React & React-Redux

Révisions React

Qu'est-ce que React ?

- Une bibliothèque JavaScript pour construire des interfaces utilisateur.
- Permet de créer des UI réactives et dynamiques en composant.

Concepts Clés :

1. **JSX** : Syntaxe qui permet d'écrire du HTML dans du JavaScript.
2. **Composants** : Les briques de base de React. Ils peuvent être des classes ou des fonctions.
3. **Props** : Données passées d'un composant parent à un composant enfant.
4. **State** : Données propres à un composant, généralement gérées avec `useState` ou dans les classes avec `this.state`.

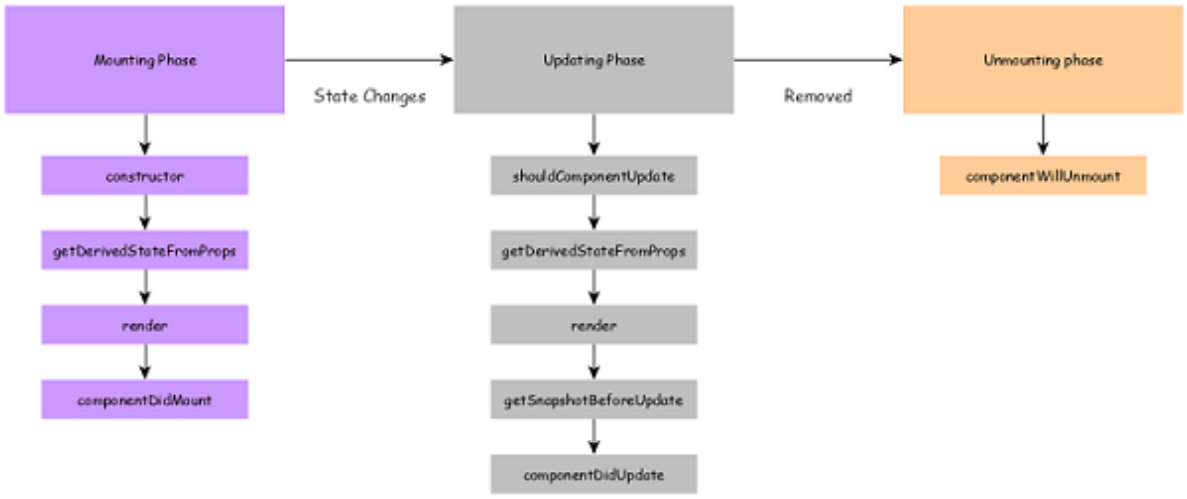
Cycle de Vie des Composants

Étapes du cycle de vie :

1. **Mounting (Montage)** :
   - `constructor`
   - `static getDerivedStateFromProps`
   - `render`
   - `componentDidMount`
2. **Updating (Mise à jour)** :
   - `static getDerivedStateFromProps`
   - `shouldComponentUpdate`
   - `render`
   - `getSnapshotBeforeUpdate`

- `componentDidUpdate`

3. **Unmounting (Démontage)** :
   - `componentWillUnmount`



Les composants React passent par une série d'étapes, appelées cycle de vie d'un composant. Ces étapes sont divisées en trois phases principales: le montage (mounting), la mise à jour (updating) et le démontage (unmounting).

1. Le Montage (Mounting)

Cette phase intervient lorsque l'instance d'un composant est créée et insérée dans le DOM.

2. La Mise à jour (Updating)

Cette phase se produit lorsqu'un composant est mis à jour et peut inclure des changements d'état ou de props.

- **constructor(props)** : C'est le premier appel fait lors de la création du composant. Il est utilisé pour initialiser l'état et lier les méthodes (si

nécessaire).

```
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
```

- **static getDerivedStateFromProps(props, state)** : Cette méthode statique est appelée avant chaque rendu, que ce soit pour le montage ou la mise à jour. Elle permet de faire correspondre l'état aux props.

```
  static getDerivedStateFromProps(props, state) {
    return { ...derivedStateFromProps }; // Retourne un
objet pour mettre à jour l'état, ou null pour ne rien c
hanger.
  }
```

- **render()** : Méthode obligatoire pour chaque composant. Elle retourne l'arbre de composants React à rendre dans le DOM.

```
  render() {
    return <div>Composant affiché</div>;
  }
```

- **componentDidMount()** : Appelée immédiatement après que le composant est monté dans le DOM. Utilisée pour effectuer des opérations DOM, lancer des requêtes réseau, etc.

```
  componentDidMount() {
    // Exécute du code après le rendu initial
  }
```

2. La Mise à jour (Updating)

Cette phase se produit lorsqu'un composant est mis à jour et peut inclure des changements d'état ou de props.

