React

React est un framework frontend très populaire pour la création d'applications web. Couplé avec NextJS, il permet de créer des applications web performantes rendues coté serveur et optimisées pour le SEO.

Il permet aussi la création d'applications native avec React Native.

- Reference officielle: https://react.dev/reference/react
- Tutoriels d'apprentissage: https://react.dev/learn

Utilisation

React est rarement utilisé seul aujourd'hui. Il est généralement utilisé avec des frameworks comme NextJS, Gatsby, Expo, ou bien seul avec Vite et React Router

---

Librairies

Voici une liste de librairies très utilisées en complément de React:

- React Hook Form (formulaires): https://react-hook-form.com/
- Zod (validation de données): https://zod.dev/
- React Router (routing, si utilisé sans framework): https://reactrouter.com/
- Axios (data fetching): https://axios-http.com/
- React Query (data fetching): https://tanstack.com/query/latest/docs/framework/react/overview
- SWR (data fetching): https://swr.vercel.app/
- Apollo (GraphQL): https://www.apollographql.com/
- URQL (GraphQL): https://nearform.com/open-source/urql/
- Jest (tests unitaires): https://jestjs.io/

- Cypress (tests d'integration): https://www.cypress.io/

Design & UI

TailwindCSS est aujourd'hui une reference pour gérer le style de pages web.

Pour ce qui est des composants, il existe de nombreux librairies pour React:

- Material UI: https://mui.com/
- Shadcn: https://ui.shadcn.com/
  - Theme builder: https://tweakcn.com/
- Chakra UI: https://chakra-ui.com/

Pour les animations, on peut utiliser:

- Framer Motion: https://www.framer.com/motion/
- React Spring: https://react-spring.dev/

Composants

Une application React est constituée de composants imbriqués. Séparer une application en composant permet de découper le code par rôle (single responsibility) et factoriser des zones réutilisables.

JSX

En React, un composant est une fonction. Pour décrire le rendu d'un composant, on utilise la syntaxe JSX, qui ressemble à du HTML mais est en réalité une extension de JavaScript.

https://react.dev/learn/your-first-component

Un composant peut être imbriqué dans un autre composant, et ainsi de suite.

Pour afficher des données dans un composant (interpolation), on utilise les accolades `{}`.

```tsx
function App() {
  return (
    <div>
      <Counter />
    </div>
  )
}
function Counter() {
  const count = 10;
  return <div>Count is {count}</div>
}
```

Lifecycle

Une fonction de composant est appelée à chaque fois que le composant est rendu. React décide de quand ce rendu est déclenché, généralement par un changement d'état ou une modification de props.

Cela signifie que l'état des variables locales est perdu à chaque rendu, on va donc en general utiliser des constantes.

```tsx
function Counter() {
  let count = 10;
  count++; // la valeur sera remise à 10 à chaque rendu
  return <div>Count is {count}</div>
}
```

https://react.dev/learn/keeping-components-pure

Props

Pour passer des données à un composant, on utilise les props. Les props sont passées en argument à la fonction du composant.

```tsx
function App() {
    const name = "John";
    return <Hello name={name} />
}
function Hello({ name }) {
    return <div>Hello {name}</div>
}
```

https://react.dev/learn/passing-props-to-a-component

Différence notables entre HTML et JSX

- `class` → `className`

- `for` → `htmlFor`
- `tabindex` → `tabIndex`
- `key` : identifiant unique pour les éléments du rendu en liste
- `ref` : référence à un élément du DOM
- `onClick` : gestionnaire d'événement de clic
- `onChange` : gestionnaire d'événement de changement
- ...

Interpolation avancée

Dans le JSX, le code placé entre accolades est interprété comme du JavaScript, il est donc possible d'utiliser des expressions complexes.

```tsx
function App() {
    const name = "John";
    const data = [0, 1, 2]
    return (
        <div>
            {/* Interpolation simple */}
            <p>{`Hello ${name}`}</p>
            <p>{1+1}</p>

            {/* Conditionnel */}
            {data[0] > 0 && <p>data[0] is greater than 0</p>}

            {/* Conditionnel ternaire */}
            <p>{Math.random() > 0.5 ? "Hello" : "World"}</p>

            {/* Rendu d'une liste */}
            <p>{data.map((item) => <span key={item}>{item}</span>)}<
        </div>
    )
}
```

Le rendu sera:

```html
<div>
    <p>Hello John</p>
    <p>2</p>

    <p>World</p>

    <p><span>0</span><span>1</span><span>2</span></p>
</div>
```

