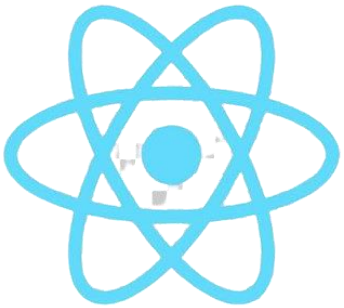
React JS



En bref !

React JS est une bibliothèque JavaScript utilisée pour créer des interfaces utilisateurs. Développée par Facebook (Meta) en 2013, elle doit répondre à une problématique, une meilleure synchronisation des états avec l'interface utilisateur (UI). Pour cela, React JS sépare l'état de la vue.

React JS n'est pas un Framework MVC comme ses concurrents peuvent l'être, c'est une bibliothèque qui encourage à utiliser des composants réutilisables, qui peuvent évoluer dans le temps.

Pourquoi utiliser React JS ?

React JS n'est pas le « meilleur » des Framework JS, il est juste différent des autres, de par son approche plus simple, mais pas nécessairement dénué de complexité, de par son écosystème, React est présent depuis plusieurs années et n'a été que peu modifié, ce qui a permis la mise en place d'une pléthore d'outils et de librairie facilitant la création de composants réutilisables. ReactDOM pour des applications Web, React Native pour des applications mobile ou React Desktop pour des applications de bureau. Ensuite, React se base sur une syntaxe spécifique, JSX (JavaScript Syntax Extension) qui est une petite extension du JavaScript pour la templatisation, l'avantage c'est ça facilité d'apprentissage.

Et enfin, React JS doit rester avant tout une préférence personnelle, il fonctionne d'une manière qui pourra plaire ou non. Il est donc important de se plonger dans d'autres Framework pour se faire son propre avis.

Les prérequis

Le prérequis principal est bien-sûr d'être à l'aise avec JavaScript, et surtout avec la version ES6.

Installer Node JS, pour cela référez-vous à se lien :
https://welovedevs.com/fr/articles/tuto-install-node-js-windows/

Connaitre les commandes de base de NPM :

```
//connaitre la version de node
node –v

//connaitre la version de npm
npm –v

//installer la dernière version de npm
npm i npm –g

// initialiser un package
npm init -y

// installer un package
npm i nom_du_paquet

//ou un package globalement
npm i –g nom_du_paquet

// installer des paquets en tant que devDependencies
npm i nom_du_paquet -D
```

Commencer son premier projet React JS

Il y a plusieurs manières de démarrer un projet React, mais celle que je préconise pour gagner du temps c'est d'utiliser create-react-app.

create-react-app est un outil permettant de configurer votre environnement de développement, le squelette de votre code. Il embarque un certain nombre d'outils préconfigurés, tels que Webpack, Babel et ESLint, afin de vous garantir la meilleure expérience de développement possible.

Pour initialiser votre projet, nous allons faire :

```
npx create_react_app 'mon projet' //Pas de majuscule
```

Parfait votre projet est désormais initialisé. Maintenant vous devez entrer dans votre projet avec la commande :

```
cd 'mon projet'
```

À partir de là, vous pouvez ouvrir votre éditeur de texte préféré, moi j'utilise VS Code, mais libre à vous de choisir le vôtre.

Vous trouverez trois dossiers :

- node_modules : c'est là que sont installées toutes les **dépendances** de notre code. Ce dossier peut vite devenir très volumineux.
- public : dans ce dossier, vous trouverez votre **fichier index.html** et d'autres fichiers relatifs au référencement web de votre page.
- src : vous venez de rentrer dans le cœur de l'action. **L'essentiel des fichiers que vous créerez et modifierez seront là**.

Et faisons maintenant un petit tour des fichiers importants :

- package.json : situé à la racine de votre projet, il vous permet de **gérer vos dépendances** (tous les outils permettant de construire votre projet), vos scripts qui peuvent être exécutés avec yarn, etc. Si vous examinez son contenu, vous pouvez voir des dépendances que vous connaissez : React et ReactDOM :
    - vous y trouverez react-scripts, créé par Facebook, qui permet d'installer Webpack, Babel, ESLint et d'autres pour vous faciliter la vie ;
- dans /public, vous trouvez index.html. Il s'agit du **template de votre application**. Il y a plein de lignes de code, mais vous remarquez ? Comme dans les chapitres précédents, nous allons y ancrer notre app React ;
- dans /src, il y a index.js qui permet d'**initialiser notre app React** ;
- et enfin, dans /src, vous trouvez App.js qui est **notre premier composant React**.

Deux fichiers que nous n'utiliserons pas directement mais qui ne font pas de mal à garder :

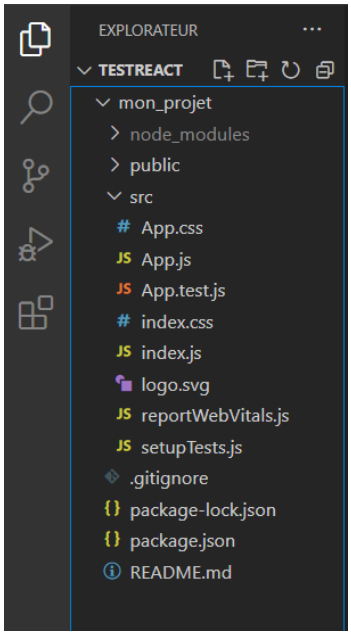
- le README.md qui permet d'afficher une page d'explication si vous mettez votre code sur GitHub, par exemple ;
- et le fichier .gitignore qui précise ce qui ne doit pas être mis sur GitHub, typiquement le volumineux dossier des node_modules.

