

FORMATION REACT.JS REDUX

Victor Dupré



Panorama

Introduction à Redux



Redux

Redux est une bibliothèque de gestion d'état pour les applications JavaScript. Il est souvent utilisé avec des frameworks tels que React et Angular pour gérer l'état de l'application de manière centralisée. Redux permet de stocker l'état de l'application dans un store global, qui peut être consulté et modifié par des actions déclenchées par l'utilisateur ou par l'application elle-même.



Installation

`npm install redux react-redux`

Création d'actions

La première étape consiste à créer les actions nécessaires pour gérer les articles. Les actions sont des objets JavaScript qui décrivent les modifications à apporter au store Redux. Voici un exemple d'actions pour gérer les articles :

```
export const ADD_POST = 'ADD_POST';
export const DELETE_POST = 'DELETE_POST';

export function addPost(title, content) {
  return {
    type: ADD_POST,
    payload: { title, content },
  };
}

export function deletePost(id) {
  return {
    type: DELETE_POST,
    payload: { id },
  };
}
```

Création du reducer

Une fois les actions créées, la prochaine étape consiste à créer un reducer pour gérer les articles. Le reducer est une fonction pure qui prend en entrée l'état actuel du store Redux et une action, et renvoie un nouvel état qui reflète les modifications apportées par l'action.

Voici un exemple de reducer pour gérer les articles :

```
import { ADD_POST, DELETE_POST } from './actions';

const initialState = {
  posts: [],
};

function postsReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_POST:
      return {
        ...state,
        posts: [...state.posts, action.payload],
      };
    case DELETE_POST:
      return {
        ...state,
        posts: state.posts.filter(post => post.id !== action.payload.id),
      };
    default:
      return state;
  }
}

export default postsReducer;
```


Création du store

La dernière étape consiste à créer le store Redux lui-même. Le store est un objet qui contient l'état de l'application et les méthodes nécessaires pour le modifier. Voici un exemple de création du store "posts" :

```
import { createStore } from 'redux';  
import postsReducer from './postsReducer';  
  
const store = createStore(postsReducer);  
  
export default store;
```

Création d'un Provider

```
import React from 'react';
import { Provider } from 'react-redux';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import store from './store';
import Home from './Home';
import PostList from './PostList';
import PostDetails from './PostDetails';

function App() {
  return (
    <Provider store={store}>
      <Router>
        <Switch>
          <Route exact path="/" component={Home} />
          <Route exact path="/posts" component={PostList} />
          <Route exact path="/posts/:id" component={PostDetails} />
        </Switch>
      </Router>
    </Provider>
  );
}

export default App;
```




Comment utiliser Redux avec des composants fonctionnels ?

Redux peut être utilisé avec des composants fonctionnels en utilisant la bibliothèque react-redux. react-redux fournit des hooks qui permettent aux composants fonctionnels de se connecter au store Redux. Voici les principaux hooks que vous utiliserez pour connecter des composants fonctionnels au store Redux :

Exemple d'utilisation de useSelector

Dans cet exemple, nous utilisons useSelector pour accéder à la liste des articles (posts) stockés dans le store Redux et les afficher dans un composant PostList.

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

function PostList() {
  const posts = useSelector(state => state.posts);

  return (
    <div>
      {posts.map(post => (
        <div key={post.id}>
          <h2>{post.title}</h2>
          <p>{post.content}</p>
        </div>
      ))}
    </div>
  );
}

export default PostList;
```

Exemple d'utilisation de useDispatch

```
import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { addPost } from './actions';

function AddPostForm() {
  const dispatch = useDispatch();
  const [title, setTitle] = useState('');
  const [content, setContent] = useState('');

  const handleSubmit = e => {
    e.preventDefault();
    dispatch(addPost(title, content));
    setTitle('');
    setContent('');
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Titre :
        <input type="text" value={title} onChange={e => setTitle(e.target.value)} />
      </label>
      <label>
        Contenu :
        <textarea value={content} onChange={e => setContent(e.target.value)} />
      </label>
      <button type="submit">Ajouter</button>
    </form>
  );
}

export default AddPostForm;
```

Dans cet exemple, nous utilisons useDispatch pour déclencher une action qui ajoute un nouvel article (post) dans le store Redux. Le composant AddPostForm permet aux utilisateurs de saisir un titre et un contenu pour un nouvel article, puis de cliquer sur le bouton "Ajouter" pour déclencher l'action.

Plusieurs reducers ?


