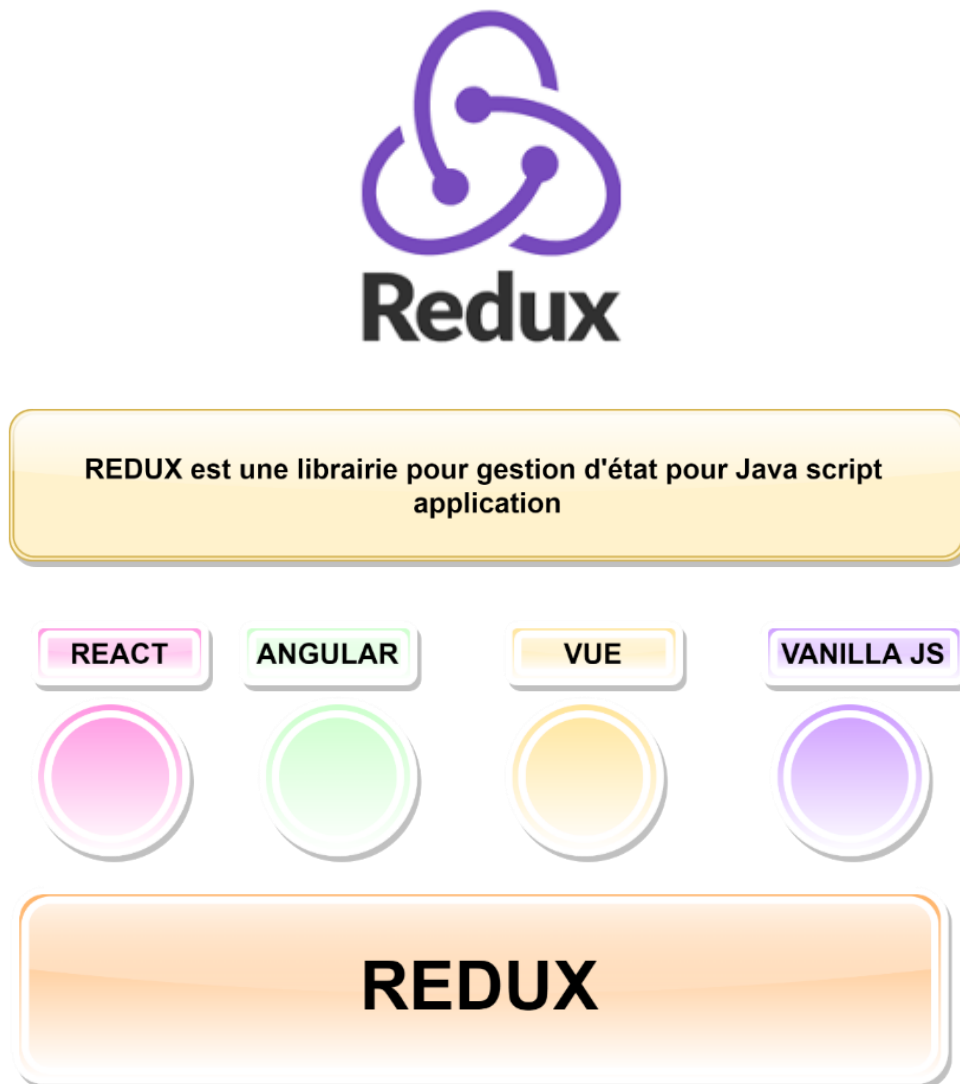


## 17.1. Manipulation des states complexes



Redux n'est pas lié à une librairie ou Framework User Interface

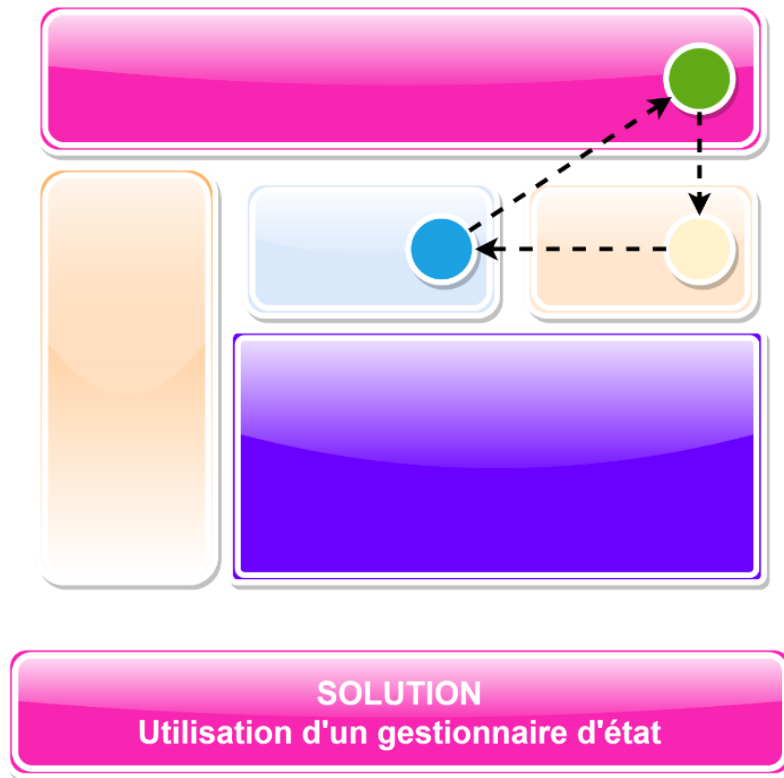
Mais pourquoi on a besoin à un gestionnaire d'état comme REDUX.

Quand on crée une application avec un complexe UI, l'application est constituée par plusieurs composants dont on a besoin de garder leurs états asynchrones, un changement d'état d'un composant pourra provoquer immédiatement la modification d'état d'autres composants.

Dans ce cas la modification des données dans un composant, déclenchera la modification d'état d'autres composants.

Dans le cas d'un scénario complexe, les données peuvent être aussi modifiées par une arrivée des données provenant d'une requête réseau API ou d'un background tâche.

Dans cette situation les données peuvent circuler d'une partie à une autre d'UI, puis les données sont changées d'une manière imprévisible, en conséquence il faut écrire beaucoup de code pour garder l'état des composants asynchrone. Quand quelque chose ne va pas, il faut comprendre d'où elles viennent les données, chose qui est très complexe à faire directement avec du code



En Redux au lieu de stocker les données sur plusieurs partie de l'UI, on stocke les données dans un centrale repository il s'agit d'un objet java script nommé store, on peut le considérer comme une petite base de données pour le front end, en conséquence pour cette architecture, les différentes partie de l'UI ne maintiendront plus leur propre état au lieu de cela, ils obtiendront ce dont ils ont besoin dans le **store**.