Lorsque vous vous trouvez à la racine de votre projet, vous pouvez exécuter 'npm start' qui va démarrer votre application en mode développement.

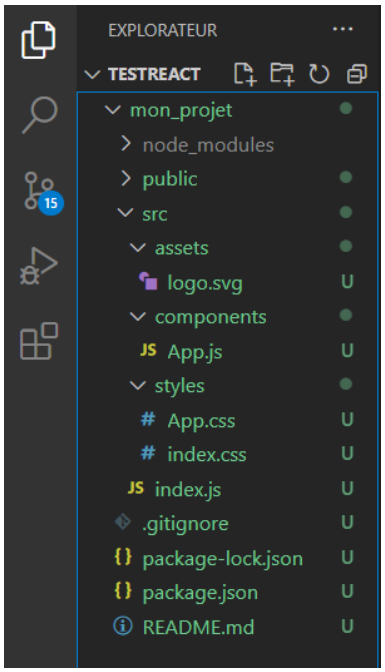
Un onglet a dû s'ouvrir dans votre navigateur à l'URL http://localhost:3000/.

Maintenant que votre application est fonctionnelle, nous allons réorganiser un peu notre arborescence. On va commencer par créer un dossier /components dans /src, où nous mettrons tous nos composants. On y glisse App.js et on en profite pour changer le chemin d'import dans index.js. Ensuite, ont créé le dossier /styles, où nous mettrons toutes nos feuilles de style, on peut y glisser App.css. Et enfin, un dossier /assets, qui regroupera toutes les images ou documents que l'on utilisera dans notre application, vous pouvez y glisser logo.svg. Le reste des fichiers est inutile à part index.js qu'il faut absolument garder.

Votre arborescence devrait passer de ça :



à ça :



Cette bonne pratique permet de rendre votre code lisible par le plus grand nombre et surtout de vous y retrouver et d'être plus efficace.

Voilà vous êtes parait pour commencer à développer sur React JS

Les nouveautés ESNext à maîtriser dans React JS

Variables et Portée :

Une des premières nouveautés de JavaScript a été la façon dont on peut appeler des variables. Habituellement on déclarait toujours une nouvelle variable avec le mot-clé **var**.

La portée (le scope en anglais) de **var** est énorme et du coup il est très facile de polluer son application.

Déclarer une variable avec **let** permet de limiter sa portée dans la fonction ou le bloc (un if, un for…) dans lequel elle a été déclarée. Du coup on va désormais préférer utiliser **let** face à **var**, car il convient dans la grande majorité des cas.

Enfin, lorsque l'on déclare une variable qui au final ne va pas varier, on utilise **const**.

```
var a = 1 ;
```

```
let b = 2 ;
const c = 3 ;

function test(){
        console.log(a) ; // 1
        console.log(b) ; // 2
        console.log(c) ; // 3

        if(a == 1) {
                let d = 4;
                console.log(d); // 4
        }
        console.log(d); // error: undefined

}
test();

c+= 3; // error: assignement to constant variable
```

Tout comme **var**, **let** et **const** sont « vus » par les fonctions déclarées en dessous : dans la fonction test() on peut bien afficher la valeur de a, b et c.

Par contre comme d est défini dans le if, il est impossible d'y accéder de l'extérieur.

Pour faire simple, si on déclare quelque chose qui ne va pas changer, on utilise const. Si on déclare une variable, en général let fera l'affaire.

Fonction fléchée ou Arrow Function :

Pour déclarer une fonction, jusque-là on faisait simplement function toto() { … }. Rien de plus simple. Mais désormais on utilise la double flèche, ou fat arrow en anglais et ça ressemble à ça :

```
// Js standard
function truc( x, y ) { … }

// ESNext
const toto = ( x, y ) => { … } // Fonction avec plusieurs paramètres
const momo = x => { … } // Fonction avec un seul paramètre
const bobo = () => { … } // Fonction sans paramètre
```

Cela permet d'être plus concis et c'est plus court à écrire.

Notez que l'on stocke la fonction dans une variable via l'utilisation du mot-clé const. Et lorsqu'il n'y a qu'un seul paramètre attendu, on peut ne pas mettre les parenthèses après le =.

Un autre avantage c'est qu'en JS on utilise beaucoup le mot-clé **this** pour faire référence à l'objet qu'on est en train de manipuler, et quand on le passe dans une fonction (par exemple un événement) **this** ne fait plus référence à l'objet mais à l'événement. Du coup auparavant il fallait bricoler un peu. Cette nouvelle notation conserve la bonne référence à **this**.

Concaténation simplifiée :

En JS la concaténation a toujours était contraignant. Mais désormais on a une méthode pour le faire simplement. Il faudra utiliser les guillemets renversés (ou **backtick** en anglais), avec **alt gr + 7**.

```
 // JS
"Bonjour, je m'appelle " + name + " et j'ai " + age + " ans";

// ESNext
`Bonjour je m'appelle ${name} et j'ai ${age} ans`;
```

De plus il est désormais tout à fait possible d'écrire une chaine sur plusieurs lignes (alors qu'auparavant il fallait absolument séparer avec un + entre chaque ligne.

