Lecture individuelle n° 1

React

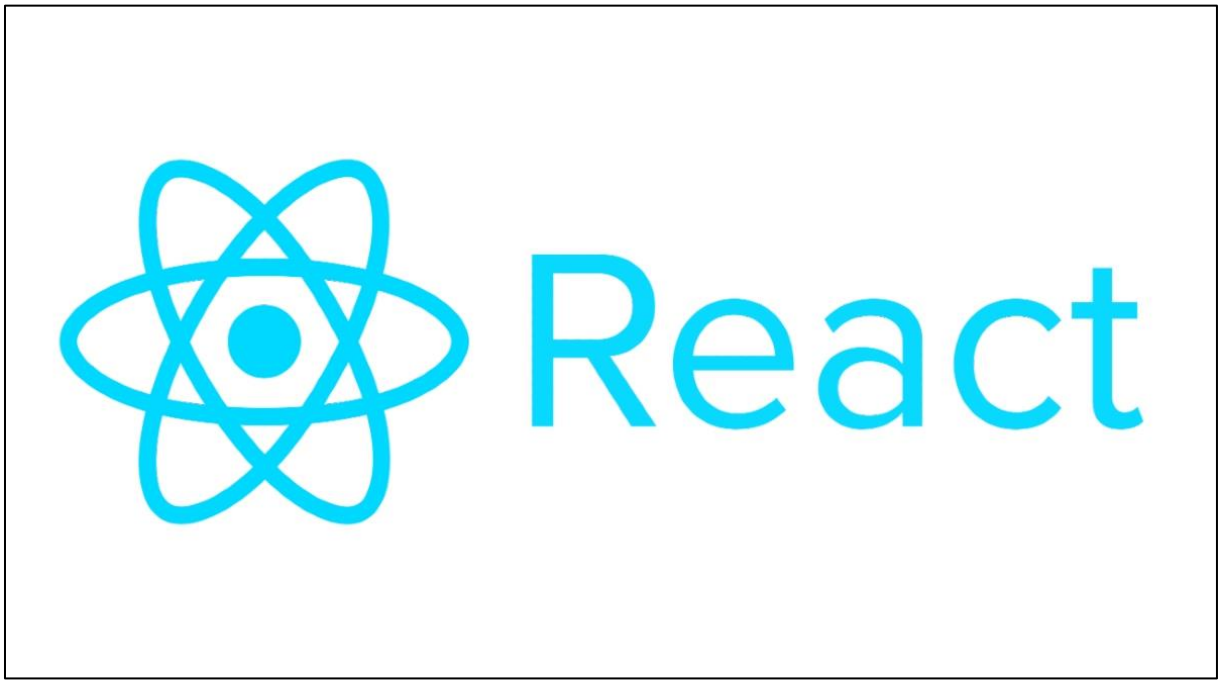


Table des matières

Introduction

React est une des bibliothèques JavaScript les plus populaires pour construire des interfaces web. Son approche par composants réutilisables en fait un outil particulièrement modulaire pour développer vos applications.

Cette lecture individuelle s'est basée sur un cours OpenClassRoom « Débutez avec React », mis à jour le 29 septembre 2023 et réalisé par Aurélien Antonio et Alexia Toulmet.

Dans cette lecture individuelle, nous commencerons par nous initier au fonctionnement de React et à la syntaxe JSX. Puis nous créerons les bases de notre application complète avec Create React App, à laquelle nous ajouterons du style, et du contenu ! La dernière partie sera l'occasion d'aborder le state et les effets de bord, afin de rendre l'application interactive pour l'utilisateur.

Dans ce cours, nous verrons les différents outils de React tout en se basant sur la création d'un site qui vend des plantes.

1. Présentation de React

React est un projet open-source, distribué sous la licence MIT et piloté par Facebook. Leurs produits web et mobile tels que Facebook, Messenger, Instagram, reposent en grande partie sur cette technologie. Comme React est open-source, vous pouvez accéder au code source directement sur GitHub, proposer une feature, ou même notifier d'un problème (*issue*).

L'ambition de React est de créer des interfaces utilisateurs, avec un outil rapide et modulaire. L'idée principale derrière React est que vous construisiez votre application à partir de composants. Un composant regroupe à la fois le HTML, le JS et le CSS, créés sur mesure pour vos besoins, et que vous pouvez réutiliser pour construire des interfaces utilisateurs.

React a différents avantages :

- Sa communauté très active
  Cela facilite énormément la résolution de problèmes, car il est presque impossible qu'en cherchant sur internet, on ne trouve pas quelqu'un d'autre qui a déjà fait face à notre problématique.

- Opportunités professionnelles
  Etant donné qu'il s'agit d'une des bibliothèques les plus répandues, les opportunités professionnelles sont particulièrement nombreuses.

- Documentation
  La documentation de React est riche, régulièrement mise à jour et intégralement traduite en français.

2. Appréhendez la logique de React

Les frameworks front-end

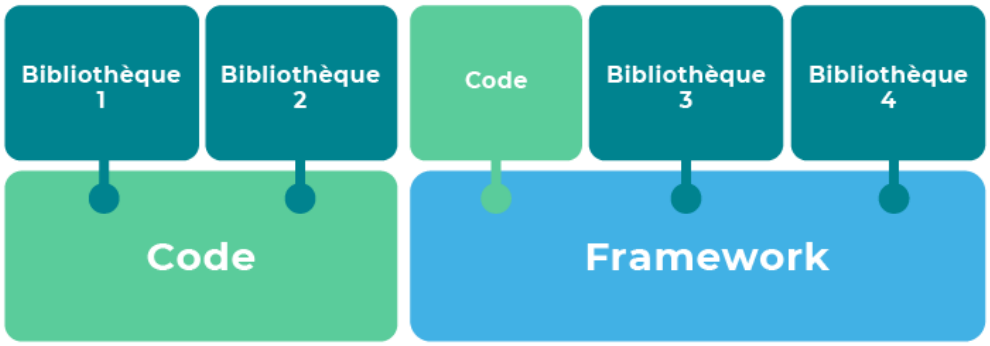
Un framework front-end est un ensemble de classes, fonctions et utilitaires qui nous facilite la création d'applications web. Ces outils sont compatibles avec tous les navigateurs.

Leur but est d'éviter de réinventer la roue pour n'importe quel besoin classique d'une application : gestion de l'interface utilisateur, des événements, du DOM, des formulaires, de l'évolution dans le temps des données manipulées par l'interface, etc. En plus, initialiser une base de code avec un framework simplifie non seulement votre prise de marques, mais aussi l'intégration d'une nouvelle personne sur la codebase.

Différence framework et bibliothèque

Un framework est un ensemble d'outils ultra complets permettant de créer une application de A à Z et fournissant tous les outils nécessaires au développement d'une application. Alors qu'une bibliothèque s'ajoute à une partie de votre application.

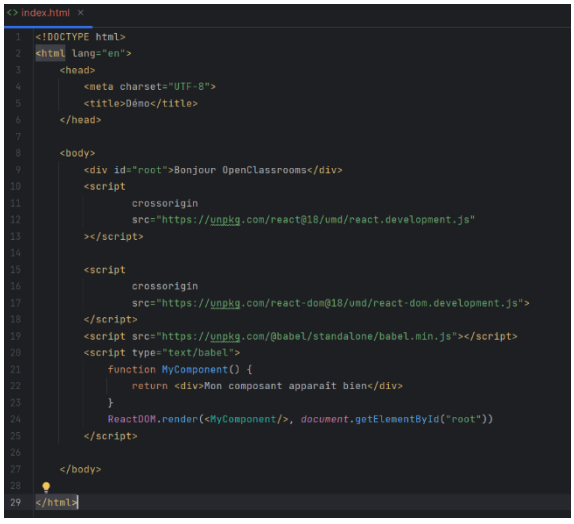
Une bibliothèque est beaucoup plus flexible, elle fournit un ensemble de ressources que nous pouvons combiner avec d'autres bibliothèques pour construire notre application.



