

# Le middleware "Thunk" et `createAsyncThunk`` dans Redux Toolkit

## Qu'est-ce que Thunk dans Redux ?

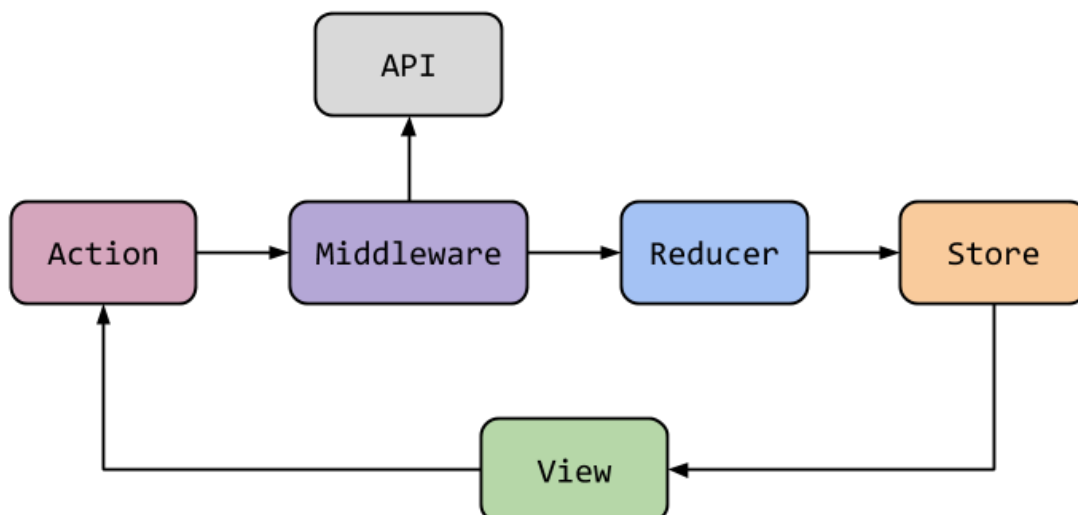
**Thunk** est utilisé pour récupérer des données d'une API et stocker la réponse dans le state Redux qui augmente le nettoyage du code.

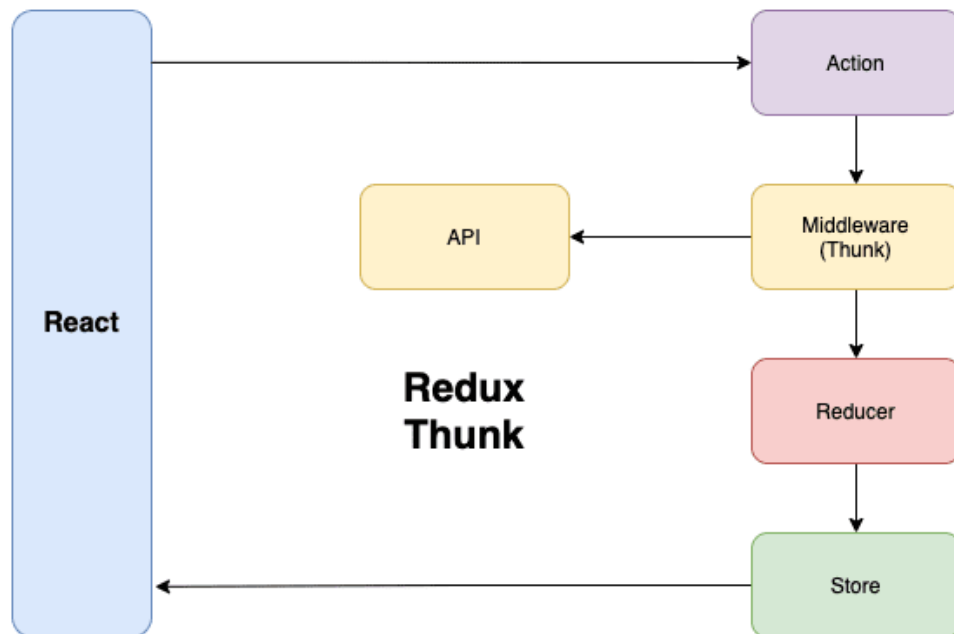
Par défaut, les actions de Redux sont expédiées de façon synchrone, ce qui pose un problème pour toute application non triviale qui doit communiquer avec une application externe ou effectuer des effets secondaires. Redux permet également au middleware qui se trouve au milieu d'une action d'être distribué et à l'action d'atteindre les réducteurs.

