## Introduction à React

React est une bibliothèque JavaScript développée par Facebook qui permet de construire des interfaces utilisateur interactives. Elle est centrée sur le concept de composants, qui sont des blocs de construction réutilisables d'une interface. React se charge de rendre ces composants de manière efficace lorsque l'état de l'application change, garantissant des performances optimales.

## Le Rendering en React

Le rendering est le processus par lequel React affiche vos composants à l'écran. Lorsque l'état ou les données d'un composant changent, React met à jour l'interface utilisateur pour refléter ces modifications, sans avoir besoin de recharger toute la page. Cette approche améliore les performances et l'expérience utilisateur.

## Utilisation de TypeScript

Dans nos projets, nous utilisons TypeScript, un sur-ensemble de JavaScript qui ajoute une vérification statique des types. Cela améliore la qualité du code, facilite la détection des erreurs et renforce la maintenabilité.

## Qu'est-ce que le TSX ?

Le TSX (TypeScript JSX) est une extension de syntaxe qui permet d'écrire du JSX (le langage utilisé par React pour décrire l'interface utilisateur) avec TypeScript. Cela signifie que vous bénéficiez de la flexibilité de JSX avec la sécurité de TypeScript. Les fichiers TSX ont l'extension .tsx et permettent d'ajouter des types à vos composants React.

## Le cycle de vie d'une application React

Le **cycle de vie d'une application React** correspond aux différentes étapes par lesquelles passe un composant React, de sa création à sa destruction. Il est divisé en trois phases principales :

1. **Montage (Mounting)** : C'est le moment où le composant est créé et inséré dans le DOM. C'est ici que React prépare tout pour afficher le composant pour la première fois.
2. **Mise à jour (Updating)** : Pendant cette phase, le composant réagit aux changements d'état ou de propriétés (props). Chaque mise à jour entraîne un nouveau rendu du composant, qui adapte son contenu visuel en fonction des nouvelles données.
3. **Démontage (Unmounting)** : C'est lorsque le composant est supprimé du DOM, souvent parce qu'il n'est plus nécessaire. Cela permet de libérer les ressources et de nettoyer correctement tout ce qui a été mis en place durant son montage.

Le cycle de vie permet donc à React de gérer efficacement les changements dans l'application tout en optimisant les performances.

Les hooks de React, et en particulier useEffect, permettent de gérer les cycles de vie des composants fonctionnels, remplaçant ainsi les méthodes de cycle de vie des composants de classe comme componentDidMount, componentDidUpdate, et componentWillUnmount.

## Les hooks

Les **Hooks** en React sont des fonctions qui permettent d'ajouter des fonctionnalités comme la gestion de l'état à des composants fonctionnels, sans avoir besoin d'utiliser des classes. Ils simplifient le code tout en permettant aux composants fonctionnels d'avoir des capacités avancées, telles que la gestion de l'état, le cycle de vie et l'accès direct au DOM. Grâce aux Hooks, les composants fonctionnels peuvent faire tout ce qu'un composant de classe peut faire, tout en restant plus lisibles et maintenables.

**Hooks Standards :**

**useState**

Permet de déclarer une variable d'état locale dans un composant fonctionnel. Cette variable d'état est ensuite utilisée pour stocker des valeurs qui peuvent changer au fil du temps.

const [state, setState] = useState(initialValue);

* ***Bonnes pratiques du useState :***
  + Utiliser useState pour des valeurs locales et spécifiques au composant.
  + Initialiser l'état avec une valeur ou une fonction de valeur initiale.
  + Utiliser le setter retourné par useState pour mettre à jour l'état.

**useEffect**

Permet d'exécuter du code après le rendu du composant. Il est utilisé pour les effets de bord tels que les appels API, les mises à jours suite à des événements ou la manipulation directe du DOM.

useEffect(() => {

// Code à exécuter

return () => {

// Cleanup (optionnel)

};

}, [dependencies]);

* ***Bonnes pratiques du useEffect***
  + Spécifier un tableau de dépendances pour contrôler quand l'effet doit être exécuté.
  + Nettoyer les effets en retournant une fonction dans le callback de useEffect pour éviter les fuites de mémoire.
  + Utiliser plusieurs hooks useEffect si nécessaire pour séparer les logiques indépendantes.

**useContext**

Permet d'accéder à des valeurs partagées via un Context React. Il est utile pour passer des données à travers l'arborescence de composants sans avoir à les passer explicitement par chaque niveau.Il s’agit de la version native de [Redux](https://destinygroup.atlassian.net/wiki/spaces/DSTNYFR/pages/1356365829), seulement il peut être plus délicat à mettre en place par conséquent on privilégiera l’utilisation de Redux à celle du useContext.

const value = useContext(MyContext);

* ***Bonnes pratiques du useContext***
  + Créer un Context pour les données globales que plusieurs composants doivent utiliser.
  + Utiliser useContext pour accéder à ces données dans les composants enfants.
  + Éviter d'utiliser useContext pour des valeurs qui ne changent pas fréquemment pour minimiser les rerendus inutiles.

**useRef**

useRef est un outil que React fournit pour garder une information "en mémoire" sans que cela ne force le composant à se réactualiser. Il est souvent utilisé pour deux raisons principales :

1. **Accéder aux éléments de la page** : Par exemple, si on a besoin de manipuler directement un élément visuel (comme un champ de saisie de texte), useRef permet d'y accéder sans déclencher de re-rendu de la page.
2. **Garder des données entre deux affichages** : Si on souhaite garder une information (comme un compteur ou un délai) qui ne devrait pas forcer React à réafficher la page quand elle change, useRef est une bonne option.

Exemple :

const ref = useRef(initialValue);

ref peut être utilisé pour accéder à quelque chose (comme un champ de saisie) ou stocker une information qui ne doit pas influencer l'affichage de la page.

* ***Bonnes pratiques du useRef***
  + Utiliser useRef pour accéder aux éléments DOM.
  + Utiliser useRef pour stocker des valeurs qui doivent persister entre les rendus sans causer de re-rendu (par exemple, pour implémenter des timers).
  + Ne pas utiliser useRef pour des données qui doivent déclencher un re-rendu lorsqu'elles changent.

**useCallback**

Mémorise une fonction et la renvoie. La fonction mémorisée ne sera recréée que si une des dépendances a changé. Cela est utile pour éviter de recréer des fonctions à chaque rendu, ce qui peut être coûteux.

const memoizedCallback = useCallback(() => {

// Function body

}, [dependencies]);

* ***Bonnes pratiques du useCallback***
  + Utiliser useCallback pour des fonctions passées en prop à des composants enfants qui utilisent React.memo pour optimiser les rerendus.
  + Spécifier un tableau de dépendances approprié pour s'assurer que la fonction est mise à jour correctement lorsque les dépendances changent.
  + Éviter d'utiliser useCallback à chaque fois, mais seulement quand il y a un réel bénéfice en termes de performance.

**useMemo**

Mémorise la valeur retournée par une fonction de calcul. Cette valeur mémorisée ne sera recalculée que si une des dépendances a changé. Cela est utile pour optimiser des calculs coûteux ou pour éviter de recréer des objets complexes.

const memoizedValue = useMemo(() => {

return computeExpensiveValue(a, b);

}, [a, b]);

* ***Bonnes pratiques du useMemo***
  + Utiliser useMemo pour des calculs lourds ou pour mémoriser des objets complexes.
  + Spécifier un tableau de dépendances approprié pour s'assurer que la valeur mémorisée est mise à jour correctement lorsque les dépendances changent.
  + Ne pas abuser de useMemo, mais l'utiliser uniquement lorsque les calculs sont réellement coûteux ou les objets complexes.

**Hooks Personnalisés :**

Un custom hook est une fonction JavaScript réutilisable qui utilise les hooks de React pour encapsuler et partager la logique d'état et d'effets entre plusieurs composants. Il commence toujours par le préfixe use et permet de centraliser des logiques complexes ou répétitives, rendant les composants plus simples et plus modulaires. Les custom hooks peuvent gérer des appels API, ou toute autre logique réutilisable qui nécessite des hooks standard comme useState, useEffect, etc.

