le state.

Nous allons maintenant voir que la librairie React Redux permet de gérer cette liaison de manière plus simple et automatisée. React Redux ne remplace pas Redux, il est nécessaire d'installer les deux outils dans un nouveau projet :

```
1 npm install --save redux react redux
```

Exemple

Imaginons alors un store très simple, dans un fichier **store.js** :

```
import { createStore } from 'redux'

const initialState = {
  products: [
    {
      id: Math.random(),
      name: 'iPhone XR - 64gb',
      brand: 'Apple',
      price: 599
    },
    {
      id: Math.random(),
      name: 'Macbook Pro',
      brand: 'Apple',
      price: 2129
    },
    {
      id: Math.random(),
      name: 'Airpods Pro',
      brand: 'Apple',
      price: 279
    }
  ],
  cart: []
}

const shopReducer = (state = initialState, action) => {
  switch(action.type) {
    case 'BUY_ITEM':
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    case 'EMPTY_CART':
      return {
        ...state,
        cart: []
      }
    default:
      return state
  }
}

const store = createStore(shopReducer)

export default store
```

Ce store possède un reducer qui permet de gérer une boutique en ligne. Le state est constitué de la liste des produits disponibles sur le site, et du contenu du panier de l'utilisateur. Deux actions sont présentes dans le reducer : `BUY_ITEM` qui permet d'ajouter un item au panier, et `EMPTY_CART` qui permet de réinitialiser le panier à un tableau vide.

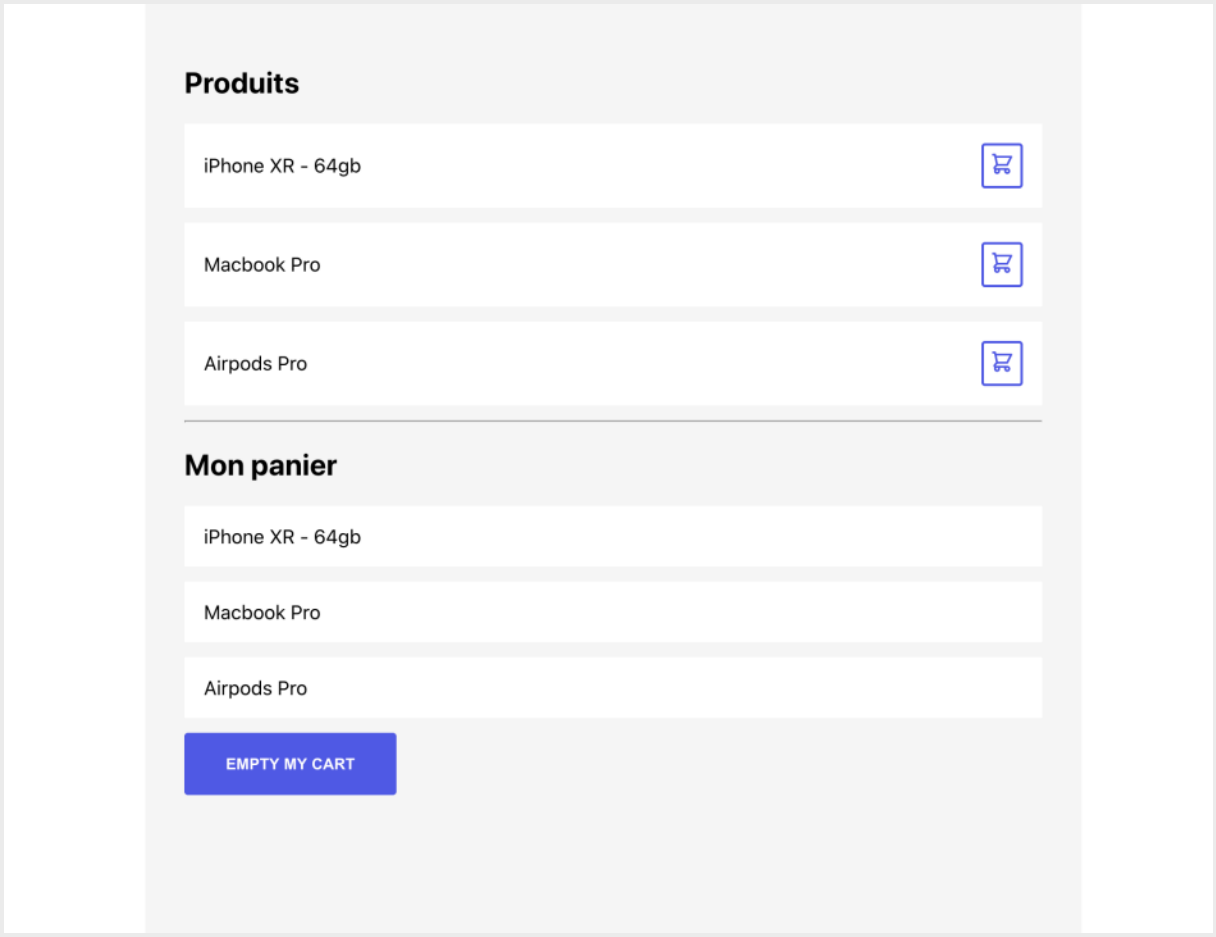
En utilisant React Redux, nous allons maintenant mettre au point le composant `<Shop />` qui affichera la liste des produits. Un bouton en face de chaque produit permettra de l'ajouter au panier. Les éléments du panier seront listés juste en dessous, avec un bouton unique permettant de vider le panier.

React Redux va se reposer sur le modèle de la « *double view* » proposé par Flux. Comme nous avions pu le voir dans le pattern Flux, il existe deux types de views : les presentation views, responsables d'afficher les éléments sur la page (elles peuvent s'apparenter à nos composants React), et les container views, qui gèrent une partie de la logique d'affichage liée au state global et qui font office de passe-plat entre le store et la presentation view.