Il existe deux bibliothèques de middleware très populaires qui prennent en charge les effets spéciaux et les actions asynchrones : [Redux Thunk](#) et [Redux Saga](#). Cette publication vous emmène à la découverte de Redux Thunk.

**Thunk** est un concept de programmation dans lequel une fonction est utilisée pour retarder l'évaluation/le calcul d'une opération.

Redux Thunk est un middleware qui vous permet de faire un appel à l'action auprès des créateurs qui renvoie une fonction au lieu d'un objet d'action. Cette fonction reçoit la méthode de distribution du store. Elle permet donc d'envoyer des actions synchrones régulières dans le corps de la fonction une fois que les opérations asynchrones ont été terminées.





## Qu'avez-vous utilisé pour récupérer les données ?

Tout d'abord, en utilisant le hook `useEffect` et dans le cycle de vie `componentDidMount`, vous avez récupéré les données d'une API. Qu'en est-il du stockage dans Redux ? Vous auriez utilisé le hook `useDispatch` pour stocker et ensuite utiliser `useSelector` pour récupérer les données.

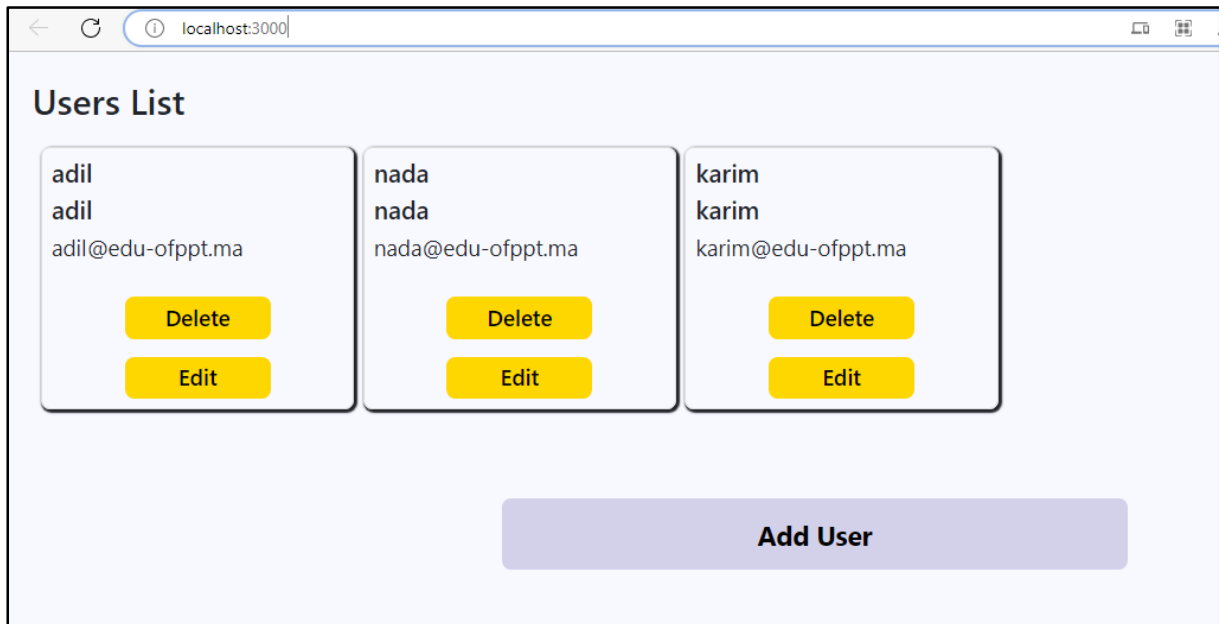
OK ? Maintenant, ces opérations sont assignées à Thunk et vous n'avez pas besoin de fouler chaque composant dans lequel vous utilisez les données que vous avez appelé une API.

Après cela, vous devez vérifier les résultats pour les statuts qui peuvent être remplis, rejetés et en attente, ce qui peut être fait plus facilement en utilisant Thunk.

