React avancé

bstorm TRAINING

Table des matières

Single-Page Application

Mise en place avec « React-Router »

Single-Page Application

Le package « React-Router »

Pour mettre en place une application SPA (Single-Page Application), il est conseillé d'utiliser une librairie pour faciliter sa mise en place en React.

Une des librairies les plus utilisée pour le créer une SPA est « React-Router »

Celle-ci permet:

- Définir le routing de l'application.
- Naviguer entre les différentes pages.
- Récupérer des informations depuis l'url.

Mise en place du routing

Pour ajouter le routing dans l'application, il faut installer le package « react-router » .

" **npm install —save react-router-dom** "

Ensuite, il est nécessaire de :

- Connecter l'application à l'URL.
- Créer des composants React qui symbolisent les pages de l'application.
- Permettre à l'utilisateur de naviguer entre les pages.
- Configurer les différentes routes :
  - En JSX via les composants « Routes » et « Route ».
  - En JS à l'aide du hook « useRoutes »

Connecter l'application à l'URL

En premier lieu, il est nécessaire de connecter l'application à l'URL.

Pour cela, dans le fichier « index.js », il faut encapsuler l'App avec « BrowserRouter ».

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { BrowserRouter } from 'react-router-dom';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

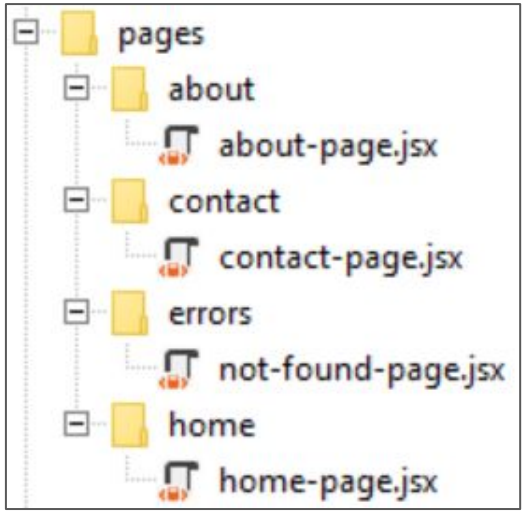
Ajouter des pages à l'application

Création des composants qui seront utilisés pour définir le contenu des pages

Exemple d'un composant « Page » :

```jsx
const HomePage = () => {

    return (
        <>
            <h1>Home</h1>
            <p>Lorem ipsum dolor sit ...</p>
        </>
    );
};

export default HomePage;
```

Structure de fichier :



Remarque : La structure utilisée dans ce support est « Par types d'éléments ».

Permettre à l'utilisateur de naviguer entre les pages

Le composant « Link » de React-Router permet de mettre en place de la navigation entre les pages de l'app.

Il est également possible d'utiliser le composant « NavLink ». Celui-ci à le même fonctionnement que le « Link ».

Sa particularité est qu'il détecte la route active de l'application. Cela permet d'interagir facilement le visuel.

Exemple de NavBar

```jsx
import { Link } from 'react-router-dom';
import style from './nav-bar.module.css';

const NavBar = () => (
    <nav className={style.nav}>
        <ul>
            <li>
                <Link to=''>Demo React-Router</Link>
            </li>
            <li>
                <Link to='/about'>About</Link>
            </li>
            <li>
                <Link to='/contact'>Contact</Link>
            </li>
        </ul>
    </nav>
);

export default NavBar;
```

Configuration des routes : En JSX

Exemple de création des routes en JSX via les composants « Routes » et « Route »

```jsx
const App = () => {
  return (
    <div className="App">
      <NavBar />
      <Routes>
        <Route path='' element={<HomePage />} />
        <Route path='contact' element={<ContactPage />} />
        <Route path='about' element={<AboutPage />} />
        <Route path='*' element={<NotFoundPage />} />
      </Routes>
    </div>
  );
};

export default App;
```

Configuration des routes : En JavaScript

Exemple de configuration des routes à l'aide du hook « useRoutes »

Fichier JS de configuration des routes

```
const appRoutes = [
    {
        path: '',
        element: <HomePage />
    },
    {
        path: 'contact',
        element: <ContactPage />
    },
    {
        path: 'about',
        element: <AboutPage />
    },
    {
        path: '*',
        element: <NotFoundPage />
    }
];

export default appRoutes;
```

Injection de routes à l'aide du hook

```
import { useRoutes } from 'react-router-dom';
import appRoutes from './routes';
import NavBar from './containers/nav-bar/nav-bar';

const App = () => {
  const routes = useRoutes(appRoutes);

  return (
    <div className="App">
      <NavBar />
      {routes}
    </div>
  );
};

export default App;
```

Les Routes Enfants

Single-Page Application

Objectif

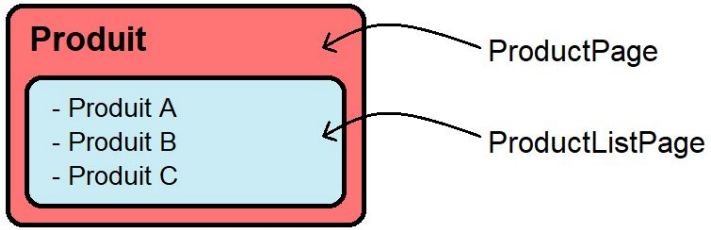
Pour éviter la répétition de code dans les différentes « pages », il est possible d'intégrer un composant dans une page qui dépendra de la route active.