**Nomenclature des Hooks Personnalisés**

Un hook personnalisé doit toujours commencer par le préfixe use.

* ***Raison***: Cela permet à React de reconnaître qu'il s'agit d'un hook et de s'assurer que les règles des hooks sont appliquées.

function useCustomHook() {

// Logique du hook

}

**Exploiter la Puissance de la Composition**

Utilisez les hooks existants (comme useState, useEffect, etc.) pour créer des hooks personnalisés qui encapsulent des logiques réutilisables.

* ***Raison***: Cela permet de réutiliser du code et de simplifier vos composants.

import { useState, useEffect } from 'react';

function useFetchData(url) {

  const [data, setData] = useState(null);

  const [loading, setLoading] = useState(true);

  useEffect(() => {

    async function fetchData() {

      const response = await fetch(url);

      const result = await response.json();

      setData(result);

      setLoading(false);

    }

    fetchData();

  }, [url]);

  return { data, loading };

}

**Exporter et Importer les Hooks Personnalisés**

Placez vos hooks personnalisés dans des fichiers séparés pour faciliter leur réutilisation et leur maintenance.

* ***Raison***: Cela améliore la structure de votre projet et rend vos hooks plus accessibles.

export function useHook() {}

**Isoler la Logique et l'État**

Les hooks personnalisés (custom hooks) doivent encapsuler toute la logique nécessaire à leur fonctionnalité et gérer leur propre état de manière indépendante. Cela signifie que tout le code nécessaire pour accomplir une tâche spécifique est contenu dans le hook lui-même, plutôt que d'être dispersé dans le composant qui utilise ce hook. Cette isolation rend les hooks personnalisés autonomes, réutilisables et faciles à comprendre.

***Exemple :***

*Sans hook :*

import React, { useState } from 'react';

function Counter() {

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

  const increment = () => setCount(count + 1);

  const decrement = () => setCount(count - 1);

  return (

    <div>

      <button onClick={decrement}>-</button>

      <span>{count}</span>

      <button onClick={increment}>+</button>

    </div>

  );

}

export default Counter;

*Avec hook :*

import { useState } from 'react';

// Custom Hook

function useCounter(initialValue = 0) {

  const [count, setCount] = useState(initialValue);

  const increment = () => setCount(count + 1);

  const decrement = () => setCount(count - 1);

  return { count, increment, decrement };

}

export default useCounter;

// Composant utilisant le Hook Personnalisé

import React from 'react';

import useCounter from './useCounter';

function Counter() {

  const { count, increment, decrement } = useCounter(0);

  return (

    <div>

      <button onClick={decrement}>-</button>

      <span>{count}</span>

      <button onClick={increment}>+</button>

    </div>

  );

}

export default Counter;

***Points Clés :***

* ***Encapsulation***: Toute la logique et l'état liés au compteur sont encapsulés dans le hook personnalisé useCounter.
  + ***Réutilisable***: Le hook useCounter peut maintenant être réutilisé dans d'autres composants sans dupliquer la logique du compteur.
  + ***Simplicité***: Le composant Counter est simplifié car il n'a plus besoin de gérer directement l'état ou la logique du compteur.

***Avantages* :**

* ***Modularité***: Les hooks personnalisés rendent votre code plus modulaire et plus facile à maintenir.
* ***Réutilisable***: La même logique peut être utilisée dans plusieurs composants, réduisant ainsi la duplication de code.
* ***Lisibilité***: Les composants deviennent plus simples et plus faciles à comprendre, car ils délèguent la gestion de la logique et de l'état aux hooks personnalisés.

**Règles des hooks :**

**N'appeler les hooks qu'au niveau supérieur.**

Les hooks doivent être appelés au niveau supérieur du composant fonctionnel, et non à l'intérieur de boucles, de conditions ou de fonctions imbriquées.

* ***Raison***: Cela garantit que les hooks sont appelés dans le même ordre à chaque rendu

function MyComponent() {

  // Correct

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

  useEffect(() => {

    setCount(0) // Code d'effet

  }, []);

  // Incorrect

  if (someCondition) {

    useState(0); // Ne faites pas cela

  }

}

**Respectez les règles des dépendances des hooks**

Il faut toujours spécifier toutes les dépendances utilisées à l'intérieur de useEffect, useCallback, useMemo et d'autres hooks similaires.

* ***Raison***: Cela garantit que les effets et les valeurs mémorisées sont recalculés correctement lorsque leurs dépendances changent.

useEffect(() => {

// Code d'effet qui dépend de 'prop' et 'state'

}, [prop, state]);

**Divisez les effets complexes en plusieurs useEffect**

Dans le cas d’un effet complexe avec plusieurs responsabilités, divisez-le en plusieurs hooks useEffect pour chaque responsabilité.

* ***Raison***: Cela rend votre code plus lisible, plus facile à déboguer et à maintenir.

useEffect(() => {

  // Code d'effet pour les appels API

}, [dependency1]);

useEffect(() => {

  // Code d'effet pour les mis à jours d'évènements

}, [dependency2]);

**Exemple global**

Voici un exemple d’utilisation de ces hooks dans le cadre d’une application. (useContext n’a pas été utilisé puisqu’il doit être remplacé par redux :

)

import React, { useState, useRef, useCallback, useEffect, useMemo } from 'react';

export default MyComponent({ initialCount } : { initialCount: number }) => {

// useState pour gérer l'état du compteur

const [count, setCount] = useState<number>(initialCount);

// useRef pour garder une référence à un élément DOM

const inputRef = useRef<HTMLInputElement>(null);

// useMemo pour mémoriser une valeur dérivée

const computedValue = useMemo(() => {

return count \* 2; // Calcul coûteux, ici simple pour l'exemple

}, [count]);

// useCallback pour créer une fonction de gestion des clics

const handleClick = useCallback(() => {

setCount(prevCount => prevCount + 1);

}, []);

// useEffect pour exécuter un effet secondaire

useEffect(() => {

// Focus sur l'élément input lorsque le composant est monté

if (inputRef.current) {

inputRef.current.focus();

}

}, []); // Le tableau vide signifie que cet effet s'exécute une seule fois après le premier rendu

return (

<div>

<h1>Compteur: {count}</h1>

<button onClick={handleClick}>Incrémenter</button>

<input ref={inputRef} type="text" placeholder="Focus automatique" />

<p>Valeur calculée: {computedValue}</p>

</div>

);

};

**Explication des Hooks Utilisés :**

 useState :

* Utilisé pour gérer l'état du compteur (count). Permet de définir une valeur initiale et de mettre à jour cette valeur lorsque l'utilisateur clique sur le bouton.

 useRef :

* Utilisé pour obtenir une référence à un élément DOM (inputRef). Permet de mettre le focus sur l'élément input lorsque le composant est monté.

 useCallback :

* Utilisé pour mémoriser la fonction handleClick qui incrémente le compteur. Cela évite de recréer la fonction à chaque rendu, ce qui peut être utile pour les optimisations de performance.

 useEffect :

* Utilisé pour effectuer un effet secondaire après le premier rendu du composant, ici pour mettre le focus sur l'élément input. Le tableau vide ([]) signifie que cet effet s'exécute uniquement lors du premier montage du composant.
*  useMemo :
* Utilisé pour mémoriser la valeur dérivée (computedValue). Cette valeur est recalculée uniquement lorsque count change, ce qui évite de recalculer la valeur à chaque rendu.

