Développement front-web : la bibliothèque React.JS

Pour commencer…

Développement front-end

Principe :

- Programmation des éléments visibles par les utilisateurs d'applications web
  - Mise en page
  - Boutons/formulaires
  - Liens/navigation

Caractéristiques :

- Expérience utilisateur améliorée
- Performance
- Réduction de la charge serveur

React.JS

Développée par Facebook en 2013

Faciliter la création d'application web monopage, via la création de composants dépendant d'un état et générant une page HTML à chaque changement d'état.

Architecture d'un projet React.JS

index.html

```
<body>
<div id="root"></div>
</body>
```

Afficher le composant <App /> dans l'élément d'id root

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));
```

app.js

```
import React from 'react';
function App() {
  return (
    <p>Bonjour</p>
  );
}
export default App;
```

le composant <App />

Création d’un projet React

- Installation de Node.js et NPM

- Initialisation d’un projet Node : npx create-react-app mon-projet

(création du répertoire, de l’arborescence React et des bibliothèques de démarrage react, react-dom et react-scripts)

- Exécution de l’application : npm start

Structure d’une application React

Principes-clés

Les composants

But : Diviser l'interface utilisateur en morceaux réutilisables

```
<Oiseau espece="Perruche" sexe="F" />
```

app.js

```
function Oiseau(props) {

  return (
  <div>
    <p>Il reste un oiseau de type {props.espece} de sexe {props.sexe}.</p>
  </div>
  );
}
```

Oiseau.js

Composants fonctionnels vs. de classe

Composants fonctionnels :

- Déclarés comme une fonction Javascript
- Un seul paramètre : objet **props** manipulable
- Renvoient une vue à afficher en retour.