```
import React from 'react';
import { Provider } from 'react-redux';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import { createStore, combineReducers } from 'redux';
import postsReducer from './postsReducer';
import usersReducer from './usersReducer';
import Home from './Home';
import PostList from './PostList';
import PostDetails from './PostDetails';

const rootReducer = combineReducers({
  posts: postsReducer,
  users: usersReducer,
});

const store = createStore(rootReducer);

function App() {
  return (
    <Provider store={store}>
      <Router>
        <Switch>
          <Route exact path="/" component={Home} />
          <Route exact path="/posts" component={PostList} />
          <Route exact path="/posts/:id" component={PostDetails} />
        </Switch>
      </Router>
    </Provider>
  );
}

export default App;
```



```
1  import React, { useState } from "react"
2  import Article from "../components/Article/Article"
3  import AddArticle from "../components/AddArticle/AddArticle"
4
5  const Articles = () => {
6    const [articles, setArticles] = useState([
7      { id: 1, title: "post 1", body: "Quisque cursus, metus vitae pharetra" },
8      { id: 2, title: "post 2", body: "Quisque cursus, metus vitae pharetra" },
9    ])
10   const saveArticle = e => {
11     e.preventDefault()
12     // the logic will be updated later
13   }
14
15   return (
16     <div>
17       <AddArticle saveArticle={saveArticle} />
18       {articles.map(article => (
19         <Article key={article.id} article={article} />
20       )))
21     </div>
22   )
23 }
24
25 export default Articles
26
```

Reducer.js

```
1  const initialState = {  
2    articles: [  
3      { id: 1, title: "post 1", body: "Quisque cursus, metus vitae pharetra" },  
4      { id: 2, title: "post 2", body: "Quisque cursus, metus vitae pharetra" },  
5    ],  
6  }  
7  
8  const reducer = (state = initialState, action) => {  
9    return state  
10  }  
11  export default reducer
```


App.js

```
1  import React from "react"
2  import ReactDOM from "react-dom"
3  import { createStore } from "redux"
4  import { Provider } from "react-redux"
5
6  import "./index.css"
7  import App from "./App"
8  import reducer from "./store/reducer"
9
10 const store = createStore(reducer)
11
12 ReactDOM.render(

13   <Provider store={store}>
14     <App />
15   </Provider>,
16   document.getElementById("root")
17 )


```

Injection de la donnée

```
1  import React from "react"
2  import { connect } from "react-redux"
3
4  import Article from "../components/Article/Article"
5  import AddArticle from "../components/AddArticle/AddArticle"
6
7  const Articles = ({ articles }) => {
8    const saveArticle = e => {
9      e.preventDefault()
10     // the logic will be updated later
11   }
12   return (
13     <div>
14       <AddArticle saveArticle={saveArticle} />
15       {articles.map(article => (
16         <Article key={article.id} article={article} />
17       ))}
18     </div>
19   )
20 }
21
22 const mapStateToProps = state => {
23   return {
24     articles: state.articles,
25   }
26 }
27
28 export default connect(mapStateToProps)(Articles)
```

Création d'une action

```
1  import React from "react"
2  import { connect } from "react-redux"
3
4  import Article from "../components/Article/Article"
5  import AddArticle from "../components/AddArticle/AddArticle"
6  import * as actionTypes from "../store/actionTypes"
7
8  const Articles = ({ articles, saveArticle }) => (
9    <div>
10      <AddArticle saveArticle={saveArticle} />
11      {articles.map(article => (
12        <Article key={article.id} article={article} />
13      ))}
14    </div>
15  )
16
17  const mapStateToProps = state => {
18    return {
19      articles: state.articles,
20    }
21  }
22
23  const mapDispatchToProps = dispatch => {
24    return {
25      saveArticle: article =>
26        dispatch({ type: actionTypes.ADD_ARTICLE, articleData: { article } }),
27    }
28  }
29
30  export default connect(mapStateToProps, mapDispatchToProps)(Articles)
```

Reducer.js

