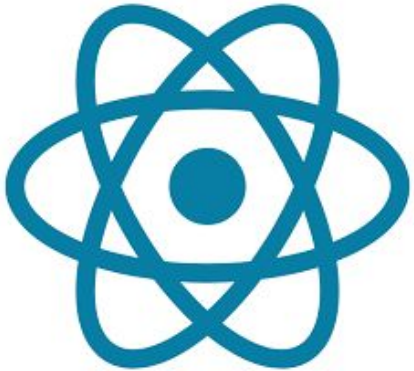
# React



## Troisième partie

# Rappels React - bases

React est une **bibliothèque JavaScript** pour le développement de SPA **réactives**.

- Repose sur une hiérarchie de **composants**
- Composants : prennent des valeurs d'entrée via des **props**
- Propagation des props d'un composant parent à ses enfants
- Remontée d'information via des **callbacks**
- Changement de props ou de state : **re-rendu des composants concernés** (et *uniquement* ceux-là)

# *Hooks avancés*

# Hooks : règles à suivre

Les hooks présentent de nombreux avantages, et sont aujourd'hui très largement répandus au sein de l'écosystème React. Cependant, leur utilisation est soumise à certaines règles :

- Réservés à des composants React uniquement, et *pas* à des **fonctions javascript**
- Ne peuvent *pas être utilisés* dans des **boucles, structures conditionnelles**, ou au sein de **fonctions imbriquées**

En effet, **l'ordre** dans lequel les hooks sont appelés **doit être préservé** entre 2 renders successifs d'un composant (ce qui n'est pas garanti si on les utilise dans des boucles ou structures conditionnelles).

# Règles sur les hooks

```
import { useState, useEffect } from "react";

function DataComponent(props) {
      const [numOne, setNumOne] = useState(0);
      const [numTwo, setNumTwo] = useState(0);

      if (props.bindNumbers) {
      useEffect(function updateNumTwo() {
            setNumTwo(numTwo + 1);
            }, [numOne])
      }

      const [numThree, setNumThree] = useState(0);
      return (
      <div>
            <button onClick={() => setNumOne(numOne + 1)}>
                  Click me
            </button>
            <p> {numOne}, {numTwo}, {numThree} </p>
      </div>
      );
};
```

```
// premier rendu du composant

useState('numOne')
useState('numTwo')
useEffect(updateNumTwo)
useState('numThree')


// deuxième rendu du composant

useState('numOne')
useState('numTwo')
// useEffect('updateNumTwo')
useState('numThree')


// L'ordre des hooks n'est plus maintenu !
```

# Règles sur les hooks - code correct

```
import { useState, useEffect } from "react";

function DataComponent(props) {
      const [numOne, setNumOne] = useState(0);
      const [numTwo, setNumTwo] = useState(0);

      useEffect(function updateNumTwo() {
            if (props.bindNumbers) {
                  setNumTwo(numTwo + 1);
                  }
            }, [numOne])

      const [numThree, setNumThree] = useState(0);
      return (
      <div>
            <button onClick={() => setNumOne(numOne + 1)}>
                  Click me
            </button>
            <p> {numOne}, {numTwo}, {numThree} </p>
      </div>
      );
};
```

# Hooks personnalisés : principes

Pour l'instant, nous avons vu les hooks standards de React : useState, useEffect, useContext, useRef…

*(et plus tard dans le cours, 2 hooks venant d'une librairie externe)*

Mais il est également possible de créer ses **propres hooks** : permet de réutiliser de la logique sur les données.

*Syntaxe* : un hook commence systématiquement par *use* suivi d'une majuscule

# Hooks personnalisés : exemple

*Hook permettant de mettre à jour la position du curseur en continu*

```
import { useState, useEffect } from "react";

function useCursorPosition() {
      const [mousePosition, setMousePosition] = useState({ x: 0, y: 0 });
  }

      useEffect(() => {
      const updateMousePosition = (evt) => {
            setMousePosition( { x: evt.clientX, y: evt.clientY } )
                  }
      window.addEventListener( 'mousemove', updateMousePosition )
      return () => { window.removeEventListener( 'mousemove', updateMousePosition ) }
      }, [])

      return mousePosition;
};

export default useCursorPosition;
```

# *Requêtes asynchrones*

# Application Programming Interface

Dans plusieurs cas, on aimerait pouvoir *connecter* plusieurs services les uns avec les autres .

