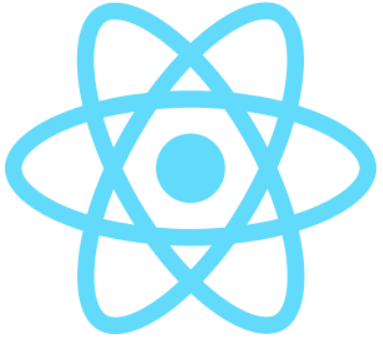
React



Deuxième partie

Rappels

React est une **bibliothèque JavaScript** pour le développement d'applications web dynamiques.

- On déclare des **composants**, responsables de **rendre** des fragments de HTML.
- On utilise la **syntaxe JSX**, qui facilite l'écriture de HTML et l'utilisation d'autres composants.
- Une application React est constituée d'un **arbre de composants**.
- Les composants prennent des **valeurs en entrée** via les ***props***.
- Les composants peuvent maintenir un **état interne** grâce au ***state***.
- Les **modifications de *props* ou *state*** entraînent le **re-rendu des composants concernés**.
- Les **changements de *state* se propagent via les *props*** dans l'arbre **de haut en bas**, ce qui garantit la cohérence de l'application à tout moment.
- La **modification effective du DOM** est gérée toute seule par React via le ***Virtual DOM***.

Hooks

Cycle de vie des composants

Les composants simples sont des **fonctions pures** (*props* => HTML) :

- la sortie dépend uniquement des entrées (*props*)
- pas d'influence depuis/sur un état externe (effet de bord)

Mais dans certains cas, on veut pouvoir :

- interagir avec un **état local**.
  Ex : garder l'état ouvert/fermé d'un panneau dépliant.
- interagir avec un **état global**.
  Ex : connaître l'utilisateur connecté.
- avoir des **effets extérieurs**.
  Ex : changer le titre de la page, faire une requête à une API, etc.
- accéder aux **éléments DOM** rendus.
  Ex : obtenir la taille à l'écran d'un élément.

Les *hooks*

Pour gérer ces cas, React introduit la notion de ***hooks*** :

- Fonctions spécialisées exécutées **à chaque rendu**
- "S'accrochent" au cycle de vie d'un composant et à sa gestion d'état locale
- Permettent d'exécuter du code en dehors du flux de rendu pur
- Permettent de contrôler finement les conditions d'exécution de ce code
- Leur nom commence par "`use`"

On verra ici les plus communs.

Référence complète

`useState`

Permet de gérer une valeur de *state*.

```
const [stateValue, setStateValue] = useState(initialValue);
```

- retourne un tableau comprenant :
  - la **valeur courante** de l'état pour ce cycle de rendu (`stateValue`)
  - une **fonction de mise à jour** (`setStateValue`)
- prend en entrée la **valeur initiale** (`initialValue`), qui sera utilisée lors du premier cycle

Par convention, on nomme `valeur` / `setValeur` :

- `const [nbClicks, setNbClicks] = useState(0);`
- `const [date, setDate] = useState(new Date());`

On peut utiliser plusieurs `useState` dans un même composant pour diviser le *state* en unités.

`useState` - Exemple

```jsx
import { useState } from "react";

const StateExample = () => {
  const [nbClicks, setNbClicks] = useState(0);
  const [detailsOpen, setDetailsOpen] = useState(false);

  const toggleDetails = () => {
    setDetailsOpen(!detailsOpen);
  };

  const incNbClicks = () => {
    setNbClicks(nbClicks + 1);
  };

  return (
    <div>
      <button onClick={toggleDetails}>{detailsOpen ? "▲ Masquer" : "▼ Voir"}</button>
      {detailsOpen && (
        <div>
          <p>Nombre de clics : {nbClicks}</p>
          <button type="button" onClick={incNbClicks}>
            Click!
          </button>
        </div>
      )}
    </div>
  );
};
```

`useEffect`

Hook générique de gestion d'effets de bord impératifs.

```
useEffect(callback);
```

- Prend en argument une **fonction callback**
- Le callback va être appelé **après chaque rendu**.

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

const TitleCounter = () => {
  const [nbClicks, setNbClicks] = useState(0);

  useEffect(() => {
    document.title = `${nbClicks} clicks`;
  });

  return (
    <button type="button" onClick={() => setNbClicks(nbClicks + 1)}>
      Click!
    </button>
  );
};
```

`useEffect` - Dépendances

⚠️ En pratique, on veut contrôler finement à quel moment un effet se déclenche ou non.

```
useEffect(callback, dependencies);
```

- `useEffect` prend en second argument un **tableau de dépendances**.
- Le callback va être appelé seulement **si les valeurs du tableau ont changé**.

