



19.1 Redux Toolkit

Pourquoi Redux toolkit

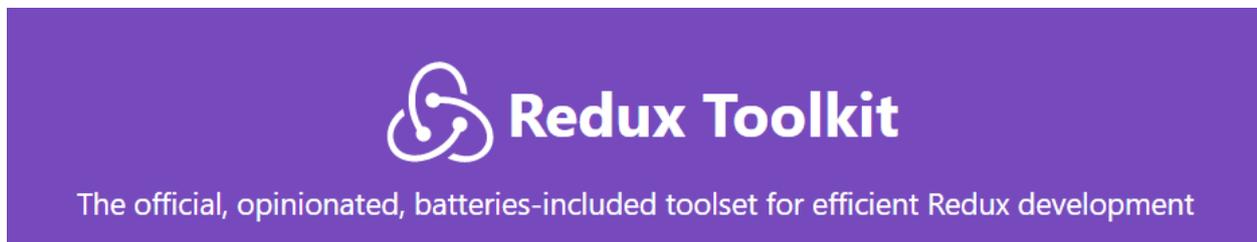
redux nécessite beaucoup de notions :

- ✓ Action
- ✓ Action Objet
- ✓ Action createur
- ✓ Switch déclaration dans le reducer

Beaucoup de package qui doivent être installé pour travailler avec redux

- ✓ Redux-thunk
- ✓ Immer
- ✓ Redux-devtool

La solution c'est l'utilisation de Reduxjs/toolkit



 **Redux Toolkit**

The official, opinionated, batteries-included toolset for efficient Redux development

 Simple	 Opinionated	 Powerful	 Effective
Includes utilities to simplify common use cases like store setup , creating reducers , immutable update logic , and more.	Provides good defaults for store setup out of the box , and includes the most commonly used Redux addons built-in .	Takes inspiration from libraries like Immer and Autodux to let you write "mutative" immutable update logic , and even create entire "slices" of state automatically .	Lets you focus on the core logic your app needs, so you can do more work with less code .

Le package Redux Toolkit est destiné à être le moyen standard d'écrire la logique Redux. Il a été créé à l'origine pour répondre à trois préoccupations courantes concernant Redux :

- ✓ Configurer Redux est trop compliqué
- ✓ on doit ajouter beaucoup de packages pour que Redux fasse quelque chose d'utile.
- ✓ Redux nécessite trop de code passe-partout

Par ailleurs Redux Toolkit comprend également une puissante capacité de récupération et de mise en cache des données que nous avons baptisée "Requête RTK". Il est inclus dans le package en tant qu'ensemble distinct de points d'entrée. C'est facultatif, mais cela peut éliminer le besoin d'écrire vous-même la logique de récupération des données.

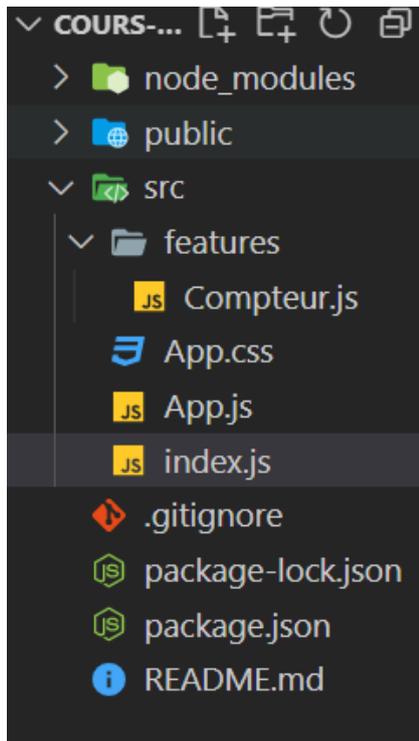


Installation

Pour utiliser Redux toolkit on doit installer

- ✓ `npx create-react-app cours-redux-toolkit`
- ✓ `npm install react-redux`
- ✓ `npm install @reduxjs/toolkit`

La structure projet compte pour expliquer les concepts de base de reduxjs/toolkit



