Avant React

Avant React

- Des sites construits avec HTML/CSS/JS
- HTML → affiche la sémantique
- CSS → mise en forme
- JS → ajoute de l'interactivité aux pages

Les données était demandé au serveur.

La problématique était la compatibilité entre les différents navigateurs → arrivé de Jquery

Le JS a pris de plus en plus de place jusqu'à la naissance de SPA

En 2010 → naissance d'Angular → application complexe

2013 → sortie de ReactJS par Facebook

Pourquoi choisir React

Pourquoi choisir React

- Avec React plus d’interaction direct avec le DOM

```
let desc = document.createElement("p");
desc.style.cssText = "font-size: 13px"
desc.innerHTML = "Description :  BlablaBla";
```

- Naissance du JSX (copie du DOM)

```
<h1 className={}>Titre</h1>
```

- Programmation Déclarative

UI Interface

UI Layer

- Facebook dit, on ne touche plus au DOM , on crée un Dom virtuel
- Chaque balise est un composant React
- Découpage en composant ré-utilisable à l'interieur du projet ou dans d'autres projets
- Chaque composant aura ça logique , son état ses propriété

Architecture

Architecture

Structure minimal d’un projet React après `npx create-react app myapp`

```
my-app/
node_modules/
public/
index.html
favicon.ico
src/
App.js
index.js
package.json
package-lock.json
README.md
```

- Le répertoire `node_modules` contient toutes les dépendances npm que vous avez installées pour votre projet.
- Le répertoire `public` contient les fichiers statiques tels que `index.html`, qui est la page d'accueil de votre application, et `favicon.ico`, qui est l'icône du site.
- Le répertoire `src` contient les fichiers source de votre application, tels que les composants React et les fichiers de configuration. Le fichier `index.js` est le point d'entrée de votre application, et le fichier `App.js` est le composant principal de votre application.
- Le fichier `package.json` décrit les dépendances et les scripts de votre application.
- Le fichier `package-lock.json` décrit les versions exactes des dépendances installées pour votre projet.
- Le fichier `README.md` est un fichier de documentation pour votre projet.

Les composants

Les composants

Un composant parent est un composant qui contient d'autres composants enfants. Les enfants peuvent être des composants simples ou des composants plus complexes qui contiennent également des enfants.

Le composant parent fournit des données et des fonctionnalités aux composants enfants à l'aide de `props` (propriétés). Les composants enfants peuvent être modifiés en retour grâce à des événements déclenchés par les interactions utilisateur.

Ce modèle de composition permet une structure claire et hiérarchique pour les applications React, ce qui facilite la maintenance et la mise à jour du code. Les composants peuvent être réutilisés dans différentes parties de l'application, ce qui permet une meilleure organisation et une maintenance plus efficace du code.

Par exemple, un composant "Card" peut être défini en tant que composant parent qui contient des composants enfants pour afficher une image, du texte et des boutons. Ce composant peut être utilisé à plusieurs endroits dans l'application pour afficher différentes informations. Les propriétés du composant parent sont passées aux composants enfants pour personnaliser leur contenu.

Class Component

Class component

```
class App extends Component {

constuctor(){
super()
this.state = {name: "Rachid"}
}


render() {
return(
   <div className="App">
     <h1> Hello {this.state.name} </h1>
     <button @OnClick={this.setState({name: "Yanis")}>Changer de nom</button>
   </div>
}

}
```

Un composant de classe React est un composant défini en utilisant une classe ES6. Il a une méthode de rendu obligatoire qui décrit ce que le composant affiche. Il peut également avoir des méthodes supplémentaires pour mettre à jour son état et gérer des effets secondaires.