Nous allons commencer par construire le composant d'affichage `<Shop />`. Pour le moment, le composant n'aura pas accès au global state, et nous allons présumer que notre composant aura accès aux variables `products` qui portera la liste des produits, `cart` qui portera la liste des éléments de notre panier, et enfin `onAddItemToCart` et `onEmptyCart`, qui sont deux fonctions permettant respectivement d'ajouter un item au panier et de vider le panier. Nous allons simplement mettre en place les éléments d'affichage et nous reviendrons plus tard sur ce fichier.

```
import React from 'react'
import styled from 'styled-components'

const Shop = () => {
  return (
    <Wrapper>
      <h2>Produits</h2>
      <ul>
        {products.map((product) => (
          <Product key={product.id}>
            {product.name}
            <ButtonIcon
              type="button"
              onClick={() => onAddItemToCart(product)}
            >
              <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke="currentColor">
                <path strokeLinecap="round" strokeLinejoin="round" strokeWidth={2} d="M3 3h2l.4 2M7 13h10l4-8H5.4M7 13L5.4 5M7 13l-2.293 2.293c-.63.63-.184 1.707.707 1.707H17m0 0a2 2 0 100 4 2 2 0 000-4zm-8 2a2 2 0 11-4 0 2 2 0 014 0z" />
              </svg>
            </ButtonIcon>
          </Product>
        ))}
      </ul>

      <hr />

      <h2>Mon panier</h2>

      {cart.length === 0
        ? <Empty>Aucun article dans votre panier</Empty>
        : (
          <ul>
            {cart.map((cartItem) => (
              <Product key={`${cartItem.id}-cart`}>{cartItem.name}</Product>
            ))}
          </ul>
        )
      }

      <Button
        type="button"
        onClick={onEmptyCart}
      >
        Empty my cart
      </Button>
```

```
    </Wrapper>
  )
}

const Wrapper = styled.div`
  width: 48rem;
  min-height: 100vh;
  margin: 0 auto;
  background: #F5F5F5;
  padding: 2rem;
`

const Product = styled.li`
  display: flex;
  align-items: center;
  justify-content: space-between;
  background: #fff;
  margin: 0.75rem 0;
  padding: 1rem;
`

const Button = styled.button`
  border-radius: 3px;
  border: none;
  border: 2px solid #5352ed;
  background: #5352ed;
  color: #fff;
  font-weight: bold;
  text-transform: uppercase;
  font-size: 0.8rem;
  padding: 1rem 2rem;
  cursor: pointer;
`

const ButtonIcon = styled(Button)`
  background: none;
  padding: 5px;

  svg {
    width: 20px;
    color: #5352ed;
  }
`

const Empty = styled.span`
  display: block;
  margin: 2rem auto;
`

export default Shop
```



Mettons maintenant en place le container dans un fichier **Shop.js** placé dans un nouveau répertoire `/src/Container`. Le fait que le nom soit commun entre notre composant d'affichage et son container est volontaire, nous verrons pourquoi dans un instant.

```
import { connect } from 'React Redux'
import Shop from '../Shop'

const mapStateToProps = state => {
  return {
    products: state.products,
    cart: state.cart
  }
}

const mapDispatchToProps = dispatch => {
  return {
    onAddItemToCart: item => {
      dispatch({ type: 'BUY_ITEM', payload: item })
    },
    onEmptyCart: () => {
      dispatch({ type: 'EMPTY_CART' })
    }
  }
}

const ShopContainer = connect(mapStateToProps, mapDispatchToProps)(Shop)

```

```
export default ShopContainer
```

Reprenons les éléments de ce container pas-à-pas. Nous importons deux éléments : la fonction `connect` de React Redux, et notre composant d'affichage `<Shop />`.

Deux fonctions anonymes sont ensuite définies : `mapStateToProps` et `mapDispatchToProps`, qui sont responsables de créer une liaison entre les éléments du state et notre composant.

`mapStateToProps` reçoit en paramètre le state et retourne un objet avec les propriétés `products` et `cart`, qui correspondent à leurs équivalents dans le global state.

Même chose du côté de `mapDispatchToProps`, qui reçoit la fonction `dispatch` en paramètre et qui nous permet de définir deux fonctions, `onAddItemToCart` et `onEmptyCart`, qui vont dispatch les actions correspondantes dans le store.

`mapStateToProps` et `mapDispatchToProps` sont ensuite liées à notre composant d'affichage, `<Shop />`, via la fonction `connect` importée depuis React Redux.

Il faut comprendre que le composant d'affichage `<Shop />` et son container associé (le fichier `/src/Container/Shop.js` que nous venons de créer) sont très fortement couplés : il est donc nécessaire, pour pouvoir bénéficier de la liaison avec Redux, d'importer le container qui importera lui-même le composant d'affichage. Nous devons donc modifier le composant parent `<App />` en conséquence :

```
import React from 'react';
import './App.css';
import Shop from './Container/Shop';
import { Provider } from 'React Redux';
import store from './store';

function App() {
  return (
    <div className="App">
      <Provider store={store}>
        <Shop />
      </Provider>
    </div>
  );
}

export default App;
```

Le store est injecté dans l'arbre de composants par le biais d'un nouveau composant `<Provider />` qui prend le store en props. Le composant `<Shop />` représenté juste en dessous n'est pas notre composant d'affichage (le fichier `/src/Shop.js`), mais bien le container (le fichier `/src/Container/Shop.js`). Pour faciliter la maintenabilité du code, il est généralement recommandé de copier la structure de fichiers de nos composants d'affichage à l'identique dans le répertoire `/src/Container`, afin que chaque composant ait sa contrepartie et qu'il soit simplement nécessaire d'ajouter `/Container/` dans les chemins d'imports pour refléter ce changement.

Revenons à présent sur notre composant d'affichage, `<Shop />`, qui reçoit maintenant sous la forme de props tous les éléments définis dans les fonctions `mapStateToProps` et `mapDispatchToProps` du container :

```
import React from 'react'
import styled from 'styled-components'

const Shop = ({ products, cart, onAddItemToCart, onEmptyCart }) => {
  return (
    /*...*/
  )
}

```

```
export default Shop
```

Syntaxe **À retenir**

- React Redux permet une meilleure intégration de l'outil Redux dans une application React, en nous autorisant à créer une couche de « container » pour nos composants d'affichage. Cette couche de container sera responsable de la liaison des éléments du store Redux, ainsi que des actions de mutation avec notre composant, qui recevra ces éléments sous la forme de props.
- Il sera cependant nécessaire de bien penser à modifier nos imports existants, qui pointaient vers le composant d'affichage, pour les faire maintenant pointer vers les containers qui se chargeront d'importer à leur tour le composant d'affichage.
- Afin de réduire la charge mentale liée à ce processus, il est généralement recommandé de reproduire à l'identique la structure de fichiers des composants dans un nouveau répertoire `/src/Container`. De cette manière, ajouter un container passera simplement par la modification des lignes d'imports de `./MonComposant` à `./Container/MonComposant`.

