Intro

Principe: Redux est un outil similaire au context Api de React, il sert des gérér des states globaux a toute l'application

Context API VS Redux: Redux va être plus utiliser sur des grosses application avec état qui ont changements complex et/ou des changements nombreux. Cela permet d'avoir un code plus clean/simple.

Context API	Redux
Built-in tool that ships with React	Additional installation Required, driving up the final bundle size
Requires minimal Setup	Requires extensive setup to integrate it with a React Application
Specifically designed for static data, that is not often refreshed or updated	Works like a charm with both static and dynamic data
Adding new contexts requires creation from scratch	Easily extendible due to the ease of adding new data/actions after the initial setup
Debugging can be hard in highly nested React Component Structure even with Dev Tool	Incredibly powerful Redux Dev Tools to ease debugging
UI logic and State Management Logic are in the same component	Better code organization with separate UI logic and State Management Logic

Api

Principe: Redux de base possède 6 fonctions dont 5 helpers (like lodash).

Compose: Prends 3 paramètre qui sont des functions et qui seront exécuté du dernier au premier. Return une fonction qui prend un paramètre et appele avec dans l'odre précédent avec ce paramètre

```
const smallString = (string) => string.toLowerCase()
const reapeat3Times = (string) => string.reapeat(3)
const boldString = (string) => string.bold()

const createSmallReaptedBoldString = compose(boldString, reapeat3Times, boldString)

const theNewSmallReapeatedBoldString = createSmallReaptedBoldString("HELLO")
// result: "**hellohellohello**" Pour dire que c bold enfaite
```

bindActionCreatores: Prend un object qui prend en propriété(: action creator) avec des actions creator et les binds avec le second paramètre qui sera la fonction store.dispatch. Retourne un object avec les actions creator bind avec le store souhaiter. Ce qui permet de les appeler directement par l'objet retourner

```
const addTask = (task) => { type: "ADD_TASK": payload: task }

const changePassword = (newPass) => { type: "CHANGE_PASSWORD": payload: newPass }


const store = creatore(reducer) //A random reducer

const actionStore = bindActionsCreator(addTask, changePassword, store.dispatch)

//TO USE

actionStore.addTask({name: "Leanr Redux"}) // Trigger "ADD_TASK"
action.changePassword("theBestPassword") // Trigger "CHANGE_PAS"
```

CreateStore

createStore: Prend un reducer et retoune un store, un object avec plusieurs propriété afin de gérer les states en second paramètre on prend la valeur initial du reducer (deprécié)
Init: Lors de l'initisation redux call un dispatch "INIT" pour recupere les state initiaux

Reducer: une fonction qui prend l'état initial/état avant action et une action et retourne le nouvelle selon action donnée. Il interpretra cette objet(dispatch) avec les méthode switch case en JS

```
/*
        action: un dispatch de forme {type: ..., ...}
*/
const initialState = [
        {
                name: "I'm a task"
                id: 4
        }
]
const taskReducer = (tasks = initalState, action) => {
        switch(action.type) {
                case "ADD_TASK":
                        return [...tasks, action.payload]
                case "DELELTE_TASK":
                        return tasks.map(tasks => task.id !== action.taskId)
                        //Voir si on peut mettre les propriétés qu'on veut
                default:
                //Obliger de mettre un default
                return tasks
        }

}

const store = createStore(taskReducer) //Le state initial est initalState

//Pour Trigger un changement
store.dispatch({type:"ADD_TASK", payload: {name:"The task from the dispatch"}})

console.log(store.getState())
// [{name: "I'm a task"}, {name:"The task from the dispatch"}]
```

combineReducer: Permet de combiner de différer reducer qui géra chacun une proprité de l'objet.

```
const initalState = {
        tasks: [
                {name: "I'm a task"}
        ]
        userInfo: {
                name:"TheBestUser",
```

```
                password:"TheBestPassword",
                mail:"TheBestMail@gmail.com"
        }
}
const taskReducer = (tasks = initalState = initialState.tasks, action) => {
        switch(action.type) {
                case "ADD_TASK":
                        return [...tasks, action.payload]
                case "DELELTE_TASK":
                        return tasks.map(tasks => task.id !== action.taskId)
                        //Voir si on peut mettre les propriétés qu'on veut
                default:
                //Obliger de mettre un default
                return tasks
        }

const userInfoReducer = (userInfo = initialState.userInfo, action) => {
        switch(action.type) {
                case "CHANGE_INFO":
                        return {...userInfo, [action.info]: action.payload}
                default:
                        return userInfo
        }
}

const rootReducer = combineReducer({tasks: taskReducer, userInfo: userInfoReducer})

const store = createStore(rootReducer)
console.log(store.createStore())
//The initalState is what we expected thanks to combine reduce

store.dispatch({type:"ADD_TASK", payload: {name:"New Task"}})
store.disptach({
type:"CHANGE_INFO",
proprety:"password",
payload:"goodPasword"
})

console.log(store.createStore())
/** Result
{
        tasks: [
                {name: "I'm a task"}
                {name: "New Task"}
        ]
        userInfo: {
                name:"TheBestUser",
                password:"goodPassword",
                mail:"TheBestMail@gmail.com"
        }
}
*/
```