- **static getDerivedStateFromProps(props, state)** : Appelée lors de chaque mise à jour, pour synchroniser l'état avec les props.
- **shouldComponentUpdate(nextProps, nextState)** : Méthode pour décider si le composant doit être mis à jour ou non. Retourne un boolean.

```
shouldComponentUpdate(nextProps, nextState) {
      return true; // ou false pour éviter la mise à jour
    }
```

- **render()** : Appelée à nouveau pour effectuer un nouveau rendu du composant avec les nouvelles props ou état.
- **getSnapshotBeforeUpdate(prevProps, prevState)** : Utilisée pour capturer certaines informations du DOM (par exemple, position du scroll) juste avant que les changements soient rendus. Retourne une valeur, utilisée comme argument pour `componentDidUpdate`.

```
  getSnapshotBeforeUpdate(prevProps, prevState) {
    return snapshotData; // ex: la position du scroll.
  }
```

- **componentDidUpdate(prevProps, prevState, snapshot)** : Appelée immédiatement après la mise à jour. Utilisée pour réaliser des opérations sur le DOM après la mise à jour.

```
  componentDidUpdate(prevProps, prevState, snapshot) {
    // Effectue les opérations après la mise à jour
  }
```

3. Le Démontage (Unmounting)

Cette phase se produit lorsqu'un composant est retiré du DOM.

- **componentWillUnmount()** : Appelée immédiatement avant que le composant soit retiré du DOM. Utilisée pour nettoyer les ressources, annuler les requêtes, etc.

```
componentWillUnmount() {
  // Nettoyage du composant
}
```

Les Hooks :

1. **useState** : Gère l'état local d'un composant fonctionnel.

```
const [count, setCount] = useState(0);
```

2. **useEffect** : Permet d'effectuer des effets de bord dans les composants fonctionnels.

```
useEffect(() => {
    // Effet de bord (fetch, eventListener, etc.)
}, [dependencies]);
```

3. **useRef** : Accède aux éléments DOM ou stocke des valeurs persistantes.
4. **useContext** : Accède au contexte global.
5. **useReducer** : Gère l'état complexe avec des functions de réduction.

La mémoïsation :

La mémoïsation dans React est une technique d'optimisation des performances qui consiste à mémoriser (ou mettre en cache) les résultats de certaines opérations coûteuses afin de les réutiliser ultérieurement sans avoir à les recalculer. Cela peut être particulièrement utile dans le contexte de composants React, où des calculs ou des rendus peuvent être déclenchés fréquemment par des re-renders.

1. Mémoïsation de composants avec `React.memo`

`React.memo` est une fonction d'ordre supérieur qui peut être utilisée pour mémoïser des composants fonctionnels. Cela signifie que React va se rappeler du résultat du dernier rendu et ne re-rendra le composant que si ses props ont changé.

```
import React from 'react';

const MyComponent = React.memo((props) => {
  return <div>{props.value}</div>;
});
```

Dans cet exemple, `MyComponent` ne sera re-rendu que si la prop `value` change.

2. Mémoïsation de fonctions avec `useMemo`

`useMemo` est un hook qui permet de mémoïser une valeur calculée. Vous fournissez une fonction et une liste de dépendances, et `useMemo` ne recalculera la valeur que si l'une des dépendances a changé.

```
import React, { useMemo } from 'react';

const MyComponent = ({ items }) => {
  const sortedItems = useMemo(() => {
    return items.sort();
  }, [items]);
```

```
  return (
    <ul>
      {sortedItems.map(item => (
        <li key={item}>{item}</li>
      ))}
    </ul>
  );
};
```

Dans cet exemple, `sortedItems` sera recalculé uniquement lorsque `items` changera, ce qui peut éviter des calculs inutiles lors des re-renders.

3. Mémoïsation de fonctions avec `useCallback`

`useCallback` est un hook similaire à `useMemo`, mais il est utilisé pour mémoïser des fonctions plutôt que des valeurs. Cela peut être utile pour éviter de recréer des fonctions à chaque rendu, surtout si ces fonctions sont passées comme props à des composants enfants.

```
import React, { useCallback } from 'react';

const MyComponent = ({ onClick }) => {
  const handleClick = useCallback(() => {
    onClick();
  }, [onClick]);

  return <button onClick={handleClick}>Click me</button>;
};
```

Ici, `handleClick` ne sera recréé que si `onClick` change, ce qui peut éviter des re-renders inutiles dans des composants enfants.