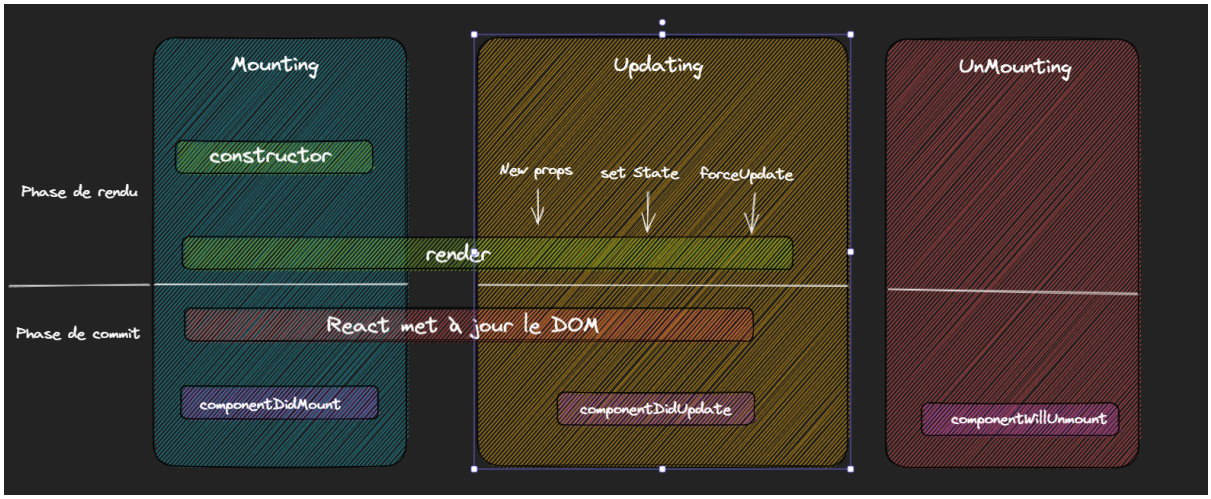
```
// async methode
this.setState(
() => {
return {
name: "Yanis"
}
},
 () => {
console.log(this.state)
})
```

Cycle de vie

Cycle de vie

Les cycles de vie en React sont des méthodes spécifiques qui sont appelées à des moments déterminés de la vie d'un composant React :

1. `Mounting` : lorsque le composant est créé et inséré dans le DOM pour la première fois, la méthode `constructor` est appelée, suivie de la méthode `render`.
2. `Updating` : lorsque les `props` ou l'état du composant sont mis à jour, la méthode `shouldComponentUpdate()` peut être appelée pour déterminer si le composant doit être mis à jour, puis la méthode `render` est à nouveau appelée pour refléter les modifications.
3. `Unmounting` : lorsque le composant est retiré du DOM, la méthode `componentWillUnmount` est appelée pour nettoyer tout ce qui est nécessaire avant de sortir.



Functional component

Functional component

- Peut être déclaré en fonction fléchée
- Plus besoin de `constructor` ni de la méthode `render()`
- Fonction pure (top to bottom)
- Retourne l'Interface Utilisateur en JSX
- Plus de cycle de vie comme avec les Class

```
function App () {
const [name, setName] = useState("Rachid")
changeName = () => {
    this.setState({ name: "Yanis" });
  };

return(
   <div className="App">
     <h1> Hello {name} </h1>
     <button @OnClick={setName("Yanis")}>Changer de nom</button>
   </div>


}
```

Hooks

Hooks

useState()

`useState` va permettre d'encapsuler un état dans mon composant fonctionnel

```
// destructuration
const [name, setName] = useState()
```

Les valeurs sont individuels contrairement l'état des class

- React va relancer toute la fonction du composant à chaque fois qu'il détecte un changement de l'état ou des propriétés

useEffect()

`useEffect` est un Hook de React qui permet de synchroniser un composant avec un système externe.

Il vous permet d'exécuter du code à chaque fois qu'un composant se rend.

C'est une façon de gérer les side effects (par exemple, la récupération de données, la mise à jour de l'état d'un composant) dans les composants fonctionnels.

Il prend deux arguments: une fonction de rappel à exécuter après le rendu et une liste de dépendances qui indique à React quand réexécuter l'effet.

```
useEffect( () => {
 console.log("a chaque rendu")
})

// componentDidMount alternative
useEffect( () => {
 console.log("rendu seulement une seule fois !")
}, [])

// componentDidUpdate alternative
useEffect( () => {
```

```
 console.log("rendu seulement une prémiere fois + rendu a chaque fois que " + toto + "change !")
}, [toto])

// componentWillUnmout alternative
useEffect( () => {

  window.addEventListener("scroll", console.log("scroll detecté")

 return () => {
   // dès que le composant ce demonte, ce code est executé
    windowr.removeEventListener("scroll", console.log("scroll supprimé")
}

}, [])
```