- https://react.dev/learn/conditional-rendering
- https://react.dev/learn/rendering-lists

Evenements

Les événements sont des actions effectuées par l'utilisateur ou le navigateur, qui déclencheront des fonctions.

```tsx
function App() {
    function handleClick() {
        alert("Hello")
    }
    function handleChange(event) {
        console.log(event.target.value)
    }
    return (
        <div>
            <input type="text" onChange={handleChange} />
            <button onClick={handleClick}>Click me</button>
        </div>
    )
}
```

Fragments

Il est parfois necessaire de retourner plusieurs éléments d'un composant, sans pour autant les placer dans une balise HTML commune. Pour cela, on peut utiliser un fragment.

```tsx
function App() {
    return (
        // equivalent à <Fragment>
        <>
            <p>Hello</p>
            <p>World</p>
        </>
    )
}
```

Typescript

Il est vivement recommandé d'utiliser TypeScript pour tous les projets Javascript, nottament avec React. Les composants seront alors écrits en TSX.

Exemple de composant correctement typé:

```tsx
interface ButtonProps {
  label: string;
  onClick?: () => void;
}

const Button: React.FC<ButtonProps> = ({ label, onClick }) => {
  return (
    <button
      onClick={onClick}
    >
      {label}
    </button>
  );
};
```

```
export default Button;
```

Hooks

En react, les hooks sont des fonctions (systematiquement commençant par `use`) qui permettent de gérer l'état et les effets secondaires des composants. La librairie integre déjà un ensemble de hooks indispensable, et il est possible de créer ses propres hooks.

---

useState

https://react.dev/reference/react/useState

Comme vu precedement, les variables locales des composants sont réinitialisées à chaque rendu. Pour conserver une valeur entre les rendus, on peut utiliser le hook `useState`. Celui ci retourne une paire de valeurs: la valeur courante et une fonction pour la modifier.

Lorsqu'une variable créée avec `useState` est modifiée avec son `set...`, tous les composants affichant cette variable seront re-rendus.

```tsx
function Counter() {
    const [count, setCount] = useState(0);
    return <div>
        <p>Count is {count}</p>
        <button onClick={() => setCount(count + 1)}>Increment</butto
    </div>
}
```

Attention: il ne faut pas modifier directement l'état d'une variable (`count` ici)

Pour modifier des valeurs dans un objet ou un tableau, on peut déstructurer, ou utiliser le prototype `set...(prevState ⇒ newState):

```tsx
const [data, setData] = useState({
    value: 0,
    name: "John"
});
setData({
    ...data,
    value: data.value + 1
})
setData(prev => ({
    ...prev,
    value: prev.value + 1
}))
```

---

useEffect

https://react.dev/reference/react/useEffect

Ce hook servira a executer des actions qui auront des effets secondaires, tel que des appels API, des événements, des animations, etc.

Il prend en paramètre une fonction et un tableau de valeurs. La fonction sera appelée à chaque fois qu'une des valeurs du tableau change. Si un tableau vide est passé, la fonction sera appelée une seule fois après que le composant ait été créé.

Cette fonction peut retourner une fonction de nettoyage qui sera appelée lorsque l'effet sera relancé, ou lorsque le composant sera démonté.

```ts
useEffect(() => {
    console.log("Component mounted")

    return () => console.log("Component unmounted") // optionnel
}, [])

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

```
    console.log("count changed", count)
}, [count])
```

Attention: Toutes les variables qui peuvent potentiellement changer dans la fonction doivent être dans le tableau de dépendances.

---

useMemo

https://react.dev/reference/react/useMemo

useMemo est un Hook React qui vous permet de mettre en cache le résultat d'un calcul d'un rendu à l'autre. Comme useEffect, il prend en paramètre une fonction et un tableau de valeurs. La fonction sera appelée à chaque fois qu'une des valeurs du tableau change.

Cela evite d'effectuer le même calcul à chaque rendu du composant.

tsx
```tsx
const memoizedValue = useMemo(() => {
    return expensiveComputation(a, b);
}, [a, b]);
```

---

useCallback

https://react.dev/reference/react/useCallback

useCallback est similaire à useMemo, sauf qu'il conserve une fonction plutot qu'une valeur.

tsx
```tsx
const memoizedCallback = useCallback(() => {
    doSomething(a, b);
}, [a, b]);
```

Cela permet d'optimiser les performances des composants et des hooks personnalisés en évitant de re-créer les fonctions à chaque rendu.

---

useRef

https://react.dev/reference/react/useRef

useRef est un Hook React qui vous permet de créer une référence vers une valeur qui n'a pas d'incidence sur le rendu du composant. Si cette valeur change, le rendu ne sera pas ré-executé. Il est souvent utiliser pour stocker des references vers des elements du DOM