Enhancer: Cela permet de gérér des feature en plus comme des plugin, gérér des perfomance. C'est le troisième paramètre de createStore. Après les deux appels on retourne simplement un nouveau store avec notre nouveau reducer qui sera parent à celui de qui gère les states.
Condition (bon fonctionnement): Retourner une fonctionne, appeler le reducer durant le process
Second argurment: Si on passe en second argurment une fonction redux le definera comme si c'était un enhancer
Autrement Dit: Un enchancer c'est juste un reducer parent qui ne change pas l'été (l'enfant le gère) juste intrerprète les infos.

```
const reducer = (state = 0, action) => {
        switch (action.type) {
                case "INCREMENT":
                        return state + 1;

                case "DECREMENET":
                         return state - 1;
                default:
                        return state;
        }
```

```
};


//
const timeLogEnchancer = (createStore) => (reducer, initState) => {
        const timeLogReducer = (state, action) => {
                const prevState = state;

                console.log(`beforeChange ${prevState}`);
                const newState = reducer(state, action); //Call reducer
                console.log(`afterChange ${newState}`);

                return newState;
        };



return createStore(timeLogReducer, initState);

};



const store = createStore(reducer, 0, timeLogEnchancer);



store.dispatch({ type: "INCREMENT" });
```

applyMiddlewere: C'est une abstraction des enhancer qui permet plus facilement d'avoir plusieurs enhancer maintenant pour cela on créer des middlewere function.
Technique: La fonction reproduit le principe d'un enhancer (createStore) ⇒ {reducer, initState} et retourne un store avec un dispatch qui sera la composition de tous les middleware qu'on a mis, en attente du troisième call (param: action)

```
const reducer = (state = 0, action) => {
        switch (action.type) {
                case "INCREMENT":
                        return state + 1;

                case "DECREMENET":
                         return state - 1;
                default:
                        return state;
        }

};

//next reprénsente la fonction store.dispatch
//action représente le disptacher
const logMiddlewere = (store) => (next) => (action)  => {
        console.log("old State", store.getState());
        next(action) // Pour exécuter les changements
        console.log("The New State", store.getState())
}

const perfMiddleware = (store) => (next) => (action) => {
        const start = performance.now()
        next(action)
        const end = performance.now()

        const diff = end - start
        console.log("perf is", diff)
}
//La fonction applyMiddlewere va compose elle même tous ses argurments
//Beaucoup plus simple qu'avec les enhancers

const store = createStore(reducer, applyMiddlewere(logMiddlewere, perfMiddlewere))
```

Store

Dispatch: prend un object avec la propriété type qui donne le *type* d'action donnée.
Payload? pour des info plus (any) pour le futur état. Et appele la fonction reducer donné avec l'état avant l'action et l'objet en second agurments pour le reducer.
Action Creator(Maintenable Code): Une fonction qui créer un dipastch très utile lors qu'on utilise un dipatch a plein d'endroit diff du code cela permet de rajouter une feature plus facilement au lieu de modif tous les dispatch.
CONSTANT (Maintenable Code): pour décrire les type des dispatch on utilise des variable(const) en lettre capitale pour éviter des fautes de frappe
Subscriber: prend en argurment un callback qui sera appeler à chaque fois le state change

```
//Disptach Form
type dipsatch = {action: string, payload?: any} /// ...

//Subscribe
const store = createStore(reducer);
store.subcribe(() => console.log("SUBSCRIBE"))

store.disptach({type:"ADD_TASK", payload: {name: "new_Task"}}) // Log Subscribe
store.disptach({type: "A not handle Type"}) // Log Subscribe

//Subscribe by default trigger after the change of the state
```

React-Redux

Hooks

useSelector: fonction qui prend un callback avec le state en paramètre en retourne le state ou une partie

```
const Component = () => {
        const state = useSelector(state => state)

        return <h1>${state} from useSelector hooks</h1>
}
```

useDispatch: retourne la fonction dispatch

```
const Component = () => {
        const dispatch = useDispatch

        return <button onClick={dispatch({type:"ADD_ITEM"})}>Add </button>
}
```

Connect Api

Connect: Permet de connecter les états, dipatch avec nos composants. c'est une fonction currying qui prend au premier appel mapStateToProps et mapStateToDispatch et au second appel le composant.
Container: Tous le buiness se passe dans un fichier ComponentContainer qui va gérer tous la logique pour les état et se conecter avec le composant voulue

mapStateToProps: prend en paramètre l'état et retourne sa forme voulue
mapDispatchToProps: prend paramètre la fonction dispatch et retourner un objet avec en propriétés les noms des props voulue et en valeur une fonction qui dispatch.
Le second paramèter: le second paramètre sera les props de composant.

```
//MenuItemContainer.js

const mapDispatchToProps = (dispatch, ownProps) => ({
        remove: () => dispatch(deleteItem(ownProps.uuid)),
        updatePrice: (price) => dispatch(updatePrice(ownProps.uuid, price)),
        updateQuantity: (quantity) =>
        dispatch(updateQuantity(ownProps.uuid, quantity))
```

```
});



const mapStateToProps = (state, ownProps) => ({
        total: ownProps.price * ownProps.quantity
});



export const MenuItemContainer =
          connect(mapStateToProps,mapDispatchToProps)(MenuItem);
```

Redux ToolKit