III. Exercice : Appliquez la notion

Question

[solution n°1 p.21]

À partir du store fourni, écrivez le contenu du fichier `/src/Container/Todo.js` pour réaliser la liaison avec un composant d'affichage `<Todo />`. Le composant devra recevoir les props suivantes :

- `todos` : Tableau de la liste des *todo*
- `user` : Objet contenant les informations utilisateur
- `onAddTodo` : Fonction qui appelle l'action `ADD_TODO`
- `onMarkTodoAsDone` : Fonction qui appelle l'action `MARK_TODO_AS_DONE`

```
import { createStore } from 'redux'

const initialState = {
  todos: [],
  user: {
    name: 'John',
    username: 'itsjohn',
    token:
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIn0.dozjgNryP4J3jVmNHl0w5N_XgL0n3I9PlFUP0THs
  }
}

const todoReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return [...state.todos, action.payload]
    case 'MARK_TODO_AS_DONE':
      const targetTodo = state.todos.find(todo => todo.id === action.payload)
      return [
        ...state.todos.filter(todo => todo.id !== action.payload),
        { ...targetTodo, done: true }
      ]
    default:
      return state
  }
}
```

```
26
27 const store = createStore(todoReducer)
28
29 export default store
```

IV. Redux Toolkit

Objectifs

- Découvrir un nouvel outil pour simplifier la syntaxe de Redux
- Apprendre comment définir une action avec `createAction`
- Apprendre comment simplifier l'écriture d'un reducer avec `createReducer`
- Voir une nouvelle structure : la *slice* permettant de rassembler le state, les reducers et les actions liés par un thème commun

Mise en situation

La syntaxe de Redux est souvent intimidante pour les nouveaux développeurs : entre les multiples reducers, la syntaxe `switch...case`, les actions qui sont des objets, on peut vite se sentir submergé. Redux propose un nouvel outil officiel, Redux Toolkit, visant à simplifier l'utilisation de Redux par le biais d'une syntaxe plus proche de ce que avons l'habitude d'utiliser.

Pour installer Redux Toolkit, il est simplement nécessaire de saisir la commande :

```
1 npm install @reduxjs/toolkit
```

Cet outil ne remplace par Redux, et il est nécessaire de l'installer également dans le projet, si ce n'est pas déjà fait. Redux Toolkit est bien sûr compatible avec React Redux, mais il sera nécessaire d'adapter un peu notre container. Nous verrons ces modifications à la fin de cette section.

Exemple

Une fois installé, nous allons nous concentrer sur notre fichier `store.js`. Redux Toolkit met à disposition une nouvelle fonction pour déclarer le store, `configureStore` :

```
1 import { configureStore } from '@reduxjs/toolkit'
2
3 const shopReducer = (state, action) => state
4
5 const store = configureStore({
6   reducer: shopReducer
7 })
8
9 export default store
```

Ce n'est pas un changement radical, mais la structure objet de `configureStore` va nous permettre de faire des choses intéressantes.

Dans le cas de plusieurs reducers, il est possible de passer à la propriété `reducer` un objet contenant tous les reducers. Ceux-ci seront automatiquement combinés en un reducer racine avec `combineReducers()`. La propriété `middleware` permet d'injecter très rapidement des middleware à Redux (nous verrons l'utilisation de middleware dans la dernière partie). Le devTools est activé par défaut avec `configureStore`, mais il est possible de le désactiver en passant explicitement `false` à la propriété `devTools`.

Concentrons-nous maintenant sur la définition du reducer `shopReducer`.

Nous allons importer deux nouvelles fonctions mises à disposition par Redux Toolkit : `createAction` et `createReducer`.

`createAction` va nous permettre de définir nos types d'actions sous la forme de constantes dans le store. De cette manière, elles sont plus facilement réutilisables et le risque d'erreur de saisie du nom diminue.

Reprenons nos actions précédemment définies et transformons-les en utilisant `createAction` :

```
import { configureStore, createAction, createReducer } from '@reduxjs/toolkit'

const buyItem = createAction('BUY_ITEM')
const emptyCart = createAction('EMPTY_CART')

/* store */
```

Modifions maintenant le reducer en lui-même en utilisant `createReducer` :

```
const initialState = { /*...*/ }
const shopReducer = createReducer(initialState, {
  [buyItem]: state => ({
    ...state,
    cart: [...state.cart, action.payload]
  }),
  [emptyCart]: state => ({
    ...state,
    cart: []
  })
} )
```

La syntaxe `switch...case` n'est plus présente, elle est remplacée par un système de tableau associatif où les actions sont désignées directement par les constantes `buyItem` et `emptyCart`, déclarées juste avant.

Redux Toolkit apporte une nouvelle entité, la slice, qui correspond à la fois à un reducer, à son state initial et à la déclaration de ses actions. Utiliser une slice va nous aider à simplifier encore notre syntaxe en remplaçant la déclaration de nos actions via `createAction`. Notre code complet devient alors :

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

const shopSlice = createSlice({
  name: 'shop',
  initialState: {
    products: [
      {
        id: Math.random(),
        name: 'iPhone XR - 64gb',
        brand: 'Apple',
        price: 599
      },
      {
        id: Math.random(),
        name: 'Macbook Pro',
        brand: 'Apple',
        price: 2129
      },
      {
        id: Math.random(),
        name: 'Airpods Pro',
        brand: 'Apple',
        price: 279
      }
    ],
    cart: []
  },
  reducers: {
```

```
    buyItem(state, action){
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    },
    emptyCart: state => ({
      ...state,
      cart: []
    })
  }
})

const store = configureStore({
  reducer: shopSlice.reducer
})

export const { buyItem, emptyCart } = shopSlice.actions
export default store
```

Maintenant que nous avons mis en place une slice dans notre store Redux, notre container ne va plus être en mesure de déclencher des actions. Le « type » des actions ayant changé, il sera alors nécessaire d'importer les actions depuis le fichier **store.js** et de modifier le code du container comme suit :

```
import { buyItem, emptyCart } from '../store'