```
function Oiseau(props) {

  return (
  <div>
    <p>Il reste un oiseau de type {props.espece} de sexe {props.sexe}.</p>
  </div>
  );
}
```

Composant fonctionnel

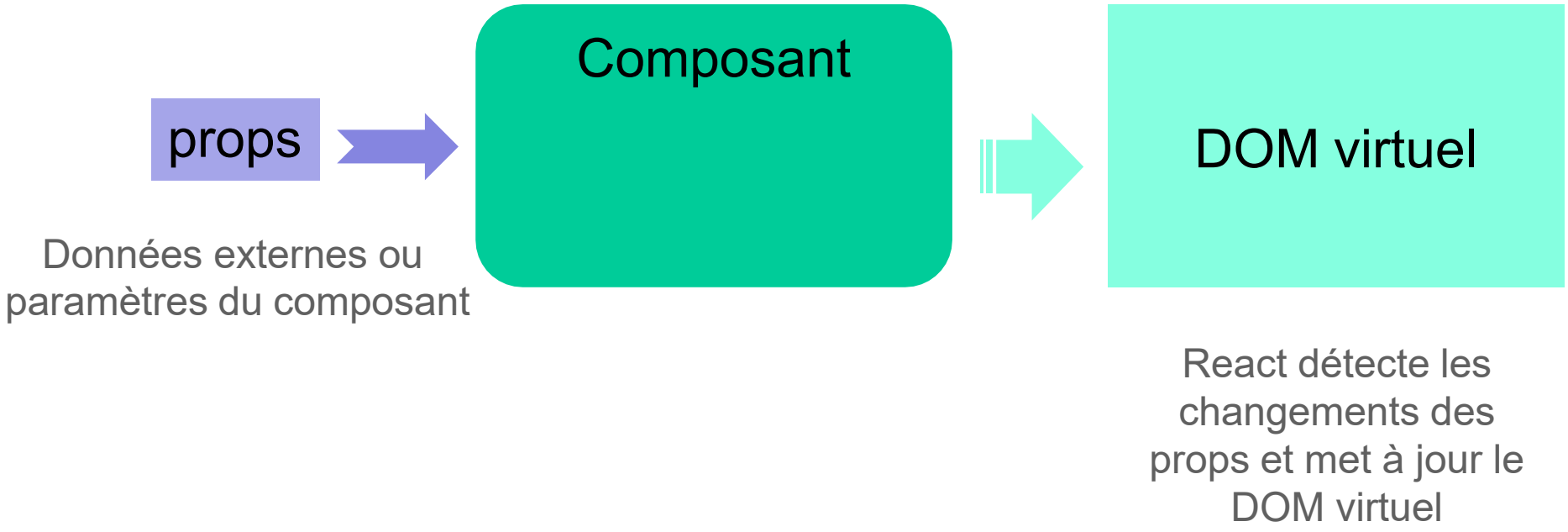
Composants de classe :

- Déclarés comme une classe qui étend React.Component
- Accèdent aux **props** via **this.props**
- Doivent implémenter une méthode render ( ) pour renvoyer une vue à afficher

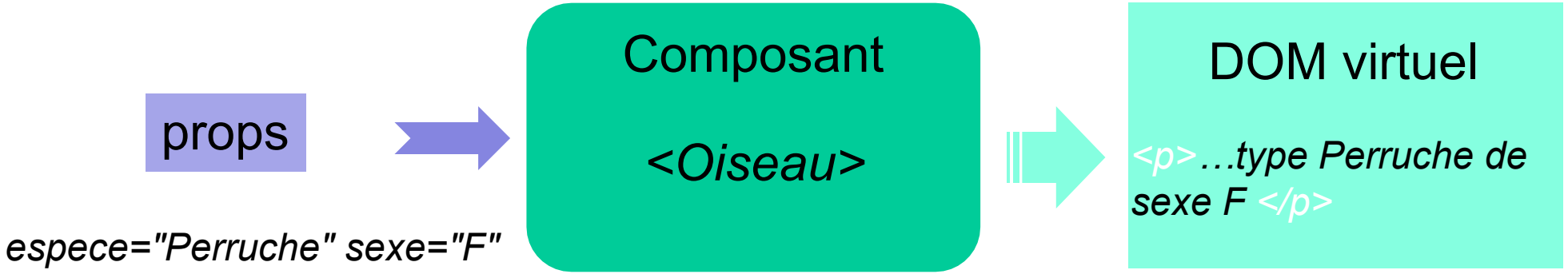
```
class Oiseau extends Component {
  render() {
    return (
      <div>
        <p>Il reste un oiseau de type {this.props.espece}
        de sexe {this.props.sexe}.</p>
      </div>
    );
  }
}
```

Composant de classe

1) Les composants de base (sans état)



1) Les composants de base (sans état) : un exemple



app.js

```
<Oiseau espece="Perruche" sexe="F" />
```

Oiseau.js

```
function Oiseau(props) {

  return (
  <div>
    <p>Il reste un oiseau de type {props.espece} de sexe {props.sexe}.</p>
  </div>
  );
}
```

1) Les composants de base

- A noter que via les props un composant parent peut aussi transmettre une fonction à un composant enfant :

```
const changerEspece = () => {
    ...
};