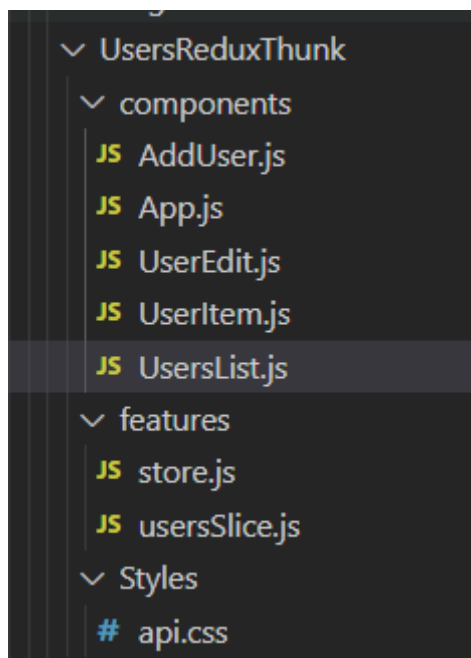
Cela abstrait l'approche standard recommandée pour gérer les cycles de vie des demandes asynchrones.

Ainsi, le code est plus propre, plus standard et plus flexible dans l'écriture.

## Exemple d'utilisation



## La structure du dossier :



Dans **App.js** vous configurez le router et le provider.

**<UserItem/>** est un child component de **<UsersList/>** qui affiche un seul utilisateur dans une carte, celle-ci contient toutes les informations et deux boutons **Delete** et **Edit**.

Après mettez quelques données dans db.json sur le serveur JSON :

```
{
  "users": [
    {
      "name": "adil",
      "username": "adil",
      "email": "adil@edu-ofppt.ma",
      "id": 1
    },
    {
      "name": "nada",
      "username": "nada",
      "email": "nada@edu-ofppt.ma",
      "id": 2
    },
    {
      "name": "karim",
      "username": "karim",
      "email": "karim@edu-ofppt.ma",
      "id": 3
    }
  ]
}
```

Considérons que j'ai une tranche appelée `usersSlice.js`. **`createAsyncThunk`** sera utilisé et créé comme indiqué ci-dessous. Supposons que nous voulons récupérer la liste des utilisateurs à partir d'une API.

## 1. **`getUsers` (pending, fulfilled, rejected)**

Tout d'abord, vous créez une variable appelée **`getUsers`** qui est assignée à **`createAsyncThunk`** (remarquez le mot clé export avant de déclarer la variable). **`createAsyncThunk`** a 2 arguments. Le premier est une chaîne de caractères pour spécifier le nom du Thunk et le second est une fonction asynchrone qui retournera une promesse.

Ensuite, vous créez une tranche en utilisant *createSlice*. Dans *extraReducers* (remarquez que la propriété reducers est différente) vous spécifiez 3 états probables de la promesse qui sont *pending*, *fulfilled et rejected*. Vous décidez ce que Redux doit faire dans ces 3 états différents de l'API.

- Pending signifie que la manipulation de l'API est en cours.
- Fulfilled signifie que la réponse a été obtenue de l'API.
- Rejected signifie que l'appel à l'API a échoué.

```
import { createSlice, createAsyncThunk } from "@reduxjs/toolkit";
import axios from "axios";
const initialState={value:[],isLoading:null,err:null}
export const getUsers=createAsyncThunk("user/getUser",
async(log,thunkAPI)=>{
  const {rejectWithValue}=thunkAPI
  try{
const resp=await axios.get('http://localhost:3006/users')
console.log(resp)
return resp.data
  }catch(error)
  {
    return rejectWithValue(error.message)
  }
})

const userSlice=createSlice({
  name:"user",
  initialState,
  reducers:{

  },
  extraReducers:{
    [getUsers.pending]:(state,action)=>{
      state.value=[]
      state.isLoading=true;
    },
    [getUsers.fulfilled]:(state,action)=>{

      state.value=action.payload

      state.isLoading=false;
    }
  }
})
```

```

    },
    [getUsers.rejected]:(state,action)=>{
        state.err=action.payload
        state.isLoading=false;
    }
}
}))

export default userSlice.reducer;

```

**rejectWithValue(error.message)** : rejectWithValue est une fonction utilitaire que vous pouvez renvoyer dans votre créateur d'action (**createAsyncThunk**) pour renvoyer une réponse rejetée avec une charge utile et des méta définis. Elle transmet la valeur que vous lui donnez et la renvoie dans les données utiles de l'action rejetée puis on sauvegarde l'erreur dans le state.