## Convention de nommage

* Noms de dossiers en camelCase (ex : myFolder)
* Noms des variables et des fonctions en camelCase (ex : let myValue)
  + Utiliser des noms descriptifs et significatifs pour comprendre à quoi est associée la fonction / la variable
* Noms de fichiers en PascalCase (ex : MyFile.tsx)
* Noms de composants en PascalCase (ex : MyComponent() )

## Linter

Installations nécessaires :

* npm install eslint-plugin-react --save-dev
* npm install eslint-plugin-import --save-dev
* l’extension eslint sur vscode ou sur intelliJ

Afin de pouvoir lancer une analyse eslint sur notre projet, il faudra rajouter dans le package.json le scripts suivant :

"scripts": {

"lint": "eslint . --ext .js,.jsx,.ts,.tsx"

},

Le linter est utilisé pour maintenir la qualité du code en imposant des règles de style et des meilleures pratiques. ESLint est un outil qui analyse votre code pour trouver des erreurs et des problèmes de style. En configurant ESLint pour une application React, vous pouvez appliquer des règles spécifiques à JavaScript et React. Cela aide à identifier et corriger les erreurs potentielles avant même d'exécuter le code, assure la conformité aux conventions de codage, et renforce les bonnes pratiques de développement. Prettier est un formateur de code qui s'assure que votre code est formaté de manière cohérente selon des règles définies. En intégrant Prettier avec ESLint, il est possible d’automatiser le formatage du code, ce qui réduit les différences de style entre les développeurs et permet de se concentrer sur la logique du code plutôt que sur son apparence.

***Avantages du Linter et de Prettier***

* ***Détection Précoce des Erreurs*** *:* ESLint peut détecter des erreurs de syntaxe, des problèmes de type, et des violations des règles de React, comme les dépendances manquantes dans les hooks.
* ***Respect des Conventions de Codage*** *:* En forçant le respect des conventions de codage, ESLint assure que tout le code est écrit de manière cohérente, ce qui améliore la lisibilité et la maintenabilité.
* ***Meilleures Pratiques*** *:* ESLint avec les plugins React et React Hooks peut appliquer des règles spécifiques qui encouragent les bonnes pratiques, telles que l'utilisation correcte des hooks.
* ***Formatage Cohérent*** *:* Prettier garantit que le code est formaté de manière cohérente, ce qui réduit les différends entre développeurs concernant le style de code et permet de se concentrer sur les aspects fonctionnels.

Configuration eslint nécessaire dans chaque projet React. Le fichier concerné est .eslintrc.cjs

module.exports = {

root: true,

env: { browser: true, es2020: true },

extends: [

'eslint:recommended',

'plugin:@typescript-eslint/recommended',

'plugin:react/recommended',

'plugin:react-hooks/recommended',

'plugin:import/errors',

'plugin:import/warnings',

'plugin:import/typescript',

],

ignorePatterns: ['dist', '.eslintrc.cjs'],

parser: '@typescript-eslint/parser',

plugins: ['react-refresh', 'import'],

rules: {

'react-refresh/only-export-components': ['warn', { allowConstantExport: true }],

'@typescript-eslint/no-unused-vars': ['error', { argsIgnorePattern: '^\_' }],

'react/jsx-uses-react': 'off',

'react/react-in-jsx-scope': 'off',

'@typescript-eslint/explicit-function-return-type': 'off',

'@typescript-eslint/no-explicit-any': 'warn',

'import/no-unresolved': 'error',

'import/newline-after-import': 'warn',

},

};

Explications :

* **‘root: true’ :** signifie que la configuration ESLint est le fichier racine et qu’il ne cherchera pas à hériter d’autres fichiers de configuration ESLint situés plus haut dans l’arborescence de fichiers.
* **‘env: { browser: true, es2020: true }’:** il spécifie les environnements pour lesquels le code est écrit. Ici nous avons :
  + ‘***browser: true’:*** qui indique le code est destiné à être exécuté dans un navigateur
  + ***‘es2020: true’:*** qui indique que le code utilise les fonctionnalités d’ES2020.
* **‘extends’:** cette section inclut des configurations de bases qui seront héritées par le projet
  + ***‘eslint:recommanded’:*** active les règles recommandées par ESLint qui couvrent les erreurs de base. (comme les variables non définies ou l’utilisation incorrecte de certains opérateurs)
  + ***‘plugin:@typescript-eslint/recommanded’:*** active les règles recommandées par typescript via le plugin @typescript-eslint
  + ***‘plugin:react/recommanded’:*** active les règles recommandées pour React. Cela garantit que les composants sont bien définis et que des pratiques communes sont suivies.
  + ***‘plugin: react-hooks/recommanded’:*** active les règles spécifiques aux hooks de React (par exemple, les règles sur l’utilisation correcte de useEffect, useState, etc)
  + ***‘plugin:import/errors’ & ‘plugin:import/warnings’ & ‘plugin:import/typescript’:*** Ces trois plugins permettent de gérer et de vérifier les imports. Ils vérifient que les chemins di’mportation sont corrects et que les modules existent bien, en tenant compte de TypeScript.
* **‘ignorePatterns: [ ‘dist’, ‘.eslintrc.cjs’]’:** il indique à ESLint d’ignorer les fichiers et les dossiers listés ici
* **‘parser: ‘@typescript-eslint/parser’’:** Ici on précise que le parser utilisé est celui de @typescript-eslint ce qui permet à ESLint de comprendre et d’analyser le code TypeScript
* **‘plugins: [’react-refresh', ‘import’]':** Ce plugin est lié à react-refresh qui est un outil permettant de faire du rechargement à chaud (Hot Module Replacement) dans des projets React. Le plugin contient des règles spécifiques à l’utilisation de ce module, mais aussi à import qui est utilisé pour valider les importations dans le code afin de s’assurer qu’elles sont correctement ordonnées et que les modules référencés existent vraiment.
* **‘rules’:** Cette section définit des règles spécifiques pour ESLint. Chaque règle peut-être configurée avec des niveaux d’avertissement (warn) ou d’erreur (error). Voici les règles présentes dans notre fichier:
  + ***‘react-refresh/only-export-components: ['warn', { allowConstantExport: true }]':*** cette règle avertit si on exportes autre chose que des composants React dans les fichiers où react-refresh est utilisé, toutefois elle autorise l’export des constantes.
  + ***‘typescript-eslint/no-unused-vars: ['error', { argsIgnorePattern: ‘^\_’ }]':*** signe une erreur lorsque des variables ou des arguments de fonction ne sont pas utilisés dans le code. Cependant, les arguments commençant par un underscore (\_) sont ignorés, ce qui est une pratique courante pour signaler des arguments intentionnellement non utilisés.
  + ***‘react/jsx-uses-react: ‘off’’:*** désactive cette règle car depuis React 17+ il n’est plus nécessaire d’importer explicitement React dans les fichiers TSX. Cette règle est donc inutile pour les projets modernes.
  + ***‘react/react-in-jsx-scope: ‘off’':*** similaire à la règle précédente, elle vérifie que React est dans la portée des fichiers TSX. Depuis React 17+ cette règle est obsolète.
  + ***‘typescript-eslint/explicit-function-return-type: ‘off’’:*** désactive l’obligation explicite de spécifier les types de retour dans les fonctions. Cela allège la syntaxe, surtout dans des situations où TypeScript peut inférer correctement le type de retour.
  + ***‘typescript-eslint/no-explicit-any: ‘warn’’:*** cette règle émet un avertissement si le type any est utilisé dans le code. Le type ‘any’ est souvent considéré comme une mauvaise pratique car il désactive les avantages de la vérification des types de TypeScript.
  + ‘***import/no-unresolved: ‘error’':*** génère une erreur si un module importé ne peut pas être résolu, c’est-à-dire si le chemin d’importation est incorrect ou si le module n’existe pas alors on obtient une erreur.
  + ***‘import/newline-after-import: ‘warn’’:*** avertit si une nouvelle ligne n’est pas ajoutée après les déclarations d’importation, ce qui est une convention de style courante pour améliorer la lisibilité du code.

## Extensions recommandées

**ESLint** : pour le linting du code (

* )
* **Prettier** : pour le formatage du code (

 )

 **ReactJS code snippets** : Pour des snippets de code React (pas d’équivalent à ma connaissance)

 **ReactJS** : pour une autocomplétion des outils React (natif sur IntelliJ)

 **Bracket Pair Colorizer**: Pour une meilleure lisibilité des parenthèses et des accolades

## Mapping

Technique utilisée pour transformer des données en une liste d'éléments React. Par exemple, si on a un tableau de données, on peut utiliser map() pour créer un composant React pour chaque élément du tableau, produisant ainsi une liste dynamique d'éléments affichés à l'écran. C'est une manière efficace de rendre des listes d'éléments basées sur des données variées ou dynamiques.