```
useEffect(() => {
  // faire quelque chose seulement si user ou language a changé
  // par rapport au render précédent
}, [user, language]);
```

```
useEffect(() => {
  // faire quelque chose seulement au premier rendu
}, []);
```

`useEffect` - Exemple, appel d'API

```jsx
import { useState, useEffect } from "react";
import axios from "axios"; // bibliothèque d'appels HTTP qu'on va utiliser en TME

const SongLyrics = ({ artist, song }) => {
  const [lyrics, setLyrics] = useState(null);

  useEffect(() => {
    axios
      .get(`https://api.lyrics.ovh/v1/${artist}/${song}`)
      .then((result) => {
        setLyrics(result.lyrics);
      })
      .catch((error) => {
        setLyrics(null);
      });
  }, [artist, song]); // Redéclenche l'effet si l'une des valeurs change

  return (
    <div>
      <h2>
        {song} (by {artist})
      </h2>
      <pre>{lyrics || "Not found"}</pre>
    </div>
  );
};
```

`useEffect` - Fonction de nettoyage

- Le callback fourni à `useEffect` peut retourner lui-même un callback de nettoyage
- Exécuté avant le démontage du composant
- Exécuté avant la ré-exécution de l'effet

Exemple typique : un timer

```
useEffect(() => {
  const timerID = setInterval(() => {
    setDate(new Date());
  }, 1000);

  return () => {
    // fonction de nettoyage
    clearInterval(timerID);
  };
}, []); // premier rendu seulement
```

`useRef`

Hook de stockage de valeur persistante entre cycles de rendu.

```
const myRef = useRef(null);
```

- Prend en entrée la valeur initiale
- la valeur est accessible en lecture et écriture via `current` : `myRef.current`

Assimilable à une propriété d'instance dans une classe (`this.myValue = …`)

Usage courant : garder une référence directe vers un élément DOM rendu, grâce à la prop spéciale `ref`

```
const RefToDOM = () => {
  const domElt = useRef(null);

  return <div ref={domElt}>...</div>;
};
```

`useRef` - Exemple

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

const Measure = () => {
  const domElt = useRef(null);

  useEffect(() => {
    const onResize = () => {
      // domElt.current contient une référence vers le vrai nœud DOM de la div
      const width = domElt.current.clientWidth;
      console.log(width);
    };

    window.addEventListener("resize", onResize);

    return () => {
      window.removeEventListener("resize", onResize);
    };
  }, []);

  return <div ref={domElt} style={{ border: "1px solid", height: 50 }} />;
};
```

`useContext`

Hook de connexion à un **contexte** (état global qui ne passe pas par les *props*).

Si le composant est enfant d'un fournisseur du contexte `MyContext` :

```
const value = useContext(MyContext);
```

- Prend en entrée le type de contexte
- Retourne sa valeur courante

Mécanisme souvent utilisé pour les données transverses à toute une application :

- utilisateur connecté
- langue de l'interface
- thème d'interface
- …

https://react.dev/learn/passing-data-deeply-with-context

`useContext` - Exemple

```jsx
import React, { useContext } from "react";

const LanguageContext = React.createContext({ lang: "en" });

function App() {
  return (
    <LanguageContext.Provider value={{ lang: "fr" }}>
      <Header />
    </LanguageContext.Provider>
  );
}

function Header() {
  return (
    <header>
      <Greeting />
    </header>
  );
}

function Greeting() {
  const { lang } = useContext(LanguageContext);
  return <div>{lang === "fr" ? "Bonjour !" : "Hello!"}</div>;
}
```

`useMemo` / `useCallback`

Hooks de mémoïsation.

- évitent de recalculer une valeur ou redéclarer une fonction à chaque rendu.
- réduisent les changements de *props* des enfants en conservant des références stables.

Même mécanisme que `useEffect` pour les dépendances.

```
import { useMemo } from "react";

function Factorial({ number }) {
  const factorial = useMemo(() => {
    return costlyFactorialImpl(number);
  }, [number]);

  return (
    <div>
      Le carré de {number} est {number * number}, sa factorielle est {factorial}
    </div>
  );
}
```

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

function Clicker({ number }) {
  const [clicks, setClicks] = useState(0);

  const handleClick = useCallback(() => {
    setClicks(clicks + 1);
  }, []);

  return (
    <div>
      <p>{clicks} clics !</p>
      <MyButton onClick={handleClick} />
    </div>
  );
}
```

Hooks personnalisés

On peut **définir ses propres *hooks***, par composition des autres (on les nommera aussi `use*`).

```
function useCurrentUser() {
  const [user, setUser] = useState(null)

  useEffect(() => {
    // code complexe (ex: appel réseau, lecture de cookies, authentification, etc.)
    setUser(…);
  }, []);

  return user;
}

function App() {
  const user = useCurrentUser(); // utilisation du hook

  return <div>...</div>;
}
```

Mécanisme très utilisé dans les bibliothèques tierces pour React.

Formulaires

Gestion de l'état dans les formulaires

⚠️ Rappel : les champs de formulaires HTML sont *stateful*. Ils ont un état interne stocké dans le DOM (ex : la valeur d'un champ texte, le statut d'une case à cocher, etc.) 

=> Conflit conceptuel avec React sur la gestion d'état de l'application

Deux approches possibles :

Non contrôlée

- Utilisation de l'état interne géré par le DOM
- Lecture de la valeur quand c'est nécessaire
- Nécessite une référence explicite sur l'élément DOM (via `useRef` par exemple)

Contrôlée

- Gestion de la valeur en tant que *state* React
  "Source unique de vérité"
- Mise à jour via le cycle de rendu de React
- Facilite les opérations avant le rendu
  Ex. : formatage ou validation
- Plus lourd (*state* + événements)

Formulaire non contrôlé