Pour cela, il est nécessaire que :

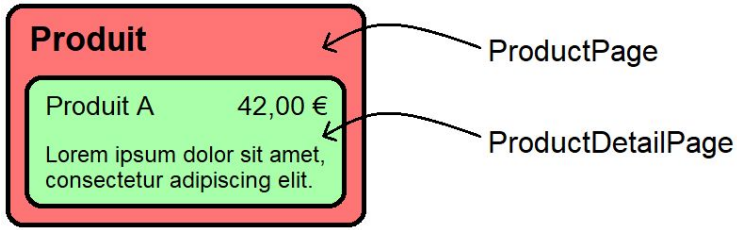
- Le composant page doit contenir un composant « Outlet »
- La configuration de la route contient les différentes routes enfants

Exemple :

➔ /exemple.com/product



➔ /exemple.com/product/42



Page avec un « Outlet »

Pour intégrer des composants dans une page à l'aide de routes enfants, il est nécessaire d'ajouter le composant « Outlet » fourni par « react-router ».

```
const { Outlet } = require('react-router-dom');

const ProductPage = () => {

    return (
        <>
            <h1>Product</h1>
            <Outlet />
        </>
    );
};

export default ProductPage;
```

Configuration de routes enfants

Exemple de configuration d'une route avec des routes enfants

En JSX

```jsx
<Routes path='product' element={<ProductPage />}>
  <Route index element={<ProductListPage />} />
  <Route path=':productId' element={<ProductDetailPage />} />
  <Route path='new' element={<ProductFormPage />} />
</Routes>
```

En Javascript (avec « useRoutes »)

```jsx
{
   path: 'product',
   element: <ProductPage />,
   children: [
      {
         index: true,
         element: <ProductListPage />
      },
      {
         path: ':productId',
         element: <ProductDetailPage />
      },
      {
         path: 'new',
         element: <ProductFormPage />
      }
   ]
},
```

La route configurée en « index » permet d'afficher le composant lorsque l'url correspond à la route parent.

Les Hooks de « React-Router »

Single-Page Application

Quelques Hooks utiles de « React-Router »

- useParams

Permet d'obtenir les données dynamiques contenues dans la route.

```
// exemple.com/product:productId
const { productId } = useParams();
```

- useSearchParams

Permet de lire ou modifier le « Query String » de l'url.

```
// exemple.com/demo?nb=42
const [searchParams, setSearchParams] = useSearchParams();
```

Quelques Hooks utiles de « React-Router »

- useNavigate

Permet d'obtenir une méthode pour naviguer par programmation.

```
const navigate = useNavigate();

const handleDemo = () => {
    // Navigation vers la page "/new-page"
    navigate('/new-page');
};
```

State Management

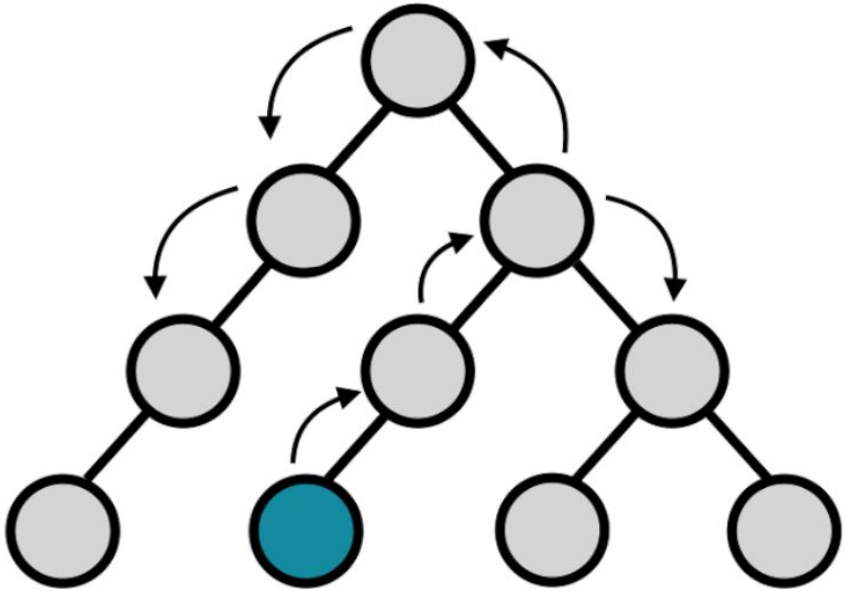
Introduction

State Management

Pourquoi utiliser un « state management »

Manipuler les données dans différents composants React peut s'avérer compliqué, car il faut faire transiter les données au sein de l'arborescence.