Si nous devons mettre à jour les données, il y a un seul endroit à mettre à jour dans le store, ce qui résout immédiatement le problème de synchronisation des données entre différentes parties de l'UI.

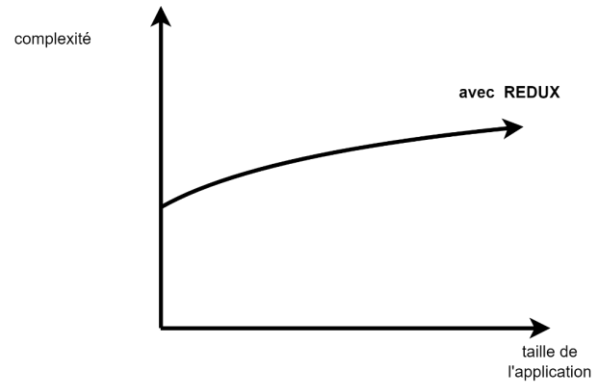
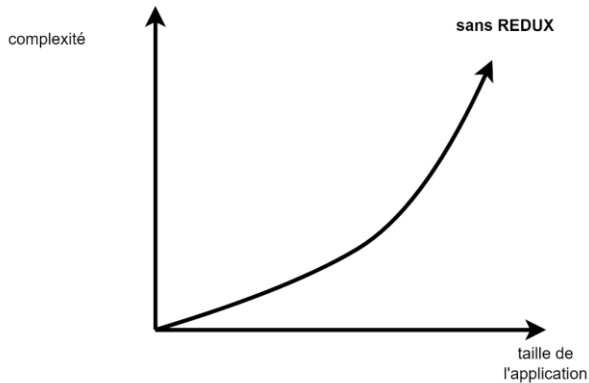
L'architecture redux permet également de comprendre facilement comment les données changent dans nos applications si quelque chose ne va pas, nous pouvons voir exactement comment les données ont changé comment pourquoi quand et où.

#### Avantages REDUX :

- ✓ Changements d'état prévisibles
- ✓ état centralisé
- ✓ Débogage facile
- ✓ conserver l'état de la page
- ✓ undo/redu
- ✓ modules complémentaires de l'écosystème angular,vue,react ,vanilla js

### Inconvénients REDUX :

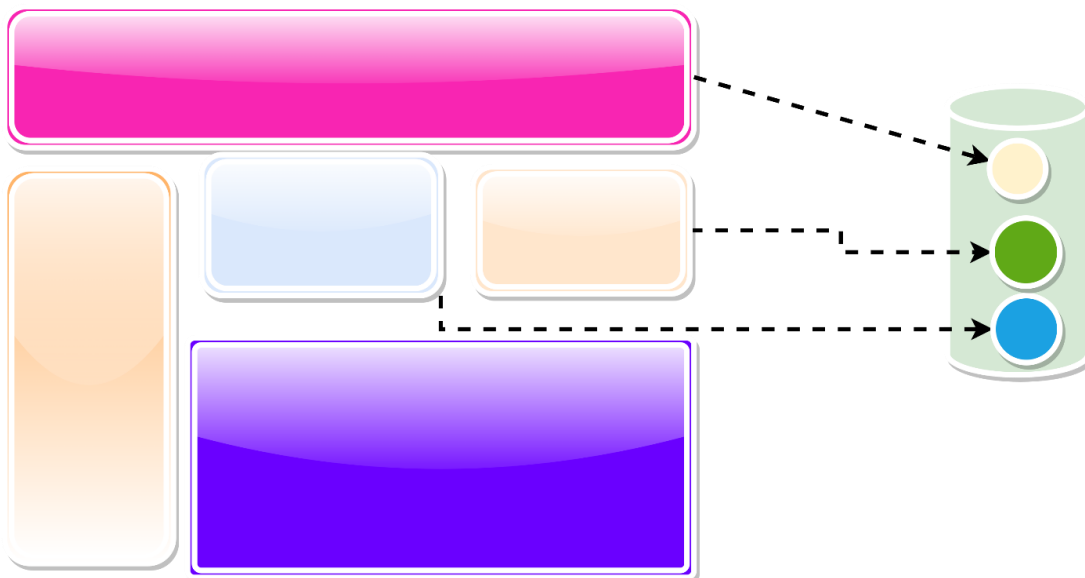
- ✓ Complexité
- ✓ Verbosité



Pour tout projet ou application REDUX, nous devons déterminer le problème qu'on doit résoudre, quelles sont les contraintes, quelle est la solution optimale pour le résoudre efficacement.

Quand on ne doit pas utiliser REDUX :

- application de petite ou moyenne taille
- budget serré
- petit flux de données d'interface utilisateur
- données statiques



L'idée de Redux est de centraliser la gestion des états dans un objet unique appelé store, qui centralise tous les états de tous les composants. Cet objet store (unique pour toute l'application) est géré par les méthodes de Redux.

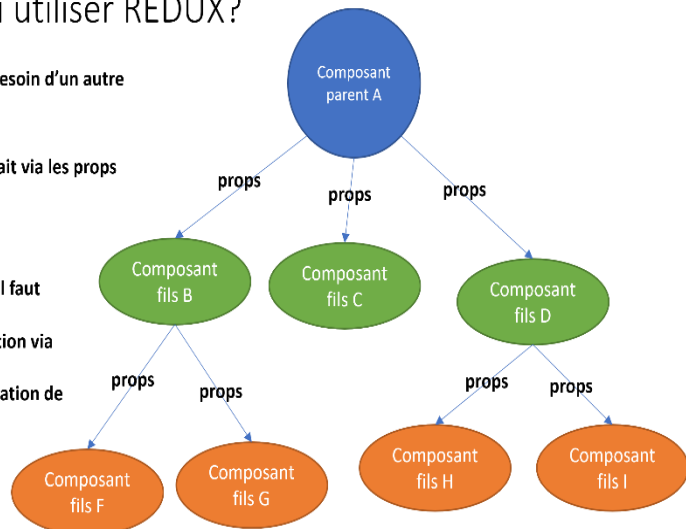
## Pourquoi utiliser REDUX?