```tsx
const intervalRef = useRef(null);
const inputRef = useRef(null);

function handleClick() {
    inputRef.current.focus();
}

useEffect(() => {
    intervalRef.current = setInterval(() => {
        console.log("tick");
    }, 1000);
    return () => clearInterval(intervalRef.current);
}, []);

return <div>
    <input ref={inputRef} />
    <button onClick={handleClick}>Focus</button>
</div>
```

---

useContext

https://react.dev/reference/react/useContext

useContext est un Hook React qui vous permet d'acceder aux données d'un contexte.

```tsx
const MyContext = createContext(null);

function App() {
    return (
        <MyContext.Provider value="Hello">
            <MyComponent />
        </MyContext.Provider>
    )
}

function MyComponent() {
    const value = useContext(MyContext);
    return <div>{value}</div>;
}
```

Les contextes permettent de partager des données entre les composants sans avoir à les passer explicitement à chaque composant en tant que props.

Hooks personnalisés

Il est possible de créer ses propres hooks. Pour cela, il faut suivre les règles de hooks et utiliser les hooks existants.

```tsx

function useCounter() {
    const [count, setCount] = useState(0)

    function increment() {
        setCount(count + 1)
    }

    useEffect(() => {
        console.log("count changed", count)
    }, [count])
```

```
    return [count, increment]
}

function MyComponent() {
    const [count, increment] = useCounter();
    return <div>
        <p>Count is {count}</p>
        <button onClick={increment}>Increment</button>
    </div>
}
```

Rule of hooks

Les hooks doivent impérativement être utilisés à la racine d'un composant ou d'un autre hook, c'est à dire jamais de manière conditionnelle ou imbriquée.

jsx

```
function Counter() {
  const [count, setCount] = useState(0); // ✅
  if(count > 10) {
    const [count2, setCount2] = useState(0); // ❌
  }
  for (let i = 0; i < 10; i++) {
    const theme = useContext(ThemeContext); // ❌
  }
  if(count > 10) {
    return <div>Count is greater than 10</div>
  }
  const [count3, setCount3] = useState(0); // ❌
  //...
}
```

https://react.dev/reference/rules/rules-of-hooks

Server Components

Depuis la version 18, React propose une nouvelle approche pour le rendu des composants : les Server Components (**RSC**).

- https://react.dev/reference/rsc/server-components
- https://nextjs.org/docs/app/getting-started/server-and-client-components

Cette approche permet de rendre des composants directement sur le serveur, ce qui permet de gagner en performance et de réduire la quantité de code JavaScript à envoyer au client. Cette technique a initialement été portée par NextJS, mais qui a maintenant été integrée diretement dans React.

Comment ça marche ?

Désormais, tout composant React est **par défaut** un Server Component.

Cela signifie que le rendu (html) sera generé au moment du build de l'application, ou bien a chaque fois que l'utilisateur chargera la page, par le serveur, au choix. Ce que recevra le client sera quasiment uniquement du HTML et CSS, et n'aura pas besoin d'executer React pour afficher ce composant, ce qui augmente les performances significativement.

Data fetching

Avant les server component, il était habituel de récupérér les données côté client, via des appels API principalement. Cela signifie un temps de chargement plus long de la page, avec plusieurs allers retours entre le client et le serveur, des loaders, etc.

```tsx
'use client'
function Component({id}) {
  const [data, setData] = useState(null);
  useEffect(() => {
```

```
    fetch(`/api/data`).then(data => {
      setData(data);
    });
  }, [id]);

  return (
    <div>
      <p>{data ? data.message : "Loading..."}</p>
    </div>
  );
}
```

Désormais, les données peuvent être récupérée directement par le serveur lui-meme, qui est plus proche de la base de données, et donc plus rapide, et le client recevra la page déjà remplies des données:

tsx
```
async function Component({id}) {
  const data = await fetch(`/api/data`);
  return (
    <div>
      <p>{data}</p>
    </div>
  );
}
```

La contrepartie est que les server component **ne sont pas interactifs**: Il n'est plus possible d'utiliser des hooks tels que useState pour rendre le composant dynamique. En effet, ils sont rendus côté serveur une fois, et ne peuvent pas être modifiés ensuite.

tsx
```
export default function Expandable({children}) {
  const [expanded, setExpanded] = useState(false); // 🚨 pas possibl
  return (
      <button
        onClick={() => setExpanded(!expanded)}
      >
        Toggle
```

```
      </button>
      {expanded && children}
    )
  }
```

Client components

Il est evidemment encore possible de créer des composants interactifs, qui seront rendus côté client. Pour ce faire, il faut utiliser le mot-clé `use client` devant le composant.

