

# Redux



# Définition



- Redux est un 'state container' pour les applis javascript
- Redux Toolkit: approche officielle recommandé pour écrire la logique Redux:
  - Il entoure le noyau Redux et contient des packages et des fonctions essentielles pour créer une application Redux
  - Il s'appuie sur de bonnes pratiques suggérées par Redux, simplifie la plupart des tâches , **évite les erreurs courantes** et facilite l'écriture d'applications Redux.

# Logique Redux



- L'ensemble de l'état global de l'app. est stocké dans une arborescence d'objets à l'intérieur d'un seul '*store*'. La seule façon de modifier l'arborescence des états est de créer une action, un objet décrivant ce qui s'est passé, et de l'envoyer au '*store*'.



# Vocabulaire Redux

- **Thermes important à connaître pour l'utilisation de Redux:** (les termes ci-dessous seront en anglais pour utiliser le vocabulaire Redux)

- *Actions*
- *Action creator*
- *Reducer*
- *Store*
- *Dispatch*
- *Selectors*



# Actions (les 'actions object')

- Une action est un simple **objet JavaScript** doté d'un **champ de type**
- Une action est **un événement décrivant quelque chose qui s'est produit** dans l'application
- Le champ type doit être **une chaîne qui donne à cette action un nom descriptif**, comme "todos/todoAdded" (Nous écrivons généralement cette chaîne de type comme "domain(eventName")

```
const addTodoAction = {  
  type: 'todos/todoAdded',  
  payload: 'Buy milk'  
}
```

fonctionnalité ou la  
catégorie à laquelle  
appartient cette action

événement spécifique  
qui s'est produit

# Action creator



- Un *créateur d'action* est une fonction qui crée et renvoie un objet d'action. Nous les utilisons généralement pour ne pas avoir à écrire l'objet d'action à la main à chaque fois :

```
const addTodo = text => {
  return {
    type: 'todos/todoAdded',
    payload: text
  }
}
```



# Reducers

- Un réducteur **est une fonction**
  - qui reçoit le '*state*' actuel et un objet d'action
  - décide si et comment mettre à jour l'état
  - renvoie le nouvel état :  $(state, action) \Rightarrow newState$ .
- On peut considérer un *reducer* comme un écouteur d'événements ('*event listener*') qui gère les états en fonction du type d'action (événement) reçu.

 **INFO**

"Reducer" functions get their name because they're similar to the kind of callback function you pass to the `Array.reduce()` method.

# Reducers



- Les reducers doivent **obligatoirement** suivre les règles suivantes:
  - Ils ne doivent calculer la nouvelle valeur d'état qu'en fonction des arguments 'state' et 'action'
  - **Ils ne sont pas autorisés à modifier le 'state' existant.** Au lieu de cela, ils doivent effectuer des mises à jour **immuables\***, en **copiant le state existant et en modifiant les valeurs copiées.**
  - Ils ne doivent pas faire de logique asynchrone, calculer des valeurs aléatoires ou provoquer d'autres «side effects\*\* »

\*Un objet **immutable**, en programmation orientée objet et fonctionnelle, est un objet dont l'état ne peut pas être modifié après sa création.

\*\* Les side effects dans React sont des actions qui ont lieu dans un composant en dehors du cycle de rendu normal et sont souvent utilisés pour gérer des tâches asynchrones ou des interactions avec le monde extérieur au composant. En react, les "side effects" sont souvent gérés par le hook 'useEffect'



# Reducers: la logique

- La logique à l'intérieur des fonctions du *reducter* suit généralement la même série d'étapes :
- Vérifiez si le *reducer* se soucie de cette action
  - Si tel est le cas, faites une copie de l'état, mettez à jour la copie avec de nouvelles valeurs et renvoyez-la
  - Sinon, renvoyez l'état existant inchangé



# Reducers: la logique

- Exemple:

```
const initialState = { value: 0 }

function counterReducer(state = initialState, action) {
  // Check to see if the reducer cares about this action
  if (action.type === 'counter/increment') {
    // If so, make a copy of `state`
    return {
      ...state,
      // and update the copy with the new value
      value: state.value + 1
    }
  }
  // otherwise return the existing state unchanged
  return state
}
```



# Store

- L'état actuel de l'application Redux réside dans un objet appelé '*store*'
- Le *store* est créé en passant dans un '*reducer*'
- Le *store* possède une méthode appelée *getState* qui renvoie la valeur de l'état actuel