→ Les composants ont leur propre état c'est pourquoi avons-nous besoin d'un autre outils pour aider à gérer cet état

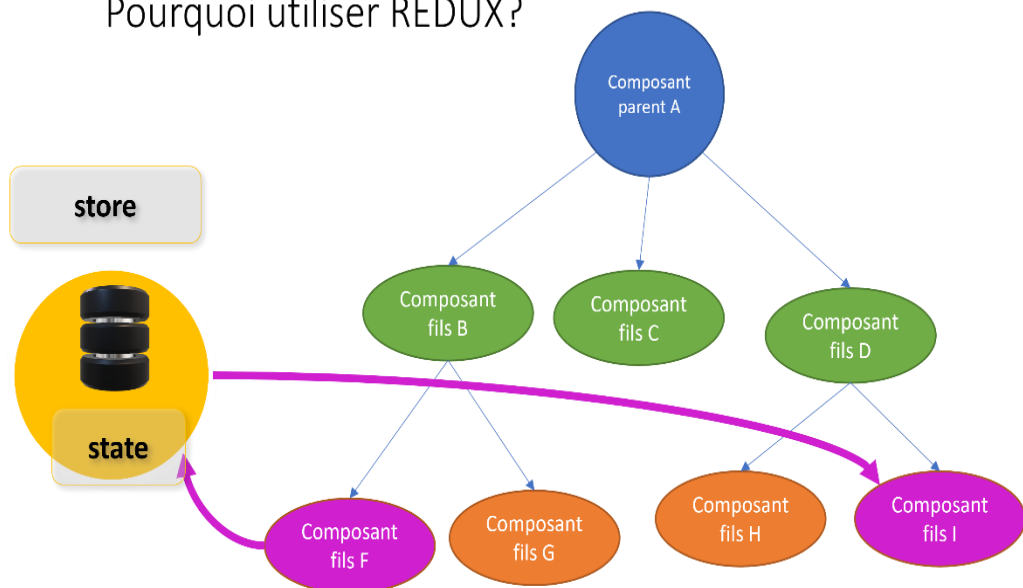
→ En react le passage d'information de père vers descendants se fait via les props

En React pour récupérer des informations F dans G par exemple il faut passer  
Un callback fonction dans le props de B→F puis passer l'information via  
pros B→G  
Figurer vous combien de travail nous faudra pour passer l'information de  
F→I

→



## Pourquoi utiliser REDUX?



Avec redux le passage d'information d'un composant à l'autre se fait d'une manière très facile et fluide

Exemple le composant F change le state de store, le composant I reçoit les données mise à jours



## 17.2. Utilisation de REDUX

### `createStore`

La fonction **`createStore(reducer)`** prend en paramètre une fonction de callback (appelée un reducer) ayant les paramètres state et action. La fonction `createStore(reducer)` retourne un objet appelé store.

La fonction de **callback** utilisée en paramètre de `createStore()` est donc appelée un reducer. On étudie ci-après cette fonction.

Les données de store ne peuvent être lu et mis à jour qu'à travers les méthodes suivantes, fournies sur l'objet store

### `store.dispatch(action)`

La méthode **`store.dispatch(action)`** qui permet d'effectuer une modification de l'état, en fonction de l'action indiquée. En effet, seules les actions permettent d'effectuer une modification de l'état. Par exemple, une action pourrait être « Ajouter cet élément dans la liste » ou « Retirer le deuxième élément de la liste ». La différence avec React est que la modification de l'état n'est pas faite directement en indiquant sa valeur (par `this.setState()`), mais plutôt en exécutant une action qui provoquera la modification de l'état (par `store.dispatch(action)`).

### `store.getState()`

La méthode **`store.getState()`** permet de récupérer sous forme d'objet l'état (stocké dans l'objet store). Cette méthode `store.getState()` ressemble au fonctionnement de l'objet `this.state` utilisé dans React, à la différence qu'elle retourne l'état de toute l'application et pas seulement celui qui est associé à un composant.

### `store.subscribe(listener)`

La méthode **`store.subscribe(listener)`** permet d'effectuer un traitement lors de chaque action. Le traitement est effectué dans la fonction de callback `listener()`.


Globalement, on voit donc que l'on a des actions qui mettent à jour l'état et que l'on peut « écouter » les éventuels changements d'état lorsque les actions sont exécutées. La mise à jour de l'état est effectuée suite aux actions « dispatchées », mais la procédure de mise à jour de l'état est centralisée dans la fonction de callback indiquée en paramètre de la fonction `createStore(reducer)`. Le reducer (méthode de la forme `reducer(state, action)`) est donc une partie importante de l'application car il permet d'indiquer comment l'état est mis à jour (en fonction de l'ancien état et de l'action effectuée).

→ **store**

REDUX

Le store contient les données de l’application donc il faut déterminer la structure des données de l’application qui doivent être dans un objet racine. Cet objet peut être appelé state  
Exemple: pour une application e-commerce  
`initialState={articles:[],utilisateur:{nom:null,isLogged:false}}`