L'assignation déstructurée :

Comme on va le voir on utilisera souvent des objets et des sous-objets dans notre code. On va notamment récupérer les données de nos blocs qui seront stockées dans un objet **props**, puis un sous-objet attributes.

```
// Au lieu d'utiliser
props.attributes.number
props.attributes.title
props.attributes.radius
props.attributes.width

// On pourra extraire
const { number, title } = props.attributes

<RichText.Content value={ number } /> // On utilise seulement number
```

Avec l'assignation déstructurée on indique à JS que l'on veut extraire les éléments, ou seulement une partie d'entre eux.

Cela permet aussi dès le début du code de tenir un inventaire des objets dont on va faire usage par la suite.

Les classes :

Les classes permettent une programmation orientée objet (POO) comme dans tous les langages de programmation.

Jusqu'à récemment JS n'avait pas exactement cette approche par classe. C'est aujourd'hui corrigé. Lorsque l'on fait du React, chaque composant de notre application sera représenté par une classe.

Voici une classe représentant un composant:

```
export default class Gmap extends Component {
  createMap = () => {
    // méthode pour créer la map
  }

  createMarker = () => {
    // méthode pour créer un marker et le placer sur la map
  }

  render() {
    return( … ) // affiche le rendu du composant
  }
}
```

Le mot clé **export** permet de rendre cette classe (et donc ce composant dans le cas de React) disponible ailleurs, il suffira alors de l'importer avec import, que l'on a vu un peu avant.

Vous verrez parfois certains développeurs déclarer simplement la classe et faire le **export** default à la fin du fichier. C'est une autre façon de l'écrire mais sachez que ça revient exactement au même.

Notez enfin que dans une classe on n'est pas obligé de mettre const avant la déclaration d'une fonction.

L'assignation simplifiée :

Parfois quand on assigne des valeurs à un objet, on a tendance à se répéter inutilement : la clé et la valeur ont le même nom. Heureusement en ESNext on peut simplifier cela :

```
// Dans un objet
{ city: city, country: country } // écriture classique
{ city, country } // écriture simplifiée

// Dans un composant

// Au lieu de :
<Inspector
        onChangeContent={ onChangeContent }
        defaultColor={ defaultColor }
/>

// On peut écrire :
<Inspector
        { ...{ onChangeContent, defaultColor } }
/>
```

Dans le premier exemple on cherche à assigner la valeur de city dans un objet dont la clé sera également city. L'écriture simplifiée permet d'éviter la répétition.

Pareil pour les composants React. On va très souvent leur passer en paramètre des valeurs et des fonctions, et au lieu d'assigner une par une chaque valeur, on peut utiliser l'écriture simplifiée.

Notez dans ce cas la présence des ...{} qui permet d'indiquer à React que l'on veut une écriture attribut=valeur.

Les syntaxes React :

La première chose à savoir c'est qu'un composant React dispose d'une fonction render() dans laquelle on va retourner du HTML, ce qui n'est pas possible avec du JS natif.

Voici à quoi peut ressembler le render d'un composant React :

```
render() {
  return(
    <div className="my-component">
            {this.title}
    </div>
  )
}
```

Si vous avez remarqué, on n'utilise pas le mot **class** pour appeler une classe CSS dans la div mais **className**, parce que mine de rien **class** est un mot-clé réservé du langage JavaScript et à la compilation ça causerait des erreurs. Du coup React a créé le substitut **className**.

Les composants React :

Comme on l'a vu un composant React est avant tout une classe, qui va être un peu spécifique. Imaginons que j'ai une liste de tâches qui va appeler une tâche, je n'aurais qu'à appeler la balise <Task />.

```
import Task from './task'

//…

render(){
  return (
    <li>
      <Task />
    </li>
  }
)
```

```
export default class Task extends Component {

  render() {
    return(
     <p>Ma tâche</p>
    )
  }
}
```

Dans index.js, j'appelle mon composant Task en l'important, et lors du render()j'appelle la balise <Task /> pour indiquer que c'est ici que je veux afficher mon composant.

Cela permet de bien découper mon application en plusieurs composants, et d'aérer un peu le code : chaque composant embarque ses propres fonctions. Et on ne se gênera pas pour le faire, afin de garder notre code super lisible !

Un seul bloc à la fois :

React n'aime pas que vous rendiez un composant qui contient plusieurs balises HTML concomitantes. La règle ici est simple : lorsque vous rendez du code via la fonction render(), assurez-vous d'avoir toujours un bloc englobant :

```
// Pas bon
return (
        <Bidule />
        <Truc />
)

// Ok !
return (
```

```
  <div className="container">
            <Bidule />
            <Truc />
  </div>
)
```

Le ternaire :

Vous connaissez l'opérateur ternaire, qui permet de faire un if et juste derrière dire : si c'est vrai j'affiche ceci et si c'est faux j'affiche cela. En JSX il s'écrit comme ça :

```
  render() {
  return(
    { props.attributes.postID ? (
      <Post>
    ) : (
      <p>Choisissez un article</p>
    ) }
  )
}
```

Dans cet exemple on regarde si on a défini un postID : si oui on va afficher le sous-composant <Post />, et sinon on va afficher un message qui incite à d'abord aller sélectionner un article.