Transformez un simple fichier HTML en app React

Ajouté dans une balise script, les liens CDN (*content delivery network*) permettent notamment d'importer une bibliothèque directement dans le code HTML.

A partir d'un index.html, j'ai créé une petite application React avec un composant :



```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Démo</title>
    </head>

    <body>
        <div id="root">Bonjour OpenClassrooms</div>
        <script
                crossorigin
                src="https://unpkg.com/react@18/umd/react.development.js"
        ></script>

        <script
                crossorigin
                src="https://unpkg.com/react-dom@18/umd/react-dom.development.js">
        </script>
        <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
        <script type="text/babel">
            function MyComponent() {
                return <div>Mon composant apparaît bien</div>
            }
            ReactDOM.render(<MyComponent/>, document.getElementById("root"))
        </script>

    </body>

</html>
```

Ici, on a importé 3 liens CDN indispensables pour notre app React :

- React – c'est l'API qui permet de gérer les composants
- React Dom – c'est l'API qui est responsable de générer les composants dans le DOM
- Babel – cet outil permet d'utiliser les dernières syntaxes de JS dans le navigateur (ES6+)

Nous avons également créé notre premier composant, qu'on appelle « functional components » c'est-à-dire une fonction qui retourne un élément React :

```
function MyComponent() {
  return (<div>Mon composant apparaît bien</div>)
}
```

Le Document Object Model (DOM)

Il est généré par votre navigateur depuis le HTML pour afficher une page. Il correspond à une sorte d'arbre de nœuds.

React lui-même ne manipule pas directement le DOM du navigateur. Cependant, il génère un DOM virtuel, distinct du DOM des navigateurs. Au moment venu, il réconcilie les deux, en prenant soin de minimiser le nombre d'opérations nécessaires. C'est ce qui nous permet d'avoir de super performances, et d'utiliser React dans des nombreux contextes, et pas seulement au sein du navigateur même, typiquement les applications mobiles natives, etc.

En résumé

Un framework JS est un ensemble de classes, fonctions et utilitaires qui nous facilitent la création d'applications pour les navigateurs ou mobiles.

L'un des outils les plus populaires, React, qui est une bibliothèque aussi bien qu'un framework, permet de créer des interfaces utilisateurs.

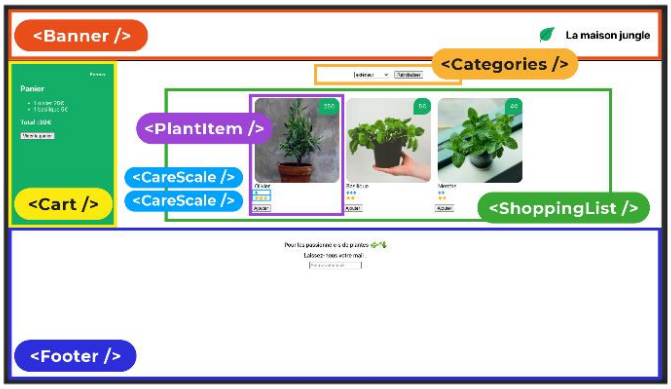
L'approche technique de React est de créer du code modulaire, à base de composants réutilisables.

Trois des avantages de React sont sa communauté, sa documentation et ses opportunités professionnelles.

3. Ecrivez du code modulaire avec les composants en JSX

En tant que développeur, au début d'un projet, nous devons essayer de repérer quel partie de notre application constitue des composants réutilisables.

Une interface est toujours constituée de différents éléments : des boutons, des listes, des titres, des sous-titres. Une fois rassemblés, ces éléments constituent une **interface utilisateur ou *UI***. Si je prends l'exemple de la maquette de notre site de plantes, vous voyez la barre de menu, le panier, et que pour chaque article, il y a un nom, une photo, une description.



Avec React, chacune de ces parties qu'on va pouvoir réutiliser correspond à un composant. Ils contiennent tout ce qui est nécessaire à leur bon fonctionnement : **la structure, les styles et le comportement** (par exemple, les actions qui sont déclenchées quand on clique dessus).

Les composants nous permettent d'utiliser la même structure de données, et de remplir ces structures avec différents jeux de données. Peu importe le nombre de plantes que vous aurez à mettre dans La maison jungle, vous pourrez les exploiter pour afficher vos données sans aucun effort. Et si dans le futur, vous avez besoin de créer une nouvelle page avec la même présentation, vous pourrez réutiliser le même composant.

C'est donc la mission des développeurs et développeuses React de découper toute interface utilisateur en éléments réutilisables, imbriqués les uns dans les autres. La majorité de nos composants sont eux-mêmes créés en **combinant d'autres composants** plus simples.

Initiez-vous au JSX

JSX est une extension de JavaScript créée par React, permettant d'utiliser notre syntaxe sous forme de tags <> directement dans le code Javascript.

Il s'agit d'ailleurs de **la spécificité de React** : contrairement aux autres frameworks où on écrit du HTML enrichi, les équipes de React ont créé le JSX, leur propre syntaxe basée sur JavaScript, qui permet de **mêler HTML et JS**.

Manipulez des données dans vos composants JSX

En React, les accolades { } sont également particulièrement utiles. Dès qu'il s'agit d'expressions JavaScript, elles sont écrites entre accolades.
Ça nous permet d'**appliquer des expressions JavaScript** directement dans notre JSX pour :

- faire des maths :
  La grande réponse sur la vie, l'univers et le reste est { 6 * 7 }
- modifier des chaînes de caractères :
  { alexia.toUpperCase() }

- utiliser des ternaires :
  { 2 > 0 ? 'Deux est plus grand que zéro' : 'Ceci n'apparaîtra jamais } }

Ou même tout simplement pour afficher une variable JS :

- pour une string : { myTitle }
- pour un nombre : { 42 }

Exercice :

Créer un composant <Cart /> (panier) qui viendra sous notre titre.

Le panier contient 3 plantes : un monstera, un lierre et un bouquet de fleurs.

Créer 3 variables pour les prix des plantes : le monstera coûte 8, le lierre coûte 10, et le bouquet de fleurs coûte 15.

Le panier contient une liste (<ul>), et chaque élément présente le nom de l'article, et le prix.

Le total du panier additionne les trois prix.

Solution de l'exercice :

```
function Header() {
   const text = "la maison jungle"
   return (<h1>{text.toUpperCase()}</h1>)
}

function Description() {
   return (<p>Ici achetez toutes les plantes dont vous avez toujours rêvé 🌵🪴🪴</p>)
}

function Cart() {
 const monstera = 8
 const lierre = 10
 const bouquet = 15
 return(<div>
    <h2> Panier </h2>
    <ul>
     <li> Monstera : {monstera} </li>
     <li> Lierre : {lierre} </li>
     <li> Bouquet : {bouquet} </li>
    </ul>
    Total du panier : {monstera+lierre+bouquet}$$
    </div>
    )
}

function Banner() {
   return (<div>
      <Header />
```

```
      <Description />
    </div>)
}

ReactDOM.render(<React.Fragment><Banner />><Cart />></React.Fragment>,
document.getElementById("root"))
```

En résumé

- une interface utilisateur (ou *UI*) est **constituée de multiples composants React** qui :
    - sont **réutilisables** ; par exemple, un bouton, un élément dans une liste, un titre
    - **regroupent** la structure, les styles et le comportement d'un élément
    - sont **traduits par React** en gros objets, qui sont **ensuite greffés au DOM** ;
- le JSX est une syntaxe créée par React permettant d'écrire du JavaScript. Il faut suivre quelques règles :
    - deux composants doivent toujours être wrappés dans **un seul composant parent**
    - les noms des composants **commencent par une majuscule**
    - les balises des composants **doivent être refermées**.