Pour éviter cela, il est conseillé de mettre en place un mécanisme de « state management ».



State management

Un système de « state management » a pour objectif de simplifier le stockage de données au sein d'une application.

En Javascript, il existe plusieurs solutions :

- **Data Store** : Flux (Non Conseillé), Redux, MobX, Zustand.
- **Observer pattern** : RxJS
- **État partagé** : Recoil, Jotai.

Dans le cadre de ce cours, nous allons utiliser la librairie « Redux ».

La librairie Redux

State Management

Un store, ça sert à quoi ?

Redux est une librairie JavaScript qui met en place le mécanisme de « Store ».

Le principe de fonctionnement d'un Store est :

- Création d'un conteneur centralisé pour les données.
- Interaction avec les données uniquement à travers de méthodes.
- Notification des changements des données via des événements.

Redux, ça fonctionne comment ?

Le fonctionnement du Store Redux se base sur l'utilisation de deux éléments :

- **Les Actions**
  Ce sont des objets Javascript qui représentent les actions à réaliser.
  Celle-ci contiennent :
  - Le type d'action.
  - Des données *(si cela est nécessaire)*.

- **Les Reducers**
  Ce sont des fonctions qui se déclencheront lorsque qu'une action est reçue.
  Leur objectif est de mettre à jour les données du Store.

Procédure pour créer un Store de Redux

Pour utiliser le store Redux, il faut installer le package « redux » .

“ npm install –save redux ”

Une fois installé, il faut :

➔ Définir les différents Actions.

➔ Créer les Reducer pour résoudre les actions.

➔ Configurer l’instance du store Redux.

Exemple de mise en place d'un store Redux

Pour cet exemple de création d'un store, nous allons mettre en place un compteur.

Les actions que le store devra être capable de résoudre sont les suivantes :

- « cpt/increment »
  Permet d'augmenter la valeur du compteur de X.
- « cpt/reset »
  Permet de définir la valeur du compteur à 0.

Exemple de mise en place d'un store Redux

```
// Configuration et création du store
// **********************************
import { legacy_createStore as createStore } from 'redux';

// Etat initial du state
const initialState = {
    count: 0
};

// Création du Reducer
const reducer = (state = initialState, action) => {
    switch (action.type) {
        case "cpt/increment":
            return {
                count: state.count + action.payload
            };
        case "cpt/reset":
            return {
                count: 0
            };
        default:
            return state;
    }
};

// Création du Store
const store = createStore(reducer);
```

```
// Utilisation du store
// ********************

// Affichage du contenu
const state1 = store.getState();
console.log('State 1', state1);

// Abonnement à la réception d'action
const unsubscribe = store.subscribe(() => {
    console.log('Action traité');
});

// Envoi d'une action dans le store
store.dispatch({ type: 'cpt/increment', payload: 1 });

// Affichage du contenu
const state2 = store.getState();
console.log('State 2', state2);

// Envoi d'action dans le store
store.dispatch({ type: 'cpt/exemple', payload: 'Hello' });
store.dispatch({ type: 'cpt/increment', payload: 4 });

// Affichage du contenu
const state3 = store.getState();
console.log('State 3', state3);

// Désabonnement
unsubscribe();
```

```
State 1 { count: 0 }
Action traité
State 2 { count: 1 }
Action traité
Action traité
State 3 { count: 5 }
```