**Utiliser des Clés Uniques**

Lors du rendu de listes, chaque élément doit avoir une clé unique.

* ***Raison***: Les clés aident React à identifier quels éléments ont changé, ont été ajoutés ou supprimés, optimisant ainsi le processus de réconciliation.

{items.map((item, index) => (

<li key={item}>{item}</li> // Utilisez des clés uniques

))}

**Éviter d'utiliser les Index comme Clés**

Utiliser l'index d'un tableau comme clé peut causer des problèmes si l'ordre des éléments change.

* ***Raison***: Les clés doivent être stables et uniques pour chaque élément. Les index peuvent entraîner des bugs subtils et des comportements inattendus lors de réarrangements de la liste.

**Rendre des Listes Complexes avec des Composants Enfants**

Pour des éléments de liste complexes, créez des composants enfants.

* ***Raison***: Cela améliore la lisibilité, la réutilisabilité et la maintenabilité du code.

function ListItem({ item }) {

  return <li>{item.name}</li>;

}

const items = [

  { id: 1, name: 'Apple' },

  { id: 2, name: 'Banana' },

  { id: 3, name: 'Cherry' },

];

return (

  <ul>

    {items.map((item) => (

      <ListItem key={item.id} item={item} />

    ))}

  </ul>

);

**Filtrage et Transformation avant le Rendu**

Filtrez et transformez les données avant le rendu pour éviter une logique complexe dans le TSX.

* ***Raison***: Cela rend le TSX plus propre et sépare la logique de manipulation des données du rendu.

const items = [

  { id: 1, name: 'Apple', available: true },

  { id: 2, name: 'Banana', available: false },

  { id: 3, name: 'Cherry', available: true },

];

const availableItems = items.filter(item => item.available);

return (

  <ul>

    {availableItems.map((item) => (

      <li key={item.id}>{item.name}</li>

    ))}

  </ul>

);

**Utiliser les Fragment (<React.Fragment> ou <> </>) pour les Listes sans Conteneur**

Utiliser les Fragment (<React.Fragment> ou <> </>) pour les Listes sans Conteneur

* ***Raison***: Cela permet de rendre des listes sans ajouter de nœuds DOM supplémentaires inutiles.

<>

{items.map((item) => ( <li key={item.id}>{item.name}</li> ))}

</>

**Manipuler les Données Avant le Rendu**

Cela rejoint le filtrage des données, on prépare ou on transforme les données avant de les passer au TSX.

* ***Raison***: Maintient la logique de transformation de données hors du rendu JSX, rendant le code plus lisible et plus maintenable.

const items = [

  { id: 1, name: 'apple' },

  { id: 2, name: 'banana' },

  { id: 3, name: 'cherry' },

];

const capitalizedItems = items.map(item => ({

  ...item,

  name: item.name.charAt(0).toUpperCase() + item.name.slice(1)

}));

return (

  <ul>

    {capitalizedItems.map((item) => (

      <li key={item.id}>{item.name}</li>

    ))}

  </ul>

);

## Optimisation des ressources

L'optimisation des ressources en React vise à améliorer les performances des applications en réduisant la consommation excessive de ressources comme la mémoire et le CPU.

**Promise - Manipuler les états de chargement et d’erreur de manière appropriée**

Dans une application il est courant de travailler avec des promesses (promise) pour gérer les appels API ou d’autres tâches asynchrones.

**Gestion des états de chargement et d'erreur :**

Il est important de toujours suivre les **états de chargement** (loading) et **d'erreur** (error) dans les composants lorsqu’on effectue des appels asynchrones. Cela permet d'informer l'utilisateur de l'état du traitement (exemple : montrer un spinner pendant le chargement ou un message d'erreur en cas de problème).

Utiliser des hooks comme useState pour gérer ces états :

const [loading, setLoading] = useState(false);

const [error, setError] = useState<Error | null>(null);

const [data, setData] = useState<DataType | null>(null);

const fetchData = async () => {

setLoading(true);

setError(null);

try {

const response = await fetch('/api/data');

const result = await response.json();

setData(result);

} catch (err) {

setError(err as Error);

} finally {

setLoading(false);

}

};

**Promise.all :**

Lors de plusieurs appels asynchrones qui peuvent être exécutés en parallèle, il est recommandé d'utiliser Promise.all pour les effectuer de manière concurrente, ce qui peut accélérer considérablement les temps de réponse et économiser des ressources :

const fetchMultipleData = async () => {

try {

const [data1, data2] = await Promise.all([

fetch('/api/data1').then((res) => res.json()),

fetch('/api/data2').then((res) => res.json()),

]);

// Utilise data1 et data2 ici

} catch (err) {

// Gérer les erreurs

}

};

Cela permet de lancer les deux requêtes en même temps et de ne pas attendre l'une après l'autre, réduisant ainsi le temps total de chargement.

Dans le cas où vous souhaitez faire un trop grand nombre de requêtes en même temps, il se peut que le Promise.all soit rendu très lent (problème de RAM, de réseau ou autre).

Dans ce cas il est intéressant de faire des lots de requêtes. Exemple :

const batchRequest = async (requests: Array<() => Promise<any>>, batchSize: number) => {

const results = [];

for (let i = 0; i < requests.length; i += batchSize) {

const batch = requests.slice(i, i + batchSize);

const batchResults = await Promise.all(batch.map(req => req()));

results.push(...batchResults);

}

return results;

};

// Exemple d'utilisation avec des fetch:

const requests = [

() => fetch('/api/data1').then(res => res.json()),

() => fetch('/api/data2').then(res => res.json()),

// ...

];

const results = await batchRequest(requests, 5); // Exécute 5 requêtes simultanément

**Lazy loading**

Le **lazy loading** est une technique permettant de charger les composants de manière différée, uniquement lorsqu'ils sont nécessaires. Cela réduit le poids initial de l'application, améliorant ainsi les temps de chargement initiaux.

**React.lazy et Suspense :**

React.lazy permet de charger un composant de manière dynamique, ce qui est utile pour ne pas importer tout le code au début de l'application. C'est souvent utilisé pour des routes ou des composants qui ne sont pas immédiatement nécessaires.

const LazyComponent = React.lazy(() => import('./LazyComponent'));

const MyComponent = () => {

return (

<Suspense fallback={<div>Loading...</div>}>

<LazyComponent />

</Suspense>

);

};

Ici, Suspense est utilisé pour indiquer à React ce qu'il doit afficher pendant le chargement du composant paresseux (exemple : un spinner ou un message "Loading...").

Lazy loading est à utiliser principalement pour les composants volumineux ou rarement utilisés (comme des pages ou des widgets spécifiques).

**Code Splitting :**

Cette approche est généralement utilisée avec React.lazy pour diviser l'application en petits chunks (morceaux de code). Ces morceaux ne sont chargés que lorsque c'est nécessaire, améliorant ainsi les performances de l'application.

**Virtual DOM**

Le **Virtual DOM** est une abstraction du DOM réel, que React utilise pour optimiser les mises à jour de l'interface utilisateur. Plutôt que de modifier directement le DOM, ce qui est coûteux en performances, React utilise un Virtual DOM pour effectuer des comparaisons efficaces et minimiser les changements réels.

**Comprendre le fonctionnement :**

Lorsqu'un état ou des props changent, React crée une nouvelle représentation du DOM dans sa mémoire, appelée le Virtual DOM. React compare alors la nouvelle version du Virtual DOM avec l'ancienne version (diffing algorithm). Seuls les changements nécessaires sont appliqués au vrai DOM, minimisant ainsi les rendus inutiles et améliorant les performances.