/* ... */

const mapDispatchToProps = dispatch => {
  return {
    onAddItemToCart: item => {
      dispatch(buyItem(item))
    },
    onEmptyCart: () => {
      dispatch(emptyCart())
    }
  }
}
```

Ce sont maintenant des fonctions qui sont placées en paramètres de `dispatch` et non plus les objets des actions. Nous devrions alors retrouver un fonctionnement normal entre Redux Toolkit et React Redux.

Syntaxe **À retenir**

- Redux Toolkit est un outil officiel dont l'usage est recommandé par l'équipe de Redux. Redux Toolkit a pour objectif de simplifier la syntaxe et de permettre au développeur de construire le store, les actions, les reducers et les slices avec moins de code. Redux Toolkit est compatible avec les autres outils de l'écosystème puisqu'il ne modifie pas le fonctionnement de Redux, mais seulement la syntaxe permettant de créer les diverses entités.

V. Exercice : Appliquez la notion

Question

[solution n°2 p.21]

Vous allez devoir transformer le store Redux fourni en utilisant Redux DevTools. Vous utiliserez une slice et exporterez vos actions sous la forme de fonctions.

```
import { createStore } from 'redux'

const initialState = {
  todos: [],
  user: {
    name: 'John',
    username: 'itsjohn',
    token:
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIn0.dozjgNryP4J3jVmNHl0w5N_XgL0n3I9PlFUP0THs
  }
}

const todoReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return [...state.todos, action.payload]
    case 'MARK_TODO_AS_DONE':
      const targetTodo = state.todos.find(todo => todo.id === action.payload)
      return [
        ...state.todos.filter(todo => todo.id !== action.payload),
        { ...targetTodo, done: true }
      ]
    default:
      return state
  }
}

const store = createStore(todoReducer)

export default store
```

VI. Persistance locale avec Redux Persist

Objectifs

- Faire un point sur ce que permet de faire et de ne pas faire `localStorage`
- Voir comment installer et configurer Redux Persist
- Vérifier le bon fonctionnement de la persistance

Mise en situation

Comme nous l'avons dit, le state, qu'il soit localisé sur un composant ou qu'il soit global et géré par Redux, est volatil et non-persisté. Redux Persist est un outil qui va nous permettre de persister le contenu de notre state global dans le `localStorage`, la couche de persistance côté navigateur client.

localStorage

Il faut être clair sur ce qu'est et n'est pas le `localStorage`. Afin d'illustrer cette analyse, nous allons nous concentrer sur deux cas d'utilisation.

La persistance en `localStorage` n'est pas adaptée pour un stockage de longue durée, ni pour remplacer d'aucune manière une application back-end et une base de données. Il faut toujours garder à l'esprit qu'en plus d'avoir une taille limite qui ne nous permettrait pas de stocker énormément de données, le `localStorage` peut être vidé à tout instant par l'utilisateur. Les données sont également liées à la machine et à l'ordinateur avec lequel elles ont été stockées, ce qui signifie qu'il n'existe aucun moyen de récupérer des données depuis le `localStorage` d'un autre navigateur, encore moins depuis une autre machine.

En revanche, dans le cadre d'une application pouvant fonctionner en mode hors-ligne, il peut être intéressant de rapidement persister le contenu de notre state dans le `localStorage` du navigateur client lorsqu'on n'est pas en mesure de contacter le serveur back-end, et ce afin de pouvoir reprendre un fonctionnement nominal quelques minutes plus tard. Dans le cas où les données et leur sauvegarde ne seraient pas critiques – par exemple, un objet lié aux préférences utilisateur (thème, langue...) –, il peut être également envisageable d'utiliser le `localStorage`.

Installation

Redux Persist est une extension de Redux permettant très simplement de créer une copie du state global de l'application dans le `localStorage`. Pour l'installer, il suffit simplement d'exécuter la commande :

```
npm i redux-persist
```

Exemple

Une fois l'outil installé, il faudra venir modifier notre fichier `/src/store.js`, que nous pouvons reprendre depuis les sections précédentes.

```
import { configureStore, createSlice } from '@reduxjs/toolkit'
import { persistStore, persistReducer } from 'redux-persist'
import storage from 'redux-persist/lib/storage'

const shopSlice = () => { /* ... */ }

const persistConfig = {
  key: 'root',
  storage,
}

const persistedReducer = persistReducer(persistConfig, shopSlice.reducer)

const store = configureStore({
  reducer: persistedReducer
})

export const persistor = persistStore(store)
export const { buyItem, emptyCart } = shopSlice.actions
export default store
```

Redux Persist nous laisse importer deux fonctions : `persistStore` et `persistReducer`. Il est nécessaire de « wrapper » nos reducers avec la fonction `persistReducer()` avant de passer le reducer au store et, de même, de wrapper le store avant d'exporter le résultat.

Il est nécessaire de définir un objet de configuration pour la persistance en indiquant quelle méthode sera choisie par Redux Persist. C'est le rôle de l'import `storage` qui, sur une utilisation web, utilise par défaut `localStorage`.

Il est également nécessaire de modifier notre composant d'entrée `<App />` pour y ajouter un composant `<PersistGate />`, tel ce que nous avions déjà fait pour le composant `<Provider />`, qui expose le state aux composants enfants.

```
import React from 'react';
import './App.css';
import Shop from './containers/Shop';
import { Provider } from 'react-redux';
import store from './store';
import { PersistGate } from 'redux-persist/integration/react';
import { persistor } from './store';

function App() {
  return (
    <div className="App">
      <Provider store={store}>
        <PersistGate loading={null} persistor={persistor}>
          <Shop />
        </PersistGate>
      </Provider>
    </div>
  );
}