Après cela, vous déclarez le reducer que vous avez créé dans configureStore :

### store.js

```

import { configureStore } from "@reduxjs/toolkit"
import usersReducer from "../usersSlice"
export const store=configureStore({
    reducer:{
        users:usersReducer
    }
})

```

Ensuite, créez un composant appelé **UsersList.js** et ensuite, vous ferez comme ceci :

```

import React, { useEffect, useState } from "react";
import { useDispatch, useSelector } from "react-redux";
import { NavLink } from "react-router-dom";

import { getUsers } from "../features/usersSlice";
import UserItem from "../UserItem";
export default function UsersList() {
    const dispatch = useDispatch()
    useEffect(() => {
        dispatch(getUsers())
    }, [])
}

```

```

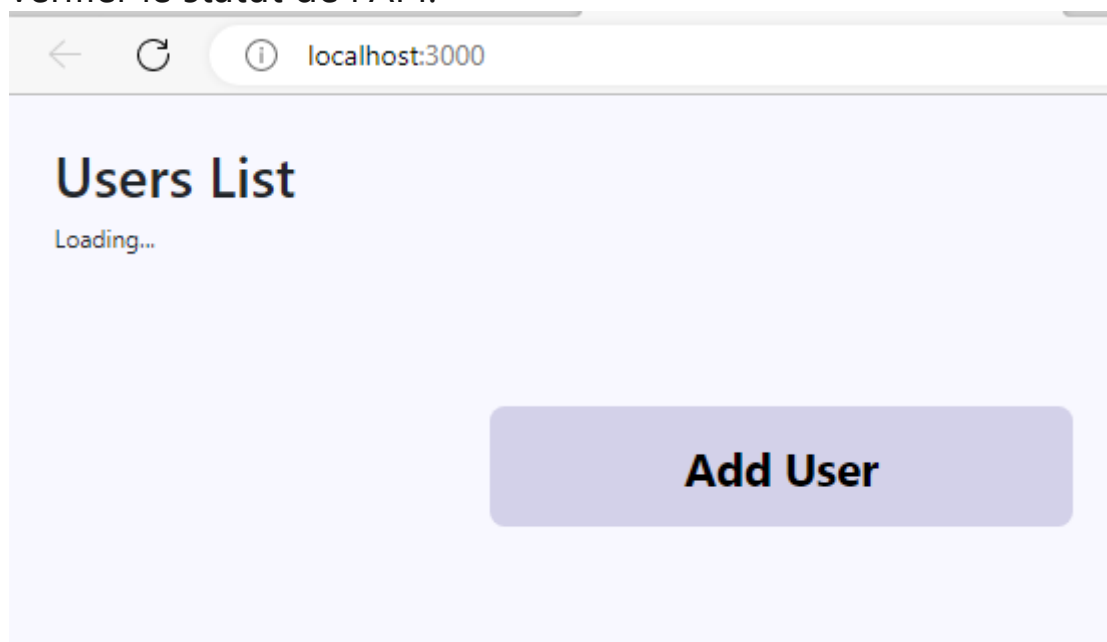
const { value, isLoading, err } = useSelector(state => state.users)

const users = <div>
{value.length>0 ? value.map((item) => (<UserItem user={item} key={item.id}
/>)) : null}
</div>
if (err !== null)
  return <h1>{err}</h1>
return (<
  <h1>Users List</h1>
  {isLoading ? "Loading..." : users}
  {value.length===0 && isLoading===false && <h2>There is no users</h2>}
  <NavLink to="/addUser">Add User</NavLink>
</>)
}

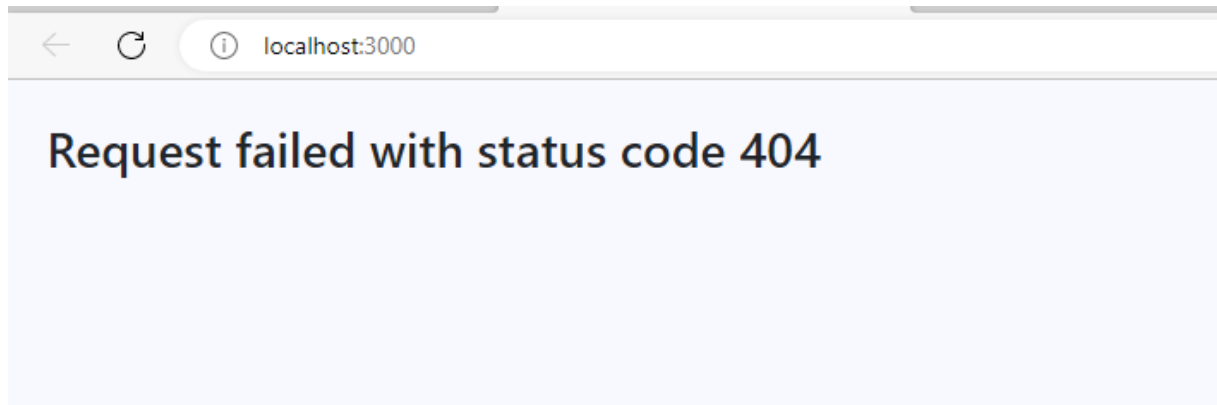
```