store



state

→ Un seul store pour l’application

→ **responsabilités**

→ Un seul store pour l’application

→ Permet l’accès au state via `getState()`

→ Inscrit un ecouteur (listeners) via `subscribe(listener)`

→ Stocke l’état state de l’application

→ Permet la modification de state via `dispatch(action)`


REDUX

Redux Architecture


Action

store

reducer

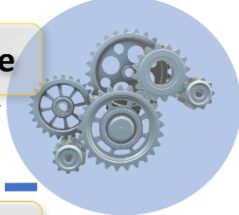


dispatch



state

Previews state



new state

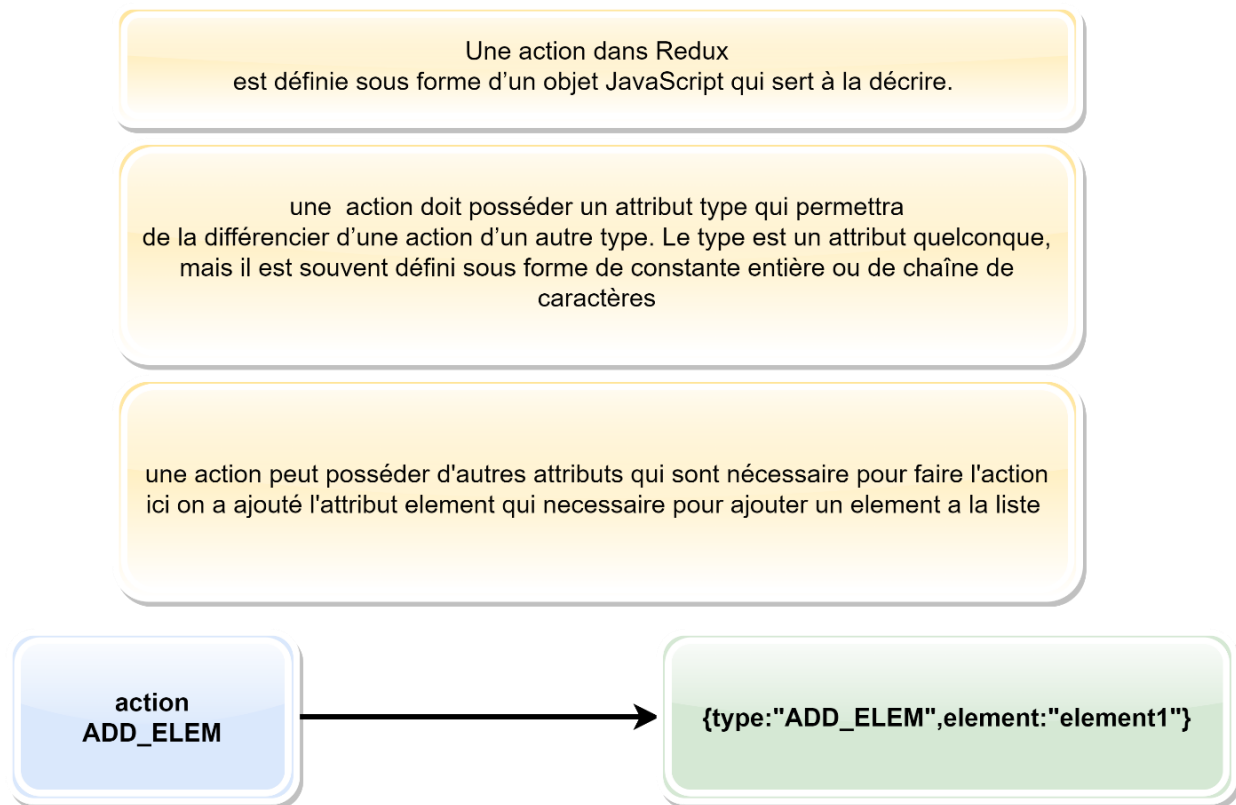
Réalisé par ABDELMOUNIM BENDAOU et NOUREDDINE ABABAR

6 / 25

## 17.3. Actions dans Redux

### Actions dans Redux

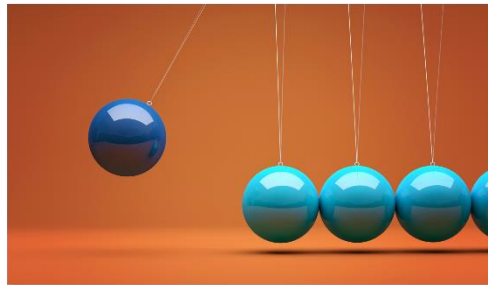
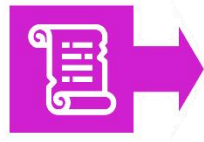
Les actions sont les éléments de l'application qui permettent de modifier l'état. Elles correspondent à tout ce qui peut produire un changement d'état de notre application. Par exemple, en supposant que notre application gère une liste d'éléments, des actions possibles pourraient être :



Une action dans Redux est définie sous forme d'un objet JavaScript qui sert à la décrire. La seule contrainte est que chaque action doit posséder un attribut type qui permettra de la différencier d'une action d'un autre type. Le type est un attribut quelconque, mais il est souvent défini sous forme de constante entière ou de chaîne de caractères

## REDUX ACTION

➔ Définir les actions



Ajouter Article au panier

Changer la quantité

Supprimer Article du panier



➔ Définir les actions



## REDUX

Ajouter Article au panier

```

const action = {
  type: "ADD_ARTICLE",
  payload: {
    id: 10,
    description: "Article1",
    prix: 200,
    quantite: 2
  }
}
  
```

Payload contient seulement les informations nécessaires pour l'exécution de l'action ADD\_ARTICLE





Définir les actions



## REDUX

Modifier la quantité d'un article

```
const action = {  
  type: "UPDATE_QUANTITE",  
  payload: {  
    id: 10,  
    quantite: 2  
  }  
}
```

Payload contient seulement le minimum d'informations pour l'exécution de l'action UPDATE\_QUANTITE



Définir les actions



## REDUX

Supprimer un article du panier

```
const action = {  
  type: "REMOVE_ARTICLE",  
  payload: { id: 10 }  
}
```

Payload contient seulement le minimum d'informations nécessaires pour l'exécution de l'action REMOVE\_ARTICLE



## Créateur d'actions dans Redux :

# REDUX

## ➔ Action créateur

un créateur d'action est une fonction qui crée et retourne une action

```
import * as actions from './actionTypes'
export function
addArticle(id,description,prix,quantite){

    return {
        type:actions.ADD_ARTICLE,
        payload:{
            id:id,
            description:description,
            prix:prix,
            quantite:quantite
        }
    }
}
```

```
export const
addArticle=(id,description,prix,quantite)=>({
    type:actions.ADD_ARTICLE,
    payload:{
        id:id,
        description:description,
        prix:prix,
        quantite:quantite}
})
```

```
store.dispatch(addArticle(14,'article4',200;2))
```

## 17.4. les reducers dans Redux

Le reducer est la fonction de callback utilisée en paramètre de la fonction createStore(reducer). Elle permet d'indiquer les changements d'états de l'application, en fonction des actions effectuées.

Son fonctionnement est le suivant : la fonction **reducer(state, action)** utilise l'état actuel de l'application, et selon l'action indiquée en paramètre, la fonction retourne un nouvel état.

Elle ne doit rien faire d'autre, et en particulier elle ne doit surtout pas :

- ➔ mettre à jour des valeurs externes (bases de données, variables, etc.) ;
- ➔ utiliser des données variables autres que celles indiquées en paramètres (state et action) ;
- ➔ modifier l'état actuel (donc on s'interdit d'effectuer une action quelconque dans le reducer, ce qui modifierait l'état).

En fait, un reducer est ce que l'on appelle une « **fonction pure** » :

Si l'on exécute plusieurs fois la fonction avec les mêmes arguments, son comportement doit toujours être identique. Et pour cela, le monde extérieur ne doit pas influencer sur son comportement.