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

const store = configureStore({ reducer:
  counterReducer })

console.log(store.getState())
// {value: 0}
```

# Dispatch

- Le 'store' Redux a une méthode appelée '*dispatch*'.
- **La seule façon de mettre à jour l'état est d'appeler store.dispatch() et de transmettre un objet 'action'**  
=> Le magasin exécutera sa fonction '*reducer*' et enregistrera la nouvelle valeur '*state*'. Ainsi, nous pourrons utiliser '*getState()*' pour récupérer la valeur mise à jour

```
store.dispatch({ type: 'counter/increment' })  
  
console.log(store.getState())  
// {value: 1}
```

- **On peut considérer la répartition des actions comme un "triggering an event" (« déclenchement d'un événement » en fr) dans l'application.**
  - Les '*reducers*' agissent comme des auditeurs d'événements ('*event listener*', et lorsqu'ils entendent une action qui les intéresse, ils mettent à jour l'état en réponse.



# Dispatch

- On appelle généralement les créateurs d'actions pour qu'ils envoient la bonne action :

```
const increment = () => {
  return {
    type: 'counter/increment'
  }
}

store.dispatch(increment())

console.log(store.getState())
// {value: 2}
```



# Selectors

- Les *sélectors* sont des fonctions qui savent extraire des informations spécifiques, à partir de la valeur d'un '*state*' d'*un store*.
- A mesure qu'une application grandit, cela permet d'éviter de répéter la logique, car différentes parties de l'application lisent les même données



# Flux de données

## Configuration initiale:

- Le store est créé grâce à une fonction '*root reducer*' (-> Fonction '*reducer*' racine)
- Le '*store*' appelle le '*root reducer*' une fois et enregistre la valeur de retour comme '*state*' initial
- Lorsque l'interface utilisateur est rendue pour la première fois, les composants de l'interface utilisateur accèdent au *state* actuel du *store* Redux et utilisent ces données pour décider quoi restituer. Ils s'abonnent également à toutes les futures mises à jour du magasin afin de savoir si le '*state*' a changé

# Flux de données

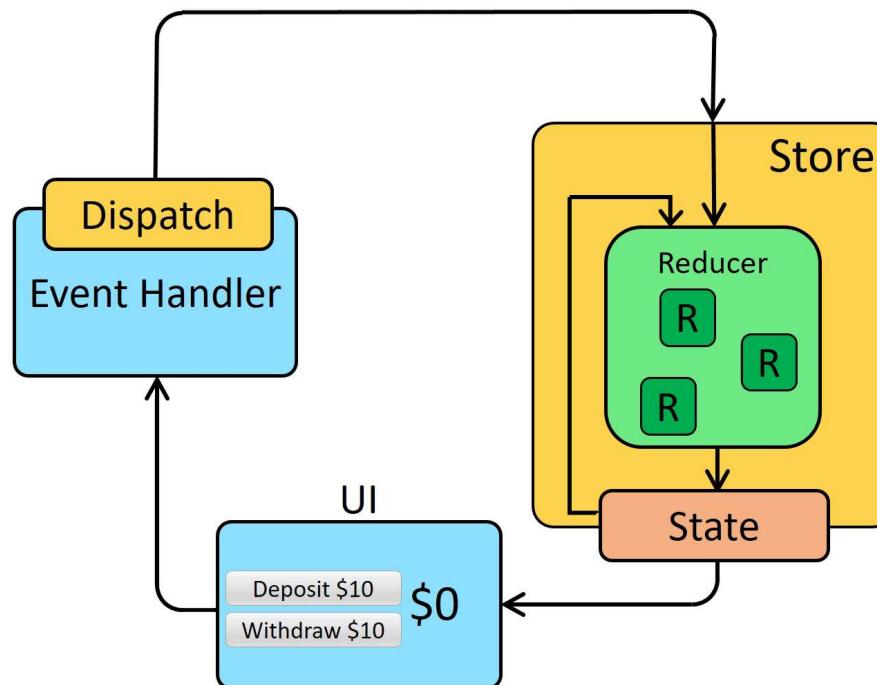


## Mises à jour:

- Quelque chose se passe dans l'application, par exemple un utilisateur cliquant sur un bouton
- Le code de l'application envoie une action au *store Redux*, comme

```
dispatch({type: 'counter/increment'})
```
- Lorsque l'interface utilisateur est rendue pour la première fois, les composants de l'interface utilisateur accèdent au *state* actuel du *store Redux* et utilisent ces données pour décider quoi restituer. Ils s'abonnent également à toutes les futures mises à jour du magasin afin de savoir si le '*state*' a changé

# Flux de données



# Basic Redux and UI Integration



- L'utilisation de Redux nécessite quelques étapes cohérentes :
  - Créer une boutique Redux
  - Abonnez-vous aux mises à jour
  - Dans le rappel d'abonnement :
    - Obtenir l'état actuel du magasin
    - Extrayez les données nécessaires à cet élément d'interface utilisateur
    - Mettre à jour l'interface utilisateur avec les données
  - Si nécessaire, restituez l'interface utilisateur avec son état initial
  - Répondez aux entrées de l'interface utilisateur en envoyant des actions Redux

# Basic Redux and UI Integration