Tout d'abord, vous devez dispatcher la fonction asynchrone que vous avez créée en utilisant ***createAsyncThunk***. Toutes les opérations seront effectuées par Redux! Tout est prêt et vous pouvez utiliser le hook `useSelector` pour obtenir des données et les utiliser comme vous le souhaitez.

Vous pouvez également utiliser l'état du statut (**`isLoading`**) pour vérifier le statut de l'API.



En cas d'erreur on affichera le message renvoyé par le serveur qu'on a stocké dans le state Redux **'err'**. Essayer par exemple de stopper l'exécution du serveur JSON pour voir cette erreur.



## 2. addUser (pending, fulfilled, rejected)

En suivant la même logique on va insérer un nouvel utilisateur via API.

A screenshot of a web browser window. The address bar shows 'localhost:3000/addUser'. The main content area displays a form titled 'Add User :'. The form has three input fields: 'Name :', 'Username :', and 'Email :'. Each field has a placeholder text 'name ...', 'username ...', and 'email ...' respectively. Below the input fields are two blue buttons: 'Add' and 'back'.



vous créez une variable appelée ***addUser*** qui est assignée à ***createAsyncThunk*** (remarquez le mot clé export avant de déclarer la variable). ***createAsyncThunk*** a 2 arguments. Le premier est une chaîne de caractères pour spécifier le nom du Thunk et le second est une fonction asynchrone qui retournera une promesse.

Ensuite, Dans ***extraReducers*** (vous spécifiez 3 états probables de la promesse qui sont ***pending***, ***fulfilled*** et ***rejected***. Vous décidez ce que Redux doit faire dans ces 3 états différents de l'API.

- Pending signifie que la manipulation de l'API est en cours.
- Fulfilled signifie que la réponse a été obtenue de l'API.
- Rejected signifie que l'appel à l'API a échoué.

```
export const addUser=createAsyncThunk("user/addUser",
async(arg,thunkAPI)=>{
  const {rejectWithValue}=thunkAPI
  try{
const resp=await axios.post(`http://localhost:3006/users`,arg)
console.log(resp)
return arg
  }catch(error)
  {
    return rejectWithValue(error.message)
  }
})

const userSlice=createSlice({
  name:"user",
  initialState,
  reducers:{

  },
  extraReducers:{
    [addUser.pending]:(state,action)=>{

      state.isLoading=true;

    },
    [addUser.fulfilled]:(state,action)=>{
      state.value.push(action.payload)
      state.isLoading=false;
    },
    [addUser.rejected]:(state,action)=>{
```

```
        state.err=action.payload
        state.isLoading=false;
    }
}
})
```