<Oiseau changerEspece={changerEspece} espece="Perruche" sexe="F"/>
```

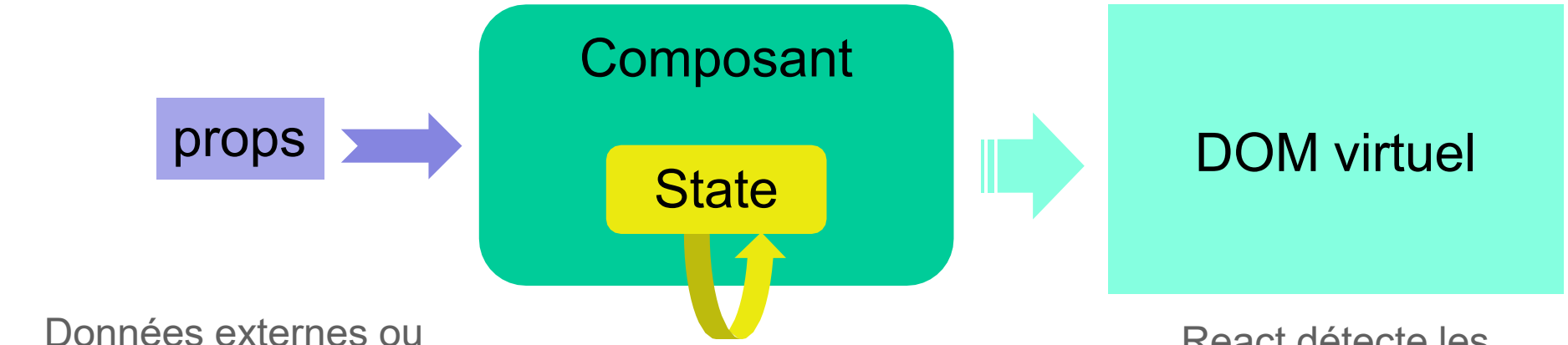
app.js

- Qui y accédera de la façon suivante :

```
props.changerEspece();
```

Oiseau.js

2) Les composants à état

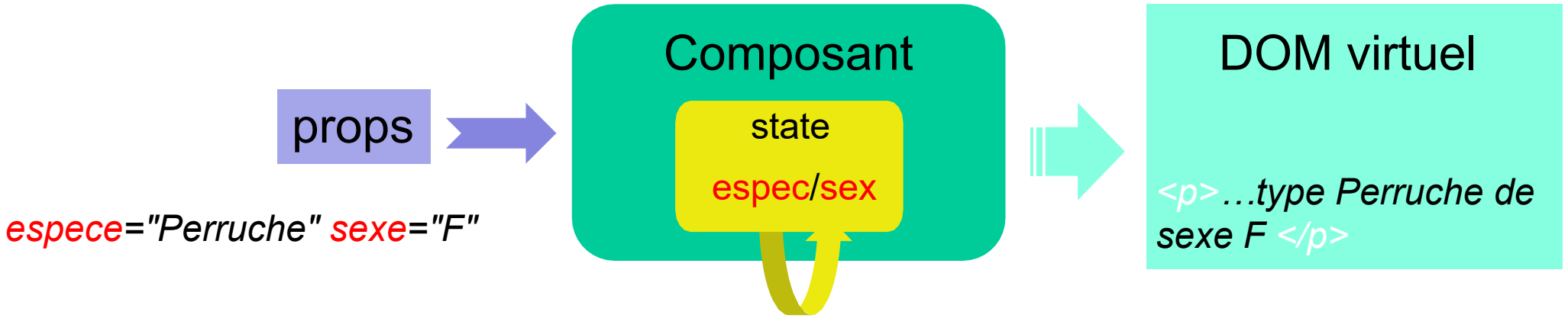


Données externes ou paramètres du composant

Données internes du composant

React détecte les changements du state ou des props et met à jour le DOM virtuel

2) Les composants à état: un exemple




```jsx
import React, { useState } from 'react';

function Oiseau(props) {
  // Déclaration de l'état avec useState
  const [espec, setEspec] = useState(props.especе);
  const [sex, setSex] = useState(props.sexe);

  return (
    <div>
      <p>Il reste un oiseau de type {espec} de sexe {sex}.</p>
      {/* Boutons pour changer l'espèce et le sexe */}
      <button onClick={() => setEspec('Aigle')}>Changer en Aigle</button>
      <button onClick={() => setSex('Mâle')}>Changer en Mâle</button>
    </div>
  );
}

export default Oiseau;
```

Composants à état : le hook useState

- Intérêt

  Changer la valeur d’une variable de composant.

  Sur la base d’actions utilisateur

Le hook useState

- Définir un état