- Nécessité de passer par une source externe pour stocker un grand nombre de données
- Bénéficier de fonctionnalités d'autres applications

API : *Application Programming Interface*

- **Protocoles+règles** permettant à des applications **d'échanger** des données, des fonctionnalités, des méthodes et fonctions …
- Exemples d'API :
  - Composants React
  - Web service (par exemple, l'API YouTube permet de rechercher des vidéos, et intégrer certaines fonctionnalités dans une autre application)

# Requêtes asynchrones

Lorsqu'une application souhaite échanger des données avec un serveur :

- L'application va envoyer une *requête* : action que l'on souhaite réaliser sur une ressource spécifique (exemple HTTP : GET pour récupérer la ressource, POST pour l'envoyer)
- Pour garantir une application *réactive* : on ne veut pas bloquer l'exécution en attendant la réponse du serveur
- Programmation *synchrone* : exécution *séquentielle* des tâches les unes après les autres
- Programmation **asynchrone** : plusieurs tâches effectuées *simultanément* sans bloquer l'exécution des autres
- Exemple Web : l'utilisateur peut continuer à *interagir* avec la page pendant qu'une requête API est en cours de traitement

# React asynchrone

Intégration de fonctions asynchrones dans des composants React :

- très similaire à javascript
- **async** permet de déclarer une fonction asynchrone
- **Promise** : valeur ou opération qui sera (peut être) disponible/effectuée dans le futur
- 3 états : *pending* (en attente), *fulfilled* (opération effectuée) ou *rejected* (l'opération a échoué)
- **await** permet d'attendre le résultat d'une **Promise** (fulfilled ou rejected) avant de passer à l'opération suivante
- uniquement utilisable au sein une fonction asynchrone
- **fetch** : interface permettant d'effectuer des requêtes http à des API externes

# React asynchrone - exemple 1

```
async function fetchData() {
  try {
    const response = await fetch('https://...=fr')
    const result = await response.json()
    return result
  } catch (err) {
    console.error(err)
    return {}
  }
}
```

```
async function logData() {
    const result = await fetchData()
    console.result(result)
}
```

# React asynchrone et composants

On aimerait pouvoir utiliser le résultat d'une fonction asynchrone dans un composant React

- Exemple : récupérer un profil d'utilisateur depuis un serveur et mettre à jour la page web
- Mais on veut pas bloquer l'UI en attendant le résultat !
- On va donc passer par le hook **useEffect**

**Attention** : le premier argument de useEffect **ne peut pas** être une fonction asynchrone, il doit renvoyer soit une fonction de nettoyage, soit rien du tout.

- Première approche : faire la requête asynchrone avec *useEffect*
- Deuxième approche : composants *serveurs* (nouveau paradigme React)

# React asynchrone et composants

*Requête asynchrone avec useEffect*

```
import { useState, useEffect } from “react”;

function DataComponent() {
      const [data, setData] = useState(“”);
  }
      useEffect(() => {
            async function fetchData() {
                  const response = await fetch('https://randomuser.me/api/’);
                  const result = await response.json();
                  setData(result.results.name.first);
            }
      }, [])

      return (
      <div>
            <p> {data} </p>
      </div>
      );
};
```

# React asynchrone et composants

*Requête asynchrone avec composants serveurs*

- Nouveau paradigme de React, introduit dans Next.js 13.4 (framework React)
- Dans un **composant serveur**, le rendu est effectué côté *serveur* avant d'être envoyé au client
- Cela signifie que ces composants ne sont jamais re-rendus : incompatibles avec *useState*, *useEffect*
- **Avantages** : simplicité d'écriture, rendu beaucoup plus performant
- **Attention** : au sein de ce paradigme, tous les composants seront des composants serveurs *par défaut*
- Composants clients : le rendu est partagé entre le client et le serveur : permet d'utiliser les hooks *useState*, *useEffect*

```
async function ServerComponent() {
      const data = await fetch('https://...=fr');
      return (
      <div>
            <p> {data} </p>
      </div>
      );
};
```

```
'use client';
import {useState} from "react";

function ClientComponent() {
      const [data, setData] = useState('0') ;
      return (
      <div>
            <p> {data} </p>
      </div>
      );
};
```

# React asynchrone : POST

Jusqu'à présent, nous avons vu des requêtes permettant de *récupérer* des données.