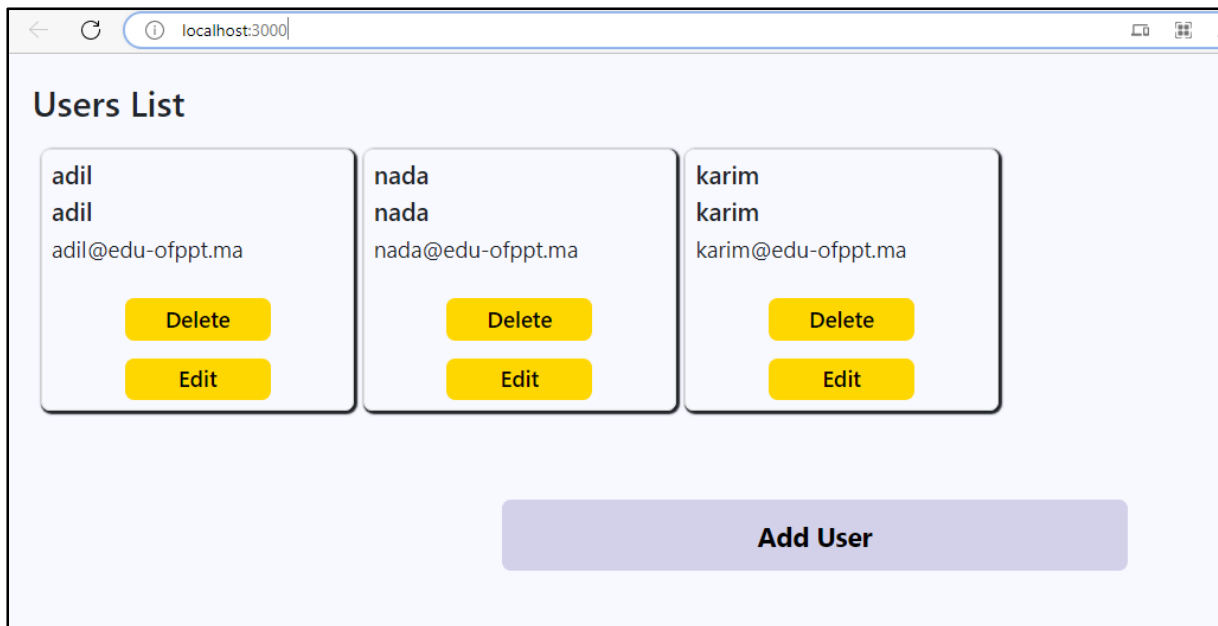
La méthode **createAsyncThunk** sera appelée avec deux arguments :

**arg** : une valeur unique, contenant le premier paramètre qui a été passé au créateur d'action **thunk** lorsqu'il a été envoyé. Ceci est utile pour passer des valeurs telles que les ID des éléments qui peuvent être nécessaires dans le cadre de la requête. Si vous devez passer plusieurs valeurs, passez-les ensemble dans un objet lorsque vous distribuez le thunk, comme

```
dispatch(addUser({name : 'Adil', username : 'Adil', email : 'adil@edu-ofppt.ma'})).
```

### 3. deleteUser (pending, fulfilled, rejected)

En suivant la même logique on va supprimer un utilisateur en utilisant la méthode **delete** de l'API axios.



vous créez une variable appelée ***deleteUser*** qui est assignée à ***createAsyncThunk*** (remarquez le mot clé export avant de déclarer la variable). ***createAsyncThunk*** a 2 arguments. Le premier est une chaîne de caractères pour spécifier le nom du Thunk et le second est une fonction asynchrone qui retournera une promesse.

Ensuite, Dans ***extraReducers*** (vous spécifiez 3 états probables de la promesse qui sont ***pending***, ***fulfilled*** et ***rejected***. Vous décidez ce que Redux doit faire dans ces 3 états différents de l'API.

- Pending signifie que la manipulation de l'API est en cours.
- Fulfilled signifie que la réponse a été obtenue de l'API.
- Rejected signifie que l'appel à l'API a échoué.

```
export const deleteUser=createAsyncThunk("user/deleteUser",
async(id,thunkAPI)=>{
  const {rejectWithValue}=thunkAPI
  try{
const resp=await axios.delete(`http://localhost:3006/users/${id}`)
console.log(resp)
return log
  }catch(error)
  {
    return rejectWithValue(error.message)
  }
})
const userSlice=createSlice({
```

```

name: "user",
initialState,
reducers: {

},
extraReducers: {

  [deleteUser.pending]: (state, action) => {

    state.isLoading = true;

  },
  [deleteUser.fulfilled]: (state, action) => {

    state.value = state.value.filter((item) => item.id !== action.payload)
    state.isLoading = false;

  },
  [deleteUser.rejected]: (state, action) => {

    state.err = action.payload
    state.isLoading = false;

  }
}
})