```
const [compteur, setCompteur] = useState(0);
```

- Modifier un état

```
const incrementCompteur = () => {
    setCompteur(compteur + 1);
};
```

Le hook useState

Exemple : Changer la valeur d'une variable par une valeur saisie par l'utilisateur

```
function FormTexte (props){

    const [prenom, setPrenom]= useState("Entrez ici votre valeur");

    const modifierValeur = (e) => {
        setPrenom (event.target.value);
    };

    return (
        <div>
            <p>Entrez votre prénom</p>
            <input type="text" value={prenom} onChange={modifierValeur}/>
        </div>
    );

}
```

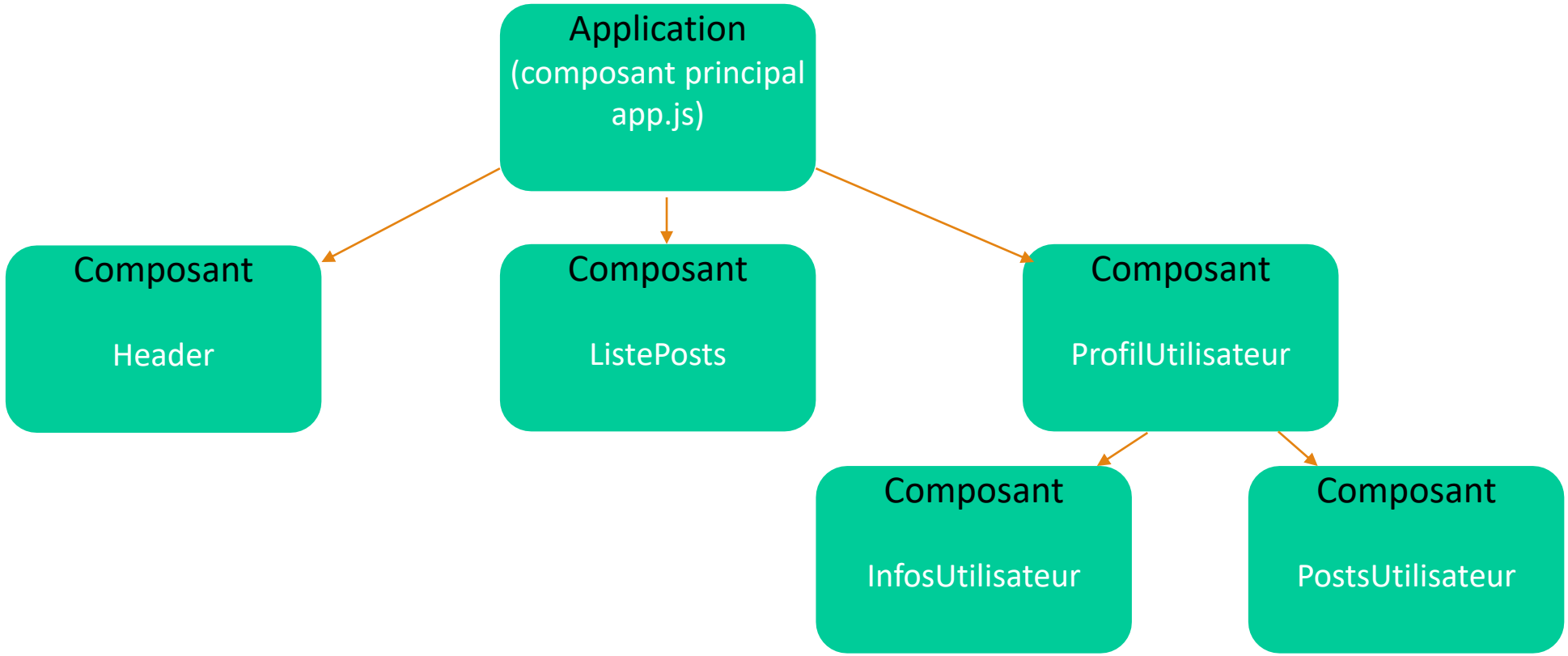
Rappel JS : la propriété onClick

```
function handleClick() {
  alert('Bouton cliqué !');
}

<button onClick={handleClick}>Clique-moi</button>
```

Structure de React

- Une application est un arbre de composants :



JSX

Syntaxe JSX

- JavaScript XML (JSX)
- Syntaxe proche HTML pour écrire les éléments React en JavaScript
- Uniquement utilisée dans le rendu des composants (dans le return d'un composant)
- Possibilité d'intégrer un style avec syntaxe proche CSS
- Possibilité d'intégrer du code JS entre accolades
- Juste un langage de confort pour le développeur : JSX converti en JS par un compilateur avant interprétation par le navigateur

JSX vs JS

```jsx
<div>
  <p>Il reste un oiseau de type {espec} de sexe {sex}.</p>
  {/* Boutons pour changer l'espèce et le sexe */}
  <button onClick={() => setEspec('Aigle')}>Changer en Aigle</button>
  <button onClick={() => setSex('Mâle')}>Changer en Mâle</button>
</div>
```

JSX

```js
const div = document.createElement('div');
const p = document.createElement('p');
p.textContent = `Il reste un oiseau de type ${espec} de sexe ${sex}.`;

// Création des boutons
const buttonAigle = document.createElement('button');
buttonAigle.textContent = 'Changer en Aigle';
buttonAigle.onclick = () => setEspec('Aigle');

const buttonMale = document.createElement('button');
buttonMale.textContent = 'Changer en Mâle';
buttonMale.onclick = () => setSex('Mâle');

// Ajouter les éléments au DOM
div.appendChild(p);
div.appendChild(buttonAigle);
div.appendChild(buttonMale);