Compteur.js

```
import { createSlice } from '@reduxjs/toolkit'
const initialState = { value: 0 }
const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment:(state)=> {
      state.value++
    },
    decrement:(state)=> {
      state.value--
    },
    incrementByAmount:(state, action)=> {
      state.value += action.payload
    },
  },
})
export const { increment, decrement, incrementByAmount } = counterSlice.actions
export default counterSlice.reducer
```



createSlice :

createSlice est une fonction qui accepte un état initial, un objet de fonctions de réduction et un "nom de slice", et génère automatiquement des créateurs d'action et des types d'action qui correspondent aux réducteurs et à l'état.

Cette API est l'approche standard pour écrire la logique Redux.

En interne, il utilise **createAction** et **createReducer**, vous pouvez donc également utiliser Immer pour écrire des mises à jour immuables "mutantes"

InitialState :

La valeur d'état initiale pour cette slice d'état.

Il peut également s'agir d'une fonction qui doit renvoyer une valeur d'état initiale lorsqu'elle est appelée. Ceci sera utilisé chaque fois que le réducteur est appelé avec undefined comme valeur d'état, et est principalement utile pour des cas comme la lecture de l'état initial à partir de localStorage.

Name :

Un nom de chaîne pour cette slice d'état. Les constantes de type d'action générées l'utiliseront comme préfixe.

reducers :

Un objet contenant des fonctions Redux : fonctions destinées à gérer un type d'action spécifique, équivalent à une seule instruction switch .

Les clés de l'objet seront utilisées pour générer des constantes de type action de chaîne, et celles-ci apparaîtront dans l'extension Redux DevTools lorsqu'elles seront envoyées. De plus, si une autre partie de l'application envoie une action avec exactement le même type de chaîne, le reducer correspondant sera exécuté. Par conséquent, vous devez donner aux fonctions des noms descriptifs.

Cet objet sera passé à createReducer, afin que les reducers puissent "muter" en toute sécurité l'état qui leur est donné.

Vous l'avez peut-être remarqué, mais lors de la déclaration des *reducers* dans notre *slice*, nous avons modifié l'état et n'avons donc pas respecté le principe d'immutabilité et le fait qu'un *reducer* doit être une fonction pure.

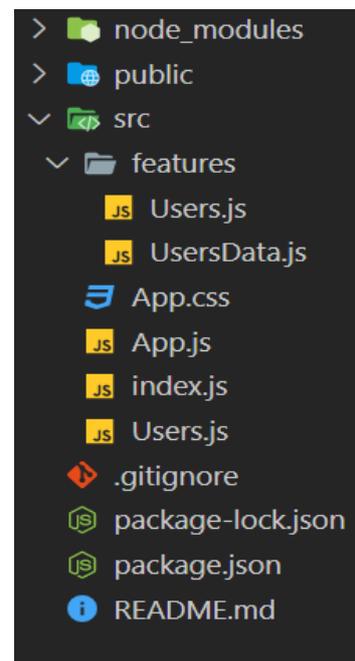
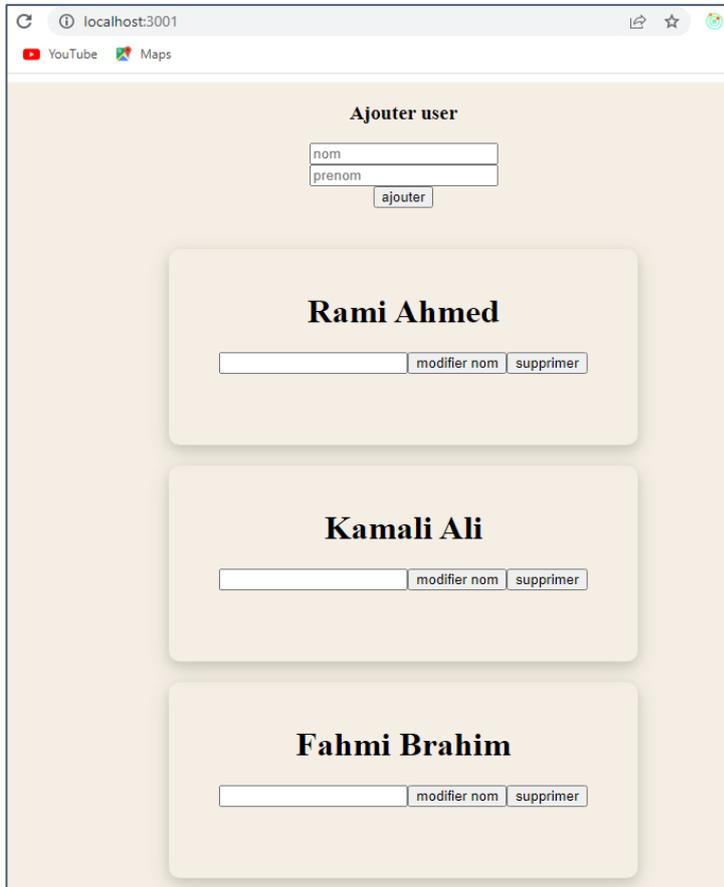
En réalité, c'est totalement acceptable, grâce à Immer qui est intégré dans @reduxjs/toolkit.