Notez bien que l'on empacte nos deux possibilités entre des parenthèses, dans lesquelles on balance directement du HTML. Et il ne faut pas oublier d'entourer le tout d'accolades.

La double négation :

```
render() {
  return(
            !! condition && ( // double négation
      <Inspector />
    )
  )
}
```

Si j'avais mis qu'un seul point d'exclamation on aurait tout de suite compris que je cherchais à faire quelque chose quand la condition n'est pas remplie.

Mais là pourquoi 2 ? Puisque que l'on cherche à faire quelque chose quand la condition est valide alors pourquoi ne pas simplement rien mettre ?

Et bien en fait c'est une astuce pour garantir d'avoir un true / false à tous les coups. Selon les cas une valeur JS peut être null, false, undefined…

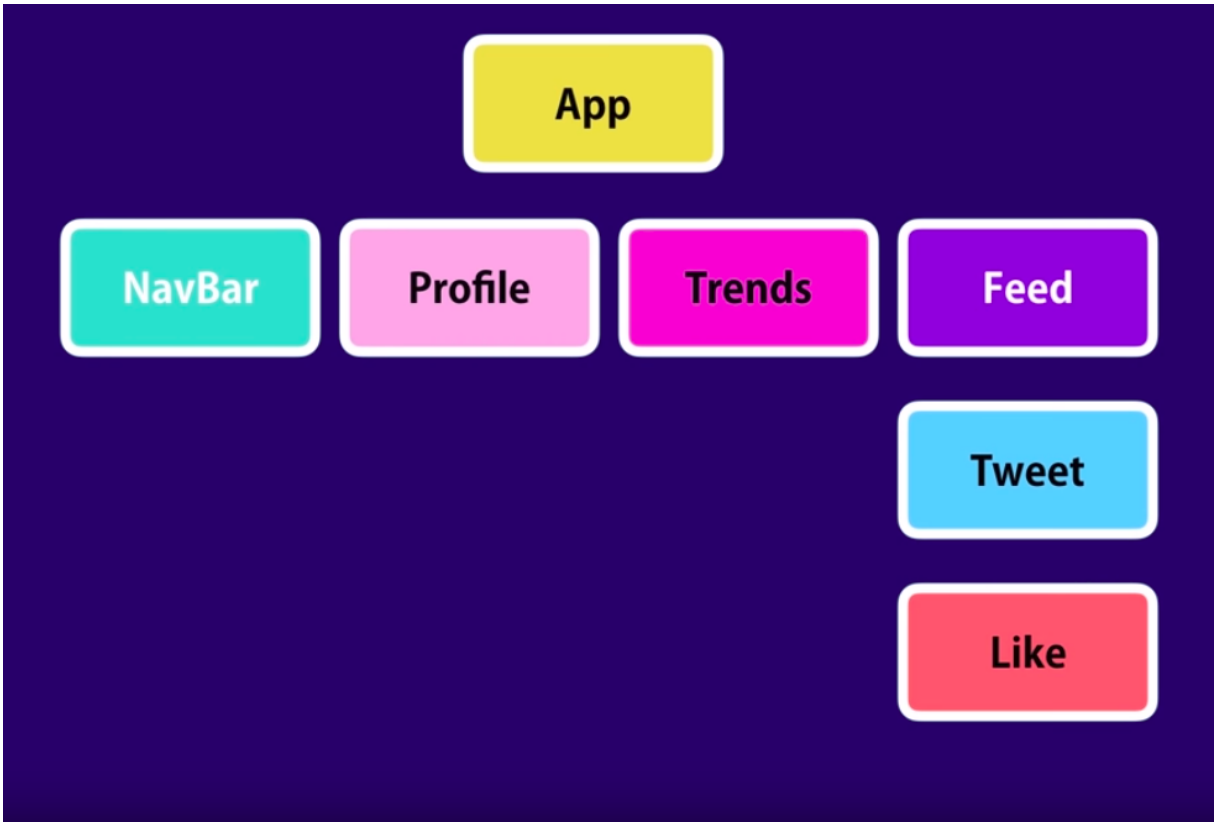
La double négation permet de dire que, peut importe l'état, si ce n'est pas true, alors c'est false. Cela nous évite de devoir tester le type de la variable, voir si elle existe… et donc nous fait gagner du temps.

En clair : **c'est un moyen pratique pour nous permettre de forcer le test d'un booléen**.

Les grands principes de React :

Les components :

Les composants sont des "morceaux" d'interface qui sont hiérarchiquement imbriqués. Conceptuellement, les composants sont comme des fonctions JavaScript. Ils acceptent des entrées (appelées « props ») et renvoient des éléments React décrivant ce qui doit apparaître à l'écran.



Le moyen le plus simple de définir un composant consiste à écrire une **fonction JavaScript** :

```
function Welcome(props) {
  return <h1>Bonjour, {props.name}</h1>;
}
```

Cette fonction est un composant React valide car elle accepte un seul argument « props » (qui signifie « propriétés ») contenant des données, et renvoie un élément React. Nous appelons de tels composants des « fonctions composants », car ce sont littéralement des fonctions JavaScript.