## EXEMPLE DE REDUCER

### ➔ Création le reducer

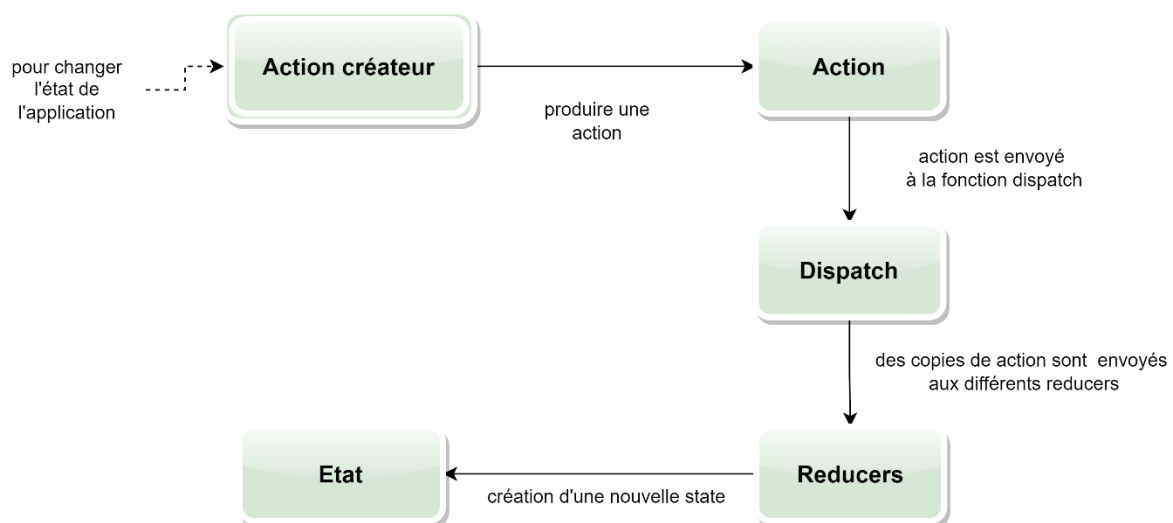


## REDUX

```
function reducer(state= initialState,action){  
  if(action.type==="ADD_ARTICLE"){  
    return [... state,action.payload]  
  }  
  if(action.type==="UPDATE_QUANTITE"){  
    const id=action.payload.id;  
    const quantite=action.payload.quantite;  
    return state.map(article=>{article.id!==action.payload.id?  
    article:{id:id,description:article.description,quantite:quantite}});  
  }  
  if(action.type==="REMOVE_ARTICLE"){  
    const id=action.payload.id  
    return state.filter(article=>article.id!==id);  
  }  
  return state;  
}
```

Reducer c'est une pure fonction qui reçoit deux arguments l'objet state et une action

## REDUX cycle



## 17.5. Application pour cours



### REACT-REDUX



Création d'une application react

**`npx create-react-app my-shop`**



Installation de redux et react-redux

**`npm install redux react-redux`**

### Explication de l'exemple :

#### Application fruiterie avec Redux

##### → Les actions



Ajouter un article dans le panier

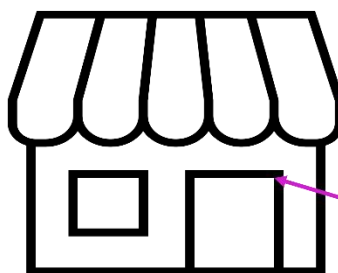


Vider panier

##### → Le store



La liste fruits



fruiterie

reducer

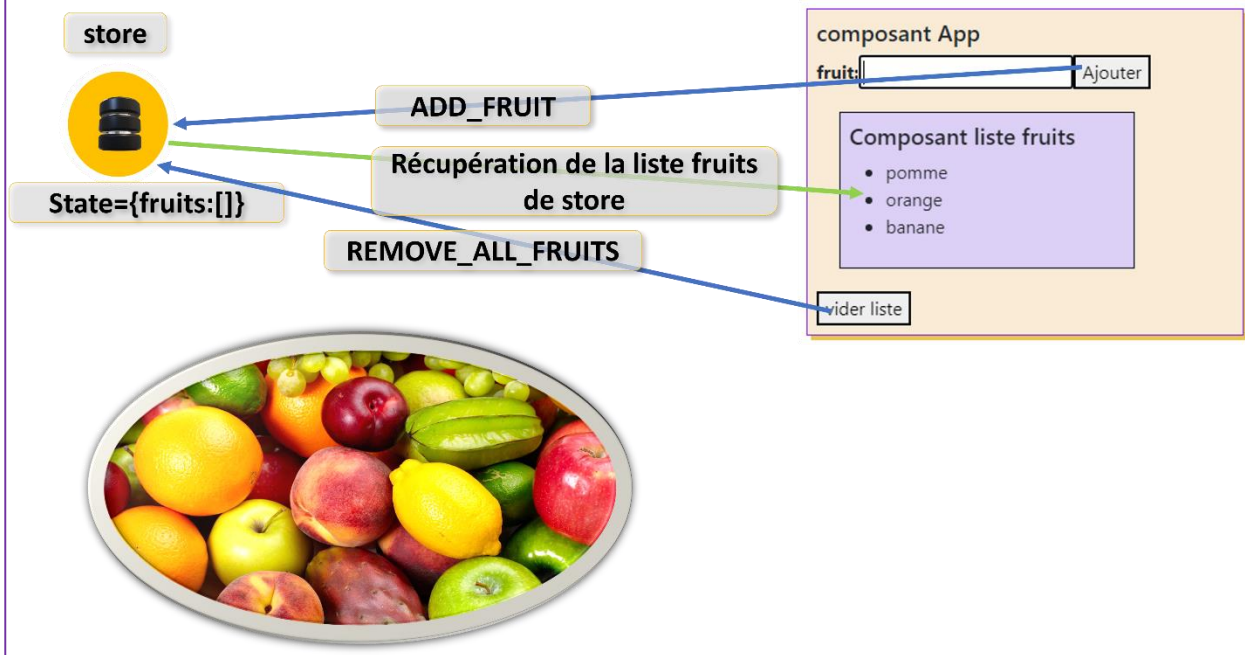
store

vendeur

Action:Achat

client

## ➔ Application fruiterie avec Redux



L'exemple est très simple, il consiste à manipuler une liste des fruits, on peut ajouter dans la liste et vider les éléments de la liste.