- **fetch** : méthode globale en javascript, permet un accès asynchrone à des ressources sur un serveur
- Fonctionne avec les méthodes http : GET est la méthode par défaut => permet de *récupérer* des données
- Compatible avec les autres méthodes http : POST pour envoyer des données, DELETE pour supprimer…
- Ces options sont spécifiées dans le 2ème argument de fetch :

```
fetch('https://randomuser.me/api/',
      {
      method: 'POST',
      headers: {'content-type': 'application/json'},
      body: JSON.stringify({user:userData})
      }
)
```

# React asynchrone : important

Certaines API demandent de s'identifier avec une clé d'authentification :

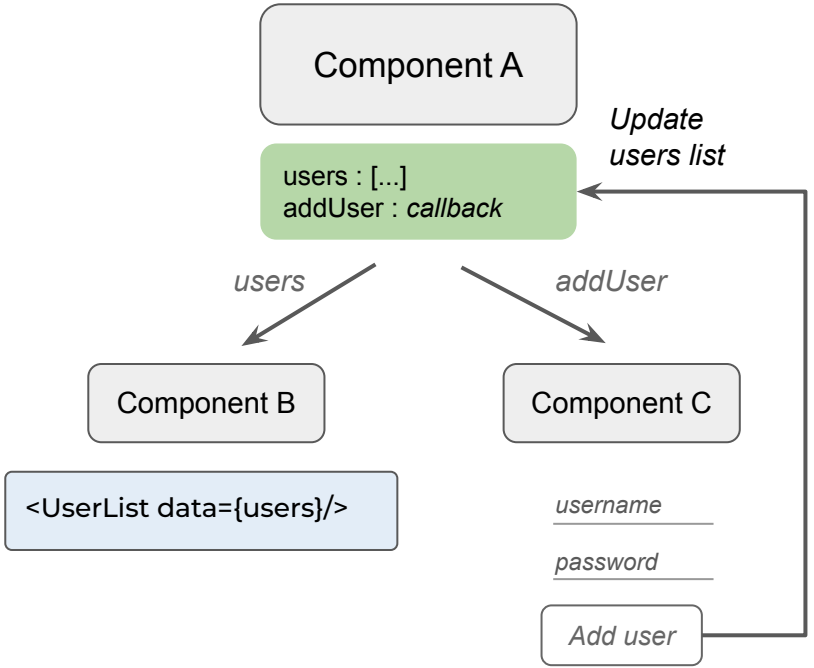
- *Exemple* : Token OAuth pour l'API Youtube
- Ces données sont à renseigner dans le 2ème argument de fetch (se référer à la documentation de l'API concernée)

**Ne jamais inclure ces données dans le code public** de votre page web : ne surtout pas les renseigner "en dur" dans votre code; si vous choisissez de les stocker dans un fichier JSON : l'inclure dans le gitignore

# *Redux*

# Gestion de states

- Rappel : les composants **parents** passent leur *state* aux composants **enfants** par *props*
- Les composants enfants peuvent faire remonter des changements de *state* aux parents grâce à des fonctions de **callbacks**
- Si 2 composants doivent partager un *state* : il faut le déclarer dans le composant parent contenant les 2 enfants le plus proche
- Application avancée : la gestion d'états peut devenir complexe : *props drilling*
- *useContext* : permet d'accéder et de changer des variables sans passer par les props…
- Mais difficile de gérer des logiques de states complexes



# Redux

**Redux** est une librairie javascript conçue pour la gestion de states à l'échelle *globale* d'un projet.

Redux est constitué de 3 éléments principaux :

- **store** : contient le state de l'application, il est accessible **globalement** grâce à un **store provider**
- chaque composant présent dans le provider peut **accéder** au store…
- … et le **modifier** en suivant des règles strictes : **actions**
- le store ne peut être que selon les actions définies. Chaque action appelle une fonction calculant le nouvel état du store : **reducers**

Ces éléments permettent une gestion **globale, contrôlée, et modulaire** des états d'une application.

**Exemple** : Application TodoList

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

# Redux - slice

Le *state* global va être découpé en *tranches* d'états (**slices**), ce qui permet la séparation de certaines données. Chaque slice est composée d'un *nom*, de la *valeur initiale* du state, et d'un ensemble de *reducers* définissant les modifications qu'on peut apporter au store.