Function Pure/Impure et Side Effects

Function Pure/Impure et Side Effects

Fonction pure

- Le code va s'exécuter de haut en bas

```
const multiply = (arg1, arg2) => {
  return arg1 * arg2
}
```

Cette fonction est dites pur car peut importe le nombre de fois que j'appelle cette méthode avec les mêmes params, j'obtiendrais toujours le même résultat :

```
multiply(2,2) // 4
multiply(2,2) // 4
multiply(2,2) // 4
multiply(2,2) // 4
```

Les Fonctions pures sont donc prévisible

Fonction impure

```
let imp = 10;

const multiply = (arg1, arg2) => {
  return arg1 * arg2 + imp
}
```

Ici ma fonction utilise une variable qui n’est pas directement dans le scope dans ma méthode, ainsi si cette valeur change je ne peut prédire le résultat du retour.

Cette fonction est alors dites impure

Side Effect

Un autre cas ou une fonction n’est plus considérer comme pure est qu’elle va provoquer un side effect

```
let imp = 10;

const multiply = (arg1, arg2) => {
   imp = arg1 * arg2
  return arg1 + arg2
}
```

Ici si ma méthode est appelée plusieurs fois avec les mêmes arguments j’aurai toujours la même valeur de retour, cependant ma let `imp` va être modifié.

C’est ce qu’on appelle un side effect, une fonction va provoquer un changement sur une variable qui n’est pas directement présentes dans son scope (porté) .

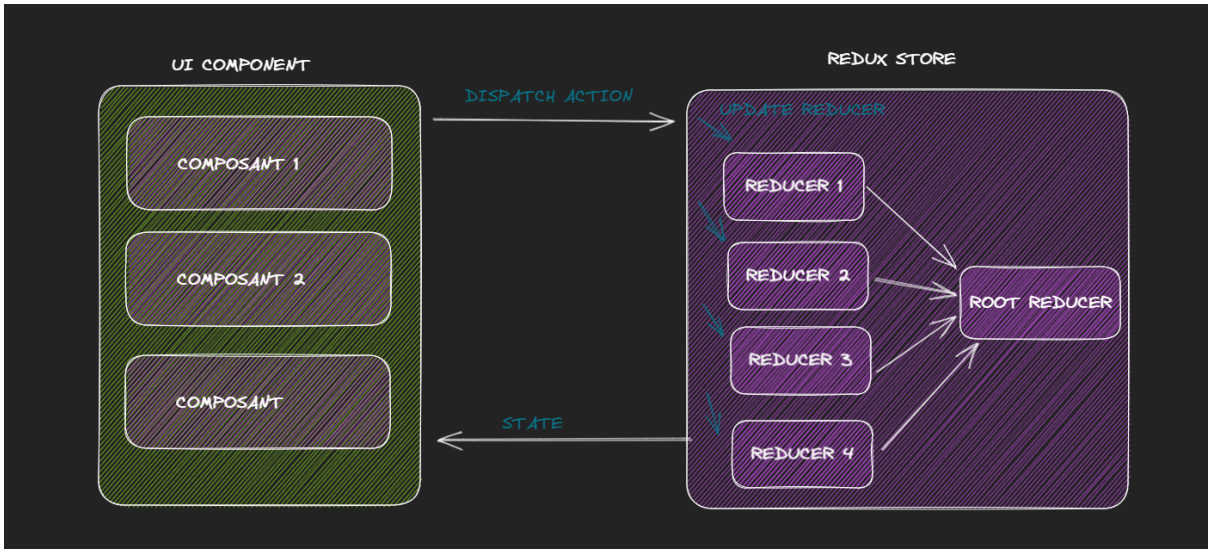
Redux

Créez rapidement un document complet.

Redux est un cadre de gestion d'état pour les applications JavaScript. Il permet de stocker et de gérer l'état de l'application dans un seul magasin centralisé, plutôt que dans plusieurs composants individuels. Cela rend plus facile la gestion de l'état de l'application, en particulier dans les applications de grande envergure avec de nombreux composants interconnectés.

Le rôle de Redux est de fournir une structure claire et prévisible pour la gestion de l'état de l'application. Il utilise des concepts tels que les actions, les réducteurs et les états pour décrire comment l'état de l'application peut être modifié de manière déterministe. Cela signifie que pour un état donné de l'application et une action donnée, la même modification de l'état sera produite, ce qui aide à éviter les bugs difficiles à reproduire.