4. Prendre en main Create React APP

Comprenez l'importance des outils automatisés

Nous avons appris à utiliser les liens CDN de React, ReactDOM, et à paramétrer Babel dans le fichier HTML pour rapidement créer une app React. Mais cette technique n'est quasiment pas utilisée dans la vie de tous les jours d'un développeur.

À la place, les développeuses et développeurs utilisent des outils automatisés pour créer une base de code, qui dispose des outils essentiels déjà préconfigurés. Pour vous citer quelques-unes des fonctionnalités de ces outils, ils permettent de :

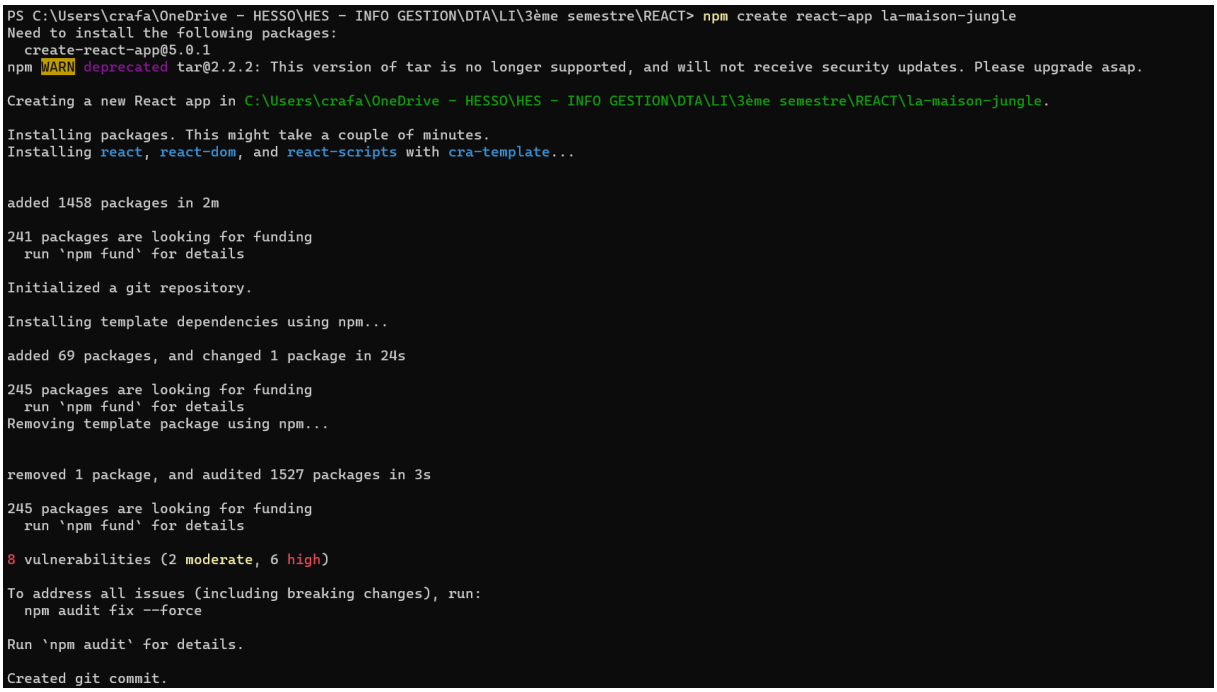
- Gérer les différentes dépendances (bibliothèques) utilisées par notre app ;
- Optimiser le chargement de notre code dans les navigateurs ;
- Importer du CSS et des images ;
- Gérer les différentes versions de JavaScript ;
- Faciliter l'expérience de développement, en rechargeant la page lorsque le code est modifié.

Create React App va vous permettre de générer un squelette de code pour votre application. Il embarque un certain nombre d'outils préconfigurés, tels que Webpack, Babel et ESLint, afin de vous garantir la meilleure expérience de développement possible.

Installation

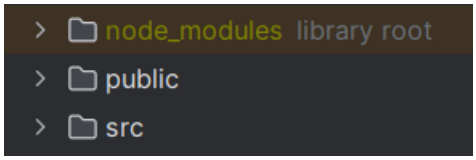
Pour manipuler Create React App ici, nous allons avoir besoin d'un gestionnaire de paquet (*package manager*) directement dans le terminal. Ici, je vais utiliser npm. Si vous utilisez une autre version, telle que yarn, je vous conseille de vous référer au guide d'utilisation de Create React App par Facebook, sur Github (en anglais).

Pour ce faire avec npm, il faut exécuter la commande « npm create react-app la-maison-jungle » à l'endroit où vous voulez que votre app React soit sauvegardée. Ici, la-maison-jungle peut être remplacé par le nom de votre projet.



```
Success! Created la-maison-jungle at C:\Users\crafa\OneDrive - HESSO\HES - INFO GESTION\DTA\LI\3ème semestre\REACT\la-maison-jungle
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd la-maison-jungle
  npm start

Happy hacking!
npm notice
npm notice New major version of npm available! 9.6.6 -> 10.2.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v10.2.0
npm notice Run npm install -g npm@10.2.0 to update!
npm notice
PS C:\Users\crafa\OneDrive - HESSO\HES - INFO GESTION\DTA\LI\3ème semestre\REACT>
```

Maintenant que cela est fait, nous pouvons ouvrir le projet avec notre logiciel de code.



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 : il s'agit du cœur de l'action. L'essentiel des fichiers que vous créerez et modifierez seront là.

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 npm, 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, permettant 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.

Organisez votre code

Nous allons maintenant modifier notre base de code pour qu'elle soit plus à l'image de notre projet. Il existe plusieurs manières d'organiser son code, et il est important de réfléchir à comment l'organiser. Ici, nous allons séparer les fichiers selon leur type : composants/style/images, etc.

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. Pour ce qui est des autres fichiers, le plus important est index.js que vous devez garder. Vous pouvez également garder index.css, mais vous pouvez supprimer les autres fichiers.

Maintenant, créons notre banner du chapitre précédent dans un fichier JavaScript à part dans /components que nous pouvons appeler Banner.js.

```
function Banner() {
 return <h1>La maison jungle</h1>
}

export default Banner
```

L' « export default Banner » est une syntaxe prévue dans l'ES6, qui vous épargnera d'utiliser les accolades au moment de l'import.

On peut maintenant adapter le code d'App.js en supprimant le code de base, et y importer notre Banner.

```
import Banner from './Banner'

function App() {
    return <Banner />
}

export default App
```

C'est « Webpack » qui nous permet d'importer notre composant aussi facilement, avec import. Cet outil particulièrement utile est essentiel pour lier les fichiers entre eux, afin qu'ils soient interprétés par le navigateur.

Ce qui nous donne (quand nous initialisons notre serveur « **npm** run start ») :

La maison jungle

En résumé

- Les développeurs utilisent des outils automatisés pour faciliter leur expérience de développement.
- Create React App (CRA) est la boîte à outils créée par Facebook, qui reste encore la référence pour initier un projet React.
- Un projet initialisé avec CRA possède toujours :
    - un fichier index.html qui est le template où vivra notre app React ;
    - un package.json qui liste les dépendances et les scripts ;
    - un fichier index.js dans lequel notre app React est initialisée, et greffée au HTML.
- CRA s'exécute avec l'aide d'un gestionnaire de paquet (dans ce cours, yarn).
- Webpack permet d'importer simplement les fichiers entre eux.