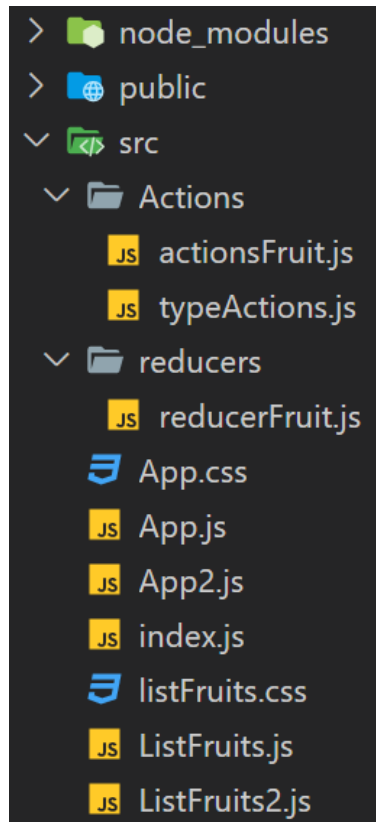
Les actions ajouter dans la liste et vider la liste se déclenchent à partir du composant App

L'affichage de la liste des fruits se fait à partir du composant ListFruits  
voir projet : [learn-redux-Cours](#) (Winrar)

Lancer **npm install** pour installer les dépendances node\_modules

## 17.6. Atelier d'application cours Redux

### 17.6.1. Redux en utilisant mapStateToProps, mapDispatchToProps



#### Préparation des types d'actions et créateur d'action :

Le fichier typeActions.js

```
export const ADD_FRUIT='ADD_FRUIT'  
export const REMOVE_ALL_FRUITS='REMOVE_ALL_FRUITS'
```

Le fichier actionsFruit.js

```
import * as type from './typeActions'  
export function addFruit(myFruit){  
  return({  
    type:type.ADD_FRUIT,  
    payload:myFruit  
  })  
}  
export function viderFruit(){  
  return({  
    type:type.REMOVE_ALL_FRUITS  
  })  
}
```



## Préparation de reducer

Fichier reducerFruit.js

```
import * as type from '../Actions/typeActions'

const initialeState={
  fruits:[]
}

export default function reducerFruits(state=initialeState,action){
  switch(action.type){
    case type.ADD_FRUIT:
      return {...state,fruits:[...state.fruits,action.payload]}

    case type.REMOVE_ALL_FRUITS:
      return {...state,fruits:[]}
    default:
      return state
  }
}
```

Fichier index.js

```
import React from "react";
import ReactDOM from 'react-dom/client'
import {Provider} from 'react-redux'
import reducerFruit from './reducers/reducerFruit'
import { legacy_createStore as createStore} from 'redux'
import App from "./App";
const root=ReactDOM.createRoot(document.getElementById("root"))
const store=createStore(reducerFruit)
root.render(
  <Provider store={store}>
    <>
      <App/>
    </>
  </Provider>
)
```

```
const store=createStore(reducerFruit)
```

createStore est désormais obsolète elle est remplacée par legacy\_createStore

comme on a vu dans le cours createStore reçoit en argument un reducer : reducerFruit

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



Avec cette instruction :

le store doit être créé via `createStore()`. Et pour que sa valeur soit transmise dans les composants par le module "react-redux", il faut créer un composant parent de tous les autres (appelé `<Provider>`), auquel la valeur du store est transmise dans la propriété `store`.  
fichier `index.js`

Le composant `Provider` enveloppe notre composant de niveau supérieur, de manière à dire chaque composant enfant de votre composant `App` aura accès au magasin.

Le fichier `App.css`

```
.container{
  margin:auto;
  margin-top:20px ;
  padding: 8px;
  width: 50%;
  border: 1px solid blueviolet;
  background-color: antiquewhite;
  box-shadow: 4px 4px rgb(232, 200, 81);
}
```

Le fichier `listFruits.css`

```
.fruits{
  margin:20px;
  padding: 8px;
  width: 260px;
  background-color: rgb(221, 208, 247);
  border:1px solid rgb(14, 19, 33)
}
```

Le fichier `ListFruits.js`

```
import {useSelector} from 'react-redux'
import {connect} from 'react-redux'
import './listFruits.css'
function ListFruits(props){
const fruits=useSelector(data=>data.fruits)
return (<div className='fruits'>
  <h5>Composant liste fruits</h5>
  <ul>
    {props.fruits.map((fruit,index)=><li key={index}>{fruit}</li>)}
  </ul>
</div>)
}
function mapStateToProps(state){
  return {fruits:state.fruits}
}
export default connect(mapStateToProps,null)(ListFruits)
```



Le fichier App.js

```
import React,{useState} from 'react'
import { connect} from 'react-redux'
import {addFruit,viderFruit} from '../Actions/actionsFruit'
import './App.css'
import ListFruits from './ListFruits'
function App(props){
  const [nomFruit,setNonFruit]=useState('')

  return(<div className='container'>
    <label htmlFor='fruit'> fruit</label><input
    onChange={(e)=>setNonFruit(e.target.value) } id='fruit' value={nomFruit} />
    <button
    onClick={()=>{props.ajouterFruit(nomFruit);setNonFruit('')}}>Ajouter</button>
    <ListFruits/>
    <button onClick={()=>props.viderList()}>vider liste</button>
  </div>)
}
function mapDispatchToProps(dispatch){
  return {
    ajouterFruit:function(myfruit){
      dispatch(addFruit(myfruit))
    },
    viderList:function(){dispatch(viderFruit())}
  }
}
export default connect(null,mapDispatchToProps)(App)
```

Le module "react-redux" comporte principalement la méthode connect(), qui permet l'accès au store :

- pour lire l'état ;
- pour déclencher les actions qui le mettent à jour.

La lecture de l'état ne se fera plus par store.getState() mais par des propriétés qui seront ajoutées au composant (c'est la méthode connect() qui les ajoute pour nous).

Par exemple, si dans l'état de Redux on a indiqué l'attribut fruits, on pourra également accéder à la propriété fruits dans un composant React (et ceci bien que cette propriété fruits n'ait jamais été créée par nous dans le composant) tous les composants fils de App peuvent y accéder.

Le déclenchement des actions de Redux suit le même principe. Des propriétés (correspondant aux actions) seront ajoutées au composant par la méthode connect(), qui permettront d'utiliser ces actions dans le composant React sans passer par le store (c'est la méthode connect() qui rend ce processus invisible). Par exemple, si dans les actions de Redux on a indiqué l'action addFruit(myFruit) qui permet d'insérer un fruit dans la liste, on pourra accéder à la propriété addFruit dans un composant React (et ceci bien que nous n'ayons jamais créé la propriété add\_inscription dans le composant).



### Utilisation de connect()

La méthode `connect(mapStateToProps, mapDispatchToProps)` retourne une nouvelle fonction, cette fonction prend en paramètre un nom de composant (ici appelé `App`). Le composant indiqué en paramètre est transformé selon les règles de correspondance indiquées dans les fonctions `mapStateToProps(state)` et `mapDispatchToProps(dispatch)`.