Data Flow



- Redux va combiner les reducers en un seul
- Redux store envoi à la vue tout son état

- Les composant dispatch des actions qui vont mettre à jour les reducers correspondant

Installation

```
npm install redux
npm install react-redux
npm install redux-logger // facultatif
```

- Créer un dossier `store`
- Créer un fichier `store.js` dedans

```
import {compose, createStore, applyMiddleware} from 'redux'
import logger from 'redux-logger'
import { rootReducer } from './rootReducer'
import thunk from 'redux-thunk'



// middleware -> code qui va s'executer avant q'une action se declenche
const middlewares = [logger, thunk]

const composedEnhancers = compose(applyMiddleware(...middlewares))

// RootReducer qui va combiner tout les reducers
export const store = createStore(rootReducer, composedEnhancers)
```

rootReducer

```
import { combineReducers } from "redux";
import { TodoReducer } from "./reducers/TodoReducer";

export const rootReducer = combineReducers({
  todo: TodoReducer ,
// tout les reducers
})
```

Reducer

Les reducers sont une partie importante de la programmation avec Redux, un cadre de gestion d'état pour les applications JavaScript.

Les reducers sont des fonctions qui prennent en entrée l'état actuel de votre application et une action, et retournent un nouvel état pour votre application.

```
state: { currentUser: null }, // etat complet
action : {
type: String,
payload: any // n'importe quel type et n'est pas requis
}
```

Imaginez que vous tenez un registre pour un boulanger. Chaque jour, le boulanger ajoute ou retire des pains de différents types (pain de seigle, pain complet, etc.) de son stock.

Vous devez tenir à jour les quantités de chaque type de pain pour vous assurer que le boulanger ne manque jamais de stock.

C'est là que les reducers entrent en jeu. Vous pouvez définir un reducer qui tient à jour les quantités de chaque type de pain en fonction des actions que le boulanger effectue.

Par exemple, si le boulanger ajoute 10 pains de seigle, vous pouvez appeler le reducer avec l'action "AJOUTER_PAIN_SEIGLE" et une quantité de 10, et le reducer retournera un nouvel état de votre registre avec 10 pains de seigle de plus.

```
const initialState = {
  painComplet: 50,
  painSeigle: 30,
};

function breadReducer(state = initialState, action) {
  switch (action.type) {
    case "AJOUTER_PAIN_SEIGLE":
      return {
        ...state,
        painSeigle: state.painSeigle + action.quantité,
      };
    case "RETIRER_PAIN_SEIGLE":
      return {
        ...state,
        painSeigle: state.painSeigle - action.quantité,
      };
    case "AJOUTER_PAIN_COMPLET":
```

```
      return {
        ...state,
        painComplet: state.painComplet + action.quantité,
      };
    case "RETIRER_PAIN_COMPLET":
      return {
        ...state,
        painComplet: state.painComplet - action.quantité,
      };
    default:
      return state;
  }
}
```

Vous pouvez voir que le reducer prend en entrée l'état actuel (ou `initialState` si aucun état n'a été défini) et une action, et retourne un nouvel état en fonction de l'action. Le reducer utilise un `switch` pour déterminer quelle action est en cours d'exécution et met à jour l'état en conséquence.

Types

Les fichiers de types aident à documenter le code et à fournir une aide au codage en indiquant les types d'actions attendus et les types de données associées à chaque action.

```
// todos/todoTypes.js
export const ADD_TODO = 'ADD_TODO';
export const TOGGLE_TODO = 'TOGGLE_TODO';
```

Dispatch et Action

Les actions dans Redux sert à centraliser la définition des actions qui peuvent être déclenchées dans une application Redux.

Les actions représentent des modifications à apporter à l'état de l'application et sont généralement déclenchées par des événements tels que des clics sur des boutons, des soumissions de formulaires, etc.

L'utilisation d'un fichier d'actions permet de séparer la logique de déclenchement des actions de la logique de mise à jour de l'état de l'application.