Projet crud :

Pour vocation pédagogique, le projet consiste à faire les opérations CRUD sur une liste locale, le projet utilisé : **cours-redux-toolkit-crud**

La structure du projet crud :



Pour l'atelier de cours on va utiliser les données suivantes :

usersData.js

```
export const usersData=[  
{id:1,nom: 'Rami',prenom: "Ahmed"},  
{id:2,nom: 'Kamali',prenom: "Ali"},  
{id:3,nom: 'Fahmi',prenom: "Brahim"}];
```

index.js

```
import ReactDOM from "react-dom/client";  
import App from './App';  
import {Provider} from 'react-redux';  
import {configureStore} from '@reduxjs/toolkit'  
import userReducer from './features/Users'  
  
const root=ReactDOM.createRoot(document.getElementById("root"));  
const store=configureStore({  
  reducer:{ users:userReducer}  
})
```



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

On remarque que le store est créé par la méthode `configureStore` de `@reduxjs/toolkit`, qui reçoit en argument un objet.

```
{
  reducer:{
    users:userReducer
  }
}
```

Cet objet contient l'attribut `reducer`, ce dernier et aussi un objet qui contient le reducer `userReducer`

users:userReducer l'attribut `users` sera le nom de state de reducer 'userReducer'

Préparation de reducer

Soit le fichier `Users.js` qui se trouve dans le dossier `features`

Users.js

```
import {createSlice} from '@reduxjs/toolkit'
import {usersData} from './UsersData'
export const userSlice=createSlice({
  name:"users",
  initialState:{value:usersData},
  reducers:{
    addUser:(state,action)=>{
      state.value.push(action.payload);
    },
    deleteUser:(state,action)=>{
      state.value=state.value.filter(user=>user.id!=action.payload.id)
    },
    updateNom:(state,action)=>{
      state.value=state.value.map(user=>{if(user.id===action.payload.id){ return
({...user,nom:action.payload.nom}) }else{return user;}})
    }
  },
});

export const {addUser,deleteUser,updateNom}=userSlice.actions;
export default userSlice.reducer;
```



vous allez remarquer que createSlice de reduxjs/toolkit utilise Immer pour écrire des mises à jour immuables "mutantes".