5. Incorporez du style et des assets à votre projet

Comme en HTML, nous pouvons associer des attributs à nos éléments. Les attributs HTML tels qu'id, href pour un lien <a />, src pour une balise <img />, fonctionnent normalement en JSX.

En revanche, il existe des mots réservés en JavaScript, tels que class. Comment faire donc, pour attribuer du style a un élément avec une classe CSS ?

Il suffit pour cela d'utiliser l'attribut « className » (il s'agit du lien entre React et CSS) , et de lui préciser une string. D'ailleurs, vous pouvez utiliser plusieurs classes sur un élément en les mettant à la suite, séparées par un espace.

Exemple :

Dans le banner.js on modifie notre composant :

```
<div className='lmj-banner'>
<h1>La maison jungle</h1>
</div>
```

Et on crée le CSS du composant dans banner.css :

```
.lmj-banner {
color: black;
text-align: right;
padding: 32px;
border-bottom: solid 3px black;
}
```

Ensuite, il ne faut pas oublier d'importer le style CSS dans la page du banner.js :

import '../styles/Banner.css'

Découvrez l'attribut style

```
<div style={{
    color: 'black',
    textAlign: 'right',
    padding: 32,
    borderBottom: 'solid 3px black'
}}>
    <h1>La maison jungle</h1>
</div>
```

Cet attribut peut être pratique pour tester rapidement quelque chose, mais il n'est pas recommandé d'en faire une utilisation plus poussée. Donc, pour styliser votre application, privilégiez davantage la méthode des classNames, ou d'autres méthodes avec des librairies tierces, par exemple.

Utilisez des images

Créer un dossier /assets dans lequel on vient mettre notre fichier logo.png

Pour l'importer dans votre code, vous pouvez maintenant faire de la manière suivante. Dans Banner.js :

```
import logo from '../assets/logo.png'
```

Vous voyez ici, on déclare en fait une variable logo à laquelle on assigne le contenu de notre image.

Puis vous pouvez l'utiliser dans un élément img, ce qui nous donne pour notre Banner.js :

```
import logo from '../assets/logo.png'
import '../styles/Banner.css'

function Banner() {
const title = 'La maison jungle'
return (
<div className='lmj-banner'>
```

```
<img src={logo} alt='La maison jungle' className='lmj-logo' />
<h1 className='lmj-title'>{title}</h1>
</div>
)
}
export default Banner
```

La propriété alt permet ici de respecter les normes d'accessibilité que j'évoquais au dernier chapitre de la première partie. C'est une bonne pratique qui donne un texte alternatif pour que les utilisateurs malvoyants accèdent au contenu.

En résumé

- L'attribut className permet de préciser une classe à un élément React pour lui indiquer du CSS.
- Le fichier CSS correspondant peut être importé directement dans un fichier .js.
- L'attribut style permet d'intégrer du style directement, on appelle cela du *inline style*.
- Les images sont importées par React grâce à Webpack. Il suffit d'importer l'image souhaitée.

6. Gagnez en temps et en efficacité grâce aux listes et aux conditions

La méthode JavaScript map() passe sur chaque élément d'un tableau. Elle lui applique une fonction, et renvoie un nouveau tableau contenant les résultats de cette fonction appliquée sur chaque élément.

Par exemple, pour une fonction qui doublerait la valeur d'un élément, cela donne :

```
const numbers = [1, 2, 3, 4]


const doubles = numbers.map(x => x * 2) // [2, 4, 6, 8]
```

Dans notre cas, elle va nous permettre de **prendre une liste de données**, et de la **transformer en liste de composants**.

La méthode map() permet facilement d'itérer sur des données et de retourner un tableau d'éléments. Comme elle, les méthodes forEach(), filter(), reduce(), etc., qui permettent de manipuler des tableaux, seront également vos alliés en React.

Précisez votre key

Les keys (clés) aident React à identifier quels éléments d'une liste ont changé, ont été ajoutés ou supprimés. Vous devez donner une clé à chaque élément dans un tableau, afin d'apporter aux éléments une identité stable.

La key doit impérativement respecter deux principes :

- Elle doit être unique au sein du tableau.
- Et stable dans le temps (pour la même donnée source, on aura toujours la même valeur de key=).

La key permet d'associer une donnée au composant correspondant dans le DOM virtuel qui permettra ensuite de générer les composants.

Il existe plusieurs méthodes pour générer une key unique :

- La méthode la plus simple et la plus fiable consiste à utiliser l'id associée à votre donnée dans votre base de données.
- Vous pouvez également trouver un moyen d'exploiter la valeur de la donnée, si vous avez la certitude qu'elle sera toujours unique, et stable dans le temps.
- En dernier recours, vous pouvez définir une string et la combiner avec l'index de la data dans votre tableau.

Utilisation de la fonction reduce

La fonction reduce est une fonction de haut niveau en JavaScript qui est utilisée pour réduire un tableau (ou autre type de liste) à une seule valeur en appliquant une fonction de réduction à chaque élément du tableau.

Exercice avec la fonction reduce : faire en sorte de ne pas afficher à double les catégories.

```
const categories = plantList.reduce(
  (acc, plant) =>
    acc.includes(plant.category) ? acc : acc.concat(plant.category), []
)
```

- (acc, plant) : acc c'est l'accumulateur, il représente le tableau en cours de construction (categories), au fur et à mesure que la fonction parcourt plantList, acc contient les catégories uniques déjà ajoutées.
- acc.includes(plant.category) ? acc : acc.concat(plant.category) : cela vérifie si la catégorie de la plante est déjà incluse dans l'accumulateur. Si elle est déjà incluse, la fonction renvoie l'accumulateur actuelle, et sinon cela crée un nouveau tableau en ajoutant la nouvelle catégorie à l'accumulateur existant.
- [] il s'agit de la valeur initiale de l'accumulateur, c'est-à-dire un tableau vide. Au début l'accumulateur commence en étant un tableau vide.

Contextualisez le contenu de vos composants

React nous permet de dresser des listes de composants : un gain de temps énorme dans votre vie de développeur. Mais ce n'est pas tout ! Le JSX nous permet également d'afficher des éléments de manière conditionnelle dans nos composants.

Différentes conditions :

On peut changer l'affichage de notre composant en fonction des conditions, il y a différentes façons d'écrire ces fonctions :

Prenons l'exemple des plantes, ayant l'attribut « fait les meilleures ventes » -> oui ou non

On peut écrire la condition comme ça :

- {plant.isBestSale ? 🔥 : 👎 }

Si on ne veut rien affiché si la condition est fausse, on peut mettre null :

- {plant.isBestSale ? 🔥 : null}

Il existe une manière encore plus simple : **&&**

Indiquée entre accolades, && précède un élément JSX et précise que l'élément ne sera généré que si la condition est respectée. On peut donc écrire :

```
{plant.isBestSale && <span>🔥</span>}
```

Vous pouvez d'ailleurs chaîner les conditions.

Si par exemple, vous vouliez afficher le 🔥 pour les plantes qui sont des isBestSale et qui sont dans la catégorie classique :

```
{plant.isBestSale && plant.category === "classique" && <span>🔥</span>}
```

De la même manière, si vous aviez voulu que le 🔥 s'affiche à côté des plantes qui sont isBestSale ou dans la catégorie classique :

```
{(plant.isBestSale || plant.category === "classique") && <span>🔥</span>}
```

Il y a des techniques qui consistent à mettre les conditions dans des const, car cela rend le code plus propre, si on doit utiliser plusieurs conditions à la suite.