```
// 1) Create a new Redux store with the `createStore` function
const store = Redux.createStore(counterReducer)

// 2) Subscribe to redraw whenever the data changes in the future
store.subscribe(render)

// Our "user interface" is some text in a single HTML element
const valueEl = document.getElementById('value')

// 3) When the subscription callback runs:
function render() {
  // 3.1) Get the current store state
  const state = store.getState()
  // 3.2) Extract the data you want
  const newValue = state.value.toString()

  // 3.3) Update the UI with the new value
  valueEl.innerHTML = newValue
}

// 4) Display the UI with the initial store state
render()

// 5) Dispatch actions based on UI inputs
document.getElementById('increment').addEventListener('click', function () {
  store.dispatch({ type: 'counter/incremented' })
})
```



# Installation et tuto de prise en main

- Lancer la commande ci-dessous pour installer Redux Toolkit

```
npm install @reduxjs/toolkit react-redux
```

- Créez un fichier nommé src/app/store.js. Importez l'API configureStore depuis Redux Toolkit

```
app/store.js

import { configureStore } from '@reduxjs/toolkit'

export default configureStore({
  reducer: {}
})
```



Cela crée un magasin Redux et configure également automatiquement l'extension Redux DevTools afin que vous puissiez inspecter le magasin pendant le développement.



# Installation et tuto de prise en main

- Fournir le store créé à React:
  - Importez le magasin Redux que nous venons de créer, placez un <Provider> autour de votre <App> et transmettez le magasin comme accessoire :

```
index.js

import React from 'react'
import ReactDOM from 'react-dom/client'
import './index.css'
import App from './App'
import store from './app/store'
import { Provider } from 'react-redux'

// As of React 18
const root = ReactDOM.createRoot(document.getElementById('root'))

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



# Installation et tuto de prise en main

- Créer un Redux State Slice
  - Ajoutez un nouveau fichier nommé src/features/counter/counterSlice.js. Dans ce fichier, importez l'API `createSlice` depuis Redux Toolkit

La création d'un '*slice*' nécessite un nom de chaîne pour identifier le '*slice*', une valeur d'état initiale et une ou plusieurs fonctions '*reducer*' pour définir la manière dont l'état peut être mis à jour. Une fois qu'un '*slice*' est créé, nous pouvons exporter les *actions creator* Redux générés et la fonction '*reducer*' pour l'ensemble de la tranche.

Redux exige que **nous écrivions toutes les mises à jour d'état de manière immuable, en faisant des copies des données et en mettant à jour les copies**. Cependant, les API `createSlice` et `createReducer` de Redux Toolkit **utilisent Immer** pour nous permettre d'écrire une logique de mise à jour « en mutation » qui devient des mises à jour immuables correctes.

# Installation et tuto de prise en main



```
features/counter/counterSlice.js

import { createSlice } from '@reduxjs/toolkit'

export const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0,
  },
  reducers: {
    increment: (state) => {
      // Redux Toolkit allows us to write "mutating" logic in reducers. It
      // doesn't actually mutate the state because it uses the Immer library,
      // which detects changes to a "draft state" and produces a brand new
      // immutable state based off those changes.
      // Also, no return statement is required from these functions.
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    },
  },
})

// Action creators are generated for each case reducer function
export const { increment, decrement, incrementByAmount } = counterSlice.actions

export default counterSlice.reducer
```

# Installation et tuto de prise en main



- Ajouter des *reducer slice* au store
  - nous devons importer la fonction 'reducer' depuis le 'slice' du compteur et l'ajouter à notre *store*. En définissant un champ à l'intérieur du paramètre 'reducer', nous demandons au magasin d'utiliser cette fonction 'slice' 'reducer' pour gérer toutes les mises à jour de cet état.

```
app/store.js

import { configureStore } from '@reduxjs/toolkit'
import counterReducer from '../features/counter/counterSlice'

export default configureStore({
  reducer: {
    counter: counterReducer,
  },
})
```



# Installation et tuto de prise en main

- Utiliser les '*states*' et '*actions*' dans les composants React:

Nous pouvons désormais utiliser les hooks React Redux pour permettre aux composants React d'interagir avec le magasin Redux.

Nous pouvons lire les données du magasin avec ***useSelector*** et distribuer des actions à l'aide de ***useDispatch***.