On modifie directement le state et c'est immer qui s'en occupe de reste :

Exemple :

```
state.value=state.value.filter(user=>user.id!=action.payload.id)
```

Préparation de l'interface utilisateur :

App.js

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

import {useRef } from 'react'
import {addUser} from "../features/Users";
import "../App.css"
import Users from "../Users";

function App(){
  const dispatch=useDispatch();
  const userList=useSelector(data=>data.users.value)
  // console.log(usersList)
  const nom=useRef('');
  const prenom=useRef('');
  function ajouter(){
    const id=usersList[usersList.length-1].id;
    dispatch(addUser({id:id,nom:nom.current.value,prenom:prenom.current.value}))
    nom.current.value="";
    prenom.current.value="";
  }
  return <div className="App">
    <input type="text" placeholder="nom" ref={nom} />
    <input type="text" placeholder="prenom" ref={prenom}/>
    <button onClick={()=>{ajouter()}}>ajouter</button>
    <Users/>
  </div>
}
```

Users.js

```
import '../App.css';
import {useSelector,useDispatch} from 'react-redux';
import {deleteUser,updateNom } from '../features/Users';
import {useState,useRef } from 'react';
```



```
export default function Users(){
const userList=useSelector((state)=>state.users.value)
const dispatch=useDispatch();
const [nom,setNom]=useState('')

return(<>
<div className='displayUsers'>
  {userList.map((user,index)=>{
    return <div key={index}>
      <h1>{user.nom} {user.prenom}</h1>
      <input onChange={(event)=> setNom(event.target.value)} />
      <button onClick={()=>dispatch(updateNom({id:user.id,nom:nom})}> update
userName</button>
      <button onClick={()=>dispatch(deleteUser({id:user.id})}>delete </button>
    </div>
  )})}
</div>
</>)
}
```

App.css