```
1  import * as actionTypes from "../actionTypes"
2
3  const initialState = {
4    articles: [
5      { id: 1, title: "post 1", body: "Quisque cursus, metus vitae pharetra" },
6      { id: 2, title: "post 2", body: "Quisque cursus, metus vitae pharetra" },
7    ],
8  }
9
10 const reducer = (state = initialState, action) => {
11   switch (action.type) {
12     case actionTypes.ADD_ARTICLE:
13       const newArticle = {
14         id: Math.random(), // not really unique but it's just an example
15         title: action.article.title,
16         body: action.article.body,
17       }
18       return {
19         ...state,
20         articles: state.articles.concat(newArticle),
21       }
22     }
23   return state
24 }
25 export default reducer
```



Import de middleware

npm install redux-thunk

```
import { createStore, applyMiddleware } from 'redux';  
import thunk from 'redux-thunk';  
import rootReducer from './reducers';  
  
const store = createStore(rootReducer, applyMiddleware(thunk));  
  
export default store;
```

Utilisation dans une action

Dans cet exemple, nous avons utilisé la fonction `applyMiddleware` fournie par Redux pour appliquer `thunk` en tant que middleware à notre store Redux. `thunk` est un middleware qui permet de retourner des fonctions à la place des objets d'action. Ces fonctions peuvent effectuer des opérations asynchrones, telles que des appels à une API, avant de déclencher une action.

Pour utiliser Redux Thunk, nous devons également modifier nos actions pour qu'elles retournent des fonctions plutôt que des objets d'action. Voici un exemple d'une action qui récupère la liste des articles à partir d'une API et la stocke dans le store Redux :

```
export function fetchPosts() {
  return async dispatch => {
    try {
      const response = await fetch('https://my-api.com/posts');
      const posts = await response.json();
      dispatch({ type: FETCH_POSTS_SUCCESS, payload: posts });
    } catch (error) {
      dispatch({ type: FETCH_POSTS_FAILURE, payload: error.message });
    }
  };
}
```




Intégration de Redux Persist

Redux Persist est une bibliothèque qui permet de persister l'état du store Redux dans le stockage local (localStorage ou AsyncStorage) du navigateur ou du dispositif mobile. Cela permet de conserver les données de l'application même après une actualisation ou une fermeture de l'application.

```
npm install redux-persist
```

Configuration

Ensuite, nous devons configurer Redux Persist en créant un persisteur et en le combinant avec notre réducteur racine. Voici un exemple de configuration de Redux Persist :

```
import { createStore } from 'redux';
import { persistStore, persistReducer } from 'redux-persist';
import storage from 'redux-persist/lib/storage';
import rootReducer from './reducers';

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

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = createStore(persistedReducer);
export const persistor = persistStore(store);
```

Ajouter le persisteur à notre application

Enfin, nous devons ajouter le persisteur à notre application en enveloppant notre composant Provider avec un composant PersistGate fourni par Redux Persist. Voici un exemple de configuration de PersistGate :

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

function Root() {
  return (
    <Provider store={store}>
      <PersistGate loading={null} persistor={persistor}>
        <App />
      </PersistGate>
    </Provider>
  );
}

export default Root;
```



Gestion des erreurs avec les "Error Boundaries"

Les "Error Boundaries" sont une fonctionnalité de React qui permet de capturer les erreurs qui se produisent dans les composants enfants lors de la phase de rendu. Les "Error Boundaries" peuvent être utilisés pour empêcher l'application de planter en cas d'erreur et pour afficher un message d'erreur convivial à l'utilisateur.

Exemple d'utilisation d'"Error Boundaries"

Voici un exemple d'utilisation d'un "Error Boundary" pour capturer les erreurs qui se produisent dans un composant enfant :

```
import React, { Component } from 'react';

class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  componentDidCatch(error, info) {
    // Mettre à jour l'état pour afficher un message d'erreur convivial
    this.setState({ hasError: true });
    // Enregistrer l'erreur quelque part
    console.log(error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Oops, quelque chose s'est mal passé.</h1>;
    }
    return this.props.children;
  }
}

export default ErrorBoundary;
```




Intégration de Redux Toolkit

Redux Toolkit est un package qui vise à faciliter la configuration de Redux et à réduire la quantité de code nécessaire pour gérer l'état de votre application.

Il fournit un ensemble d'outils prêts à l'emploi pour écrire des réducteurs (reducers), des actions (actions), et des middlewares, et permet également de gérer les opérations asynchrones de manière plus simple.