Vous pouvez également utiliser **une classe ES6** pour définir un composant :

```
class Welcome extends React.Component {
  render() {
    return <h1>Bonjour, {this.props.name}</h1>;

  }
```

```
}
```

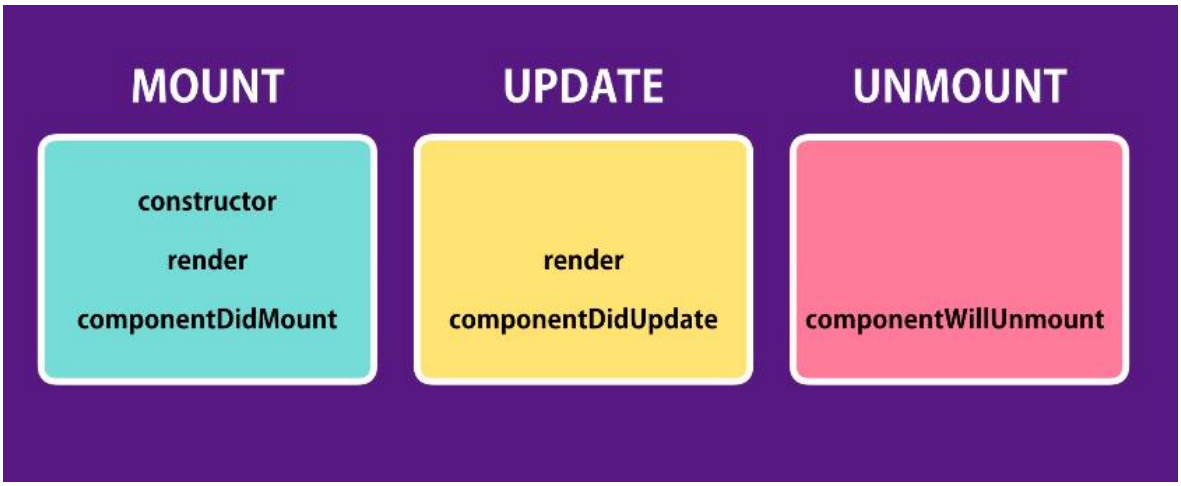
Les deux composants ci-dessus sont équivalents du point de vue de React.

Le Virtual DOM :

React crée un **DOM virtuel** qu'il est facile de créer et de modifier et qui permet de mettre à jour le DOM (en réaction… d'où le nom de la bibliothèque…).
Lorsque l'état (state) d'un élément REACT change, React crée un nouvel élément, le compare à l'ancien élément existant et modifie uniquement la partie à modifier sur le DOM virtuel (information brute) puis sur DOM réel (les balises HTML).

Le cycle de vie d'un composant :

Les composant nous offrent la possibilité d'insérer notre code (hook) à plusieurs étapes de la vie d'un component :



- **constructeur** : c'est l'endroit (moment) pour donner des valeurs de façon synchrone à la propriété state. Il sera cependant impossible d'appeler setState ici car cette méthode demande que le DOM ait déjà été chargé (après un premier render).
- **render** : c'est ici que se construit le DOM avec les éventuels changement que cela implique concernant la visualisation
- **componentDidMount** : c'est l'endroit à privilégier pour des appels asynchrones (fetch par exemple ou xhr...)

- **componentDidUpdate(prevProps, prevState, snapshot)**: à utiliser dans le cas ou l'on veut faire un appel AJAX après qu'un component a été modifié. Permet de vérifier que les props ou le state actuels concernés diffèrent des anciennes valeurs.
- **componentWillUnmount** : C'est l'endroit ou faire le ménage pour des éléments comme l'invalidation de minuteurs, l'annulation de requêtes réseau...

L'objet State et Props

Les props (diminutif de « propriétés ») et les state sont tous les deux des objets JavaScript bruts. Même s'ils contiennent tous les deux des informations qui influencent le résultat produit, ils présentent une différence majeure : props est passé au composant (à la manière des arguments d'une fonction) tandis que state est géré dans le composant (comme le sont les variables déclarées à l'intérieur d'une fonction).

Le State :

Pour faire le lien entre les propriétés d'un objet React et son rendu (le dom réel), il faut :

- Créer un objet state avec une ou des propriétés
- Utiliser ces propriété dans le render
- Modifier les propriétés de state avec la méthode setState

setState

En résumé,

- lorsque la valeur à modifier d'une propriété du state dépend de son ancienne valeur, il faut utiliser la fonction "updater" dont la signature est la suivante : (state, props) => stateChange
  Cf ex 1.
- lorsque la valeur à modifie d'une propriété du state ne dépend pas de son ancienne valeur, on peut simplement passer un objet en argument à setState. Cf ex 2.

Ex 1 - utilisation de setState avec la fonction updater :

```
import React, { Component } from 'react';
class App extends Component {
  state = {
    counter: 0
   }
   handleClickAddButton = () => {
    console.log(`Dans handleClickAddButton`);
    this.setState((state, props) => {
      return {counter: state.counter + 1};
    });
   }
  render() {
    return (
      <>
      <div className="container">
        <h1>{this.state.counter}</h1>
        <button onClick={this.handleClickAddButton}>Ajouter 1</button>
      </div>
      </>
    );
  }
}
export default App;
```

Ex 2 - utilisation de setState avec un objet

```
import React, { Component } from 'react';
class App extends Component {
  state = {
    counter: 0
   }

   handleClickSet10Button = () => {
    console.log(`Dans handleClickSet10Button`);
```

```
    this.setState({counter: 10});
  }
  render() {
    return (
      <>
      <div className="container">
        <h1>{this.state.counter}</h1>
        <button onClick={this.handleClickSet10Button}>Compteur à 10</button>
      </div>
      </>
    );
  }
}
export default App;
```

Le Props :

Un "component" père peut créer des "components" fils.
Pour cela, il faut :

- que le component père importe la classe du component fils
- créer des éléments fils dans le “render” avec à minima l’attribut key (qui pourra être retrouvé en utilisant props.id) et ensuite autant de paires “clé valeur” que nécessaire qui seront passés en argument pour initier le component fils
- que le component fils récupère ces arguments via la l'objet **props**

Les props sont en lecture seule. Un composant ne doit jamais modifier ses propres props.