```
.App {
  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: column;
  border-radius: 10px;
  background-color: rgb(245, 238, 229);
}
.displayUsers {
  margin-top: 20px;
}
.displayUsers div {
  width: 400px;
  height: 150px;
  padding: 20px;
  margin: 20px;
  border-radius: 10px;
  box-shadow: 0 4px 8px 0 rgba(45, 67, 35, 0.1), 0 6px 20px 0 rgba(77, 91, 41, 0.19);
  text-align: center;
}
```



createAsyncThunk

Aperçu

createAsyncThunk est une fonction qui accepte une chaîne de type d'action Redux et une fonction de rappel qui doit renvoyer une promesse.

Elle génère des types d'action de cycle de vie de promesse en fonction du préfixe de type d'action que vous transmettez et renvoie un créateur d'action **thunk** qui exécutera le rappel de promesse et distribuera les actions de cycle de vie en fonction de la promesse renvoyée.

Cela résume l'approche standard recommandée pour la gestion des cycles de vie des demandes asynchrones.

Il ne génère aucune fonction de réduction, car il ne sait pas quelles données vous récupérez, comment vous souhaitez suivre l'état de chargement ou comment les données que vous renvoyez doivent être traitées. Vous devez écrire votre propre logique de réduction qui gère ces actions, avec l'état de chargement et la logique de traitement appropriés pour votre propre application.

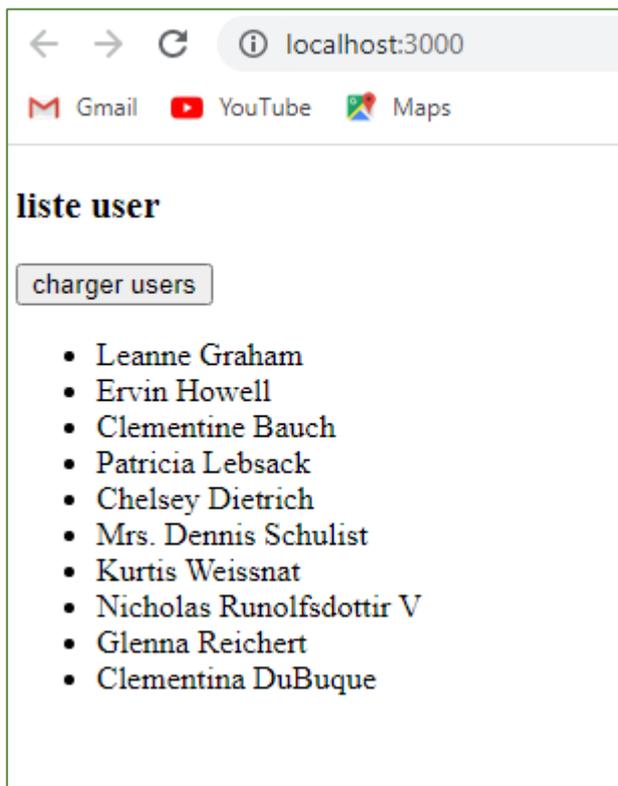
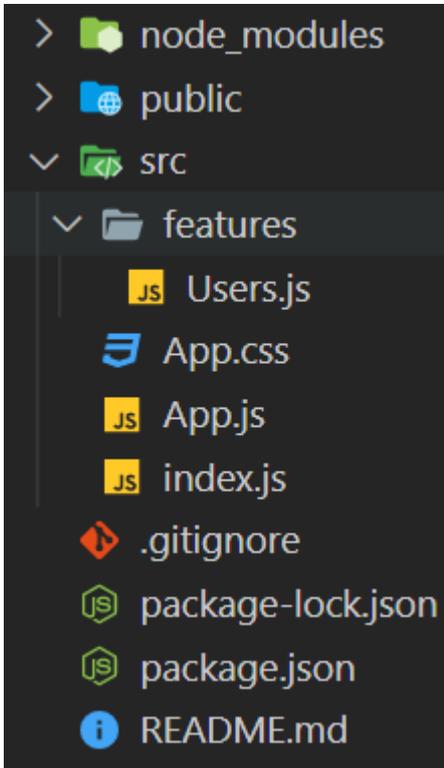
Je vous promets que vous allez très bien assimiler cette logique avec l'exemple ci-dessous 😊



Projet : consommation d'un API

Dossier projet : cours-redux-toolkit-crud-thunk (WinRAR)

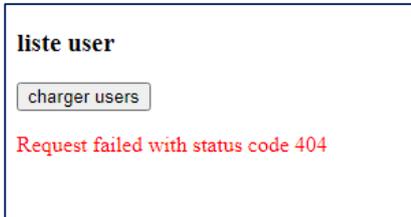
End point : <https://jsonplaceholder.typicode.com/users>





Quand on clique sur le bouton charger users le message **chargement...** s'affiche avant l'affichage de la liste des users.

Si il y a une erreur s'affiche l'erreur en couleur rouge



Features/Users.js

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

const initialState={
  loading:false,
  users:[],
  error:''
}
//elle genere pending,fulfilled et rejected action types
const fetchUsers =createAsyncThunk('users/fetchUsers', ()=>{
  return axios.get('https://jsonplaceholder.typicode.com/users')
  .then(response=>response.data)
})

export const userSlice=createSlice({
  name:"users",
  initialState:initialState,
  extraReducers:(builder)=>{
    builder.addCase(fetchUsers.pending,state=>{
      state.loading=true
    })
    builder.addCase(fetchUsers.fulfilled,(state,action)=>{
      state.loading=false;
      state.users=action.payload
    })
    builder.addCase(fetchUsers.rejected,(state,action)=>{
      state.loading=false;
      state.users=[];
      state.error=action.error
    })
  }
})

export {fetchUsers}
export default userSlice.reducer;
```



On utilise **extraReducers**

L'un des concepts clés de **Redux** est que chaque **sliceReducer** "possède" sa tranche d'état et que de nombreux **sliceReducer** peuvent répondre indépendamment au même type d'action.

extraReducers permet à `createSlice` de répondre à d'autres types d'action en plus des types qu'il a générés.

extraReducers utilise l'objet `builder` qui va modifier le state selon le cycle de vie de l'action asynchrone **fetchUsers**.

index.js

```
import ReactDOM from "react-dom/client";
import App from './App';
import {Provider} from 'react-redux';
import {configureStore} from '@reduxjs/toolkit'
import userReducer from './features/Users'

const root=ReactDOM.createRoot(document.getElementById("root"));
const store=configureStore({
  reducer:{
    users:userReducer }
})
root.render(<>
  <Provider store={store}>
    <App/>
  </Provider>

</>)
```