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


Création du store

Utilisation de la méthode configureStore :

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

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

export default store
```

Si on a plusieurs reducers :

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

const store = configureStore({
  reducer: {
    users: usersReducer,
    posts: postsReducer,
  },
})

export default store
```

Utilisation de createReducer

Si les reducers que nous utilisons précédemment sont toujours valides, nous pouvons les recréer via la méthode `createReducer`. Elle utilise la bibliothèque Immer en interne, ce qui vous permet d'écrire du code qui "mue" des données, mais qui applique en réalité les mises à jour de manière immuable. Cela rend pratiquement impossible la mutation accidentelle de l'état dans un réducteur.

```
const todosReducer = createReducer([], (builder) => {
  builder
    .addCase('ADD_TODO', (state, action) => {
      // "mutate" the array by calling push()
      state.push(action.payload)
    })
    .addCase('TOGGLE_TODO', (state, action) => {
      const todo = state[action.payload.index]
      // "mutate" the object by overwriting a field
      todo.completed = !todo.completed
    })
    .addCase('REMOVE_TODO', (state, action) => {
      // Can still return an immutably-updated value if we want to
      return state.filter((todo, i) => i !== action.payload.index)
    })
})
```

Utilisation de createReducer

Si les reducers que nous utilisons précédemment sont toujours valides, nous pouvons les recréer via la méthode `createReducer`. Elle utilise la bibliothèque Immer en interne, ce qui vous permet d'écrire du code qui "mue" des données, mais qui applique en réalité les mises à jour de manière immuable. Cela rend pratiquement impossible la mutation accidentelle de l'état dans un réducteur.

```
const todosReducer = createReducer([], (builder) => {
  builder
    .addCase('ADD_TODO', (state, action) => {
      // "mutate" the array by calling push()
      state.push(action.payload)
    })
    .addCase('TOGGLE_TODO', (state, action) => {
      const todo = state[action.payload.index]
      // "mutate" the object by overwriting a field
      todo.completed = !todo.completed
    })
    .addCase('REMOVE_TODO', (state, action) => {
      // Can still return an immutably-updated value if we want to
      return state.filter((todo, i) => i !== action.payload.index)
    })
})
```

Exemple d'action "classique"

```
case "UPDATE_VALUE":  
  return {  
    ...state,  
    first: {  
      ...state.first,  
      second: {  
        ...state.first.second,  
        [action.someId]: {  
          ...state.first.second[action.someId],  
          fourth: action.someValue  
        }  
      }  
    }  
  }  
}
```



Même action avec createReducer

```
updateValue(state, action) {  
  const {someId, someValue} = action.payload;  
  state.first.second[someId].fourth = someValue;  
}
```


Utilisation de createAction

La fonction createAction nous permet de simplifier la création d'action. La fonction génère un créateur d'action pour le type donné, qui acceptera le payload en paramètre

```
const addTodo = createAction('ADD_TODO')
addTodo({ text: 'Buy milk' })
// {type : "ADD_TODO", payload : {text : "Buy milk"}}
```

Le créateur d'action peut également servir à déterminer les types dans un reducer :

```
const actionCreator = createAction('SOME_ACTION_TYPE')

console.log(actionCreator.toString())
// "SOME_ACTION_TYPE"

console.log(actionCreator.type)
// "SOME_ACTION_TYPE"

const reducer = createReducer({}, (builder) => {
  // actionCreator.toString() will automatically be called here
  // also, if you use TypeScript, the action type will be correctly inferred
  builder.addCase(actionCreator, (state, action) => {})

  // Or, you can reference the .type field:
  // if using TypeScript, the action type cannot be inferred that way
  builder.addCase(actionCreator.type, (state, action) => {})
})
```


Les slices

Une slice est une portion du state de Redux qui contient à la fois des réducteurs et des actions associées. Elle permet de diviser l'état global en parties plus petites et plus gérables, facilitant ainsi la maintenance et la mise à jour de l'application. Les slices sont créées à l'aide de la fonction `createSlice()` fournie par le package Redux Toolkit. Cette fonction prend en entrée un objet qui décrit les réducteurs, les actions et les valeurs initiales de la slice, puis elle génère automatiquement les actions et les réducteurs Redux associés. Les actions générées sont nommées en fonction des noms des réducteurs définis dans la slice, et les réducteurs gèrent automatiquement les mises à jour de l'état pour ces actions. Les slices permettent ainsi de réduire considérablement la quantité de code nécessaire pour gérer l'état de l'application et de rendre le code plus lisible et plus facile à maintenir.