- Créez un fichier src/features/counter/Counter.js avec un composant <Counter> à l'intérieur, puis importez ce composant dans App.js et restituez-le à l'intérieur de <App>



# Installation et tuto de prise en main

features/counter/Counter.js

```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment } from './counterSlice'
import styles from './Counter.module.css'

export function Counter() {
  const count = useSelector((state) => state.counter.value)
  const dispatch = useDispatch()

  return (
    <div>
      <div>
        <button
          aria-label="Increment value"
          onClick={() => dispatch(increment())}>
          Increment
        </button>
        <span>{count}</span>
        <button
          aria-label="Decrement value"
          onClick={() => dispatch(decrement())}>
          Decrement
        </button>
      </div>
    </div>
  )
}
```



# Installation et tuto de prise en main

Désormais, chaque fois que vous cliquez sur les boutons « Incrémenteur » et « Décrémenter » :

- L'action Redux correspondante sera envoyée au magasin
- Le réducteur de tranche de compteur verra les actions et mettra à jour son état
- Le composant <Counter> verra la nouvelle valeur d'état du magasin et se restituera avec les nouvelles données

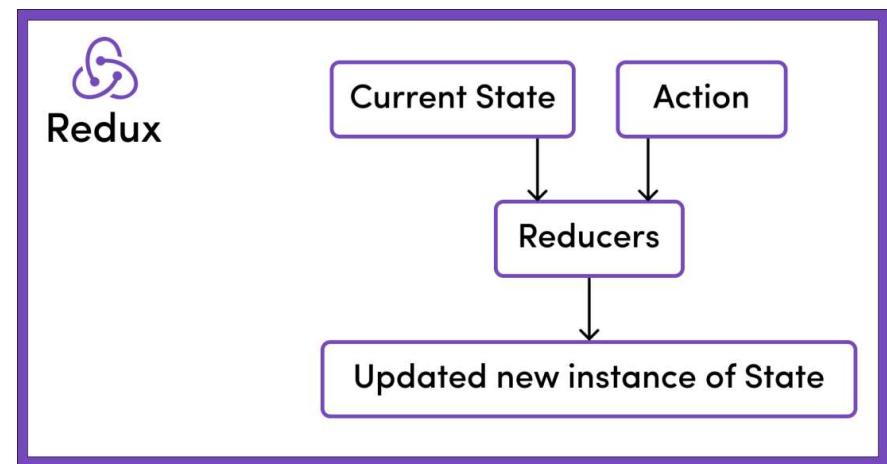
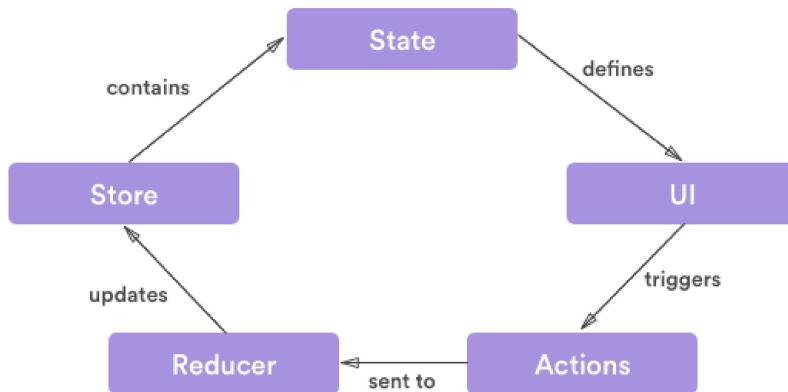


# Installation et tuto de prise en main

## Récapitulons les détails :

- Créer un magasin Redux avec configureStore
  - configureStore accepte une fonction de réduction comme argument nommé
  - configureStore configure automatiquement le magasin avec de bons paramètres par défaut
- Fournir le magasin Redux aux composants de l'application React
  - Placez un composant React Redux <Provider> autour de votre <App />
  - Transmettez le magasin Redux en tant que <Provider store={store}>
- Créez un réducteur "slice" Redux avec createSlice
  - Appelez createSlice avec un nom de chaîne, un état initial et des fonctions de réduction nommées
  - Les fonctions du réducteur peuvent "muter" l'état à l'aide d'Immer
  - Exporter le réducteur de tranche généré et les créateurs d'actions
- Utilisez les hooks React Redux useSelector/useDispatch dans les composants React
  - Lire les données du magasin avec le hook useSelector
  - Obtenez la fonction de répartition avec le hook useDispatch et répartissez les actions si nécessaire

# Reducers





# Tuto 2: Modern Redux avec *Redux Toolkit*

- [lien du tuto](#)