Il y a une autre méthode, qui consiste à mettre la condition comme un composant. Cela est pertinent par exemple, si la condition est utilisée à plusieurs endroits.

En résumé

- À partir d’une liste de données, map() permet de créer une liste de composants React.
- La prop key est indispensable dans les listes de composants.
- Si vous voulez éviter les bugs, la prop key doit :
- être unique au sein de la liste ;
- perdurer dans le temps.
- La best practice pour créer une key est d’utiliser l’ id unique associée à une donnée, et de ne pas vous contenter d'utiliser l'index de l'élément dans la liste.
- Une condition ternaire permet d’afficher un élément ou un autre dans le JSX, répondant à la condition "if… else...".
- Il existe d'autres manières de créer des conditions en React, notamment en sortant les conditions du JSX.

7. Réutilisez vos composants avec les props

La réutilisation des composants est au cœur de la logique de React. Mais, pour être réutilisés, les composants requièrent souvent une configuration. En react, il s'agit des props.

Familiarisez-vous avec la syntaxe

Et si je vous disais que vous avez déjà utilisé une prop ? Eh oui, la prop key dans le chapitre sur les listes !

Essayons de créer un nouveau composant qui va être réutilisé. L'idée est de créer une échelle d'arrosage et une échelle de luminosité pour chaque plante.

Pour cela, nous devons ajouter les données « light » et « water » dans notre base de données.

Maintenant, dans chaque item plante (dans le composant PlantItem), on vient ajouter un composant CareScale et on lui passe la prop value :

```
<CareScale scaleValue={plant.light} />
```

Comment on récupére la valeur d'un prop dans notre composant ?

Les props sont récupérées **dans les paramètres de la fonction qui définit notre composant**.

Le composant CareScale ressemblera donc à ça :

```
function CareScale(props) {
    const scaleValue = props.scaleValue
    return <div>{scaleValue}</div>
}

export default CareScale
```

Les props sont donc des **objets que l'on peut récupérer dans les paramètres de notre composant fonction.**

Créez des paramètres

Je vais commencer par préciser une prop pour le type que j'appellerai careType pour mon composant CareScale et réutiliser ce composant entre l'ensoleillement et l'arrosage :

```
<CareScale careType='water' scaleValue={plant.water} />
<CareScale careType='light' scaleValue={plant.light} />
```

Il faut maintenant que j'adapte CareScale pour récupérer le careType.

On va à présent utiliser une méthode, permise depuis l'ES6 : **la déstructuration** → elle permet directement de déclarer une variable et de lui assigner la valeur d'une propriété d'un objet.

```
const {scaleValue, careType} = props
```

On peut même faire plus simple :

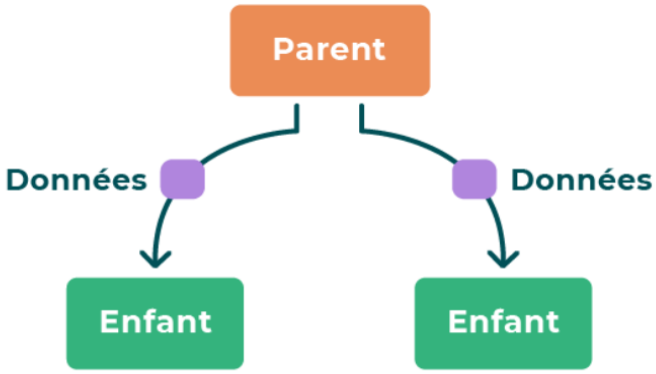
```
function CareScale({scaleValue, careType}) {
```

En pratique, une prop peut avoir n'importe quelle valeur possible en JavaScript, mais syntaxiquement, en JSX, on n'en a que deux possibilités :

- Un littéral string, matérialisé par des guillemets ;
- Ou pour tout le reste (booléen, nombre, etc…) des accolades {}

Faites descendre les données, des parents vers les enfants

Les props nous permettent donc de configurer nos composants. Elles répondent à la logique même de React selon laquelle les données descendent à travers notre arborescence de composants : il s'agit d'**un flux de données unidirectionnel**.



Les composants parents partagent leurs données avec leurs enfants

Comme vous pouviez vous en douter, un composant est le parent du composant défini dans le return().

Cela signifie qu'un parent utilise le composant children.

Pour les props, vous devez garder deux règles à l'esprit :

- Une prop est toujours passée par un composant parent à son enfant : c'est le seul moyen normal de transmission.
- Une prop est considérée en lecture seule dans le composant qui la reçoit.

Découvrez la prop technique children

Il existe chez React des props qui ont un comportement un peu particulier : nous les appelons les props techniques.

La syntaxe de cette prop est particulière, puisqu'elle n'est pas fournie à l'aide d'un attribut, mais en imbriquant des composants à l'intérieur du composant concerné.

Ce qui nous donne :

```
<Parent>
    <Enfant1 />
    <Enfant2 />
</Parent>
```

Par exemple, si on utilise children pour réécrire la Banner, cela nous donnerait dans App.js :

```
<Banner>

    <img src={logo} alt='La maison jungle' />

    <h1 className='lmj-title'>La maison jungle</h1>

</Banner>
```

Ici, img et h1 sont les nœuds enfants dans le DOM de Banner.

Et on peut **accéder à ces nœuds enfants de Banner dans ses paramètres**, un peu de la même manière qu'on récupérerait des props :

```
function Banner({ children }) {

    return <div className='lmj-banner'>{children}</div>

}
```

Cette manière d'utiliser children est particulièrement utile lorsqu'un composant ne connaît pas ses enfants à l'avance, par exemple pour une barre de navigation (Sidebar) ou bien pour une modale.

Attention aux erreurs

Les props constituent un aspect clé de React. Mais, en les manipulant, vous verrez qu'il peut être très facile de commettre des erreurs. Cela vient notamment de la flexibilité de JavaScript, qui fait du typage dynamique (les types string, int, etc.). Pour vous donner un exemple d'erreur classique :

Vous passez une prop value à un composant.

Vous utilisez une liste de valeurs, certaines valeurs sont des strings, d'autres des nombres.

Vous appliquez la méthode .toUpperCase() à votre value : **boum** !

Une erreur s'affiche : « ! .toUpperCase() n'existe pas sur un nombre ».

Pour éviter ce genre d'erreur, il faut être extrêmement rigoureux sur le type de props que vous passez à vos composants.

Pour cela, React a créé les PropTypes. (pas détaillé dans cette lecture individuelle)

En résumé

- Les props sont des objets que l'on peut récupérer dans les paramètres de notre composant fonction.
- Il existe deux syntaxes pour assigner une valeur à une prop :
    - les guillemets pour les string ;
    - les accolades pour tout le reste : nombres, expressions JavaScript, booléen, etc.
- La déstructuration est une syntaxe permettant de déclarer une variable en l'affectant directement à la valeur d'un objet (ou tableau).
- Une *prop* est :
    - toujours passée par un composant parent à son enfant ;
    - considérée en lecture seule dans le composant qui la reçoit.
- La prop children est renseignée en imbriquant les enfants dans le parent : <Parent><Enfant /></Parent>.
- children est utile lorsqu'un composant ne connaît pas ses enfants à l'avance.

8. Interagissez avec vos composants grâce aux événements

Un événement est une réaction à une action émise par l'utilisateur, comme le clic sur un bouton ou la saisie d'un texte dans un formulaire.

Avec sa syntaxe pratique et concise, React facilite énormément la gestion des événements du DOM.

Familiarisez-vous avec la syntaxe

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.