Ex :

```
import React, { Component } from "react" ;
import Counter from "./counter";
```

```
import { compileFunction } from "vm";
class Counters extends Component {
 state = {
   counters: [
     { id: 1, value: 0 },
     { id: 2, value: 1 },
     { id: 3, value: 2 },
     { id: 4, value: 3 },
     { id: 5, value: 4 }
   ]
 };
 render() {
   return (
     <div>
       {this.state.counters.map(counter => (
         <Counter key={counter.id} value={counter.value} selected={true} />
       ))}
     </div>
   );
 }
}
export default Counters;
```

Pour passer du code HTML d'un component père à un component fils, il suffit d'ajouter le code suivant sur le composant père :

```
<Counter key={counter.id} value={counter.value} selected={true} />
  <h3>Titre {counter.id} ici </h3>
</Counter>
```

Puis de récupérer ces informations dans la méthode "render" du composant fils :

```
{this.props.children}
```

Différence entre state et props

Props est un “entrant” qui est donné à un component depuis un autre component (souvent le père).
State est local à chaque component
Les autres components ne peuvent atteindre les “states” d’un component.
En revanches, les autres components peuvent donner des props à un component.
Props est “en lecture seule”. Un component ne peut changer ses props alors qu’il peut changer son “state”.

Pour propager un événement à un composant parent il suffit pour cela que le component parent passe à l’enfant le nom de la méthode qui va être appelée via les props. Ex :

```
// Père
handleDelete() {
   console.log("Efface man !");
}
...
<Counter
          onDelete={this.handleDelete}
          key={counter.id}
          value={counter.value}
          selected={true}
      />

// Fils
<button
        className="btn btn-danger btn-sm m-2"
        onClick={this.props.onDelete}
       >
```

Il est important de comprendre qu’il n’est pas possible de donner la possibilité à deux composants de modifier les mêmes données.
C’est au composant “ancêtre” de faire le travail sans quoi tout ce qui sera fait par le descendant ne sera pas pris en compte car deux “DOM virtuels”

vont cohabiter sans se connaître. On appelle cela "faire remonter l'état".
En définitive seul le component ancêtre devra :

- posséder un objet “state”
- posséder des méthodes qui modifient le DOM via la méthode setState.

Gestion des événements et des formulaires

Quelques caractéristiques de la déclaration d'un événement en React :

- l'événement s'écrit dans une balise en **camelCase**;
- vous **déclarez l'événement** à capter, et **lui passez entre accolades la fonction** à appeler ;
- contrairement au JS, dans la quasi totalité des cas, **vous n'avez pas besoin d'utiliser addEventListener**.

Par exemple, le HTML suivant :

```
<button onclick="activateLasers()">
  Activer les lasers
</button>
```

est légèrement différent avec React:

```
<button onClick={activateLasers}>  Activer les lasers
</button>
```

Autre différence importante : en React, on ne peut pas renvoyer false pour empêcher le comportement par défaut. Vous devez appeler explicitement preventDefault. Par exemple, en HTML, pour annuler le comportement par défaut des liens qui consiste à ouvrir une nouvelle page, vous pourriez écrire :

```
<a href="#" onclick="console.log('Le lien a été cliqué.'); return false">
  Clique ici
</a>
```

En React, ça pourrait être :

```
function ActionLink() {
  function handleClick(e) {    e.preventDefault();    console.log('Le lien a été cliqué.');  }
  return (
    <a href="#" onClick={handleClick}>      Clique ici
    </a>
  );
}
```

Ici, e est un événement synthétique. React le définit en suivant les spécifications W3C, afin que vous n'ayez pas à vous préoccuper de la compatibilité entre les navigateurs. Les événements React ne fonctionnent pas tout à fait comme les événements natifs. Pour en apprendre davantage, consultez le guide de référence de SyntheticEvent.
Lorsque vous utilisez React, vous n'avez généralement pas besoin d'appeler la méthode addEventListener pour ajouter des écouteurs d'événements *)* à un élément du DOM après que celui-ci est créé. À la place, on fournit l'écouteur lors du rendu initial de l'élément.
Lorsque vous définissez un composant en utilisant les classes ES6, il est d'usage que le gestionnaire d'événements soit une méthode de la classe. Par exemple, ce composant Toggle affiche un bouton qui permet à l'utilisateur de basculer l'état de "ON" à "OFF".