Pourquoi utiliser la mémoïsation ?

- **Performance** : Réduire les re-renders inutiles et les calculs coûteux améliore les performances de l'application.
- **Optimisation** : Utile dans les listes ou tableaux complexes où chaque élément est ré-rendu fréquemment.
- **Réactivité** : Rend l'application plus réactive en évitant les calculs redondants.

Quand ne pas utiliser la mémoïsation

- **Complexité inutile** : Ajouter de la mémoïsation partout peut rendre le code plus complexe sans gains de performance significatifs.
- **Coût de la mémoïsation** : La mémoïsation elle-même a un coût en mémoire et en logique, donc elle doit être utilisée de manière judicieuse.

En résumé, la mémoïsation dans React permet de gérer efficacement les performances en évitant des calculs ou rendus inutiles. `React.memo`, `useMemo`, et `useCallback` sont des outils puissants pour implémenter cette optimisation de manière pragmatique et ciblée.

React Redux

Introduction à Redux

Qu'est-ce que Redux ?

- Une bibliothèque pour la gestion de l'état global dans les applications JavaScript.

Utilisé pour des applications où l'état global doit être accessible par différents composants.

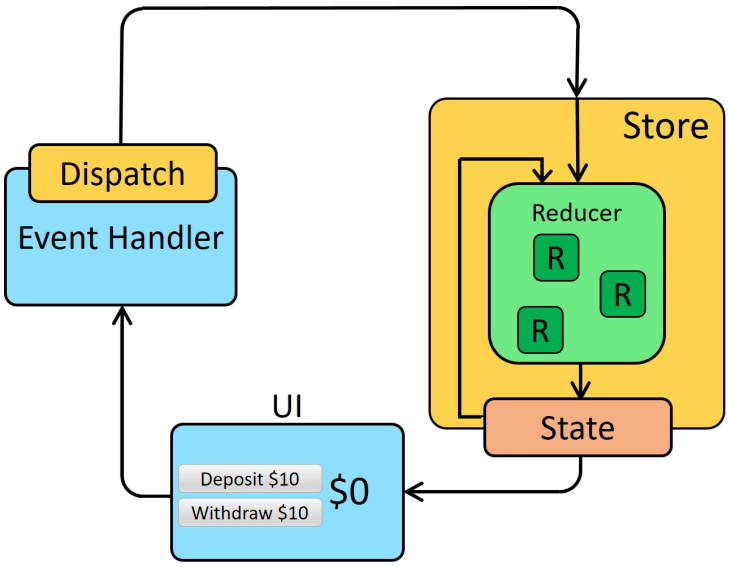
Principes Clés :

1. **Actions** : Descripteurs d'événements qui vont changer l'état.
2. **Reducers** : Fonctions pures qui prennent l'état actuel et une action, et retournent un nouvel état.

3. **Store** : Lieu unique où l'état global est stocké.
4. **Dispatch** : Méthode pour envoyer des actions aux reducers.

Flux des Données :

1. **View** déclenche une **action**.
2. **Action** est **dispatchée** à un **reducer**.
3. **Reducer** traite l'action et retourne un nouvel **état**.
4. **Store** met à jour l'état global.
5. **View** se met à jour en fonction du nouvel **état**.



Cas pratique :

Dans React, les concepts de `reducer` et de `dispatch` sont principalement utilisés avec la bibliothèque `useReducer`, qui est une alternative à l'utilisation de l'état local avec `useState`. Voici une explication de ces concepts :

`useReducer`

`useReducer` est un hook qui permet de gérer l'état local d'un composant de manière similaire à Redux. Il est particulièrement utile pour gérer des états complexes ou lorsque l'état dépend de multiples actions.

`reducer`

Un `reducer` est une fonction qui prend deux arguments : l'état actuel et une action. En fonction de l'action, le `reducer` retourne un nouvel état. Le `reducer` est une fonction pure, ce qui signifie qu'elle ne doit pas avoir d'effets secondaires et doit retourner le même résultat si elle est appelée avec les mêmes arguments.