Exemples :

```
Function handleClick() {
    console.log('✨ Ceci est un clic ✨')
}
```

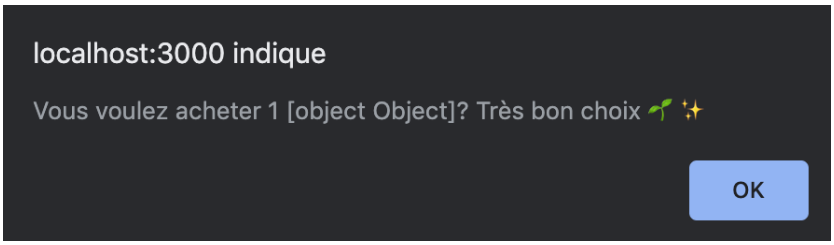
On ajoute maintenant onClick={handleClick} dans la balise li du composant PlantItem. On a donc:

```
<li className='lmj-plant-item' onClick={handleClick}>
    <img className='lmj-plant-item-cover' src={cover} alt={`${name} cover`} />
    {name}
    <div>
        <CareScale careType='water' scaleValue={water} />
        <CareScale careType='light' scaleValue={light} />
    </div>
</li>
```

Essayons maintenant de déclencher une alerte qui affiche le nom de la plante sur laquelle on a cliqué. On passe donc plantName en paramètre de handleClick comme ici :

```
function handleClick(plantName) {
    alert(`Vous voulez acheter 1 ${plantName} ? Très bon choix ⏳✨`)
}
```

Mais si je clique, ça ne marche pas :



Aucune de nos plantes ne s'appelle 1 [object Object]
En effet, React passe par défaut un objet (que nous aborderons dans quelques minutes), mais ici, nous voulons lui spécifier notre propre argument.

Pour cela, c'est très simple : on déclare une fonction directement dans onClick (les fonctions fléchées sont très pratiques pour ça).
Cette fonction appellera handleClick en lui passant name en paramètre. Donc on a :

```
onClick={() => handleClick(name)}
```

Découvrez les événements synthétiques

Donc, je vous parlais de l'objet que React passe par défaut en paramètre aux fonctions indiquées en callback des événements. Voyons voir à quoi ça ressemble.

Si je récupère le paramètre dans handleClick :

```
function handleClick(e) {
    console.log('✨ Ceci est mon event :', e)
}
```

j'obtiens ça :

Ça nous donne beaucoup d'informations, n'est-ce pas ? Mais à quoi ça correspond ? 🤔

Il s'agit en fait d'un événement synthétique. Pour faire bref, il s'agit de la même interface que pour les événements natifs du DOM, mais ils sont compatibles avec tous les navigateurs.

Nous pouvons utiliser différentes méthodes avec le paramètre récupéré dans la fonction passée à l'événement, comme par exemple «e.preventDefault() ».

Simplifiez votre création de formulaires avec React

En React, la gestion des formulaires est simplifiée : on a accès à la valeur très facilement, qu'il s'agisse d'un input checkbox, d'un textarea, ou encore d'un select avec onChange.

Il existe deux grandes manières de gérer les formulaires : **la manière contrôlée et la manière non contrôlée**.

Les formulaires non contrôlés

Sur notre app, directement dans App.js, je mets un composant QuestionForm que je vais déclarer dans un fichier à part. Nous allons ajouter un champ pour une question.
Donc pour ça je crée un form, qui englobe mon input :

```
<form onSubmit={handleSubmit}>
    <input type='text' name='my_input' defaultValue='Tapez votre texte' />
    <button type='submit'>Entrer</button>
</form>
```

Et pour handleSubmit , cela donne :

```
function handleSubmit(e) {
    e.preventDefault()
    alert(e.target['my_input'].value)
}
```

React permet de préciser une defaultValue à mon champ input. Ici, j'appelle preventDefault, sinon le submit rafraîchirait la page.
Les formulaires non contrôlés nous permettent de ne pas avoir à gérer trop d'informations. Mais cette approche est un peu moins "React", parce qu'elle ne permet pas de tout faire.

Contrôlez vos formulaires

Ici, pour vous montrer l'utilisation des formulaires contrôlés, nous aurons besoin d'une notion que nous aborderons dès le prochain chapitre : le state (état).

En bref, le state local nous permet de garder des informations. Ces informations sont spécifiques à un composant et elles proviennent d'une interaction que l'utilisateur a eue avec le composant.

Donc je vais créer ma variable inputValue et la fonction qui va permettre de changer sa valeur dans le state local avec useState.
Sachez juste que la ligne de code ci-dessous me permet de déclarer l'état initial pour inputValue et la fonction correspondante pour la modifier, et de lui préciser la valeur par défaut "Posez votre question ici" :

```
const [inputValue, setInputValue] = useState("Posez votre question ici")
```

J'ai donc mon QuestionForm comme ci-dessous :

```
import { useState } from 'react'

function QuestionForm() {
    const [inputValue, setInputValue] = useState('Posez votre question ici')
    return (
        <div>
            <textarea
                value={inputValue}
                onChange={(e) => setInputValue(e.target.value)}
            />
        </div>
    )
}

export default QuestionForm
```

Ici, je passe une fonction en callback à onChange pour qu'elle sauvegarde dans mon state local la valeur de mon input. J'accède à la valeur tapée dans l'input avec e.target.value.
inputValue a maintenant accès au contenu de mon input à tout moment. Je peux donc créer un bouton qui déclenche une alerte qui affiche le contenu de mon input, comme ici :

```
<div>
    <textarea
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
    />
    <button onClick={() => alert(inputValue)}>Alertez moi 🚨</button>
</div>
```

Et ça marche bien !

Comprenez les avantages des formulaires contrôlés

Les formulaires contrôlés permettent d'**interagir directement avec la donnée renseignée par l'utilisateur**. Vous pouvez donc afficher un message d'erreur si la donnée n'est pas valide, ou bien la filtrer en interceptant une mauvaise valeur.

Si nous décidons qu'il n'est pas autorisé d'utiliser la lettre "f", nous pouvons déclarer une variable :

- const isInputError = inputValue.includes('f')

et afficher ou non un message d'erreur en fonction de ce booléen :

```
{isInputError && (

    <div>🔥 Vous n'avez pas le droit d'utiliser la lettre "f" ici.</div>

)}
```

De la même manière, nous pouvons intercepter une mauvaise valeur entrée par l'utilisateur. Pour cela, il faut déclarer une fonction intermédiaire :

```
function checkValue(value) {
    if (!value.includes('f')) {
        setInputValue(value)
    }
}
```

et on applique la modification dans notre fonction callback :

```
onChange={(e) => checkValue(e.target.value)}
```

Ici, vous aurez beau marteler votre touche `f` autant de fois que vous voudrez, la valeur ne s'inscrira pas dans votre input.

Quand utiliser le composant contrôlé et quand utiliser sa version non contrôlée ?!

Quand vous avez un composant rapide à faire, qui n'intègre aucune complexité, un input non contrôlé peut faire l'affaire. À l'inverse, si vous avez des vérifications à faire, il vaudra sûrement mieux passer par un composant contrôlé.

Il existe également des bibliothèques qui vous permettent de gérer les formulaires et leur validation aussi proprement que possible, par exemple le très bon outil react-hook-form.