**Optimisation des mises à jour :**

**Mémorisation des composants** : Utilise React.memo pour éviter les rendus inutiles de composants. Si un composant reçoit les mêmes props, React.memo empêchera React de le rendre à nouveau.

const MyComponent = React.memo((props) => {

// Composant qui ne se rerendera que si les props changent

});

**Utilisation de** useCallback et useMemo : Ces hooks permettent de mémoriser des fonctions et des valeurs calculées pour éviter de les recalculer ou de les recréer à chaque rendu, ce qui réduit les coûts en ressources :

const memorizedValue = useMemo(() => expensiveComputation(data), [data]);

const memorizedCallback = useCallback(() => doSomething(), []);

**Éviter les rendus inutiles**

En optimisant les structures de données et les composants pour éviter les mises à jour trop fréquentes. Par exemple, lorsqu’on utilise des tableaux ou des objets, il est important de s’assurer que leurs références (key) ne changent pas inutilement, ce qui pourrait forcer un nouveau rendu. (voir le confluence sur le mapping :

)

**Autres Bonnes Pratiques de Performance et d'Optimisation**

* **Debouncing et Throttling** : Lorsqu’on a besoin de gérer des événements comme le défilement ou les entrées utilisateurs, il est possible d’utiliser des techniques comme le **debouncing** ou le **throttling** pour limiter le nombre de fois qu'une fonction est exécutée.
  + **Debounce** : Diffère l'exécution d'une fonction jusqu'à ce qu'il n'y ait plus de nouvelles invocations pendant un certain délai.
  + **Throttle** : Limite l'exécution d'une fonction à une seule fois dans un intervalle de temps donné.
* **Gestion des états globaux** : Utilise des bibliothèques comme **Redux** pour centraliser et optimiser la gestion des états globaux. Voir le confluence sur ce sujet :

**Exemple d'optimisation avec ces pratiques**

import React, { useState, useEffect, useCallback, useMemo, Suspense } from 'react';

// Chargement paresseux d'un composant

const LazyComponent = React.lazy(() => import('./LazyComponent'));

const MyComponent: React.FC = () => {

const [data, setData] = useState<number[]>([]);

const [loading, setLoading] = useState(false);

const [error, setError] = useState<Error | null>(null);

// Fonction optimisée avec useCallback

const fetchData = useCallback(async () => {

setLoading(true);

setError(null);

try {

const response = await Promise.all([

fetch('/api/data1').then((res) => res.json()),

fetch('/api/data2').then((res) => res.json()),

]);

setData(response.flat());

} catch (err) {

setError(err as Error);

} finally {

setLoading(false);

}

}, []);

useEffect(() => {

fetchData();

}, [fetchData]);

// useMemo pour une opération coûteuse

const expensiveValue = useMemo(() => {

return data.reduce((acc, num) => acc + num, 0);

}, [data]);

if (loading) return <div>Loading...</div>;

if (error) return <div>Error: {error.message}</div>;

return (

<div>

<h1>Expensive Computation: {expensiveValue}</h1>

<Suspense fallback={<div>Loading Lazy Component...</div>}>

<LazyComponent />

</Suspense>

</div>

);

};

export default MyComponent;

Cet exemple :

* Charge les données en parallèle avec Promise.all.
* Gère les états de chargement et d'erreur de manière appropriée.
* Charge un composant de manière paresseuse avec React.lazy et Suspense.
* Optimise les fonctions avec useCallback et useMemo pour éviter des rendus ou des calculs inutiles.

Ces techniques combinées permettent de maximiser les performances tout en assurant une bonne gestion des ressources dans une application React TypeScript.

## Composants

**Code Splitting**

Le **Code Splitting** (ou découpage de code) est une technique qui permet de diviser le code JavaScript en morceaux plus petits, appelés "chunks", afin de charger uniquement ce qui est nécessaire au moment opportun. Cela améliore les **temps de chargement** initiaux, réduit la taille des bundles, et optimise les performances des applications, surtout celles qui sont grandes et complexes.

**Avantages :**

* **Amélioration des temps de chargement initiaux** : Moins de code à charger signifie que l'application se lance plus rapidement pour l'utilisateur.
* **Réduction de la consommation de mémoire** : Les parties du code inutilisées ne sont pas chargées en mémoire tant qu'elles ne sont pas nécessaires.
* **Optimisation de la bande passante** : Les utilisateurs ne téléchargent que les parties de l'application dont ils ont besoin à un moment donné, ce qui est utile pour les connexions lentes.

**A combiner avec :**

* **React.lazy et Suspense** :
  + React.lazy permet de charger des composants dynamiquement (à la demande), ce qui est idéal pour implémenter le **Code Splitting**. En combinant cela avec Suspense, on peut faire le choix de montrer un spinner ou un autre indicateur à l’utilisateur pendant que le composant est chargé.

import React, { Suspense } from 'react';

const LazyComponent = React.lazy(() => import('./LazyComponent'));

const MyComponent = () => {

return (

<div>

<h1>Welcome to My Application</h1>

<Suspense fallback={<div>Loading...</div>}>

<LazyComponent />

</Suspense>

</div>

);

};

export default MyComponent;

Dans cet exemple, LazyComponent est chargé uniquement lorsqu'il est nécessaire (lorsqu'il est monté dans le DOM), et Suspense gère l'affichage d'un message de chargement pendant ce temps.

**Outils supplémentaires :**

* **Bundle Splitting** : Permet de diviser le bundle de ton projet en plusieurs fichiers JS plus petits pour des chargements plus optimisés.

**Atomic Structure (Structure Atomique)**

La **conception atomique** (ou **Atomic Design**) est une approche de conception de systèmes d'interface utilisateur qui vise à organiser les composants en unités logiques et réutilisables. Ce concept a été popularisé par **Brad Frost** et aide à structurer les composants de manière hiérarchique et modulaire, facilitant ainsi la réutilisabilité, la maintenabilité, et l'évolutivité du code.

**Principes du Design Atomique :**

La structure atomique divise les interfaces en cinq niveaux hiérarchiques, allant des plus petits composants (les atomes) aux ensembles plus complexes (les pages) :

**Atoms (Atomes) :**

Les éléments les plus basiques de l'interface, comme des **boutons**, **champs de texte**, **libellés**, ou **icônes**. Ils sont simples et ne contiennent généralement pas de logique complexe.

const Button: React.FC<ButtonProps> = ({ children }) => {

return <button>{children}</button>;

};

**Molecules (Molécules) :**

Combinaisons simples d'**atomes** qui forment des composants fonctionnels autonomes. Par exemple, un champ de formulaire avec un label et un champ d'entrée.

const InputWithLabel: React.FC<{ label: string; value: string; onChange: () => void; }> = ({ label, value, onChange }) => {

return (

<div>

<label>{label}</label>

<input value={value} onChange={onChange} />

</div>

);

};

**Organisms (Organismes) :**

Combinaisons de **molécules** qui forment des sections distinctes de l'interface, comme des barres de navigation, des formulaires de connexion, ou des cartes de contenu. Ces composants commencent à avoir une logique plus complexe.

const LoginForm: React.FC = () => {

return (

<form>

<InputWithLabel label="Username" value="" onChange={() => {}} />

<InputWithLabel label="Password" value="" onChange={() => {}} />

<Button>Submit</Button>

</form>

);

};

**Templates (Templates) :**

Ce sont des structures d'agencement plus larges composées d'**organismes**. Elles définissent la mise en page sans inclure les données spécifiques. Par exemple, une mise en page de tableau de bord.

const DashboardTemplate: React.FC = ({ children }) => {

return (

<div className="dashboard">

<Header />

<Sidebar />

<main>{children}</main>

</div>

);

};

**Pages (Pages) :**

Les **Pages** combinent des **templates** avec des données spécifiques pour rendre une vue complète. Ce sont les composants les plus complexes et englobent toute l'interface visible par l'utilisateur final.

const DashboardPage: React.FC = () => {

return (

<DashboardTemplate>

<ContentSection />

</DashboardTemplate>

);

};

**Avantages de la Structure Atomique :**

* **Réutilisabilité** : Les **atomes** et **molécules** sont conçus pour être réutilisables à travers toute l'application.
* **Cohérence** : Le respect d'une structure atomique permet d'assurer une interface utilisateur cohérente.
* **Facilité de maintenance** : Le découpage des composants en petites unités facilite leur maintenance et évolution, chaque élément étant isolé et ayant une responsabilité unique.
* **Modularité** : La conception atomique encourage une architecture modulaire, ce qui est bénéfique pour les grandes applications.