Intégration de Redux dans un projet

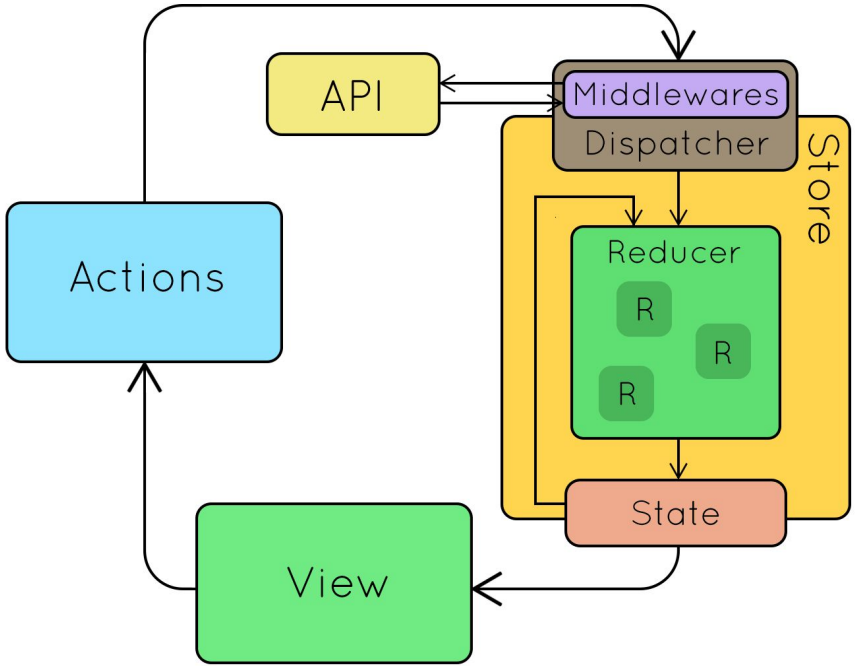
State Management

Présentation du pattern d'utilisation de Redux

Pour mettre en place le store Redux dans un projet, il est conseillé d'utiliser le pattern « one-way data flow ».

Ce pattern permet de simplifier l'utilisation de Redux :

- La vue consomme les données du store.
- Les interactions sont transmises au dispatcher via des actions dédiées.
- Les middlewares permettent d'ajouter des comportements avant les reducers.

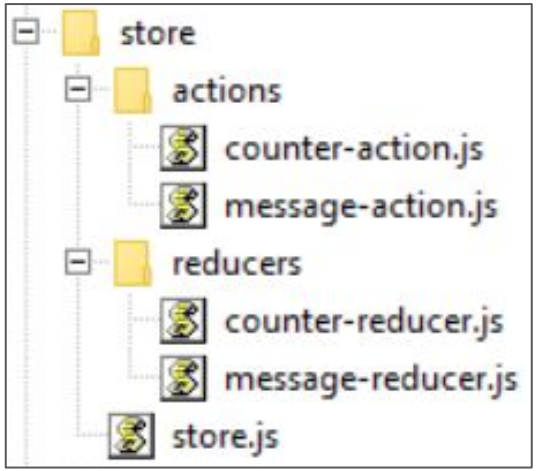


Intégration du pattern de Redux dans un projet

Pour ajouter le store dans nos projets avec ce pattern d'utilisation, il faudra mettre en place les éléments suivants :

- Des « Action Creators ».
- Des « Reducers ».
- La configuration du store Redux.

Structure de fichier :



Remarque : La structure utilisée dans ce support est « Par types d'éléments ».

Le fonctionnement des « Action Creators »

Les « Action Creators » sont des méthodes permettant de générer les objets « Action » de Redux.

L'objet généré par un « Action Creator » sera composé de :

- Un type (Obligatoire)
  L'identifiant de l'action
- Un Payload
  Les données nécessaire à l'action

```
// Action Types
export const COUNTER_INCR = 'counter/increment';
export const COUNTER_RESET = 'counter/reset';

// Action Creators
export const counterIncr = (step) => {
    return {
        type: COUNTER_INCR,
        payload: step
    };
};

export const counterReset = () => {
    return {
        type: COUNTER_RESET
    };
};
```

Le fonctionnement des « Reducers »

Un « Reducer » est une méthode qui a pour but de modifier le « State ».

Cette méthode doit posséder les deux paramètres suivants :

- State → La valeur du state actuel
- Action → L'action à réaliser

Cette méthode doit toujours renvoyer une nouvelle valeur de state.

```
import { COUNTER_INCR, COUNTER_RESET } from '../actions/counter-action';

// Initial state for "Counter"
const initialState = {
    count: 0
};

// Reducer for "Counter"
const counterReducer = (state = initialState, action) => {
    switch (action.type) {
        case COUNTER_INCR:
            return {
                ...state,
                count: state.count + action.payload
            }
        case COUNTER_RESET:
            return {
                ...state,
                count: 0
            }
        default:
            return state;
    }
};

export default counterReducer;
```

Best Practice

State Management

Bonne pratique pour « Redux »

Voici une brève liste de quelques bonnes pratiques importantes avec Redux :

- Avoir un seul store pour l'application.
- Ne jamais faire muter le State.
- Ne pas avoir de donnée non-sérialisable dans le State ou dans les actions.
- Les Reducers ne doivent pas avoir de « Side Effects »
  ➜ Pas de requête Ajax, pas de génération aléatoire, ...
- Ecrire le nom des types d'action sous la forme « domain/eventName »

Liste complete des bonnes pratiques disponible sur https://redux.js.org/style-guide/

Mise en place de Redux

Le package « Redux Toolkit »

Mise en place de Redux

Objectif de Redux Toolkit

Pour réaliser la mise en place du store, nous allons utiliser « Redux Toolkit ».

Ce package est fourni par l'équipe de développeurs de Redux.
Ses principaux objectifs sont :

- Simplifier la configuration du store Redux.
- Générer des « Action Creators ».
- Faciliter l'écriture des « Reducers ».