En résumé

- En React, un événement s'écrit dans une balise en camelCase, et on lui passe la fonction à appeler.
- Contrairement au JS, dans la quasi totalité des cas, vous n'avez pas besoin d'utiliser addEventListener.
- React passe un événement synthétique en paramètre des fonctions de callBack. Cet événement synthétique est similaire à un événement passé en natif dans le DOM, sauf qu'il est compatible avec tous les navigateurs.
- Il existe deux grandes manières de gérer les formulaires : les formulaires contrôlés ou non contrôlés. L'utilisation des formulaires contrôlés est recommandée.
- Les formulaires contrôlés sauvegardent la valeur des champs dans le state local, et accèdent à la data entrée par l'utilisateur avec onChange.
- Les formulaires contrôlés permettent de filtrer le contenu, ou d'afficher un message d'erreur en fonction de la data qui est entrée par l'utilisateur.

9. Mettez en place votre state local avec useState

Découvrez les stateful components

Le state local est présent à l'intérieur d'un composant et **garde sa valeur, même si l'application le re-render**.

Essayons par exemple d'ajouter une plante « Monstera » au panier.

Nous allons aller dans notre composant « Cart.js » et tout d'abord importer useState.

```
Import {useState} from 'react'
```

Puis, on peut créer un state cart . Avec useState , nous devons déclarer en même temps une fonction pour mettre à jour ce state (updateCart), et lui attribuer une valeur initiale, qui sera ici de 0 :

```
const [cart, updateCart] = useState(0)
```

Il faut maintenant ajouter un bouton dans le panier qui permet d'ajouter un monstera avec la fonction que nous venons de déclarer. Ce qui fait dans Cart.js :

```
function Cart() {
    const monsteraPrice = 8
    const [cart, updateCart] = useState(0)

    return (
        <div className='lmj-cart'>
            <h2>Panier</h2>
            <div>
                Monstera : {monsteraPrice}€
                <button onClick={() => updateCart(cart + 1)}>
                    Ajouter
                </button>
            </div>
            <h3>Total : {monsteraPrice * cart}€</h3>
        </div>
    )
}
```

A présent, si on clique sur "Ajouter", le montant total est modifié en fonction du nombre d'éléments sauvegardés dans le state du panier. Lorsqu'un state est modifié, alors l'affichage du composant est rafraichi et la valeur affichée est actualisée, on dit que le composant est re-render.

Notre composant Cart est maintenant devenu un **stateful** component, grâce à useState .

Familiarisez-vous avec useState

useState est un hook qui permet d'ajouter le state local React à des composants fonctions.

Un hook est **une fonction qui permet de « se brancher » (to hook up) sur des fonctionnalités React**. On peut d'ailleurs les importer directement depuis React. Après useState, nous verrons un autre hook dans cette partie : useEffect . Il existe d'autres hooks que nous n'aborderons pas dans ce cours, mais dans le suivant sur React !

Nous l'avons déjà utilisé, mais je vous le remets ici :

```
const [cart, updateCart] = useState(0)
```

Investiguons comment est construit notre state cart. 🧐

Comprenez les crochets

Tout d'abord, les crochets [] . Il s'agit de la pratique de la déstructuration. Mais ici, cela s'appelle la *décomposition*, parce qu'il s'agit d'un **tableau** et non d'un objet.

useState nous **renvoie une paire de valeurs dans un tableau de 2 éléments**, que nous récupérons dans les variables cart et updateCart dans notre exemple. Le premier élément est la valeur actuelle, et le deuxième est une fonction qui permet de la modifier.

Sans la décomposition, nous aurions aussi pu faire :

```
const cartState = useState(0)
const cart = cartState[0]
const updateCart = cartState[1]
```

Dans un tableau qu'on décompose, nous pouvons librement nommer nos variables. J'aurais tout aussi bien pu faire :

```
const [coucou, cavabien] = useState(0)
```

Initialisez votre state

Intéressons-nous maintenant au paramètre passé entre parenthèses à useState : useState(0).

Comme nous l'avons vu, il correspond à l'état initial de notre state. Cet état initial peut être un nombre comme ici, un string, un booléen, un tableau ou encore un objet avec plusieurs propriétés.

Il est important de préciser une valeur initiale dans votre state. Sinon, elle sera undefined par défaut, et ce n'est pas un comportement souhaitable : plus vous serez explicite, mieux votre application React se portera !

En résumé

- Le state local est présent à l'intérieur d'un composant : ce composant peut être re-render autant de fois que l'on veut, mais les données seront préservées.
- Un hook est une fonction qui permet de « se brancher » (*to hook up*) sur des fonctionnalités React.
- useState est un hook qui permet d'ajouter le state local React à des fonctions composants :
- Il nous renvoie une paire de valeurs dans un tableau de 2 valeurs, récupérée dans les variables entre crochets.
- Il faut initialiser votre state avec un paramètre passé entre parenthèses – un nombre, une string, un booléen, un tableau ou même un objet.

10. Partagez votre state entre différents composants

Comment faire pour **changer le comportement d'un composant en fonction du state d'un autre composant** ? Par exemple, si je veux enfin ajouter un lien entre mon Cart et mon composant ShoppingList . Je peux créer un bouton "Ajouter au panier" dans chaque PlantItem ... Mais comment faire pour venir compléter mon panier en fonction ?

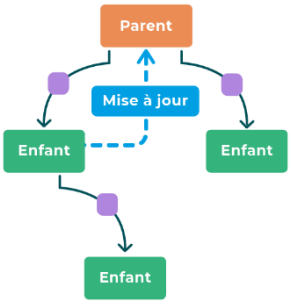
Faites remonter l'état et mettez-le à jour depuis vos composants enfants

Comme son nom l'indique, un state local… est local. Ni les parents, ni les enfants ne peuvent manipuler le state local d'un composant (ils n'en ont pas la possibilité technique).

Comment faire pour partager un élément d'état entre plusieurs composants ?

Eh bien, il faudra faire remonter ces données vers le state local du plus proche composant qui est un parent commun, et y garder le state. À partir de là, il sera possible de :

1. Faire redescendre ces infos avec des props jusqu'aux composants qui en ont besoin.
2. Faire « remonter » les demandes d'update toujours dans les props. Pour cela, on peut utiliser la fonction de mise à jour du state récupérée dans useState, en la passant en props aux composants qui en ont besoin.



Exemple :

Je remontre cart dans App.js, et je passe cart ainsi que updateCart en props :

```
function App() {
    const [cart, updateCart] = useState([])

    return (
        <div>
            <Banner>
                <img src={logo} alt='La maison jungle' className='lmj-logo' />
                <h1 className='lmj-title'>La maison jungle</h1>
            </Banner>
            <div className='lmj-layout-inner'>
                <Cart cart={cart} updateCart={updateCart} />
                <ShoppingList cart={cart} updateCart={updateCart} />
            </div>
            <Footer />
        </div>
    )
}
export default App
```

Je récupère ces props dans Cart.Js et je profite pour supprimer le bouton « Ajouter » :

```
function Cart({ cart, updateCart }) {
    const monsteraPrice = 8
    const [isOpen, setIsOpen] = useState(true)

    return isOpen ? (
        <div className='lmj-cart'>
            <button
                className='lmj-cart-toggle-button'
                onClick={() => setIsOpen(false)}
```

```
            >
                Fermer
            </button>
            <h2>Panier</h2>
            <h3>Total : {monsteraPrice * cart}€</h3>
            <button onClick={() => updateCart(0)}>Vider le panier</button>
        </div>
    ) : (
        <div className='lmj-cart-closed'>
            <button
                className='lmj-cart-toggle-button'
                onClick={() => setIsOpen(true)}
            >
                Ouvrir le Panier
            </button>
        </div>
    )
}
export default Cart
```

Je change également le ShoppingList, en utilisant les props cart et updateCart et en instaurant le bouton qui permet d'ajouter les plantes au panier.

```
function ShoppingList({ cart, updateCart }) {
    const categories = plantList.reduce(
        (acc, elem) =>
            acc.includes(elem.category) ? acc : acc.concat(elem.category),
            []
    )
    return (
        <div className='lmj-shopping-list'>
            <ul>
                {categories.map((cat) => (
                <li key={cat}>{cat}</li>
                ))}
            </ul>
            <ul className='lmj-plant-list'>
                {plantList.map(({ id, cover, name, water, light }) => (
                    <div key={id}>
                        <PlantItem cover={cover} name={name} water={water} light={light} />
                        <button onClick={() => updateCart(cart + 1)}>Ajouter</button>
                    </div>
                ))}
            </ul>
        </div>
    )
}
export default ShoppingList
```

En résumé

- Pour utiliser un même état entre plusieurs composants, il faut :
    - faire remonter l'état dans le composant parent commun le plus proche ;
    - puis faire descendre la variable

11. Déclenchez des effets avec useEffect

Découvrez useEffect

Imaginons que l'on veut créer une alerte lorsque j'ajoute une plante à mon panier, et que cette alerte affiche le montant total du panier.