```

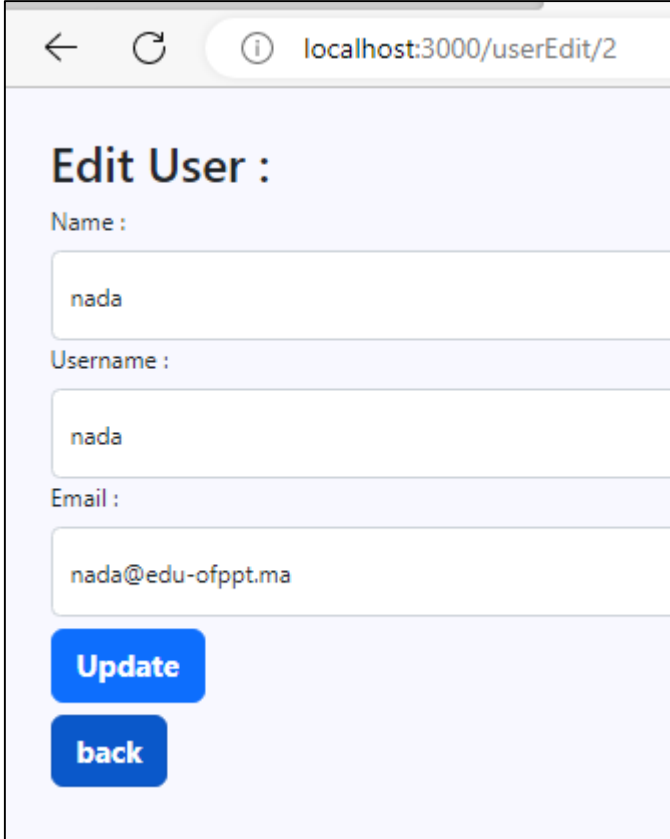
La méthode **createAsyncThunk** sera appelée avec deux arguments :

**id** : une valeur unique, contenant le premier paramètre qui a été passé au créateur d'action **thunk** lorsqu'il a été envoyé. Ceci est utile pour passer des valeurs telles que les ID des éléments qui peuvent être nécessaires dans le cadre de la requête. Si vous devez passer plusieurs valeurs, passez-les ensemble dans un objet lorsque vous distribuez le thunk, comme :

```
dispatch(deleteUser(id))
```

#### 4.    **updateUser    (pending,    fulfilled, rejected)**

En suivant la même logique on va éditer un utilisateur en utilisant la méthode **put** de l'API axios.



The screenshot shows a web browser window with the address bar displaying 'localhost:3000/userEdit/2'. The page title is 'Edit User :'. Below the title, there are three input fields labeled 'Name :', 'Username :', and 'Email :'. The 'Name' field contains 'nada', the 'Username' field contains 'nada', and the 'Email' field contains 'nada@edu-ofppt.ma'. At the bottom of the form, there are two blue buttons: 'Update' and 'back'.

On passe un paramètre id dans le **NavLink** pour qu'on puisse récupérer l'utilisateur correspondant. Et Dans le component **UserEdit** on affiche les informations par les hooks **useEffect** et **useParams**

Vous créez une variable appelée **editUser** qui est assignée à **createAsyncThunk** (remarquez le mot clé export avant de déclarer la variable).

Dans **extraReducers** (vous spécifiez 3 états probables de la promesse qui sont **pending**, **fulfilled** et **rejected**. Vous décidez ce que Redux doit faire dans ces 3 états différents de l'API.

- Pending signifie que la manipulation de l'API est en cours.
- Fulfilled signifie que la réponse a été obtenue de l'API.
- Rejected signifie que l'appel à l'API a échoué.

```
export const updateUser=createAsyncThunk("user/updateUser",
```

```

async(user, thunkAPI) => {
  const {rejectWithValue} = thunkAPI
  try {
    const resp = await axios.put(`http://localhost:3006/users/${user.id}`, user)
    return user
  } catch (error) {
    {
      return rejectWithValue(error.message)
    }
  }
})
const userSlice = createSlice({
  name: "user",
  initialState,
  reducers: {

  },
  extraReducers: {

    [updateUser.pending]: (state, action) => {

      state.isLoading = true;

    },
    [updateUser.fulfilled]: (state, action) => {

      state.value = state.value.filter((item) => item.id !== action.payload.id)
      state.value.push(action.payload)
      state.isLoading = false;

    },
    [updateUser.rejected]: (state, action) => {

      state.err = action.payload
      state.isLoading = false;

    }
  }
})

```

La méthode **createAsyncThunk** sera appelée avec deux arguments :

**user** : une valeur unique, contenant le premier paramètre qui a été passé au créateur d'action **thunk** lorsqu'il a été envoyé. Ceci est utile pour passer des valeurs telles que les ID des éléments qui peuvent être nécessaires dans le cadre de la requête. Si vous devez passer

plusieurs valeurs, passez-les ensemble dans un objet lorsque vous distribuez le thunk, comme :

```
dispatch(updateUser({name : 'Adil', username :  
'Adil ', email : 'adil456@edu-ofppt.ma'}))
```