Pour utiliser le Toolkit, il faut l'installer.

" npm install –save @reduxjs/toolkit "

Action Creators

Mise en place de Redux

Les Action Creators

Le Toolkit permet de générer les Action Creators à l'aide de « createAction ».

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

// Action Creators
export const counterIncr = createAction('counter/increment');
export const counterReset = createAction('counter/reset');
```

L'avantage de l'utiliser la méthode du toolkit est :

- Création automatique d'une méthode qui génère un objet « Action »
  { type: « action_type » , payload: « action_payload » }
- Le type peut être obtenu via de la méthode « toString() » de l'action creator.

Les Action Creators

Il est possible de customiser l'objet généré par la méthode « createAction ».
Cela permet d'ajouter dans le Payload : une valeur par défaut, un identifiant généré...

Dans ce cas, il est nécessaire d'utiliser un callback pour définir la valeur du « Payload »

```
export const messageAdd = createAction('message/add', (content) => {
    return {
        payload: {
            id: nanoid(),
            content,
            createAt: new Date().toISOString()
        }
    };
});
```

Reducers

Mise en place de Redux

Les Reducers

L'implémentation classique d'un Reducer est un bloc "switch" avec différents "case" pour résoudre chaque action.
Cette approche fonctionne bien, mais elle peut également être source d'erreur.
Quelques exemples : oublier le cas par défaut, ne pas définir l'état initial, etc...

Le Toolkit permet de créer un Reducer à l'aide de la méthode « createReducer ».
Cette méthode utilise un builder qui possède les méthodes suivantes :

- builder.addCase(actionCreator, callback)
- builder.addMatcher(matcherPredicate, callback)
- builder.addDefaultCase(callback)

Les Reducers

La méthode « createReducer » attend comme paramètres :

- L'état initial du State
- Un callback qui définit le « Builder » avec les différents évent à traiter.

Les Action Creators créés par le Toolkit sont utilisables comme valeur d'évent.

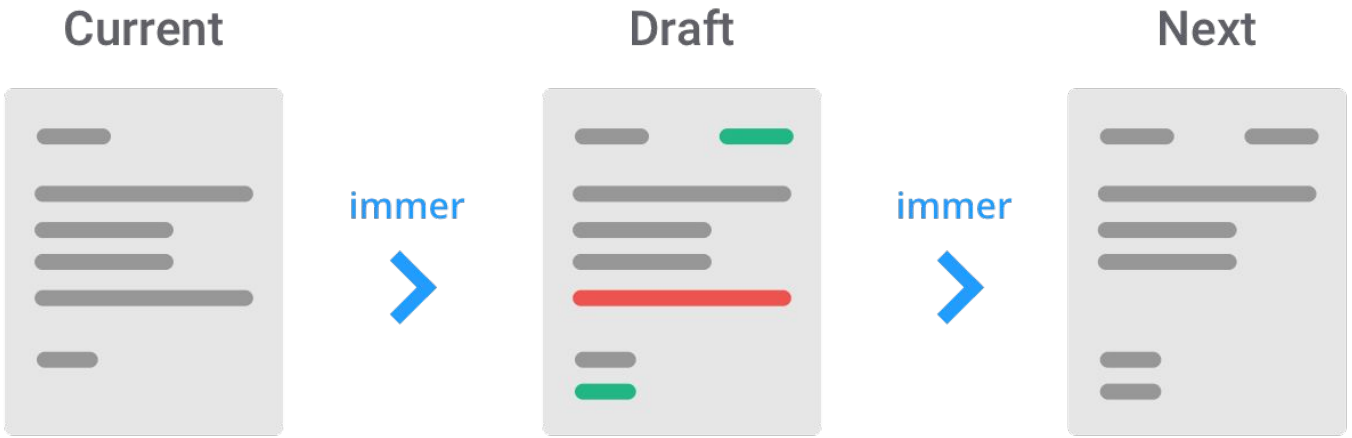
Il n'est pas obligatoire d'ajouter l'évent par défaut, s'il n'est pas customisé.

```
import { createReducer } from '@reduxjs/toolkit';
import { counterIncr, counterReset } from '../actions/counter-action';

// Initial state for "Counter"
const initialState = {
    count: 0
};

// Reducer for "Counter"
const counterReducer = createReducer(initialState, (builder) => {
    builder
        .addCase(counterIncr, (state, action) => {
            return {
                ...state,
                count: state.count + action.payload
            };
        })
        .addCase(counterReset, (state) => {
            return {
                ...state,
                count: 0
            }
        })
});

export default counterReducer;
```

Les Reducers - La bibliothèque « Immer »

La méthode « createReducer » utilise « Immer » pour faciliter l'écriture du code.

Cette bibliothèque permet de réaliser des modifications sur des données immutable, à l'aide d'une écriture « mutable » sur une copie des données.



Les Reducers - Utilisation de Immer

Grâce à l'intégration de Immer, il est donc possible de modifier les données à l'aide d'opérations simples.

- Pour les valeurs :
  - Affectation d'une nouvelle valeur
  - Incrémentation et décrémentation

- Pour les tableaux (Array) :
  - Utilisation de l'opérateur d'accès
  - Ajout d'éléments (push, unshift)
  - Retirer des éléments (pop, shift)