**JOY MUI**

Dans le cadre de notre SI et de notre conception du frontend, nous utilisons la librairie CSS JOY MUI mise en place par Arnaud. Dans le cadre de cette librairie, nous avons déjà des atomes préconçus (les boutons, les textes, les inputs, etc), mais nous avons aussi quelques Molécules (les cards par exemples). Nous partons donc de cette base et serons développer dans chaque projet que des molécules, organismes, templates et pages.

**Ressources pour Approfondir :**

Pour une meilleure compréhension des **Atomic Design Systems**, je recommande de lire l'article suivant : [Building Design Systems with Atomic Design](https://bootcamp.uxdesign.cc/building-design-systems-with-atomic-design-fd21e86f34c5). Cet article explique en détail comment appliquer ces principes dans le cadre de la création de **Design Systems** pour des interfaces complexes.

**Exemple Complet : Atomic Design et Code Splitting**

import React, { Suspense } from 'react';

// Atome : Bouton de base

const Button: React.FC<{ onClick: () => void }> = ({ onClick, children }) => {

return <button onClick={onClick}>{children}</button>;

};

// Molécule : Champ d'entrée avec label

const InputWithLabel: React.FC<{ label: string; value: string; onChange: () => void; }> = ({ label, value, onChange }) => {

return (

<div>

<label>{label}</label>

<input value={value} onChange={onChange} />

</div>

);

};

// Organisme : Formulaire de connexion

const LoginForm: React.FC = () => {

return (

<form>

<InputWithLabel label="Username" value="" onChange={() => {}} />

<InputWithLabel label="Password" value="" onChange={() => {}} />

<Button onClick={() => {}}>Submit</Button>

</form>

);

};

// Template : Page de connexion

const LoginTemplate: React.FC = ({ children }) => {

return (

<div>

<h1>Login</h1>

{children}

</div>

);

};

// Page : Chargement paresseux du composant de connexion

const LoginPage = React.lazy(() => import('./LoginForm'));

const App: React.FC = () => {

return (

<Suspense fallback={<div>Loading...</div>}>

<LoginTemplate>

<LoginPage />

</LoginTemplate>

</Suspense>

);

};

export default App;

## Redux

npm install @reduxjs/toolkit react-redux

**Introduction**

Redux est une bibliothèque de gestion d'état pour les applications JavaScript. Il est particulièrement utile dans les applications React pour gérer l'état global de manière prévisible et centralisée. Voici les concepts clés de Redux :

1. **Store** : L'objet unique qui contient l'état global de l'application.
2. **Action** : Un objet décrivant une intention de changement d'état.
3. **Reducer** : Une fonction pure qui prend l'état actuel et une action, et retourne un nouvel état.
4. **Dispatch** : Une méthode pour envoyer une action au store.
5. **Selectors** : Des fonctions pour extraire des morceaux de l'état global.

**Structure de fichier**

Nous aurons dans le dossier “utils” un sous dossier “redux”. Ce même dossier contient un premier dossier store qui va contenir un seul fichier : store.ts. Ce store.ts contient

src/

├── utils/

│ ├── redux

│ ├── store

│ └── store.ts

│ ├── reducer

│ ├── reducer1.ts

│ ├── ...

│ └── reducerXX.ts

│ ├── actions

│ ├── actionsReducer1.ts

│ ├── ...

│ └── actionsReducerXX.ts

│ └── rootReducer.ts

**Reducers**

Ce dossier contient plusieurs fichiers reducers. Généralement, ils porteront un nom de base référant à ce qu’ils vont contenir comme données. (Ex. Dans le cas de données utilisateur on pourrait avoir pour base ‘User’ ou dans le cas de mails échangés on pourrait avoir ‘Mails’). Puis cette base sera suivie de la dénomination ‘Reducer’ (Ex. UserReducer.ts ou MailsReducer.ts). Ces fichiers définissent les reducers qui traitent les actions pour mettre à jour l'état utilisateur.

En admettant que l’on souhaite faire un reducer pour retenir le nom, le prénom et l'âge d’un utilisateur. On aura donc un fichier UserReducer définissant les données à stocker et les actions qui permettront de mettre à jour l’état utilisateur. Voici ce que pourrait donner ce fichier reducer :

import { Reducer } from "@reduxjs/toolkit";

import { UserType } from "../../../components/Interfaces/UserType";

const initialState: UserType = {

name: "",

firstName: "",

age: 0

};

const userReducer: Reducer<UserType, any> = (state = initialState, action) => {

switch (action.type) {

case "UPDATE\_USER\_NAME":

return { ...state, name: action.payload };

case "UPDATE\_USER\_FIRSTNAME":

return { ...state, firstName: action.payload };

case "UPDATE\_USER\_AGE":

return { ...state, age: action.payload };

case "UPDATE\_USER\_NAME\_AND\_FIRSTNAME":

return { ...state, name: action.payload.name, firstName: action.payload.firstName };

case "RESET\_USER":

return initialState;

case "UPDATE\_USER":

return { ...action.payload };

default:

return state;

}

};

export default userReducer;

Vous pourrez constater qu’on a trois actions qui permettent de modifier un paramètre à la fois (name, firstname et age), mais qu’on a trois actions plus globales :

* **UPDATE\_USER\_NAME\_AND\_FIRSTNAME** va nous permettre de modifier deux paramètres d’un coup, sans pour autant mettre à jour l’ensemble de notre reducer. Ce n’est pas une action très utilisée, mais il est bien de l’avoir en tête.
* **RESET\_USER** va nous permettre de remettre à zéro notre reducer en lui réinjectant les données de l’initialState. On pourrait utiliser ce reset dans le cadre d’un formulaire. Une fois le formulaire complété et envoyer, cette action pourrait nous permettre de vider le formulaire pour pouvoir le remplir à nouveau
* **UPDATE\_USER** va nous permettre de modifier l’ensemble de notre reducer en lui donnant toutes les données d’un coup. On pourrait utiliser cet update général dans le cadre d’une édition d’un contenu. Au lieu de faire un dispatch par propriété, on n’en fait qu’un seul au chargement de la page

Au minimum on aura toujours une action par propriété et une action pour tout mettre à jour d’un coup. Pour le reste cela dépend de l’utilisation.

Pour éviter les doublons parmi les actions, le type (Ex. ‘UPDATE\_…’) aura la forme suivante :

‘UPDATE\_' + la base du nom du fichier (ici USER) + '\_' + la donnée modifiée

Dans nos exemples on a donc :

‘UPDATE\_USER\_FIRSTNAME’ et ‘UPDATE\_USER\_NAME’

**Actions**

Ce dossier contient plusieurs fichiers d’actions. Généralement ils porteront le même nom de base que dans le dossier reducer sauf qu’on viendra préciser “action” à la fin de celui-ci. (Exemple : UserReducer.ts donnera UserActions.ts ; MailsReducer.ts donnera MailsActions.ts, etc). Ces fichiers définissent les actions qui seront utilisées pour mettre à jour l'état du reducer auquel ils sont rattachés.

En admettant la même base que précédemment, on aura donc UserActions.ts. Voici ce que pourrait donner le fichier d’action :

import { UserType } from "../../../components/Interfaces/UserType";

export const updateName = (name: string): any => ({

type: "UPDATE\_USER\_NAME",

payload: name,

});

export const updateFirstname = (firstname: string): any => ({

type: "UPDATE\_USER\_FIRSTNAME",

payload: firstname

});

export const updateUserAge = (age: number): any => ({

type: "UPDATE\_USER\_AGE",

payload: age,

});

export const updateUserNameAndFirstname = (name: string, firstName: string): any => ({

type: "UPDATE\_USER\_NAME\_AND\_FIRSTNAME",

payload: { name, firstName },

});

export const resetUser = (): any => ({

type: "RESET\_USER",

});

export const updateUser = (user: UserType): any => ({

type: "UPDATE\_USER",

payload: user,

});

Il est primordial que le type d’une action (Ex. “UPDATE\_USER\_NAME”) corresponde à ce qui est indiqué dans le switch du reducer. Dans le cas contraire la donnée ne sera pas mise à jour puisque les actions ne seront pas reconnues !