App.js

```
import {useDispatch,useSelector} from 'react-redux';
import {fetchUsers} from "./features/Users";

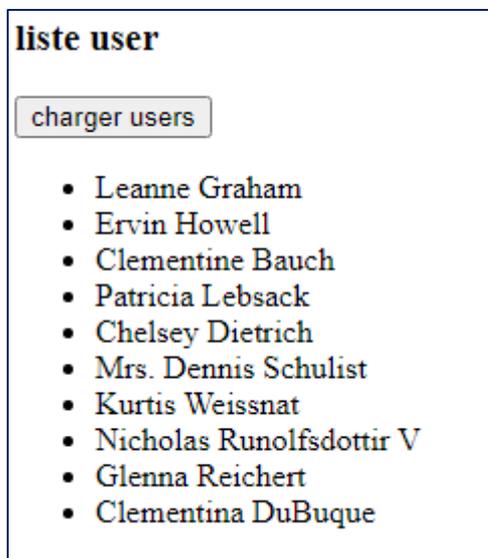
function App(){
  const dispatch=useDispatch();
  const data=useSelector(state=>state.users.users)
  const loading=useSelector(state=>state.users.loading)
  const error=useSelector(state=>state.users.error)

  function loadUsers(){
    dispatch(fetchUsers());
  }
}
```