```
import { createReducer } from '@reduxjs/toolkit';
import { counterIncr, counterReset } from '../actions/counter-action';

// Initial state for "Counter"
const initialState = {
    count: 0
};

// Reducer for "Counter" with Immer
const counterReducer = createReducer(initialState, (builder) => {
    builder
        .addCase(counterIncr, (state, action) => {
            state.count += action.payload
        })
        .addCase(counterReset, (state) => {
            state.count = 0
        })
});

export default counterReducer;
```

⚠️ Quand l'écriture « Mutable » est utilisée, il ne faut pas renvoyer de state !

La configuration du store

Mise en place de Redux

Configuration du store

Il est recommandé de générer l'instance du store avec la méthode « configureStore », celle-ci permet de réaliser la configuration du store Redux simplement.

Cette méthode a comme paramètre un objet « options » avec les valeurs suivants :

- Reducer : L'ensemble des différents reducers de l'application.
- Middleware : Les fonctionnalités à ajouter au « Dispatcher » du store.
- DevTools : Un booléen pour activer les outils pour développeur.
- PreloadedState : Une ancienne instance du state *(Restauration de données)*.
- Enhancers : Les fonctionnalités à ajouter au store Redux.

Exemple de la configuration d'un store

Dans cette exemple, la store est initialisé avec les options suivants :

- Reducer :
  Un objet qui combine les différents reducers avec un alias.
- DevTools:
  Désactivé en build de production.

Le store Redux fonctionnera donc avec un unique reducer, qui sera le résultat de la combinaison des reducers aliasés.

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

// Import des différents Reducers
import counterReducer from './reducers/counter-reducer';
import messageReducer from './reducers/message-reducer';

// Création du store avec la méthode "configureStore"
const store = configureStore({
    // Fusion des différents Reducers
    reducer: {
        counter: counterReducer,
        message: messageReducer
    },

    // Activation des outils de dev
    devTools: process.env.NODE_ENV !== 'production'
});

// Export du store créé
export default store;
```

Configuration des Middlewares dans un store

L'option « middleware » prend comme argument un tableau de Middleware à ajouter.

Lorsque cette option n'est pas présente, le « configureStore » va automatiquement appeler la méthode « getDefaultMiddleware » pour ajouter les middlewares suivants :

- **Immutability Check** [Development]
  Génère une erreur lorsqu'une mutation des données est réalisée dans le store.
- **Serializability Check** [Development]
  Génère une erreur quand le store contient des données non sérialisables.
- **Redux Thunk** [Development & Production]
  Permet d'ajouter des effets de bord *(Celui-ci sera abordé en détail plus tard)*

Configuration des Middlewares dans un store

Il est conseillé de conserver les middlewares par défaut lors de la configuration.

Pour cela, il faut utiliser un callback qui permettra de combiner le(s) middleware(s) à ajouter avec ceux définis par défaut.

```
import { configureStore } from '@reduxjs/toolkit';
import reduxLogger from 'redux-logger';

// Création du store avec la méthode "configureStore"
const store = configureStore({
    // Les options "reducers" et "devTools" sont masquées pour la lisibilité

    // Ajout des middlewares
    middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(reduxLogger)
});

// Export du store créé
export default store;
```

Interaction entre React et Redux

Le package « React-Redux »

Interaction entre React et Redux

Utilisation Redux dans une application React

Pour utiliser le store avec React, il est conseillé d'installer le package « react-redux ». Celui-ci met à disposition les éléments suivants :

- Un composant **« Provider »** qui partage le store dans l'application.
- Des hooks qui permettent d'interagir avec le store
  - Le **« useSelector() »** pour de lire les données.
  - Le **« useDispatch() »** pour envoyer des actions.
- Le HOC **« connect »** qui encapsule un composant pour le lier au store.

Pour l'utiliser, il est nécessaire de l'installer.

"npm install —save react-redux"

Le Composant Provider

Interaction entre React et Redux

Diffusion du « Store » dans l'application

Le composant « Provider » reçoit en argument un objet « Store » de Redux.

Celui-ci permet aux composants enfant d'avoir l'accès au conteneur de Redux.

Pour rendre disponible le store dans notre application, on englobe le composant « App » avec le Provider.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import './index.css';

import { Provider } from 'react-redux';
import store from './store/store';

const htmlRoot = document.getElementById('root');
const root = ReactDOM.createRoot(htmlRoot);

root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

Les hooks de « react-redux »

Interaction entre React et Redux

Consommation des données du store Redux

Pour récupérer les données du store, il faut utiliser le hook « useSelector ».
Celui-ci prend en paramètre un callback, qui définit comment obtenir les données.