Fichier : *components/TaskSlice.js*

```
import { createSlice } from "@reduxjs/toolkit";

const tasklice = createSlice({
      name : 'tasks',
      initialState : [],
      reducers : {
      addTask : ( state, action ) => {
            state.push( action.payload );
            },
      removeTask : ( state, action ) => {
            state.splice( action.payload, 1 );
            },
      }
});

export const { addTask, removeTask } = taskSlice.actions;
export const taskSelector = ( state ) => state.task;
export default taskSlice.reducer;
```

*2 actions possibles : addTask, removeTask*

*Permet de sélectionner la slice depuis le state global*

# Redux - store

Le store sera constitué de **l'ensemble des reducers**, qui permettront de **modifier** les différentes slices selon les actions définies

Fichier : *store/store.js*

```
import { configureStore } from "@reduxjs/toolkit";
import taskReducer from "../components/TaskSlice";

export default configureStore({
      reducer : {
            tasks: taskReducer
      }
```

Le store sera ensuite accessible globalement grâce à un **Provider** ainsi que 2 hooks additionnels : **useSelector** et **useDispatch**

# Redux - useDispatch

Si on veut modifier le state depuis un composant : on doit choisir la **slice** sur laquelle on veut effectuer des modifications, on choisit **l'action** à effectuer, et on effectue cette action en utilisant **useDispatch**

```
import { useState } from "react";
import { useDispatch } from "react-redux";
import { addTask } from "./TaskSlice";

export default function AddTask() {
      const [ taskName, setTaskName ] = useState('');
      const dispatch = useDispatch();

      const addNewTask = () => {
            dispatch( addTask( taskName ) );
            setTaskName( '' );
      }

      return ( <div>
                  <input onChange={ setTaskName } value={ taskName }/>
                  <button type="button" onClick={ addNewTask }> Add task </button>
               </div> )
}
```

# Redux - useSelector

Si on veut accéder à une partie du state global, on va *sélectionner* la slice correspondante avec *useSelector*.

```
import { useSelector } from "react-redux";
import { taskSelector } from "./TaskSlice";

export default function TaskList() {
      const tasks = useSelector( taskSelector );
      return (
            <ul>
               { tasks.map(task => <li> {task} </li> }
            </ul>
      )
}
```

Comme pour les autres states vus précédemment (avec *useState* ou *useContext*), si la variable **tasks** est modifiée (par une action définie dans la slice), **le composant est re-rendu**

# Redux - Provider

Le provider permettra aux différents composants d'accéder au store. Il est donc fortement conseillé de le mettre à la racine de l'application. En effet, seuls les composants enfants du provider pourront accéder au store !

Fichier : *App.js*

```
import AddTask from "./components/AddTask";
import TaskList from "./components/TaskList";
import { Provider } from "react-redux";
import store from "./store/store";

export default function App() {
  return (
      <Provider store={ store }>
            <div>
                  <AddTask />
                  <TaskList />
            </div>
      </Provider>
        );
      }
```

# Navigation

# Navigation

Souvent, une application web ne peut pas tenir sur une seule page. Pour améliorer l'UI et la simplifier, il peut être utile de séparer une application en plusieurs parties.

**React Router** : librairie de navigation, permettant des créer des routes entre différentes parties d'une page web.

Création d'un projet :

```
npx create-vite my-new-application
cd my-new-application
npm i react-router
npm run dev
```

# React router : BrowserRouter, Routes et Route

Avec React Router, on construit l'application sur le principe d'un arbre avec plusieurs branches. Chacune de ces branches sera définie dans un composant *Routes*, qui comprendra ensuite plusieurs *Route*. Chaque Route reçoit comme props un **path** (utilisé pour définir le routing) et un élément, qui correspond à l'élément de l'UI qui sera affiché en suivant cette route.

```
import { BrowserRouter, Route, Routes } from "react-router";

function App() {

      return (
            <BrowserRouter>
               <Routes>                    Route initiale
                  <Route path="/" element={<h1> Home </h1>} />
                  <Route path="other" element={<h1> Other </h1>} />
               </Routes>
            </BrowserRouter>
      )
}
```

# React router : Pages

Maintenant que nous avons différentes routes (donc différentes pages), on va relier ces différentes pages grâce à des *liens*.

Lien vers une page web externe : `<a href={ "..." }/>`

Lien vers une page interne : Composant <Link /> si le rendu du lien est statique, ou <NavLink /> si le rendu du lien est dynamique (on peut notamment savoir si lien est *active*, ou *pending*).

```
<nav>
      <Link to="/">Home</Link>
      <Link to="other">Other link</Link>
      <a href='https://en.wikipedia.org/wiki/React_(software)'>Documentation</a>
</nav>
```

On peut également passer des données avec la navigation `<Link to="other" state={ state }>Other link</Link>`

On récupère ensuite ce state avec le hook **useLocation**

# React router : useNavigate

Plutôt qu'un système de navigation par lien, on peut envisager d'autres interacteurs pour naviguer entre plusieurs pages. Par exemple, on aimerait pouvoir passer d'une page à une autre en cliquant sur un bouton, ou lorsqu'une certaine condition est vérifiée/respectée.