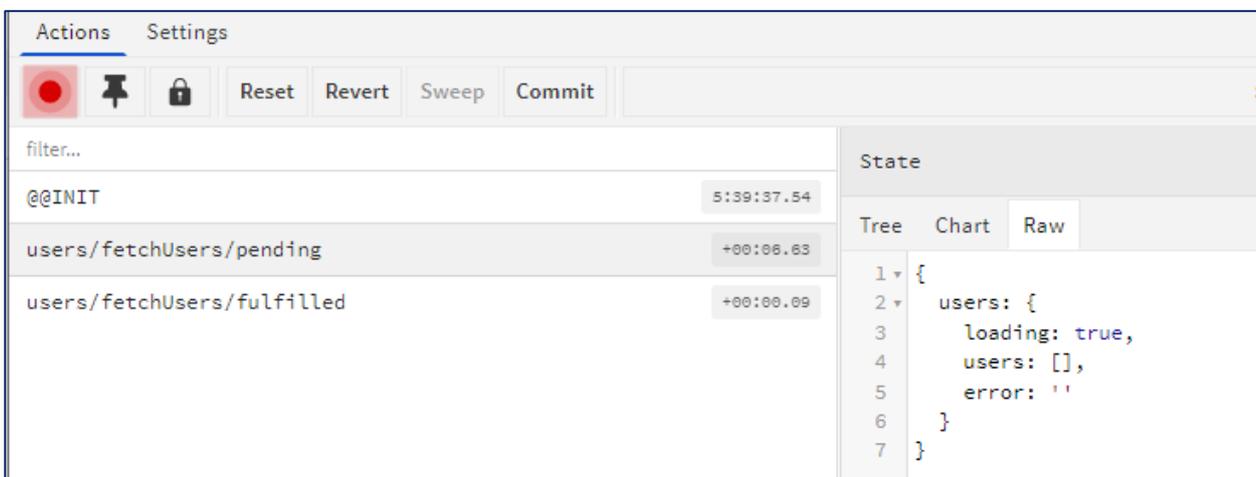


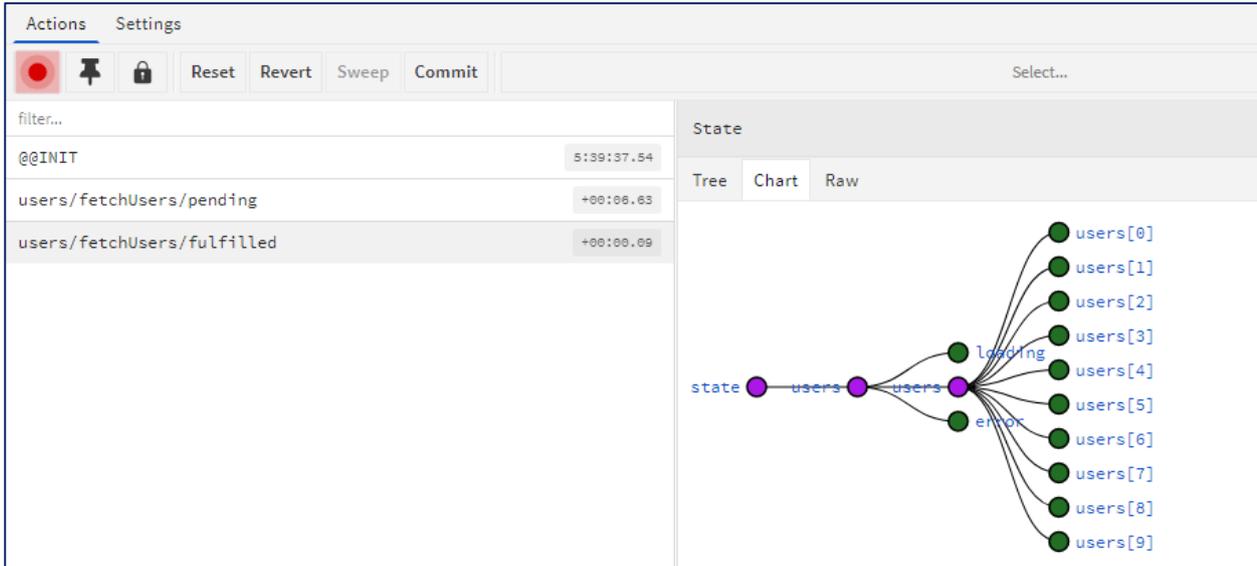
```
return <div >
  <h3>liste user</h3>

  <button onClick={loadUsers}>charger users</button>
  {loading && <h3>chargement...</h3>}
  {(!loading && data.length>0) &&
  <ul>
    {data.map(user=><li key={user.id}>{user.name}</li>)}
  </ul>}
  { (!loading && error) && <p style={{color:'red'}}>{error}</p>}
</div>
}
export default App;
```



avec '@reduxjs/toolkit' on peut utiliser directement Redux devTools sans aucune installation et configuration.





The screenshot shows Redux DevTools with the following state tree:

```

state
├── users
├── loading
└── error
  
```

The 'users' array contains 10 elements, indexed from 0 to 9.

Si on met une erreur dans l'EndPoint de l'API

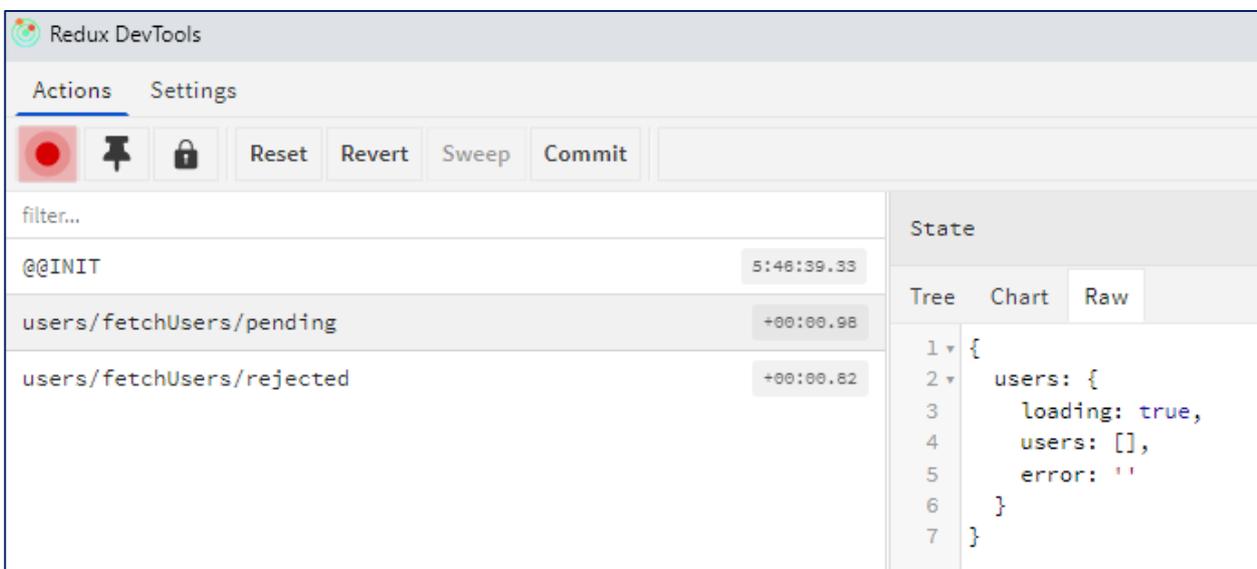
```

//elle genere pending,fulfilled et rejected action types
const fetchUsers =createAsyncThunk('users/fetchUsers', ()=>{
  return axios.get('https://jsonplaceholder.typicode.com/usersXXX')
  .then(response=>response.data)
})
  