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

const CounterDisplay = () => {

    // Lecture de la valeur de "count" contenu dans le store
    const count = useSelector(state => state.counter.count);

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

export default CounterDisplay;
```

Note : Il est possible de créer des « Selectors mémorisés » à l'aide de la méthode « createSelector » du Redux-Toolkit. https://redux-toolkit.js.org/api/createSelector

Envoyer des actions au store Redux

Pour modifier les données du store, il faut envoyer les actions à réaliser au dispatcher.
Le hook « useDispatch » permet d'interagir avec le dispatcher à l'aide d'une méthode.

```
import { useDispatch } from 'react-redux';
import { counterIncr, counterReset } from '../../store/actions/counter-action';

const CounterButton = () => {
    // Récuperation du "dispatcher" du store Redux
    const dispatch = useDispatch();

    // Envoi d'action à l'aide de la méthode dispatch et des Action Creators
    return (
        <div>
            <button onClick={() => dispatch(counterIncr(1))} >+ 1</button>
            <button onClick={() => dispatch(counterReset())} >Reset</button>
        </div>
    );
};

export default CounterButton;
```

Le HOC connect

Interaction entre React et Redux

Le HOC connect

Pour obtenir les données du state ou/et envoyer des actions au store Redux depuis React, il est également possible d'utiliser la méthode « connect ».

Cette méthode est un Higher-Order Components (HOC) qui à pour but d'encapsuler un composant, elle injecte les fonctionnalités du store aux « props » de celui-ci.

Depuis l'ajout des Hooks, il est conseillé de ne plus mettre en place l'interaction avec le store en utilisant le HOC « connect ». Bien que celui-ci reste supporté.

Par contre, il est toujours utile de comprendre son fonctionnement pour pouvoir assurer la maintenance de projet qui l'utilise.

Consommation des données via le HOC « connect »

La méthode « connect » permet de rendre accessible les données du store dans les props du composant englobé.

Pour cela, il faut configurer l'argument "mapStateToProps" à l'aide d'une méthode qui détaille la règle de liaison à réaliser avec le store Redux.

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

// ...

const mapStateToProps = (state) => ({
    count: state.compteur.count
});
export default connect(mapStateToProps)(CompteurAfficheur);
```

Envoyer des actions via le HOC « connect »

La méthode « connect » permet également d'ajouter dans les props du composant englobé des action creators, qui seront liées au store Redux.

Pour cela, il est nécessaire de configurer l'argument "mapDispatchToProps" à l'aide d'un objet qui contient les différentes actions creators.

```
import { connect } from 'react-redux';
import { incrementCpt, decrementCpt } from '../../store/actions/compteur-action';

// ...

const mapDispatchToProps = {
    incrementCpt,
    decrementCpt
};
export default connect(null, mapDispatchToProps)(CompteurInteraction);
```

Les Action Creators complexes

Le middleware « Redux-Thunk »

Les Action Creators complexes

Le middleware « redux-thunk »

Pour permettre de créer des action creators complexes, le Redux Toolkit intègre par défaut le middleware « redux-thunk ». Celui-ci permet :

- De réaliser un algorithme.
- De déclencher plusieurs actions.
- D'exécuter du code asynchrone.

En l'utilisant, le dispatcher du store devient capable de gérer des actions du type :

➔ Un objet avec un type et un payload (Fonctionnement classique du Redux).
➔ Une fonction avec en paramètres « dispatch », « getState ».

Le middleware « redux-thunk »

Exemple d'un action creator avec le déclenchement de plusieurs action

```
const { createAction } = require('@reduxjs/toolkit');

// Action creators "Simple"
export const computeStart = createAction('compute/start');
export const computeResult = createAction('compute/result');

// Action creator "Complexe"
export const computeResponse = (question) => {

    // Fonction à executer par le Dispatcher
    return (dispatch) => {
        dispatch(computeStart(question));
        setTimeout(() => {
            dispatch(computeResult(42));
        }, 236677140000000000);
    };
};
```

Action Creator asynchrone

Les Action Creators complexes

Action Creator asynchrone

Le traitement d'un Action Creator peut nécessiter d'être exécuté en asynchrone.

Par exemple, lorsqu'on consomme une API à l'aide de requêtes AJAX.

Le middleware « Redux-thunk » supporte nativement les objets
de type promesse.