Voici un exemple de `reducer` :

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}
```

Dans cet exemple, le `reducer` gère un état qui est un objet avec une propriété `count`. Il traite deux types d'actions : `increment` et `decrement`.

`dispatch`

`dispatch` est une fonction retournée par `useReducer`. Elle est utilisée pour envoyer une action au `reducer`. Lorsque `dispatch` est appelé avec une action,

cette action est envoyée au `reducer`, qui traite l'action et retourne un nouvel état.

Voici comment `useReducer` est utilisé dans un composant React :

```
import React, { useReducer } from 'react';

function Counter() {
  const initialState = { count: 0 };
  const [state, dispatch] = useReducer(reducer, initialState

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })
      <button onClick={() => dispatch({ type: 'decrement' })
    </div>
  );
}
```

Dans cet exemple :

1. `useReducer` est appelé avec le `reducer` et l'état initial (`initialState`).
2. Il retourne un tableau avec l'état actuel (`state`) et la fonction `dispatch`.
3. `dispatch` est utilisé dans les gestionnaires d'événements des boutons pour envoyer des actions de type `increment` et `decrement` au `reducer`.

Résumé

- **Reducer** : Une fonction qui prend l'état actuel et une action, et retourne un nouvel état.
- **Dispatch** : Une fonction utilisée pour envoyer une action au `reducer`.

En combinant ces deux éléments, `useReducer` permet de gérer des états complexes de manière plus structurée et prévisible dans vos composants React.

`L'action`

Une action est généralement un objet. Cet objet doit au minimum avoir une propriété `type` qui indique le type d'action à exécuter. Il peut également contenir des propriétés supplémentaires pour fournir des informations ou des données nécessaires à la mise à jour de l'état.

Structure d'une Action

Une action typique ressemble à ceci :

```
{
  type: 'ACTION_TYPE',
  payload: { /* données supplémentaires */ }
}
```

- `type` : Une chaîne de caractères qui identifie le type d'action. Cette propriété est obligatoire.
- `payload` : (Optionnel) Un objet contenant des données supplémentaires nécessaires pour traiter l'action.

Exemple d'Actions

Voici quelques exemples d'actions pour un compteur :

```
// Action pour incrémenter le compteur
const incrementAction = {
  type: 'increment'
};

// Action pour décrémenter le compteur
const decrementAction = {
```

```
  type: 'decrement'
};

// Action pour définir une valeur spécifique du compteur
const setCountAction = {
  type: 'set',
  payload: { count: 10 }
};
```

Utilisation dans le Reducer

Le `reducer` utilise ces actions pour déterminer comment mettre à jour l'état :

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'set':
      return { count: action.payload.count };
    default:
      throw new Error(`Unhandled action type: ${action.type}
  }
}
```

Exemple Complet avec `useReducer`

Intégrons tout cela dans un exemple complet :

```
import React, { useReducer } from 'react';

// Définir le reducer
function reducer(state, action) {
```

```
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'set':
      return { count: action.payload.count };
    default:
      throw new Error(`Unhandled action type: ${action.type}
  }
}

function Counter() {
  const initialState = { count: 0 };
  const [state, dispatch] = useReducer(reducer, initialState

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })
      <button onClick={() => dispatch({ type: 'decrement' })
      <button onClick={() => dispatch({ type: 'set', payload
    </div>
  );
}

export default Counter;
```

Résumé

- **Action** : Un objet avec au moins une propriété `type`. Il peut également contenir des données supplémentaires dans `payload` ou d'autres propriétés.
- **Reducer** : Traite les actions pour mettre à jour l'état.

- **Dispatch** : Envoie des actions au reducer pour déclencher des mises à jour de l'état.

Cette structure permet de gérer l'état de manière claire et prévisible, particulièrement pour les états complexes.

`useSelector`

Dans Redux est utilisé pour accéder à l'état du store Redux dans les composants fonctionnels de React. Il permet de sélectionner, ou extraire, une partie de l'état du store pour l'utiliser dans un composant.

Pour :

1. **Accès Direct à l'État :** Permet de lire une partie de l'état global géré par Redux directement dans les composants fonctionnels.
2. **Réactivité aux Changements d'État :** Les composants sont automatiquement ré-rendus lorsque la partie de l'état sélectionnée par `useSelector` change.
3. **Facile à Utiliser avec des Hooks :** S'intègre parfaitement avec les autres hooks de React pour créer des composants fonctionnels propres et concis.

Syntaxe et Utilisation

```
const component = () => {// Sélectionner une partie de l'éta
  const someState = useSelector((state) => state.someReducer

    return (
      <div>
        <p>{someState}</p>
      </div>
```

```
      );
    };
```

- **Sélecteur de l'État :**
  La fonction passée à `useSelector` reçoit l'état global du store Redux en argument et retourne la partie de l'état qui est pertinente pour le composant.
- Le composant sera ré-rendu chaque fois que la valeur retournée par `use selector` change. Cela signifie que si `someState` change dans le store Redux, le composant affichera automatiquement les nouvelles valeurs.