```

liste user

charger users

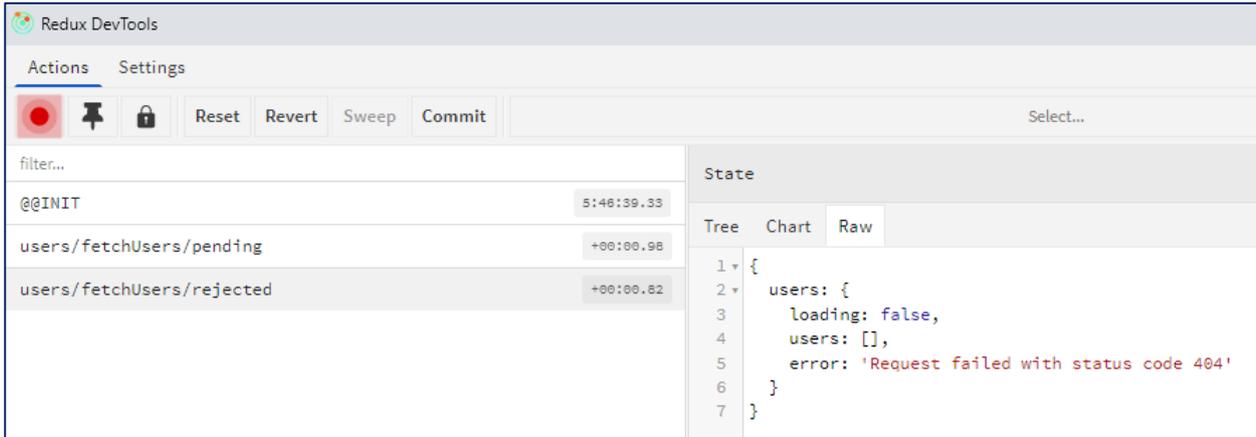
Request failed with status code 404



The screenshot shows Redux DevTools with the following state tree:

```

1 {
2   users: {
3     loading: true,
4     users: [],
5     error: ''
6   }
7 }
  
```



Vous allez remarquer que les actions

Users/fetchUsers/pending=>la requête est lancée

Users/fetchUsers/fulfilled=>la requête est exécutée avec succès et les données sont retournées

Users/fetchUsers/rejected=> il y a une erreur la requête est échouée

Sont créés et dispatcher selon le cycle de vie de promesse.

Le projet utilisé : cours-redux-toolkit-crud-thunk

Pour aller plus loin

Nous avons pu voir dans cet article comment Redux toolkit permet de réduire le temps de développement ainsi que la verbosité d'un code faisant utilisation de Redux.

Bien entendu l'article n'est pas exhaustif et présente un des atouts de cette librairie qui en cache bien d'autres.

Aussi je vous recommande de vous renseigner sur [la documentation officielle de Redux toolkit](#) qui est plutôt détaillée et qui liste bien les différentes fonctionnalités tout en les présentant avec des exemples.