**Store.ts**

Ce fichier sert à configurer et créer le store Redux pour votre application. Il importe les différentes slices (portions de l'état gérées par des reducers spécifiques) et les combine en un seul store global. Voici un exemple de fichier store.ts :

import { configureStore } from "@reduxjs/toolkit";

import { userReducer } from "../reducers/UserReducer";

import { reducerN2 } from "../reducers/reducerN2";

const store = configureStore({

reducer: {

user: userReducer.reducer,

// reducerN2: reducerN2,

},

});

export default store;

export type RootState = ReturnType<typeof store.getState>;

export type AppDispatch = typeof store.dispatch;

* **configureStore** : Importé de @reduxjs/toolkit, cette fonction simplifie la configuration du store Redux en incluant déjà les middlewares et les outils de développement appropriés.
* Les **slices** sont importés depuis les fichiers de reducers correspondants. Chaque slice contient un reducer et potentiellement des actions et des selectors associés à une fonctionnalité spécifique de votre application.
* **store** : C'est l'instance du store Redux configurée avec les reducers de chaque slice.
* **reducer** : Cette option de configureStore combine tous les reducers des slices importées en un seul objet. Chaque clé de cet objet correspond à une slice de l'état global.
* **export default store** : Cette ligne exporte l'instance du store configuré pour être utilisée dans toute l'application.
* **RootState** : Un type TypeScript qui représente la forme de l'état global de l'application. ReturnType<typeof store.getState> infère le type de retour de store.getState(), qui est l'état global.
* **AppDispatch** : Un type TypeScript pour le dispatch de Redux. typeof store.dispatch capture le type de la fonction dispatch du store. Cela permet de typer correctement les hooks useDispatch ou d'autres utilisations de dispatch dans votre application.

**Utiliser les données**

Pour utiliser les données des reducer il faut que le fichier store soit créé, puis pour les appeler dans nos fichiers tsx, nous utiliserons selector de la façon suivante :

import { useDispatch } from "react-redux";

import { updateFirstname } from "../../utils/redux/actions/UserActions"

export default function App() {

const user = useSelector((state: RootState) => state.user)

const printMyData = () => {

console.log(user.name)

console.log(user.firstName)

}

}

**Mettre à jour les données**

Pour mettre à jour les données nous utiliserons donc les actions établies précédemment. Pour utiliser ces actions dans nos fichiers tsx, nous utiliserons dispatch de la façon suivante :

import { useDispatch } from "react-redux";

import { updateFirstname } from "../../utils/redux/actions/UserActions"

export default function App() {

const dispatch = useDispatch();

const updateMyData = () => {

dispatch(updateFirstname("john"))

}

}

**Exemple concret**

Imaginons une situation où une page (Page) a un composant formulaire (Form), et ce formulaire est composé de plusieurs sous-composants (Part1, Part2, etc.). On doit passer des fonctions de mise à jour de données entre ces composants en utilisant props.

**Sans redux**

// Page.tsx

import React, { useState } from 'react';

import Form from './Form';

const Page = () => {

const [data, setData] = useState({ part1: '', part2: '' });

const refreshData = () => {

// Logic to refresh the data

console.log("Data refreshed");

};

return <Form data={data} setData={setData} refreshData={refreshData} />;

};

// Form.tsx

import React from 'react';

import Part1 from './Part1';

import Part2 from './Part2';

const Form = ({ data, setData, refreshData }) => {

return (

<div>

<Part1 value={data.part1} setValue={(val) => setData({ ...data, part1: val })} />

<Part2 value={data.part2} setValue={(val) => setData({ ...data, part2: val })} />

<button onClick={refreshData}>Refresh</button>

</div>

);

};

// Part1.tsx

import React from 'react';

const Part1 = ({ value, setValue }) => {

return <input value={value} onChange={(e) => setValue(e.target.value)} />;

};

// Part2.tsx

import React from 'react';

const Part2 = ({ value, setValue }) => {

return <input value={value} onChange={(e) => setValue(e.target.value)} />;

};

**Problèmes** :

* La gestion des états et des fonctions est partagée à travers de nombreux composants via props.
* Ce type de gestion de données devient rapidement difficile à maintenir à mesure que l'application grandit.

**Avec redux**

// store.ts (très condensé, on le séparera en trois fichiers pour plus de maintenabilité en réalité)

import { createSlice, configureStore } from '@reduxjs/toolkit';

const formSlice = createSlice({

name: 'form',

initialState: { part1: '', part2: '' },

reducers: {

setPart1: (state, action) => { state.part1 = action.payload; },

setPart2: (state, action) => { state.part2 = action.payload; },

refresh: () => { console.log("Data refreshed"); },

},

});

export const { setPart1, setPart2, refresh } = formSlice.actions;

export const store = configureStore({ reducer: { form: formSlice.reducer } });

// Page.tsx

import React from 'react';

import { useDispatch } from 'react-redux';

import { refresh } from './store';

import Form from './Form';

const Page = () => {

const dispatch = useDispatch();

return (

<div>

<Form />

<button onClick={() => dispatch(refresh())}>Refresh</button>

</div>

);

};

// Form.tsx

import React from 'react';

import Part1 from './Part1';

import Part2 from './Part2';

const Form = () => {

return (

<div>

<Part1 />

<Part2 />

</div>

);

};

// Part1.tsx

import React from 'react';

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

import { setPart1 } from './store';

const Part1 = () => {

const value = useSelector((state) => state.form.part1);

const dispatch = useDispatch();

return <input value={value} onChange={(e) => dispatch(setPart1(e.target.value))} />;

};

// Part2.tsx

import React from 'react';

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

import { setPart2 } from './store';

const Part2 = () => {

const value = useSelector((state) => state.form.part2);

const dispatch = useDispatch();

return <input value={value} onChange={(e) => dispatch(setPart2(e.target.value))} />;

};

**Avantages avec Redux** :

* **Centralisation de l'état** : Tous les états et leurs mises à jour sont gérés via Redux, évitant les passages excessifs de props.
* **Réduction de la complexité** : Les composants enfants n'ont plus besoin de recevoir les états ou les setters par props car ils peuvent directement interagir avec le store.
* **Facilité de maintenance** : Il est plus facile d'ajouter de nouveaux composants ou de nouvelles fonctionnalités sans avoir à modifier les composants parents pour leur transmettre des données ou des fonctions.

Avec Redux, la logique est beaucoup plus centralisée et organisée, ce qui facilite la maintenance à long terme de l'application.

## Router React

<https://reactrouter.com/home>

**Introduction**

react-router-dom est une bibliothèque incontournable pour la gestion de la navigation dans les applications React. Elle permet de créer des applications web monopage (Single Page Application, SPA) avec une expérience utilisateur fluide grâce à une navigation dynamique, sans recharger la page entière.

**Pourquoi utiliser react-router-dom ?**

Au lieu de charger une nouvelle page pour chaque URL, react-router-dom met à jour le contenu de la page actuelle, rendant l'expérience utilisateur plus fluide et rapide. Cette bibliothèque permet également de structurer une application en plusieurs pages ou vues, tout en gardant l'aspect d'une application React basée sur des composants. Avec react-router-dom, il devient simple de gérer des routes complexes, telles que des routes imbriquées, des paramètres dynamiques dans les URLs, et des redirections. Enfin, ke routage est défini directement dans le RSX, rendant le code plus lisible et aligné avec l'architecture React.

**Concepts de base**

* **Routes** : Chaque URL de votre application correspond à une route spécifique. Le composant <Route> permet de définir quelle vue est rendue pour une URL donnée.
* **Router** : Le composant <BrowserRouter> ou <HashRouter> enveloppe l'application et permet à react-router-dom de surveiller l'URL et de gérer les changements de navigation.
* **Link et NavLink** : Le composant <Link> permet de naviguer entre les différentes routes sans recharger la page. <NavLink> est une version améliorée de <Link> qui peut être utilisé pour appliquer des styles spécifiques lorsque le lien est actif.
* **useNavigate** : Un hook permettant de programmer des redirections.
* **useParams** : Un hook qui permet d'extraire les paramètres de l'URL (par exemple, un identifiant de produit dans une URL /product/:id).