Dans l'exemple, il faut également coder les actions qui y sont déclenchées.

```
export const requestAge = (firstname) => {

    // Traitement asynchrone
    return async (dispatch) => {

        // Action "Pending" - Demarrage du traitement
        dispatch(requestAgePending());

        try {
            // Envoi de la rêquete AJAX
            const response = await axios.get('https://api.agify.io/', {
                params: {
                    name: firstname,
                    country_id: 'BE'
                }
            });

            // Action "Fulfilled" - Terminé avec succès
            dispatch(requestAgeFulfilled(response.data));
        }
        catch (error) {
            // Action "Rejected" - Terminé en erreur
            dispatch(requestAgeRejected(error));
        }
    };
};
```

Utilisation du Toolkit

L'écriture d'Action Creator asynchrone peut s'avérer lourde lorsqu'on souhaite pouvoir interagir avec les différents états de la promesse dans le reducer.
Car il sera nécessaire de créer des action creators pour chaque état à traiter !

Pour l'éviter, il est possible d'utiliser la méthode « createAsyncThunk » du Toolkit.
Elle permet de générer une Action Creator asynchrone avec les états suivantes :

- action.pending
- action.fulfilled
- action.rejected

Le type des actions sera généré sous la forme : " < nom de l'action > / < état > ".

Utilisation du Toolkit - Exemple

La méthode « createAsyncThunk » prend obligatoirement deux paramètres :

- Le type de l'action
- Le callback « payloadCreator »

Le « payloadCreator » a en paramètres :

- Les données envoyées à l'action creator lors du dispatch.
- L'objet « ThunkAPI ».

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

export const requestAge = createAsyncThunk(
  'age/request',
  async (firstname) => {
    // Envoi de la rêquete AJAX
    const response = await axios.get('https://api.agify.io/', {
      params: {
        name: firstname,
        country_id: 'BE'
      }
    });
    return response.data;
  }
);
```

Utilisation du paramètre « ThunkAPI »

Les Action Creators complexes

Le paramètre « ThunkAPI »

Le deuxième paramètre du « payloadCreator » permet d'obtenir le « ThunkAPI ».

Celui-ci permet d'utiliser les éléments suivants :

- dispatch
  Méthode pour envoyer des actions au dispatcher.
- getState
  Méthode qui permet d'obtenir le State du store.
- extra
  Object qui contient les éléments injectés dans la config du middleware.

Utilisation du State via le « ThunkAPI »

Exemple d'une requête Ajax avec un token JWT stocké dans le store Redux

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

export const requestProduct = createAsyncThunk(
    'product/insert',
    async (data, thunkAPI) => {

        // Récuperation des valeurs du state de Redux
        const state = thunkAPI.getState();

        // Envoi d'une requete AJAX avec les informations d'authentification
        const response = await axios.post('https://demo-auth.be/product', data, {
            headers: {
                Authorization: "Bearer " + state.auth.token
            }
        });

        return response.data;
    }
);
```

Action Creator conditionnel

Les Action Creators complexes

Action Creator conditionnel

La méthode « createAsyncThunk » possède un troisième paramètre (optionnel).
Celui-ci permet d'ajouter des options aux action creators asynchrones.

Parmi les options disponibles, celles-ci permettent de rendre l'action conditionnel :

- condition
  Callback qui définit la condition à réaliser pour exécuter l'action.
  Lorsque celle-ci envoi « false », aucune action n'est dispatch au store Redux.
- dispatchConditionRejection
  Booléen qui permet d'envoyer une action « rejected » au Dispatcher quand la validation du callback de condition envoi « false »

Action Creator conditionnel

Exemple d'une requête Météo avec une de condition sur la ville et un délai de 15 min.

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

export const requestWheather = createAsyncThunk(
    'weather/now',
    async (city) => {
        // Envoi de la rêquete AJAX
        const { data } = await axios.get('https://api.openweathermap.org/data/2.5/weather?', {
            params: { appid: process.env.WEATHER_API_KEY, units: 'metric', q: city }
        });

        // Objet résultat
        return { temperature: data.main.temp, city: city, updateTime: Date.now() };
    },
    {
        condition: (city, { getState }) => {
            // Délai de 15 minutes entre deux rêquetes pour la même ville
            const { weather } = getState();
            return city !== weather.city || Date.now() - weather.updateTime > 15 * 60 * 1000;
        }
    }
);
```

Ressources

Bibliothèques utiles

Ressources

Quelques bibliothèques utiles

- **Redux Persist** - Permet de gérer la persistance de donnée dans le store Redux
  https://github.com/rt2zz/redux-persist
- **Redux Form** - Permet de gérer les formulaires au sein de Redux
  https://redux-form.com
- **Redux Observable –** Middleware basé sur RxJS pour Redux
  https://redux-observable.js.org/
- **Redux Logger** - Génération de log dans la console lors des actions du store Redux
  https://github.com/LogRocket/redux-logger

Sites utiles

Ressources

Sites utiles

- **La documentation de Redux**
  https://redux.js.org/
- **La documentation de la librairie React-Redux**
  https://react-redux.js.org/
- **La documentation de la librairie Redux Toolkit**
  https://redux-toolkit.js.org/

Merci pour votre attention.

bstorm
TRAINING