root.appendChild(div);
```

JS

Expression JS dans JSX

- Possibilité d'intégrer du code JS entre accolades

```
const name = 'Alice';
const greeting = <h1>Bonjour, {name}!</h1>;
```

```
<button onClick={() => setSex('Mâle')}>Changer en Mâle</button>
```

Syntaxe : attributs camelCase/transmission évènements

```html
<div class="container" onclick="handleClick()">Contenu</div>
<input type="text" tabindex="1">
```

HTML

```jsx
<div className="container" onClick={handleClick}>
  Contenu
</div>
<input type="text" tabIndex={1} />
```

JSX

Style de mise en page dans JSX

```jsx
  return (
    <div style={{ backgroundColor: 'lightblue', padding: '20', textAlign: 'center' }}>
      <h1 style={{ color: 'darkblue' }}>Titre avec style en ligne</h1>
      <p style={{ fontSize: '18' }}>Ceci est un paragraphe stylé en ligne.</p>
    </div>
  );
```

Style en ligne

```jsx
function App() {
  return (
    <div style={styles.container}>
      <h1 style={styles.heading}>Titre avec style
      défini dans un objet</h1>
      <p style={styles.paragraph}>Ceci est
      un paragraphe avec style dans un objet JS.</p>
    </div>
  );
}
```

Utilisation de style externe

```jsx
const styles = {
  container: {
    backgroundColor: 'lightcoral',
    padding: '20',
    textAlign: 'center',
  },
  heading: {
    color: 'darkred',
  },
  paragraph: {
    fontSize: '18',
  },
};
```

Contrainte : balise parent obligatoire

- Obligation de mettre une balise englobante JSX dans le return d'un composant

```jsx
return (
  <div>
    <p>Il reste un oiseau de type {espec} de sexe {sex}.</p>
    {/* Boutons pour changer l'espèce et le sexe */}
    <button onClick={() => setEspec('Aigle')}>Changer en Aigle</button>
    <button onClick={() => setSex('Mâle')}>Changer en Mâle</button>
  </div>
);
```

```jsx
return (
  <Fragment>
    <p>Il reste un oiseau de type {espec} de sexe {sex}.</p>
    {/* Boutons pour changer l'espèce et le sexe */}
    <button onClick={() => setEspec('Aigle')}>Changer en Aigle</button>
    <button onClick={() => setSex('Mâle')}>Changer en Mâle</button>
  </Fragment>
);
```

D’autres concepts

Transmission d'évènements