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

const counterSlice = createSlice({
  name: 'counter',
  initialState: 0,
  reducers: {
    increment: state => state + 1,
    decrement: state => state - 1,
    reset: () => 0
  }
})

export const { increment, decrement, reset } = counterSlice.actions
export default counterSlice.reducer
```

Utilisation avec Redux Persist

Pour utiliser Redux Persist, nous devons ignorer ses types d'action.

```
import {
  persistStore,
  persistReducer,
  FLUSH,
  REHYDRATE,
  PAUSE,
  PERSIST,
  PURGE,
  REGISTER,
} from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import { PersistGate } from 'redux-persist/integration/react'

import App from './App'
import rootReducer from './reducers'

const persistConfig = {
  key: 'root',
  version: 1,
  storage,
}

const persistedReducer = persistReducer(persistConfig, rootReducer)

const store = configureStore({
  reducer: persistedReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
      },
    }),
})

let persistor = persistStore(store)
```



Test Unitaire

A quoi sert le test unitaire ?

- > Eviter des régressions
- > Garantir les fonctionnalités du produit
- > Gagner du temps sur le long terme



Test Unitaire

Quelles librairies pour le test unitaire ?

- > testing-library/react et jest
- > msw pour le mock de données

Examples

```
1  import React from 'react';
2  import { render, screen } from '@testing-library/react';
3  import App from './App';
4
5  test('renders learn react link', () => {
6    render(<App />);
7    const linkElement = screen.getByText(/learn react/i);
8    expect(linkElement).toBeInTheDocument();
9  });
10
```

Examples

```
import { render, screen, fireEvent } from '@testing-library/react'
import { ThemeProvider } from '../../utils/context'
import Footer from './'

test('Change theme', async () => {
  render(
    <ThemeProvider>
      <Footer />
    </ThemeProvider>
  )
  const nightModeButton = screen.getByRole('button')
  expect(nightModeButton.textContent).toBe('Changer de mode : ☀️')
  fireEvent.click(nightModeButton)
  expect(nightModeButton.textContent).toBe('Changer de mode : 🌙')
})
```


Examples (Tests redux)

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

const initialState = [{ text: 'Use Redux', completed: false, id: 0 }]

const todosSlice = createSlice({
  name: 'todos',
  initialState,
  reducers: {
    todoAdded(state, action) {
      state.push({
        id: state.reduce((maxId, todo) => Math.max(todo.id, maxId), -1) + 1,
        completed: false,
        text: action.payload
      })
    }
  }
})

export const { todoAdded } = todosSlice.actions

export default todosSlice.reducer
```

```
import reducer, { todoAdded } from './todosSlice'

test('should return the initial state', () => {
  expect(reducer(undefined, { type: undefined })).toEqual([
    { text: 'Use Redux', completed: false, id: 0 }
  ])
})

test('should handle a todo being added to an empty list', () => {
  const previousState = []

  expect(reducer(previousState, todoAdded('Run the tests'))).toEqual([
    { text: 'Run the tests', completed: false, id: 0 }
  ])
})

test('should handle a todo being added to an existing list', () => {
  const previousState = [{ text: 'Run the tests', completed: true, id: 0 }]

  expect(reducer(previousState, todoAdded('Use Redux'))).toEqual([
    { text: 'Run the tests', completed: true, id: 0 },
    { text: 'Use Redux', completed: false, id: 1 }
  ])
})
```

Mocks de données avec msw

```
const server = setupServer(
  rest.get('http://localhost:8000/freelances', (req, res, ctx) => {
    return res(ctx.json({ freelancersList: freelancersMockedData }))
  })
)

beforeAll(() => server.listen())
afterEach(() => server.resetHandlers())
afterAll(() => server.close())

it('Should display freelancers names after loader is removed', async () => {
  render(<Freelances />)

  await waitForElementToBeRemoved(() => screen.getByTestId('loader'))
  expect(screen.getByText('Harry Potter')).toBeInTheDocument()
  expect(screen.getByText('Hermione Granger')).toBeInTheDocument()
  expect(screen.queryByTestId('loader')).not.toBeInTheDocument()
})
```