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // Cette liaison est nécéssaire afin de permettre     // l'utilisation de `this` dans la fonction de rappel.    this.handleClick = this.handleClick.bind(this);  }

  handleClick() {    this.setState(state => ({      isToggleOn: !state.isToggleOn }));  }
  render() {
    return (
      <button onClick={this.handleClick}>        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

ReactDOM.render(
```

```
  <Toggle />,
  document.getElementById('root')
);
```

Les formulaires HTML fonctionnent un peu différemment des autres éléments du DOM en React car ils possèdent naturellement un état interne. Par exemple, ce formulaire en HTML qui accepte juste un nom :

```
<form>
  <label>
    Nom :
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Envoyer" />
</form>
```

Ce formulaire a le comportement classique d'un formulaire HTML et redirige sur une nouvelle page quand l'utilisateur le soumet. Si vous souhaitez ce comportement en React, vous n'avez rien à faire. Cependant, dans la plupart des cas, vous voudrez pouvoir gérer la soumission avec une fonction JavaScript, qui accède aux données saisies par l'utilisateur. La manière classique de faire ça consiste à utiliser les « composants contrôlés ».

Composants contrôlés :

En HTML, les éléments de formulaire tels que <input>, <textarea>, et <select> maintiennent généralement leur propre état et se mettent à jour par rapport aux saisies de l'utilisateur. En React, l'état modifiable est généralement stocké dans la propriété state des composants et mis à jour uniquement avec setState().

On peut combiner ces deux concepts en utilisant l'état local React comme « source unique de vérité ». Ainsi le composant React qui affiche le formulaire contrôle aussi son comportement par rapport aux saisies de l'utilisateur. Un champ de formulaire dont l'état est contrôlé de cette façon par React est appelé un « composant contrôlé ».

Par exemple, en reprenant le code ci-dessus pour afficher le nom lors de la soumission, on peut écrire le formulaire sous forme de composant contrôlé :

```
class NameForm extends React.Component {

  constructor(props) {
```

```
    super(props);

    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);

    this.handleSubmit = this.handleSubmit.bind(this);

  }

  handleChange(event) {    this.setState({value: event.target.value});  }

  handleSubmit(event) {

    alert('Le nom a été soumis : ' + this.state.value);

    event.preventDefault();

  }

  render() {
    return (

      <form onSubmit={this.handleSubmit}>

        <label>
          Nom :
          <input type="text" value={this.state.value}
onChange={this.handleChange} />
        </label>

        <input type="submit" value="Envoyer" />

      </form>
    );
  }
}
```

Les Hooks

UseState est une fonction qui renvoie un tableau :

- Le premier élément est une variable d'état.

- Le deuxième élément est une fonction qui permet de modifier cette variable.

Les variables d'état permettent de rafraîchir le rendu de la page afin d'afficher les modifications.

Pour pourvoir l'utiliser, il faut d'abord l'importer.

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

Ensuite, il faut le déclarer les variables qui seront respectivement des référence vers la variable d'état et vers la fonction qui permet de la modifier :

```
const [maVariable, setMaVariable] = useState(valeurInitial);
```

Ici maVariable et setMaVariable peuvent prendre n'importe quel nom grâce à la décomposition (destructuring) positionnelle.

maVariable est la variable qui va s'afficher.

SetMaVariable permet de modifier MaVariable.

ATTENTION ! Il ne faut pas modifier directement la valeur de MaVariable. On est obligé de passer par une fonction.

Exemple:

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


function App() {
  const [counter, setCounter] = useState(0); // Je déclare ma variable d'état et la fonction pour la modifier.


  return (
    <div>
      <p>{counter}</p> // J'affiche la variable counter dans un paragraphe.
      <button onClick={() => setCounter(counter + 1)}>Increment</button> // à chaque clique, j'incémente de 1 la variable counter et le rendu de la page se rafraîchit.
    </div>
  );
```

```
}
```

useContext utilisé de pair avec useState, permet de partager plus facilement les variables d'états et leurs fonctions (state dans la suite du document) dans des composants profondément imbriqués.

Le state étant dans le plus haut ancêtre commun dans la pile des composants qui le requièrent, il fallait le passer à travers les props des divers composants afin d'atteindre le composant qui en a réellement besoin.

Exemple :

```
function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </>
  );
}

function Component2({ user }) {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 user={user} />
    </>
  );
}

function Component3({ user }) {
  return (
    <>
```

```
      <h1>Component 3</h1>
      <Component4 user={user} />
    </>
  );
}

function Component4({ user }) {
  return (
    <>
      <h1>Component 4</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
```

Dans cette exemple, nous voyons que le state est partagé dans les composants 2 et 3 mais n'est pas utilisé.

La solution est d'utiliser le contexte.

Pour cela, il faut d'abord l'importer et le créer.

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

const UserContext = createContext()
```

Ensuite, il faut enrober les descendants dans le founisseur de contexte (context.provider) et attribuer au fournisseur la valeur à partager dans les descenants.

```
function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
```

```
      <Component2 user={user} />
    </UserContext.Provider>
  );
}
```

Afin d'utiliser le contexte, nous importons useContext et l'appellons avec comme argument la variable instancié à la création du contexte.

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

const user = useContext(UserContext);
```

Exemple complet :

```
import { useState, createContext, useContext } from "react";

const UserContext = createContext(); //Création du contexte

function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}> //Enrobage des descendants
      <h1>{`Hello ${user}!`}</h1>
      <Component2 />
    </UserContext.Provider>
  );
}

function Component2() {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 />
    </>
```

```
  );
}

function Component3() {
  return (
    <>
      <h1>Component 3</h1>
      <Component4 />
    </>
  );
}

function Component4() {
  const user = useContext(UserContext); //Utilisation du contexte

  return (
    <>
      <h1>Component 4</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
```

Dans cette exemple, nous voyons que nous ne passons plus user dans les composants 2 et 3

useEffect permet d'exécuter des effets secondaires dans vos composants.