C'est ce composant transformé qui est retourné par la méthode `connect()(App)`. Le composant indiqué en paramètre est soit un nom de fonction, soit un nom de classe (comme on sait le faire pour créer un composant React).

La transformation du composant concerne l'ajout de nouvelles propriétés dans ce composant, ce qui lui permet d'accéder à l'état et aux actions de Redux.

### La fonction `mapStateToProps(state)`

Se trouve dans `ListFruits.js`

```
function mapStateToProps(state){  
  return ({fruits:state.fruits})  
}
```

La fonction `mapStateToProps(state)` est passée en premier argument de la méthode `connect()`.

```
export default connect(mapStateToProps,null)(ListFruits)
```

Le deuxième argument est nul car dans le composant `ListeFruits` on veut seulement récupérer le state

Elle est utilisée dans le cas où l'on souhaite permettre au composant d'accéder en lecture à l'état (ou à une partie de celui-ci) de Redux.

D'où dans le cas où le composant ne veut pas accéder en lecture à l'état de Redux, on met `null` dans le premier paramètre de la méthode `connect()`.

Dans notre exemple la propriété `fruits` est maintenant définie dans le composant `ListeFruits`, elle est donc accessible par l'objet props disponible dans celui-ci.

Dans `ListFruits.js`

```
<ul>  
  {props.fruits.map((fruit,index)=><li key={index}>{fruit}</li>)}  
</ul>
```

### La fonction `mapDispatchToProps(dispatch)`

Se trouve dans `App.js`

`mapDispatchToProps(dispatch)` est une fonction permettant de rendre les actions de Redux accessible dans le composant, sous forme de propriétés (ici des méthodes accessibles à travers l'objet props).

Ces actions permettent de mettre à jour l'état de Redux (alors que la fonction `mapStateToProps()` vue précédemment permet de rendre l'état de Redux accessible en lecture seulement).



La fonction `mapDispatchToProps()` s'inscrit en deuxième paramètre de la méthode `connect()`. Elle retourne un objet indiquant les nouvelles propriétés qui seront disponibles dans le composant, liées aux **actions de Redux**.

```
export default connect(null,mapDispatchToProps)(App)
```

Par exemple, si l'action **addFruit (myFruit)** existe dans les actions de Redux, il est possible pour un composant d'accéder à cette action en écrivant la fonction `mapDispatchToProps(dispatch)` suivante.

Extrait du code fichier `index.js`

```
import {addFruit,viderFruit} from './Actions/actionsFruit'
```

```
function mapDispatchToProps(dispatch){  
  return {  
    ajouterFruit:function(myfruit){  
      dispatch(addFruit(myfruit))  
    },  
    viderList:function(){dispatch(viderFruit())}  
  }  
}
```

Désormais la propriété **ajouterFruit** et **viderList** sont maintenant définie dans le composant `App`, elles sont donc accessibles par l'objet `props` disponible dans celui-ci.

```
<button onClick={()=>props.viderList()}>vider liste</button>
```



### 17.6.2. En utilisant les hooks useSelector et useDispatch

#### useSelector :

Le sélecteur est approximativement équivalent à l'argument mapStateToProps pour se connecter conceptuellement. Le sélecteur sera appelé avec l'état complet du magasin Redux comme seul argument. Le sélecteur sera exécuté chaque fois que le composant de fonction sera rendu (à moins que sa référence n'ait pas changé depuis un rendu précédent du composant afin qu'un résultat mis en cache puisse être renvoyé par le hook sans réexécuter le sélecteur).

useSelector() s'abonnera également au store Redux et exécutera votre sélecteur chaque fois qu'une action est envoyée.

#### useDispatch :

Ce hook renvoie une référence à la fonction dispatch du store Redux. Vous pouvez l'utiliser pour répartir les actions nécessaires.

Pour ce faire on change seulement le fichier index.js et on créera d'autres composants

ListeFruits2.js et App2.js

App2.js

```
import React from "react";
import ReactDOM from 'react-dom/client'
import {Provider} from 'react-redux'
import reducerFruit from './reducers/reducerFruit'
import { legacy_createStore as createStore} from 'redux'
import App2 from "./App2";
const root=ReactDOM.createRoot(document.getElementById("root"))
const store=createStore(reducerFruit)
root.render(
  <Provider store={store}>
    <>
      <App2/>
    </>
  </Provider>
)
```

Fichier ListFruits2.js

```
import {useSelector} from 'react-redux'
import './listFruits.css'
export default function ListFruits2(){
  const fruits=useSelector(data=>data.fruits)

  return (<div className='fruits'>
    <h5>Composant liste fruits 2</h5>
    <ul>
      {fruits.map((fruit,index)=><li key={index}>{fruit}</li>)}
    </ul>
  </div>)
}
```



le hook useSelector permet d'accéder à l'attribut fruits du store en en passant un arrow function comme argument .

```
const fruits=useSelector(data=>data.fruits)
```

chaque modification sur fruits provoque un re-render de notre composant

pour vérifier ajouter le code suivant dans la composant ListeFruits2.js

```
useEffect(()=>console.log('la liste est mise à jour'))
```

Fichier App2.js

```
import React,{useState} from 'react'
import { useDispatch } from 'react-redux'
import {addFruit,viderFruit} from './Actions/actionsFruit'
import './App.css'
import ListFruits2 from './ListFruits2'
export default function App2(props){
  const dispatch=useDispatch()
  const [nomFruit,setNonFruit]=useState('')
  return(<div className='container'>
    <h5>composant App 2</h5>
    <label htmlFor='fruit'> <b>fruit: </b></label><input
onChange={e=>setNonFruit(e.target.value)} id='fruit' value={nomFruit} />
    <button
onClick={()=>{dispatch(addFruit(nomFruit));setNonFruit('')}}>Ajouter</button>
    <ListFruits2/>
    <button onClick={()=>dispatch(viderFruit())}>vider liste</button>
  </div>
)
```

La constante dispatch permet de dispatcher facilement une action

```
<button onClick={()=>dispatch(viderFruit())}>vider liste</button>
```

## Vous pouvez utiliser le projet utilisé dans le cours

projet-dev-tools

lancer npm install

pour installer toutes les dépendances



## 17.7. Manipulation des évènements asynchrones

Dans les séances précédentes on a utilisé des données Redux, nous avons vu comment utiliser plusieurs éléments de données du store Redux dans les composants React, personnaliser le contenu des objets d'action avant qu'ils ne soient distribués et gérer une logique de mise à jour plus complexe dans nos réducteurs.