export default App;
```

Le nouveau composant `<PersistGate />` accepte deux props : `persistor`, qui prend simplement en paramètre l'élément `persistor` exporté depuis notre fichier `/src/store.js`, et `loading` qui accepte du code JSX.

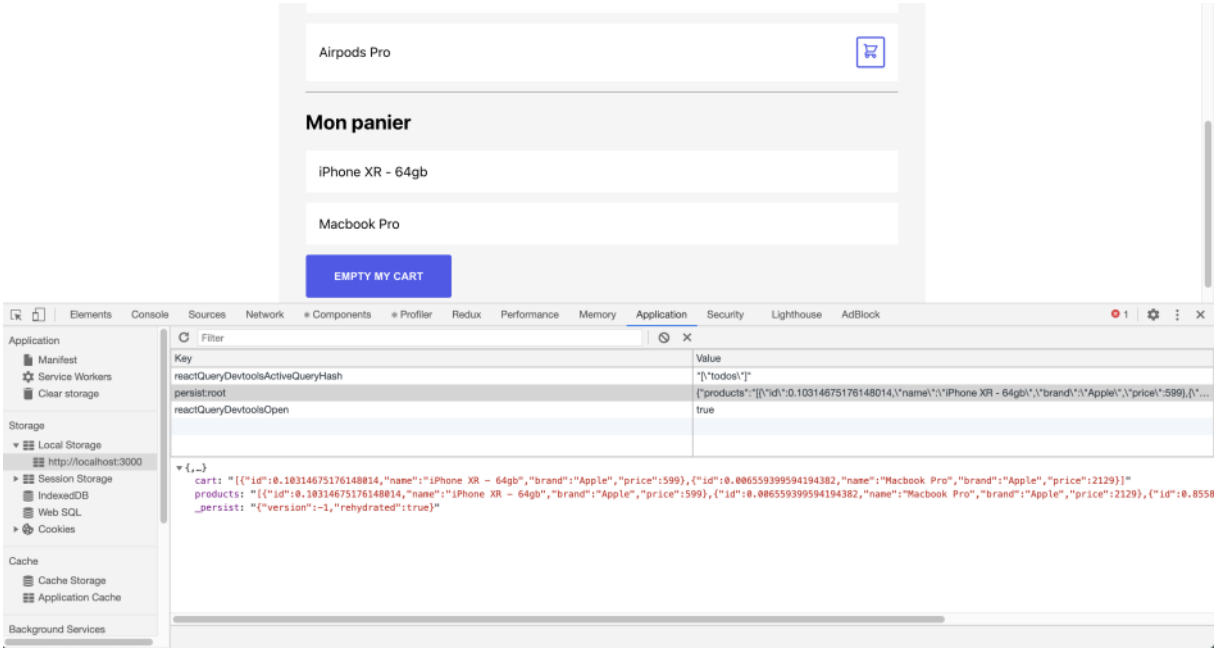
En effet, l'affichage de la page sera bloqué le temps que Redux Persist récupère le contenu du `localStorage`. Le JSX dans la prop `loading` est alors montré à l'écran jusqu'à ce que les données soient remontées.

Confirmation de la persistance

Notre store est maintenant persisté. Dans ChromeDevTools, il est possible de vérifier cela en se rendant dans l'onglet *Application*, puis dans le menu de gauche, sous *Storage*, en développant *Local Storage*, puis en sélectionnant le domaine de l'application (http://localhost:3000 par défaut pour une application en développement).

Vous devriez alors voir apparaître une clé `persist:root` qui contient tout le global state. Si nous plaçons des items dans notre panier et que nous rechargeons la page, celui-ci devrait revenir à son état initial vide. Mais, avec Redux Persist, il conserve les données dès le chargement.

En effet, Redux Persist récupère au chargement le contenu du `localStorage` et le ventile dans le global state de manière automatique.



Syntaxe **À retenir**

- Redux Persist est un outil permettant d'aller contre le fonctionnement de base du global state. En effet, Redux Persist permet de persister le global state dans une couche de stockage local, comme `localStorage`. Cette manière de faire pourrait ne pas être adaptée en fonction de la typologie de votre projet, mais pour de la persistance temporaire ou de données non-critiques, c'est une solution envisageable.

VII. Exercice : Appliquez la notion

Question

[solution n°3 p.22]

En reprenant le store créé initialement avec Redux, puis modifié avec le Redux Toolkit fourni, vous ferez en sorte que le global state soit persisté dans le `localStorage` en utilisant Redux Persist. Vous indiquerez le code modifié pour le fichier de store et le composant `<App />`.

```
import { configureStore, createSlice, createAction } from '@redux/toolkit'

const todoSlice = createSlice({
  name: 'todo',
  initialState: {
    todos: [],
    user: {
      name: 'John',
      username: 'itsjohn',
      token:
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIn0.dozjgNryP4J3jVmNHl0w5N_XgL0n3I9PlFUP0THs
    }
  },
  reducers: {
    addTodo(state, action) {
      return {
        ...state,
        todos: [...state.todos, action.payload]
      }
    },
```

```
    markTodoAsDone(state, action) {
      const targetTodo = state.todos.find(todo => todo.id === action.payload)
      return {
        ...state,
        todos: [
          ...state.todos.filter(todo => todo.id !== action.payload),
          { ...targetTodo, done: true }
        ]
      }
    }
  }
})

const store = configureStore({
  reducer: todoSlice.reducer
})