Utilisation du hook *useNavigate* :

```
let navigate = useNavigate();

return (
      <button type="button" onClick=navigate('other')>Go to other</button>
)
```

On peut, là aussi, faire passer des données d'une page à l'autre, en les ajoutant dans le 2ème argument de useNavigate

# *Structure d’un projet React*

# Application : réseau social

Le site https://randomuser.me/ propose une API publique qui permet de générer aléatoirement de faux utilisateurs, avec des profils très complet (prénom, nom, âge, adresse, mot de passe)

On va se baser sur ce site pour créer un faux réseau social, qui aura les fonctionnalités suivantes :

- **Page 1** : Affichage de la liste des utilisateurs. Pour chacun d'entre eux, on aimerait afficher leur photo de profil, leur prénom et nom, et avoir un bouton qui permet d'ajouter l'utilisateur à une liste de profils "likés". On peut cliquer sur le profil de l'utilisateur, ce qui nous emmène vers la page 2
- **Page 2** : Vue détaillée d'un utilisateur. Là, on affiche sa photo de profil en grand, son prénom, son nom, son pseudo, son âge, et son adresse. On a toujours un bouton permettant de "liker" l'utilisateur
- **Page 3** : Liste des utilisateurs likés, avec le même affichage que la page 1, et toujours la possibilité de cliquer sur un utilisateur pour voir son profil détaillé. On ajoute un bouton qui permet de supprimer un utilisateur des utilisateurs likés

# Réseau social : structure

Avant de se lancer dans une application avec de nombreuses fonctionnalités, il convient de définir une liste de fragments de code qu'on pourra réutiliser, ainsi que réfléchir aux différentes fonctionnalités, et à comment les relier.

Ici, on va tirer parti des **composants** React pour avoir à écrire le moins de code possible. On aura notamment besoin de :

- Un composant `SimpleUser` , qui prendra en props l'uri vers la photo de profil, le nom, et le prénom d'un utilisateur, et qui renverra un fragment html affichant la photo, un texte avec le prénom et le nom, et un bouton. L'effet du bouton sera également spécifié dans les props.
- Un composant `UserList`, qui prendra en props une liste d'utilisateurs, et affichera une liste de composants `SimpleUser` correspondant à cette liste.
- Un composant `DetailedUser`, qui prendra en props tous les arguments nécessaires à l'affichage détaillé d'un profil.

# Réseau social : API

On cherche à récupérer les données des utilisateurs depuis un serveur. On est donc dans un cas où on va faire des **requêtes asynchrones** en utilisant la fonction fetch. Comme on souhaite stocker la liste d'utilisateurs dans un state, et qu'on ne veut pas bloquer l'UI en attendant la réponse du serveur, on définit le callback asynchrone **directement dans un effet**

```
useEffect(() => {
    async function fetchData() {
        const response = await fetch('https://randomuser.me/api/?results=100');
        const result = await response.json();
        setData(result);
    }
}, [])
```

# Réseau social : stockage des utilisateurs likés

On cherche à enregistrer et à partager entre différents composants une liste d'utilisateurs enregistrés. On veut pouvoir ajouter un utilisateur à la liste, et à le retirer de la liste. On va donc passer par un **store** avec Redux. Le state sera constitué d'une seule Slice, sur laquelle 2 actions seront possibles :

- **addUser** : permet d'ajouter un utilisateur
- **removeUser** : permet de retirer un utilisateur de la liste

**Rappel** : on pourra effectuer ces actions depuis les composants avec le hook **useDispatch**, et on pourra lire le state stocké dans le store avec le hook **useSelector**.

En particulier, on va pouvoir passer les actions *addUser* et *removeUser* comme callback des boutons pour ajouter et supprimer des utilisateurs.

# Réseau social : vue simple/détaillée

Quand on clique sur un utilisateur depuis la liste, on veut afficher une vue détaillée de l'utilisateur sur une nouvelle page. Par ailleurs, on veut aussi disposer d'un bouton pour passer de la liste de tous les utilisateurs à celle des utilisateurs likés. On doit donc mettre en place un système de **navigation**.

Ici, on va définir 3 routes : la route **"home"** ('/') affichera la liste de tous les utilisateurs, la route **"detailed"** affichera la vue détaillée d'un utilisateur, et la route **"liked"** affichera la liste des utilisateurs likés.

On va utiliser le hook *useNavigate*, puisqu'on veut que la navigation s'effectue en cliquant sur un composant, et pas sur des liens. On passera les données utilisateurs (nom, prénom, photo…) dans le 2ème argument de ce hook.

# Réseau social : résumé