```
const handleNameChange = (event) => {
    ...
};

return (
<div>
    ...
    <input type="text" value="Prenom" onChange={handleNameChange} />
    ...
</div>
);
```

- event.type : type de l'évènement
- event.target : élément DOM qui a déclenché l'évènement
- event.clientX : position horizontale de la souris (évènement de souris onClick…)
- event.key : touche clavier pressée (évènement clavier onKeyDown…)
- D'autres à chercher…

L’opérateur de propagation …

Permet de décomposer un objet ou un tableau en ses éléments individuels

```
const [people, setPeople] = useState([
  { id: 1, name: 'Alice', age: 30 },
]);

const handleSaveName = (id, newName) => {
  const updatedPeople = people.map(person => {
    if (person.id === id) {
      return { ...person, name: newName };
    }
    return person;
  });
  setPeople(updatedPeople);
};

handleSaveName(1, 'Alicia');
```

D’autres hooks

Le hook useEffect

- Intérêt

  Définit un comportement à exécuter après chaque changement du composant.
  Gestion d'effets secondaires (appels d'API, abonnements à des évènements…)

```
function Exemple (props) {

    const [prenom, setPrenom] = useState("Entrez ici votre valeur");

    useEffect(() => {
    console.log("Le composant a été rendu !");
    });

}
```

Le hook useEffect

```
useEffect( () => {…} )
```

à chaque modification du composant

```
useEffect( () => {…}, [] )
```

une seule fois

```
useEffect( () => {…}, [age] )
```

seulement si modif état variable **age**

```
useEffect( () => {…}, [props.age] )
```

seulement si modif props **age**

Le hook useEffect

- Exemple

```
function PersonInfo() {
  const [person, setPerson] = useState({});
  const userId = 2; // à remplacer par l'ID de la personne à afficher

  useEffect(() => {
    fetch(`https://jsonplaceholder.typicode.com/users/${userId}`)
      .then((response) => response.json())
      .then((data) => setPerson(data));
  }, [userId]);

}
```

Le hook useContext

- Intérêt

  Permet de partager des valeurs entre des composants

Le hook useContext

- Utilisation

```jsx
import React, { createContext } from 'react';

// Créer un contexte pour stocker l'état global
const PeopleContext = createContext();

function PeopleProvider({ children }) {
  const [people, setPeople] = useState([]);

  return (
    <PeopleContext.Provider value={{ people }}>
      {children}
    </PeopleContext.Provider>
  );
}

export { PeopleProvider, PeopleContext };
```

PeopleProvider.js

```jsx
return (
<div>
    <PeopleProvider>
        <PersonList />
    </PeopleProvider>
</div>
);
```

app.js

```jsx
import React, { useContext } from 'react';
import { PeopleContext } from './PeopleProvider';

function PersonList() {
    const { people } = useContext(PeopleContext);

    ...
```

PersonList.js

Le hook useContext

- Utilisation : possibilité aussi de passer dans le contexte des fonctions

```jsx
import React, { createContext } from 'react';

// Créer un contexte pour stocker l'état global
const PeopleContext = createContext();

function PeopleProvider({ children }) {
  const [people, setPeople] = useState([]);

  const addPerson = (person) => {
    setPeople([...people, person]);
  };

  return (
    <PeopleContext.Provider value={{ people, addPerson }}>
      {children}
    </PeopleContext.Provider>
  );
}

export { PeopleProvider, PeopleContext };
```

PeopleProvider.js

```jsx
import React, { useContext, useState } from 'react';
import { PeopleContext } from './PeopleProvider';


function PersonList() {
    const { people } = useContext(PeopleContext);
    const { addPerson } = useContext(PeopleContext);

    ...
```

PersonList.js