```
const onlyNumbers = (str) => str.split("").every((char) => Number.isInteger(parseInt(char)));

const UncontrolledForm = () => {
  const cardNumberRef = useRef(null);
  const [hasInputError, setHasInputError] = useState(false);

  const handleSubmit = (e) => {
    e.preventDefault();
    // lecture de l'état interne de l'élément DOM pointé par cardNumberRef
    const cardNumber = cardNumberRef.current.value;
    setHasInputError(!onlyNumbers(cardNumber));
  };

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="cardNumber">Numéro de carte : </label>
      <input id="cardNumber" ref={cardNumberRef} />
      {hasInputError && <div>Entrée incorrecte !</div>}
    </form>
  );
};
```

Formulaire contrôlé

```
const groupBy4 = (str) => (str.match(/.{1,4}/g) || []).join(" ");
const onlyNumbers = (str) => str.split("").every((char) => Number.isInteger(parseInt(char)));

const ControlledForm = () => {
  const [cardNumber, setCardNumber] = useState("");
  const [hasInputError, setHasInputError] = useState(false);

  const handleChange = (e) => {
    const cleanValue = e.target.value.replace(/ /g, "");
    setCardNumber(groupBy4(cleanValue.substring(0, 16)));

    const isInvalid = !onlyNumbers(cleanValue);
    setHasInputError(isInvalid);
  };

  return (
    <div>
      <label htmlFor="cardNumber">Numéro de carte : </label>
      {/* Un champ contrôlé a les attributs value et onChange renseignés */}
      <input id="cardNumber" onChange={handleChange} value={cardNumber} />
      {hasInputError && <div>Entrée incorrecte !</div>}
    </div>
  );
};
```

Autour de React

Pour aller plus loin

React côté serveur (SSR, *Server Side Rendering*)

On peut effectuer le premier rendu d'une application React côté serveur :

1. rendu **statique**, **limité au premier cycle** (valeurs de *state* initiales)
2. envoi du HTML + code de l'application cliente au navigateur
3. chargement de l'application cliente React
4. "reprise en main" des composants et ajout de l'interactivité ("hydratation")

Deux raisons principales :

- Expérience utilisateur : page HTML complète avant chargement du JS par le navigateur.
- SEO (Search Engine Optimization) : page HTML complète pour les moteurs de recherche.

Fonctions de rendu côté serveur

Également en cours de développement par React, un nouveau modèle beaucoup plus complexe, les Server Components.

Gestion d'états complexes

Redux

- **store** : *state* global, souvent à la racine de l'application
- **actions** : opérations possibles sur ce store (ex : ADD_FRIEND, REMOVE_MESSAGE, etc.)
- **reducers** : fonctions de calcul du nouvel état du *store* en fonction d'une action
- Les composants peuvent s'abonner au store et/ou lui envoyer des actions via des *hooks* dédiés

Vite complexe, notamment pour gérer les modifications asynchrones (ex : appels d'API).

React-query

Pas de store centralisé.

Création de hooks d'abstraction des appels d'API, contrôle fin du cache et des conditions de mise à jour.

React Router

https://reactrouter.com/en/main Router déclaratif pour les SPA React

Principe : gérer la structure d'URLs et les différentes vues de l'application en React pur

```
export default function App() {
  return (
    <Router>
      <div>
        <nav>
          <Link to="/">Home</Link> · <Link to="/about">About</Link> · <Link to="/users">Users</Link>
        </nav>
        <Routes>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/users">
            <Users />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Routes>
      </div>
    </Router>
  );
}
```

React Native

https://reactnative.dev/

Framework de développement d'applications natives, également maintenu par Facebook.

- Même modèle conceptuel que React pour le web : *props/state* et propagation
- Même syntaxe JSX
- On appelle des composants visuels natifs au lieu des éléments HTML
- Principe de couche d'abstraction du rendu, similaire au VirtualDOM

Permet de cibler Android et iOS avec un même code JS de plus haut niveau.

Autres approches

Projets similaires

- Ex : Vue.js, Svelte, etc.
- Principes proches (abstraction du DOM, composants, *state*, etc.)

Critique de React

- Paradigme qui ignore en partie les standards du web (ex : Web Components)
- Lourdeur, "usine à gaz", ré-invention de la roue
- *Developer experience > user experience*
- Éclipse des solutions plus simples/adaptées
  => Critique des SPA non pertinentes et du JS partout

React est très populaire mais n'est pas la seule solution

- Émergence d'outils s'appuyant mieux sur HTML (ex : htmx, Astro)