Cela peut améliorer la lisibilité et la maintenabilité du code en permettant une définition claire des actions dans un seul endroit.

```
// todos/todoAction.js
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  };
}

export function toggleTodo(index) {
  return {
    type: 'TOGGLE_TODO',
    index
  };
}
```

`dispatch` fournie par le store Redux. Pour utiliser une action, vous devez d'abord l'importer depuis le fichier d'actions, puis l'utiliser en appelant la fonction `dispatch` et en passant l'objet d'action retourné par l'action en question en tant qu'argument.

```
function TodoList({ todos, dispatch }) {
  const handleAddTodo = text => {
    dispatch(addTodo(text));
  };

  const handleToggleTodo = index => {
    dispatch(toggleTodo(index));
  };

  // reste du composant
}
```

ou on peut également utiliser le hook `useDispatch()` de redux

Selector

`useSelector` est une hooks qui va extraire l'objet su store que l'on souhaite utilisé dans notre composant UI

```
const loading = useSelector((state) => state>.product.loading )
```

Le `selector` va se mettre à jour a chaque fois que la valeur qu'il observe changera.

Redux-Thunk

React Thunk est un middleware pour Redux qui permet d'exécuter des fonctions asynchrones dans une application Redux.

Les fonctions asynchrones peuvent inclure des opérations telles que des requêtes API, des opérations de file d'attente ou des tâches de fond.

```
npm install redux-thunk
```

Avec React Thunk, les actions peuvent retourner non seulement des objets d'action, mais également des fonctions asynchrones qui peuvent être utilisées pour déclencher des actions supplémentaires une fois qu'une opération asynchrone est terminée.

Cela permet de mieux gérer les scénarios d'application dans lesquels des actions supplémentaires doivent être déclenchées en réponse à des opérations asynchrones.

```
import axios from 'axios';

export function fetchTodos() {
  return async dispatch => {
    const response = await axios.get('/todos');
    dispatch({
      type: 'FETCH_TODOS_SUCCESS',
      todos: response.data
    });
  };
}
```

Il faut l'ajouter au store

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';
```

```
const store = createStore(rootReducer, applyMiddleware(thunk));
```

Redux-Persist

React Persist permet de conserver un état grâce au `localStorage` cela signifie que même si j'actualise ma page, mon store gardera son dernier état

UseSelector vs MapToState

It is difficult to say definitively which is "better" between `useSelector` and `mapStateToProps`, as they are used for different purposes and the appropriate choice will depend on your specific use case.

`useSelector` is a hook that allows you to retrieve values from the Redux store within a functional component, while `mapStateToProps` is a function that is used to specify which pieces of state should be passed to a component as props when using the `connect` function from React-Redux.

If you are using functional components and want an easy way to retrieve values from the Redux store, `useSelector` may be the better choice. If you are using class-based components and want to specify which pieces of state should be passed to a component as props, `mapStateToProps` may be the better choice.

Ultimately, the choice between `useSelector` and `mapStateToProps` will depend on your specific use case and the needs of your application.

Counter Exemple :

UseSelector :

```
import { createStore } from 'redux';

const initialState = {
count: 0
};

function reducer(state = initialState, action) {
switch (action.type) {
case 'INCREMENT':
```

```
return {
count: state.count + 1
};
case 'DECREMENT':
return {
count: state.count - 1
};
default:
return state;
}
}

const store = createStore(reducer);
```

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

function Counter() {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();

  function increment() {
    dispatch({ type: 'INCREMENT' });
  }

  function decrement() {
    dispatch({ type: 'DECREMENT' });
  }

  return (
    <div>
      <p>Current count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}
```

MapToState :

```
import { createStore } from 'redux';
import { connect } from 'react-redux';

const initialState = {
  count: 0
};

function reducer(state = initialState, action) {
```

```
  switch (action.type) {
    case 'INCREMENT':
      return {
        count: state.count + 1
      };
    case 'DECREMENT':
      return {
        count: state.count - 1
      };
    default:
      return state;
  }
}

const store = createStore(reducer);

function mapStateToProps(state) {
  return {
    count: state.count
  };
}

function mapDispatchToProps(dispatch) {
  return {
    increment: () => dispatch({ type: 'INCREMENT' }),
    decrement: () => dispatch({ type: 'DECREMENT' })
  };
}

function Counter(props) {
  return (
    <div>
      <p>Current count: {props.count}</p>
      <button onClick={props.increment}>Increment</button>
      <button onClick={props.decrement}>Decrement</button>
    </div>
  );
}

const ConnectedCounter = connect(mapStateToProps, mapDispatchToProps)(Counter);
```