Il faut tout d'abord importer useEffect:

```
import { useState, useEffect } from 'react'
```

et mettre ce code avant le return :

```
useEffect(() => {
    alert(`J'aurai ${total}€ à payer 💸`)
})
```

Ce qui nous donne pour Cart.js :

```
function Cart({ cart, updateCart }) {
    const [isOpen, setIsOpen] = useState(true)
    const total = cart.reduce(
        (acc, plantType) => acc + plantType.amount * plantType.price,
            0
        )
    useEffect(() => {
        alert(`J'aurai ${total}€ à payer 💸`)
    })

    return isOpen ? (
        <div className='lmj-cart'>
            <button
                className='lmj-cart-toggle-button'
                onClick={() => setIsOpen(false)}
            >
                Fermer
            </button>
            {cart.length > 0 ? (
                <div>
                    <h2>Panier</h2>
                    <ul>
                        {cart.map(({ name, price, amount }, index) => (
                            <div key={`${name}-${index}`}>
                                    {name} {price}€ x {amount}
                            </div>
                        ))}
                    </ul>
                    <h3>Total :{total}€</h3>
                    <button onClick={() => updateCart([])}>Vider le panier</button>
                </div>
            ) : (
                <div>Votre panier est vide</div>
            )}
        </div>
    ) : (
        <div className='lmj-cart-closed'>
            <button
                className='lmj-cart-toggle-button'
                onClick={() => setIsOpen(true)}
            >
```

```
                Ouvrir le Panier
            </button>
        </div>
    )
}
```

Maintenant cela fonctionne, car useEffect nous permet d'effectuer notre effet une fois le rendu du composant terminé. Et comme useEffect est directement dans notre composant, nous avons directement accès à notre state, à nos variables, nos props,

Qu'est-ce qui se passe si nous fermons notre panier ?

Notre alerte se déclenche aussi ! Cela est normal, car useEffect se déclenche après le rendu. Eh bien il se déclenche après CHAQUE rendu du composant. Sauf si vous…

Précisez quand déclencher un effet avec le tableau de dépendances

Pour décider précisément quand on veut déclencher un effet, on peut utiliser le tableau de dépendances. Il correspond au deuxième paramètre passé à useEffect.

Petit rappel : le premier paramètre passé à useEffect est une fonction. Cette fonction correspond à l'effet à exécuter. Ici, il s'agit de :

```
() => {
 alert(`J'aurai ${total}€ à payer 💎`)
}
```

Le deuxième paramètre de useEffect accepte **un tableau noté entre crochets** : il s'agit du tableau de dépendances.

Dans notre cas, si je veux que l'**alerte ne s'affiche que lorsque le total de mon panier change**, il me suffit de faire :

```
useEffect(() => {
    alert(`J'aurai ${total}€ à payer 💎`)
}, [total])
```

Vous pouvez mettre n'importe quelle variable ici. Si vous voulez afficher l'alerte quand le total change OU quand une nouvelle catégorie est sélectionnée, vous pourriez tout à fait récupérer la catégorie sélectionnée (en faisant remonter activeCategory et setActiveCategory et en les passant en props), puis mettre [total, activeCategory] dans votre tableau de dépendances.

Est-ce que l'effet est lancé au tout premier render de mon composant ?

Essayez de rafraîchir la page pour voir ! L'alerte s'affiche. Donc la réponse est **oui**.

Comment faire pour exécuter un effet uniquement après le premier render de mon composant ? Par exemple, si je veux récupérer des données sur une API ?

Il faut **renseigner un tableau de dépendances vide** :

```
useEffect(() => {
    alert('Bienvenue dans La maison jungle')
}, [])
```

À partir du moment où vous utilisez le tableau de dépendances, faites bien attention à ne pas oublier des dépendances, ou bien à ne pas en laisser qui n'ont plus rien à y faire, pour éviter d'exécuter à des moments inopportuns.

Intégrez quelques règles

Comme je vous l'ai expliqué au chapitre précédent, useEffect est un hook, une fonction qui permet de « se brancher » sur la fonctionnalité des effets de React. Mais quelques règles particulières s'appliquent au hook useEffect :

- Appelez toujours useEffect à la racine de votre composant. Vous ne pouvez pas l'appeler à l'intérieur de boucles, de code conditionnel ou de fonctions imbriquées. Ainsi, vous vous assurez d'éviter des erreurs involontaires.
- Comme pour useState, useEffect est uniquement accessible dans un composant fonction React. Donc ce n'est pas possible de l'utiliser dans un composant classe, ou dans une simple fonction JavaScript.

Par ailleurs, je vous conseille de séparer les différentes actions effectuées dans différents useEffect. Cela est plutôt une bonne pratique qu'une règle.

En résumé

- useEffectpermet d'effectuer des effets : cela permet à notre composant d'exécuter des actions après l'affichage, en choisissant à quel moment cette action doit être exécutée.
- Le hook useEffect est appelé après chaque rendu de votre composant. Il est possible de préciser quelle modification de donnée déclenche les effets exécutés dans useEffect, avec le tableau de dépendances.
- Un tableau de dépendances vide permet d'exécuter un effet uniquement au premier rendu de votre composant.

Conclusion

Grâce à cette lecture individuelle, nous avons pu avoir un bon aperçu de l'écosystème de React : ses opportunités, ses avantages et son positionnement dans l'écosystème des frameworks JS, entre bibliothèque et framework.

Nous avons pu également créer notre votre première app React et plonger dans le monde des composants.

Ensuite, nous avons tiré profit de Create React App pour créer une application complète, en maîtrisant les principes de base du JSX entre affichage conditionnel et création de listes de composants, props et événements.

Enfin, nous avons pu intégrer de la logique à vos composants à l'aide du state et de useEffect.

Avec ces connaissances fondamentales, nous avons obtenu de bonnes bases solides en React, tout en créant une app fonctionnelle pour un magasin de plantes.

Si nous voulons nous **tenir au courant de ce qui se passe dans la communauté React**, il existe plusieurs options :

- Le blog de React (en anglais GB) pour se tenir informé officiellement des nouveautés
- React Hebdo newsletter (en français FR) : vous pourrez vous tenir au courant des actualités liées à React et React Native
- Le blog de Kent C. Dodds (en anglais GB) : c'est le créateur de la bibliothèque React Testing Library, et ses articles sont très pertinents ;
- Le compte Twitter de Dan Abrahamov (en anglais GB): c'est l'un des membres de la core team React, il tweet régulièrement sur React et son environnement
- Le site CSS Tricks (en anglais GB): un excellent blog spécialisé sur le CSS qui contient toute une section dédiée à React