**Exemple basique**

import React from 'react';

import { BrowserRouter as Router, Route, Routes, Link, useParams } from 'react-router-dom';

const Home: React.FC = () => <h2>Accueil</h2>;

const About: React.FC = () => <h2>À propos</h2>;

const Product: React.FC = () => {

// On récupère l'ID passé en paramètre dans l'URL

const { id } = useParams<{ id: string }>();

return <h2>Détails du produit : {id}</h2>;

};

const App: React.FC = () => {

return (

<Router>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

<Route path="/product/:id" element={<Product />} />

</Routes>

</Router>

);

};

## React-Query

npm install @tanstack/react-query  
npm install --save-dev @types/react-query

**Introduction**

React Query est une bibliothèque essentielle pour la gestion des requêtes de données côté client dans les applications React. Elle simplifie considérablement la gestion des états asynchrones, comme les requêtes HTTP ou les WebSocket, en offrant des fonctionnalités intégrées de mise en cache, de synchronisation, de pagination, de gestion des erreurs et d'invalidation des données. Son intégration avec TypeScript permet de tirer parti du typage statique pour des requêtes plus robustes, améliorant ainsi l'expérience de développement.

**Pourquoi utiliser React Query ?**

Elle élimine la complexité d'utiliser manuellement useState, useEffect et fetch pour gérer les appels API, en encapsulant la logique dans des hooks prêts à l'emploi. React Query met en cache les données et réutilise les résultats de requêtes précédentes, ce qui améliore les performances de l'application. Elle offre également des stratégies avancées de rafraîchissement automatique des données, comme le refetching en arrière-plan, les invalidations de cache et la synchronisation avec le serveur. Enfin, grâce à TypeScript, la définition des types pour les réponses de l'API permet d'éviter les erreurs communes liées aux formats inattendus des données.

**Concepts de Base**

* **Query** : Une "query" est une requête côté client pour récupérer des données à partir d'une source externe (API, base de données, etc.). Elle est gérée par le hook useQuery.
* **Mutation** : Une "mutation" est une opération qui modifie les données côté serveur (création, mise à jour ou suppression). Elle est gérée par le hook useMutation.
* **Query Client** : C'est l'objet central de React Query qui gère l'état global des requêtes dans l'application. Il est généralement configuré à la racine de l'application à l'aide de QueryClientProvider.

**Exemple basique**

import React from 'react';

import { useQuery, QueryClient, QueryClientProvider } from '@tanstack/react-query';

// Création d'un Query Client

const queryClient = new QueryClient();

// Définition du type pour les données

interface User {

id: number;

name: string;

username: string;

email: string;

}

// Fonction pour récupérer les utilisateurs via une API

const fetchUsers = async (): Promise<User[]> => {

const response = await fetch('https://jsonplaceholder.typicode.com/users');

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json();

};

const UsersList: React.FC = () => {

// Utilisation du hook useQuery pour récupérer les données

const { data, error, isLoading } = useQuery<User[]>(['users'], fetchUsers);

if (isLoading) return <div>Chargement...</div>;

if (error) return <div>Une erreur est survenue : {error.message}</div>;

return (

<ul>

{data?.map(user => (

<li key={user.id}>{user.name} - {user.email}</li>

))}

</ul>

);

};

const App: React.FC = () => {

return (

<QueryClientProvider client={queryClient}>

<UsersList />

</QueryClientProvider>

);

};

export default App;

Explication du Code

**Query Client** : Un QueryClient est instancié et passé au QueryClientProvider, qui enveloppe toute l'application pour permettre à React Query de gérer globalement les états de requêtes.

**useQuery** : Le hook useQuery est utilisé pour exécuter une requête (ici pour récupérer des utilisateurs). La requête est identifiée par la clé 'users' et utilise la fonction fetchUsers pour obtenir les données.

**Gestion des États** : Les états asynchrones comme isLoading, error et data sont directement exposés par useQuery, simplifiant ainsi la gestion des erreurs et des chargements.

**Typage avec Typescript**

L’un des avantages majeurs de React Query avec TypeScript est la possibilité de typer explicitement les données échangées avec le backend. Cela permet de garantir la cohérence des données tout au long de l’application.

const { data, error } = useQuery<User[]>('users', fetchUsers);

Dans cet exemple, le hook useQuery est typé avec User[], ce qui signifie que TypeScript s’attend à recevoir un tableau d’utilisateurs comme réponse. Si l'API renvoie autre chose, TypeScript signalera une erreur.

**Concepts avancés**

**Utilisation des mutations**

Les mutations sont utilisées pour effectuer des opérations d'écriture, telles que l'ajout ou la suppression de données sur le serveur. Voici un exemple d'utilisation de useMutation pour ajouter un nouvel utilisateur :

import React from 'react';

import { useMutation, useQueryClient } from '@tanstack/react-query';

// Fonction pour ajouter un utilisateur

const addUser = async (newUser: Omit<User, 'id'>): Promise<User> => {

const response = await fetch('https://jsonplaceholder.typicode.com/users', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify(newUser),

});

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json();

};

const AddUserForm: React.FC = () => {

const queryClient = useQueryClient();

const mutation = useMutation(addUser, {

onSuccess: () => {

// Invalidation de la requête pour rafraîchir la liste des utilisateurs

queryClient.invalidateQueries(['users']);

},

});

const handleSubmit = () => {

mutation.mutate({

name: 'John Doe',

username: 'johndoe',

email: 'johndoe@example.com',

});

};

if (mutation.isLoading) return <div>Envoi...</div>;

if (mutation.isError) return <div>Une erreur est survenue : {mutation.error.message}</div>;

if (mutation.isSuccess) return <div>Utilisateur ajouté avec succès</div>;

return <button onClick={handleSubmit}>Ajouter un utilisateur</button>;

};

**Explication du Code**

**useMutation** : Le hook useMutation est utilisé pour déclencher une action qui modifie les données (ici l'ajout d'un utilisateur).

**Gestion des États** : Comme pour useQuery, useMutation expose des états comme isLoading, isError, et isSuccess, ce qui facilite la gestion du retour d'une mutation.

**Invalidation de Cache**

React Query permet d'invalider les caches pour garantir que les données soient mises à jour après certaines actions. Par exemple, après l'ajout d'un nouvel utilisateur, vous pouvez invalider la liste des utilisateurs :

const mutation = useMutation(addUser, {

onSuccess: () => {

queryClient.invalidateQueries(['users']);

},

});

**Refetch Automatique**

Vous pouvez configurer un rafraîchissement automatique des données :

const { data } = useQuery<User[]>('users', fetchUsers, {

refetchInterval: 5000, // Rafraîchit les données toutes les 5 secondes

});

**Pagination**

La gestion de la pagination devient également triviale avec React Query. Voici un exemple simplifié :

const fetchPaginatedUsers = async (page: number): Promise<User[]> => {

const response = await fetch(`https://jsonplaceholder.typicode.com/users?\_page=${page}&\_limit=10`);

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json();

};

const { data, isFetching } = useQuery(['users', page], () => fetchPaginatedUsers(page));

**Bonnes pratiques d’utilisation**

**Typage Strict** : Toujours typer les données échangées via les requêtes et mutations. Cela permet d’éviter les erreurs dues à des réponses inattendues de l’API.

**Clé de Query Unique** : Utilisez des clés uniques pour chaque requête afin d'éviter les conflits de cache.

**Gestion des Erreurs** : Utilisez les options onError pour gérer globalement les erreurs, en particulier pour des comportements comme des notifications d'erreurs.

**Invalidation Automatique** : Invalidez les données après une mutation réussie pour garantir que les vues affichent des données fraîches.

**Refetch on Window Focus** : Utilisez la configuration de refetchOnWindowFocus pour rafraîchir les données lorsque l'utilisateur revient sur votre application après l'avoir quittée.