export const { addTodo, markTodoAsDone } = todoSlice.actions
export default store
```

VIII. Essentiel

IX. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°4 p.23]

Exercice

Par défaut, Redux est...

- O Fortement couplé à React
- O Agnostique

Exercice

Quelle est l'entité placée entre le store Redux et un composant d'affichage permettant la liaison du global store ?

- O Un wrapper
- O Un proxy
- O Un container

Exercice

Comment sont nommées les fonctions permettant de créer la liaison avec les éléments du state global ou les actions avec un composant ?

- O `linkGlobalState` et `proxyActions`
- O `mapStateToProps` et `mapDispatchToProps`
- O `bindStoreToComponent` et `connectMutations`

Exercice

Quel est le composant, importé depuis React Redux, permettant d'injecter le store dans un arbre de composants ?

- <Provider />
- <Store />
- <MapGlobalState />

Exercice

En utilisant Redux Toolkit, quelle est la méthode qui vient remplacer `createStore()` ?

- spawnStore()
- makeStore()
- configureStore()

Exercice

Quel est l'élément de syntaxe d'un reducer que Redux Toolkit permet de remplacer par un tableau associatif ?

- Le `switch...case`
- Le fait d'être obligé de retourner le state en entier (immutabilité)
- L'action sous la forme d'un objet JavaScript

Exercice

Quelle est la méthode Redux Toolkit qui permet de transformer les actions en fonctions ?

- createReducer()
- createAction()
- connect()

Exercice

Quelle est la nouvelle entité définie par Redux Toolkit qui permet d'englober le state, les reducers et les actions liés à un même thème ?

- Le module
- Le chunk
- La slice

Exercice

Persister des éléments en `localStorage` est pertinent si...

- On souhaite un stockage de longue durée
- On traite avec des données sensibles
- Le stockage est temporaire ou les données non-critiques

Exercice

Quel est le rôle de la prop `loading` dans le composant `<PersistGate loading={null} persistor={persistor}>` ?

- Le JSX placé dans cette prop sera affiché le temps que Redux Persist récupère le contenu du `localStorage`
- La promesse placée dans cette prop se résout quand le state est ventilé

B. Exercice : Défi

En reprenant des travaux déjà avancés, vous allez devoir construire un forum en vous reposant sur un store Redux.

Question

[solution n°5 p.25]

Vous récupérerez le projet mis à disposition sous la forme d'une archive. Après avoir téléchargé et décompressé l'archive dans votre dossier **projets**, il sera nécessaire d'installer les dépendances avec la commande `npm install`.

Vous trouverez dans le projet les composants du forum déjà construits, mais qui n'affichent rien pour l'instant. Vous allez devoir construire le store Redux dans le répertoire `/src/store` en vous reposant sur Redux Toolkit. Une fois le store construit, il vous faudra modifier les composants en place pour créer la liaison avec le store en utilisant les hooks mis à disposition par React Redux.

Pour simuler une API Rest, vous utiliserez la commande `npm run api` dans un nouvel onglet de terminal. Cette commande lance un outil, `json-server`[1], qui expose sur l'URL http://localhost:3004 une API Rest. Les données exposées seront celles situées dans le fichier `/src/api/db.json`. En cas d'écriture (via une requête `POST`), les données seront ajoutées au fichier `/src/api/db.json`. Une fois lancé, `json-server` vous informera que les ressources http://localhost:3004/posts et http://localhost:3004/users sont disponibles. Vous pourrez alors déclencher une requête vers ces URL pour récupérer les données correspondantes (via une requête `GET`) et pour ajouter un nouvel élément (via une requête `POST`).

Votre store devra être composé de deux slices qui correspondent aux deux ressources disponibles sur l'API. Chaque slice définira ses reducers et actions nécessaires afin de pouvoir :

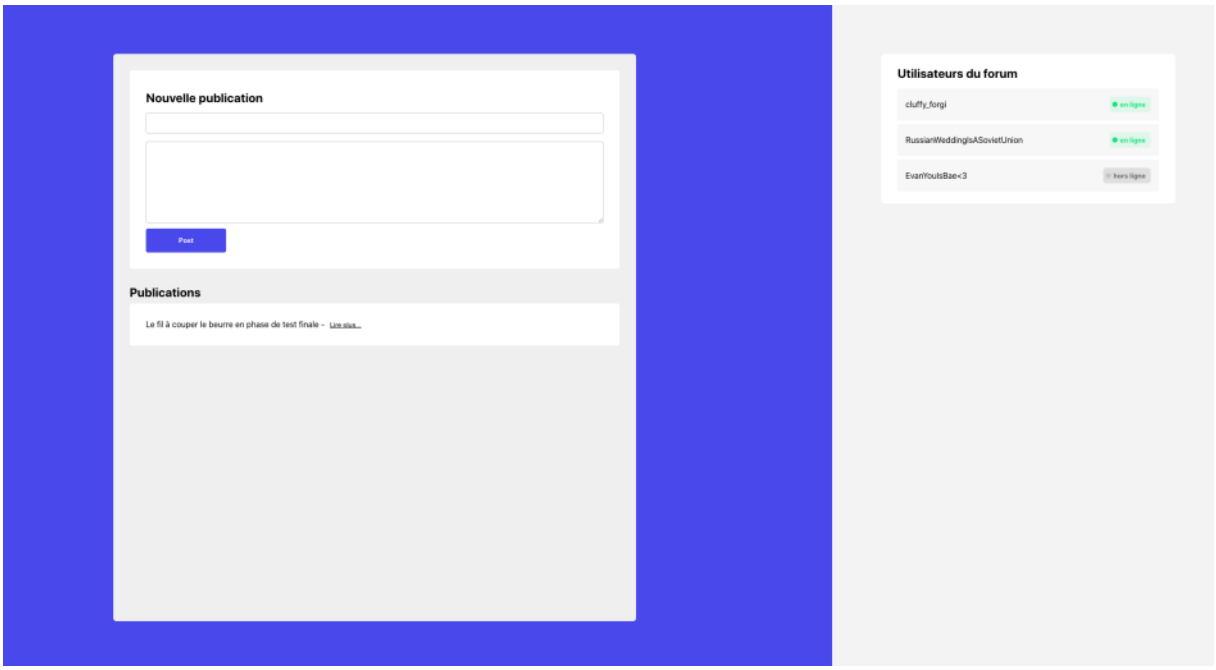
- Récupérer la liste des utilisateurs sur l'API
- Récupérer la liste des posts sur l'API
- Envoyer un nouveau post (référez-vous à la structure d'un post en explorant le fichier `/src/api/db.json` qui en contient déjà un)

L'application devra alors :

- Présenter la liste des utilisateurs dans le composant `<Sidebar />`
- Présenter un formulaire permettant d'ajouter un nouveau post dans le composant `<NewPost />`
- Présenter la liste des posts dans le composant `<Posts />`
- Présenter en détail le post sélectionné dans une nouvelle page

[cf. forum-app.zip]

1 https://github.com/typicode/json-server



Solutions des exercices

p. 9 **Solution n°1**

Container Todo - /src/Container/Todo.js

```
import { connect } from 'react-redux'
import Todo from '../Todo'

const mapStateToProps = state => {
  return {
    todos: state.todos,
    user: state.user
  }
}

const mapDispatchToProps = dispatch => {
  return {
    onAddTodo: todo => {
      dispatch({ type: 'ADD_TODO', payload: todo })
    },
    onMarkTodoAsDone: todoId => {
      dispatch({ type: 'MARK_TODO_AS_DONE', payload: todoId })
    }
  }
}

const TodoContainer = connect(mapStateToProps, mapDispatchToProps)(Todo)

export default TodoContainer
```

p. 13 **Solution n°2**

Fichier de store modifié