```tsx
'use client' // <----
export default function Expandable({children}) {
  const [expanded, setExpanded] = useState(false); // ✅ ok
  return (
      <button
        onClick={() => setExpanded(!expanded)}
      >
        Toggle
      </button>
      {expanded && children}
  )
}
```

Pour obtenir le meilleur des deux mondes, on peut utiliser les `Client Components` dans les `Server Components`.

```tsx
async function Component({id}) {
  const data = await fetch(`/api/data`);
  return (
    <div>
      <Expandable>{data}</Expandable>
    </div>
```

```
  );
}
```

```tsx
'use client'
export default function Expandable({children}) {
  const [expanded, setExpanded] = useState(false);
  return (
      <button
        onClick={() => setExpanded(!expanded)}
      >
        Toggle
      </button>
      {expanded && children}
  )
}
```

WARNING

Attention: tous les composants héritant d'un `Client Component` deviennent automatiquement des client components eux même. Pour palier ce problème, il faut passer les composants en tant que `children` ou en props.

Server functions

En plus du rendu côté serveur et des routes handlers, React propose maintenant un autre système pour effectuer des actions côté serveur, les serveur functions (aussi appelées server actions).

- https://react.dev/reference/rsc/server-functions
- https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions-and-mutations

Ces fonctions sont très similaires aux routes handlers, mais à la différence qu'au lieu de définir une API REST, avec URL et un schema de données, que le client appelera, le client peut directement appeler une fonction. Cela fait gagner énormément de temps, car il n'est plus nécéssaire de creer des routes pour chaques actions, de sérialiser les données, les valider, etc... On appelle une fonction comme si elle était présente côté client.

Ces fonctions peuvent être appelées depuis n'importe quel composant dans un event handler, useEffect, ou même passé comme directement comme action à un formulaire.

Ils peuvent être définis:

- Dans un server component, une fonction marquée avec `'use server'` sera une server function.

```tsx
async function Component() {
  async function create() {
   'use server'
    await db.create({
      name: 'John Doe',
    })
  }

  return (
      <button type="submit" onClick={create}>Create</button>
```

```
  )
}
```

- Dans un fichier commençant par `'use server'`, toutes les fonctions de ce fichier seront des server functions.

```tsx
'use server'

export async function login(formData: FormData) {
  if (formData.get('password') === PASSWORD) {
    return true
  }
  throw new Error('Invalid password')
}
```

```tsx
'use client'
import { login } from './actions'

function LoginForm() {
    return (
        <form action={login}>
            <input type="password" name="password" />
            <button type="submit">Login</button>
        </form>
    )
}
```

Gestion de l'état

Le hook `useActionState` permet de gérer l'état d'une server function.

```tsx
'use client'

import { login } from './actions'

function Component() {
  const [state, formAction, isPending] = useActionState(login, null)
```

```
  return (
    <form action={formAction}>
      <input type="text" name="name" />
      <button type="submit" disabled={isPending}>Create</button>
      {state.error && <p>{state.error}</p>}
    </form>
  )
}
```

Les exceptions ne sont pas gérés dans cette situation. Si l'action emet une erreur, le front l'ignorera, à moins qu'un fichier `error.tsx` soit présent, dans ce cas ce composant sera affiché.

Pour gérer correctement les erreurs, il est preferable d'éviter d'emetre des erreurs depuis les actions, mais plutot toujours une réponse ok avec un objet contenant des informations sur le succes de l'opération, par exemple:

tsx

```tsx
'use server'

export async function login(formData: FormData) {
  if (formData.get('password') === PASSWORD) {
    return {
      success: true,
      data: user
    }
  } else (
    return {
      success: false,
      error: 'invalid password'
    }
  )
}
```

Exercices React

https://react.dev/learn

- Creer un nouveau projet Next.js
  - `npx create-next-app@latest`
- Supprimer le contenu du composant `app/page.tsx`
- Creer un composant `Name` qui contiendra un bouton et un champ de texte
  - Lorsque le bouton est cliqué, le nom contenu dans le champ de texte s'affichera avec un `alert`
  - Lorsque le contenu du champ est changé, afficher la nouvelle valeur dans la console