Jusqu'à présent, toutes les données avec lesquelles nous avons travaillé se trouvaient directement dans notre application client React. Cependant, la plupart des applications réelles doivent fonctionner avec les données d'un serveur, en effectuant des appels d'API HTTP pour récupérer et enregistrer des éléments.

Dans cette section, nous allons convertir notre application de médias sociaux pour récupérer les publications et les données des utilisateurs à partir d'une API, et ajouter de nouvelles publications en les enregistrant dans l'API.

### Utilisation de middleware thunk pour activer asynchrone logique

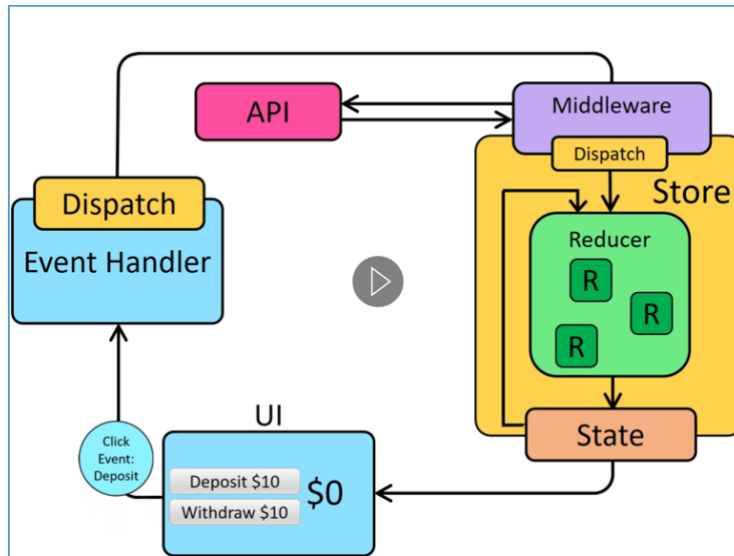
En soi, un store Redux ne sait rien de la logique asynchrone. Il sait seulement comment envoyer des actions de manière synchrone, mettre à jour l'état en appelant la fonction de réduction racine et informer l'interface utilisateur que quelque chose a changé. Toute synchronicité doit se produire en dehors du magasin.

Mais que se passe-t-il si vous souhaitez que la logique asynchrone interagisse avec le magasin en répartissant ou en vérifiant l'état actuel du magasin ? C'est là qu'interviennent les middleware Redux. Ils étendent le magasin et vous permettent de :

- Exécuter une logique supplémentaire lorsqu'une action est envoyée (telle que la journalisation de l'action et de l'état)
- Mettre en pause, modifier, retarder, remplacer ou arrêter les actions envoyées
- Écrivez du code supplémentaire qui a accès à dispatch et getState
- Enseigner au dispatch comment accepter d'autres valeurs en plus des objets d'action simples, tels que les fonctions et les promesses, en les interceptant et en envoyant des objets d'action réels à la place

La raison la plus courante d'utiliser un middleware est de permettre à différents types de logique asynchrone d'interagir avec le store. Cela vous permet d'écrire du code qui peut envoyer des actions et vérifier l'état du magasin, tout en gardant cette logique séparée de votre interface utilisateur.

Il existe de nombreux types de middleware asynchrones pour Redux, et chacun vous permet d'écrire votre logique en utilisant une syntaxe différente. Le middleware asynchrone le plus courant est redux-thunk, qui vous permet d'écrire des fonctions simples pouvant contenir directement une logique asynchrone. La fonction configureStore de Redux Toolkit configure automatiquement le middleware thunk par défaut, et nous recommandons d'utiliser les thunks comme approche standard pour écrire une logique asynchrone avec Redux.



## Préparation du projet

`npm install --save-dev redux-devtools-extension`

`npm install redux-thunk`

## Exemple

L'appel d'une action asynchrone nécessite l'utilisation de middleware thunk pour cela au niveau de création du store on doit ajouter le middleware thunk

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import {Provider} from 'react-redux'

import reducer from './Counter/CounterStore';
import {legacy_createStore as createStore} from 'redux'
import { applyMiddleware } from 'redux';
import { composeWithDevTools } from 'redux-devtools-extension';
import thunk from 'redux-thunk';
import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
const store=createStore(reducer,
  composeWithDevTools(applyMiddleware(thunk))
)
root.render(
  <Provider store={store}>
    <React.StrictMode>
      <App />
    </React.StrictMode>
  </Provider>
);
```

Le code suivant :

```
const store=createStore(reducer,
  composeWithDevTools(applyMiddleware(thunk))
)
```

Permet d'ajouter le middleware thunk et devTools ajoutant maintenant une action asynchrone : **asyncIncrementerByAmountIfOdd** a notre projet :

```
const asyncIncrementerByAmountIfOdd = (amount) => {
  return (dispatch, getState) => {
    const stateBefore = getState()
    console.log(`Compteur avant: ${stateBefore.value}`)
    if(stateBefore.value%2===1){
      dispatch(incrementerByAmount(amount))
      const stateAfter = getState()
      console.log(`Compteur après: ${stateAfter.value}`)}
    else{console.log("pas d'incrementation car !!!!!")}
  }
}
```

Pour dispatcher l'action asynchrone : **asyncIncrementerByAmountIfOdd**

```
<button
  className='button asyncButton'
  onClick={() => dispatch(asyncIncrementerByAmountIfOdd(incrementValue))} >
  Async Add Amount si compteur est impair
</button>
```

ajoutant maintenant une autre Action Asynchrone qui fait l'appel à une promise soit :

```
function fetchCount(amount = 1) {
  return new Promise((resolve) =>
    setTimeout(() => resolve({ data: amount }), 1200)
  );
}
```

qui retourne une promise en utilisant un setTimeout de 1.2 seconds

soit l'action asynchrone suivante : **asyncIncrementerByAmount**

```
const asyncIncrementerByAmount = amount => {
  return async (dispatch, getState) => {
    console.log('compteur avant changements:',getState().value)
    const response= await fetchCount(amount)
    dispatch(incrementerByAmount(response.data))
    console.log('compteur est changé après (1.2second) :',getState().value)
  }
}
```





Pour dispatcher cette action :

```
<button
  className=' button asyncButton'
  onClick={() => dispatch(asyncIncrementerByAmount(incrementValue))}
>
  Async Add Amount
</button>
```

Voir le projet complet : projet-AsyncAction

Lancer npm install pour installer les dépendances node\_modules