```
import { configureStore, createSlice } from '@redux/toolkit'

const todoSlice = createSlice({
  name: 'todo',
  initialState: {
    todos: [],
    user: {
      name: 'John',
      username: 'itsjohn',
      token:
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIn0.dozjgNryP4J3jVmNHl0w5N_XgL0n3I9PlFUP0TH
    }
  },
  reducers: {
    addTodo(state, action) {
      return {
        ...state,
        todos: [...state.todos, action.payload]
      }
    },
    markTodoAsDone(state, action) {
      const targetTodo = state.todos.find(todo => todo.id === action.payload)
      return {
```

```
        ...state,
        todos: [
          ...state.todos.filter(todo => todo.id !== action.payload),
          { ...targetTodo, done: true }
        ]
      }
    }
  }
})

const store = configureStore({
  reducer: todoSlice.reducer
})

export const { addTodo, markTodoAsDone } = todoSlice.actions
export default store
```

p. 16 Solution n°3

Fichier de store mis à jour avec la configuration Redux Persist :

```
import { configureStore, createSlice } from '@redux/toolkit'
import { persistStore, persistReducer } from 'redux-persist'
import storage from 'redux-persist/lib/storage'

const todoSlice = createSlice({
  name: 'todo',
  initialState: {
    todos: [],
    user: {
      name: 'John',
      username: 'itsjohn',
      token:
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIn0.dozjgNryP4J3jVmNHl0w5N_XgL0n3I9PlFUP0TH
    }
  },
  reducers: {
    addTodo(state, action) {
      return {
        ...state,
        todos: [...state.todos, action.payload]
      }
    },
    markTodoAsDone(state, action) {
      const targetTodo = state.todos.find(todo => todo.id === action.payload)
      return {
        ...state,
        todos: [
          ...state.todos.filter(todo => todo.id !== action.payload),
          { ...targetTodo, done: true }
        ]
      }
    }
  }
})

const persistConfig = {
```

```
  key: 'root',
  storage,
}

const persistedReducer = persistReducer(persistConfig, todoSlice.reducer)

const store = configureStore({
  reducer: persistedReducer
})

export const persistor = persistStore(store)
export const { addTodo, markTodoAsDone } = todoSlice.actions
export default store

```

Composant `<App />`:

```
import React from 'react';
import './App.css';
import Todo from './Container/Todo';
import { Provider } from 'react-redux';
import store from './store';
import { PersistGate } from 'redux-persist/integration/react';
import { persistor } from './store';

function App() {
  return (
    <div className="App">
      <Provider store={store}>
        <PersistGate loading={null} persistor={persistor}>
          <Todo />
        </PersistGate>
      </Provider>
    </div>
  );
}

export default App;

```

Exercice p. 17 **Solution n°4**

Exercice

Par défaut, Redux est...

- ○ Fortement couplé à React
- ◉ Agnostique

Redux est agnostique du framework avec lequel il est utilisé.

Exercice

Quelle est l'entité placée entre le store Redux et un composant d'affichage permettant la liaison du global store ?

- ○ Un wrapper
- ○ Un proxy
- ◉ Un container

Exercice

Comment sont nommées les fonctions permettant de créer la liaison avec les éléments du state global ou les actions avec un composant ?

- ○ `linkGlobalState` et `proxyActions`
- ◉ `mapStateToProps` et `mapDispatchToProps`
- ○ `bindStoreToComponent` et `connectMutations`

Exercice

Quel est le composant, importé depuis React Redux, permettant d'injecter le store dans un arbre de composants ?

- ◉ <Provider />
- ○ <Store />
- ○ <MapGlobalState />

Exercice

En utilisant Redux Toolkit, quelle est la méthode qui vient remplacer `createStore()` ?

- ○ spawnStore()
- ○ makeStore()
- ◉ configureStore()

Exercice

Quel est l'élément de syntaxe d'un reducer que Redux Toolkit permet de remplacer par un tableau associatif ?

- ◉ Le `switch...case`
- ○ Le fait d'être obligé de retourner le state en entier (immutabilité)
- ○ L'action sous la forme d'un objet JavaScript

Exercice

Quelle est la méthode Redux Toolkit qui permet de transformer les actions en fonctions ?

- ○ createReducer()
- ◉ createAction()
- ○ connect()

Exercice

Quelle est la nouvelle entité définie par Redux Toolkit qui permet d'englober le state, les reducers et les actions liés à un même thème ?

- ○ Le module
- ○ Le chunk
- ◉ La slice

Exercice

Persister des éléments en `localStorage` est pertinent si...

- ○ On souhaite un stockage de longue durée
- ○ On traite avec des données sensibles
- ◉ Le stockage est temporaire ou les données non-critiques

Exercice

Quel est le rôle de la prop `loading` dans le composant `<PersistGate loading={null} persistor={persistor}>` ?

- ◉ Le JSX placé dans cette prop sera affiché le temps que Redux Persist récupère le contenu du `localStorage`
- ○ La promesse placée dans cette prop se résout quand le state est ventilé

p. 19 **Solution n°5**

Fichier de store - `/store/index.js` :

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

const store = configureStore({
  reducer: rootReducer
});

export default store
```

Fichier du reducer racine - `/store/rootReducer.js` :

```
import { combineReducers } from '@reduxjs/toolkit'
import { userSlice } from './users.slice'
import { postSlice } from './posts.slice'

const rootReducer = combineReducers({
  users: userSlice.reducer,
  posts: postSlice.reducer
})

export default rootReducer
```

Slice de la ressource *utilisateurs* - `/store/users.slice.js` :

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

const fetchUsers = createAsyncThunk(
  'users/fetch',
  async () => {
    const response = await fetch('http://localhost:3000/users')
    const data = await response.json()
    return data
  }
```