Quelques exemples d'effets secondaires : récupération de données, mise à jour directe du DOM et minuteurs (setTimeout).

useEffect peut être appeler de 3 manières différentes.

```
1. useEffect(() => {
2.    //S'exécute à chaque rendu.
```

```
3. });
4. useEffect(() => {
5.   //Imite ComponentDidMount et ne s'exécute qu'au premier rendu.
6. }, []);
7. useEffect(() => {
8.   //S'exécute au premier rendu
9. //Imite ComponentDidUpdate et s'exécute à chaque mise à jour de la
   liste de dépendances.
10.}, [deps]);
```

Exemples:

```
1. import { useEffect } from 'react';
2. function Salutation({ name }) {
3.   const message = `Bonjour, ${name}!`;
4.   useEffect(() => {
5.     // Modifie le titre de la page une seule fois
6.     document.title = 'Page de salutation';
7.   }, []);
8.   return <div>{message}</div>;
9. }
10.import { useEffect } from 'react';
11.function Salutation({ name }) {
12.  const message = `Bonjour, ${name}!`;
13.  useEffect(() => {
14.    // Modifie le titre à chaque mise à jour de name
15.    document.title = 'Page de salutation';
16.  }, [name]);
17.  return <div>{message}</div>;
18.}
```

Utilisation avec un fetch:

```
import { useEffect, useState } from 'react';
function FetchEmployees() {
  const [employees, setEmployees] = useState([]);
  useEffect(() => {
    async function fetchEmployees() {
      const response = await fetch('/employees');
      const fetchedEmployees = await response.json(response);
      setEmployees(fetchedEmployees);
    }
    fetchEmployees();
  }, []);
  return (
    <div>
      {employees.map(name => <div>{name}</div>)}
    </div>
  );
}
```

useReducer est similaire au useState mais il permet de gérer plus facilement des states complexes en séparant la logique du composant.

useReducer accepte deux arguments.

```
[state, dispatch] = useReducer(<reducer>, <initialState>)
```

state : Variable d'état.

dispatch : Fonction permettant de modifier le state (fait référence à <reducer>).

<reducer> : La fonction comportant toute la logique du state.

<initialState> : La valeur initial du state.

La fonction <reducer> prend deux arguments:

```
function reducer(state, action) { }
```

state : la valeur du state courant.

action: C'est un objet contenant le type d'action à exécuter et les valeurs à modifier dans le state.

Exemple :

```
const initialTodos = [
  {
    id: 1,
    title: "Todo 1",
    complete: false,
  },
  {
    id: 2,
    title: "Todo 2",
    complete: false,
  },
];

const reducer = (state, action) => {
  switch (action.type) {
    case "COMPLETE":
      return state.map((todo) => {
        if (todo.id === action.id) {
          return { ...todo, complete: !todo.complete };
        } else {
          return todo;
        }
      });
    default:
      return state;
  }
};
```

```
function Todos() {
  const [todos, dispatch] = useReducer(reducer, initialTodos);

  const handleComplete = (todo) => {
    dispatch({ type: "COMPLETE", id: todo.id });
  };

  return (
    <>
      {todos.map((todo) => (
        <div key={todo.id}>
          <label>
            <input
              type="checkbox"
              checked={todo.complete}
              onChange={() => handleComplete(todo)}
            />
            {todo.title}
          </label>
        </div>
      ))}
    </>
  );
}
```

Les Routes :

Il faut commencer par installer la dépendance.

```
npm i react-router-dom
```

Le paquet React Router a un composant pratique appelé BrowserRouter. Nous devons d'abord l'importer depuis react-router-dom afin de pouvoir utiliser le routage dans notre application.

Il doit contenir tous les éléments de notre application où le routage est nécessaire. Cela signifie que si nous avons besoin du routage dans l'ensemble de notre application, nous devons envelopper notre composant le plus haut avec BrowserRouter.

Pour créer un itinéraire, nous devons importer Route à partir du package du routeur.

Ajoutez-le ensuite là où nous voulons afficher le contenu. Le composant Route a plusieurs propriétés. Mais ici, nous avons juste besoin de path et de render.

- path: c'est le chemin à charger lorsque l'itinéraire est atteint. Ici, nous utilisons / pour accéder à la page d'accueil.
- render: il restituera le contenu de la route. Ici, nous allons afficher un message de bienvenue à l'utilisateur.

Dans certains cas, desservir des itinéraires comme celui-ci est tout à fait correct, mais imaginez lorsque nous devons traiter un composant réel. Utiliser render ne serait pas une bonne solution.

Alors, comment pouvons-nous faire pour afficher quelque chose sans les accessoires de render ? Eh bien, le composant Route a une autre propriété nommée element.

```
import {Routes, Route } from "react-router-dom"
import Home from "./Home "
export default function App() {
 return (
  <Router>
   <main>
    <nav>
     <ul>
      <li>
          <Link to="/">Home</Link>     // lien
      </li>
     </ul>
    </nav>
```

```
        <Routes>

        <Route path="/" element={ < Home />} /> // Route vers le composant
        <Routes/>
      </main>
    </Router>

  )
}
```