```
)

const userSlice = createSlice({
  name: 'users',
  initialState: {
    entities: []
  },
  reducers: {},
  extraReducers: {
    [fetchUsers.fulfilled]: (state, action) => {
      state.entities = action.payload
    }
  }
})

export { userSlice, fetchUsers };
```

Slice de la ressource *posts* - `/store/posts.slice.js` :

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

const fetchPosts = createAsyncThunk(
  'posts/fetch',
  async () => {
    const response = await fetch('http://localhost:3000/posts')
    const data = await response.json()
    return data
  }
)

const newPost = createAsyncThunk(
  'posts/new',
  async (payload) => {
    const response = await fetch('http://localhost:3000/posts', {
      method: 'POST',
      body: JSON.stringify(payload),
      headers: {
        'Content-type': 'application/json'
      }
    })
    const data = await response.json()
    return data
  }
)

const postSlice = createSlice({
  name: 'posts',
  initialState: {
    entities: []
  },
  reducers: {},
  extraReducers: {
    [fetchPosts.fulfilled]: (state, action) => {
      state.entities = action.payload
    },
    [newPost.fulfilled]: (state, action) => {
      state.entities = [...state.entities, action.payload]
    }
  }
```

```
})

export { postSlice, fetchPosts, newPost };
```

Composant `<Sidebar />`, ajout de la liaison de données avec le store pour la récupération des utilisateurs :

```
import React, { useEffect } from 'react'
import { useDispatch, useSelector } from 'react-redux'
import { fetchUsers } from '../store/users.slice'
import styled from 'styled-components'

const Sidebar = () => {
  const dispatch = useDispatch()
  const users = useSelector(state => state.users.entities)

  useEffect(() => {
    dispatch(fetchUsers())
  }, [dispatch]);

  return (
    <Wrapper>
      <InnerWrapper>
        <h2>Utilisateurs du forum</h2>
        {users
          ? <List>
            {users.map((user) => (
              <User key={user.id}>
                {user.username}
                <Status online={user.online}>
                  {user.online ? 'en ligne' : 'hors ligne'}
                </Status>
              </User>
            ))}
          </List>
          : null
        }
      </InnerWrapper>
    </Wrapper>
  )
}

const Wrapper = styled.div`
  grid-area: 'sidebar';
  padding: 6rem;
`

const InnerWrapper = styled.div`
  background: #FFFFFF;
  border-radius: 6px;
  padding: 1rem 2rem;
`

const List = styled.ul`
  margin-top: 1rem;
`

const User = styled.li`
  margin: 0.5rem 0;
  padding: 1rem;
```

```
  background: #F8F8F8;
  display: flex;
  align-items: center;
  justify-content: space-between;
  border-radius: 4px;
`

const Status = styled.div`
  position: relative;
  display: flex;
  align-items: center;
  height: 30px;
  border-radius: 6px;
  padding-left: 20px;
  padding-right: 10px;
  font-size: 0.8rem;
  font-weight: bold;
  background: ${props => props.online ? 'hsla(152, 91%, 48%, 0.1)' : 'hsla(0, 0%, 0%, 0.1)'};
  color: ${props => props.online ? 'hsl(152, 91%, 48%)' : '#707070'};

  &:before {
    content: '';
    position: absolute;
    top: 50%;
    transform: translateY(-50%);
    left: 5px;
    width: 10px;
    height: 10px;
    border-radius: 7px;
    background: ${props => props.online ? '#0be881' : '#CCC'}
  }
`

export default Sidebar

```

Composant `<NewPost />`, ajout de la liaison de données avec le store pour la création d'un nouveau post :

```
import React, { useState } from 'react'
import { useDispatch } from 'react-redux'
import { newPost } from '../store/posts.slice'
import styled from 'styled-components'

const NewPost = () => {
  const dispatch = useDispatch()
  const [title, setTitle] = useState('')
  const [content, setContent] = useState('')

  const handleTitleInput = e => setTitle(e.target.value)
  const handleContentInput = e => setContent(e.target.value)
  const handleSubmit = () => {
    if (title && content) {
      dispatch(newPost({
        id: Math.random(),
        title,
        content,
        comments: []
      }))
    }
```

```
  }

  return (
    <Wrapper>
      <h2>Nouvelle publication</h2>
      <input
        type="text"
        value={title}
        onChange={handleTitleInput}
      />
      <textarea
        value={content}
        onChange={handleContentInput}
      />
      <Button
        type="button"
        onClick={handleSubmit}
      >
        Post
      </Button>
    </Wrapper>
  )
}

const Wrapper = styled.form`
  background: #fff;
  border-radius: 4px;
  padding: 2rem;

  input,
  textarea {
    width: 100%;
    border: 1px solid #ccc;
    border-radius: 4px;
    margin: 0.5rem 0;
    padding: 1rem;
    font-family: inherit;
  }

  input {
    height: 2.5rem;
  }

  textarea {
    height: 10rem;
  }
`

const Button = styled.button`
  background: #5352ed;
  border: none;
  padding: 1rem 4rem;
  border-radius: 4px;
  color: #fff;
  font-weight: bold;
  cursor: pointer;
`

```

```
export default NewPost

```

Composant `<Posts />`, ajout de la liaison de données avec le store pour la récupération des posts :

```
import React, { useEffect } from 'react'
import { useDispatch, useSelector } from 'react-redux'
import { fetchPosts } from '../store/posts.slice'
import styled from 'styled-components'
import NewPost from './NewPost'

const Posts = ({ onSelectPost }) => {
  const dispatch = useDispatch()
  const posts = useSelector(state => state.posts.entities)

  useEffect(() => {
    dispatch(fetchPosts())
  }, [dispatch]);

  return (
    <Wrapper>
      <InnerWrapper>
        <NewPost />
        <PostsWrapper>
          <h2>Publications</h2>
          {posts
            ? <ul>
              {posts.map((post) => (
                <Post key={post.id}>
                  {post.title} - <Link onClick={() => onSelectPost(post)}>Lire plus...</Link>
                </Post>
              ))}
            </ul>
            : null
          }
        </PostsWrapper>
      </InnerWrapper>
    </Wrapper>
  )
}

const Wrapper = styled.div`
  grid-area: 'main';
  background: #5352ed;
  display: flex;
  flex-direction: column;
  align-items: center;
  padding: 6rem;
`

const InnerWrapper = styled.div`
  background: #F1F1F1;
  width: 64rem;
  min-height: 100%;
  padding: 2rem;
  border-radius: 6px;
`

const Link = styled.button`
```

```
  background: none;
  border: none;
  text-decoration: underline;
  cursor: pointer;
`

const PostsWrapper = styled.div`
  margin-top: 2rem;
`

const Post = styled.li`
  width: 100%;
  background: #fff;
  border-radius: 4px;
  padding: 2rem;
  margin: 0.5rem 0;
`

export default Posts

```