REACT

Table des matières

Auteur : Yoann DEPRIESTER
Relu, validé & visé par :
☒ Jérôme CHRETIENNE
☒ Sophie POULAKOS
☒ Mathieu PARIS

Date création : xx-xx-20xx
Date révision : xx-xx-2023

REACT

REACT

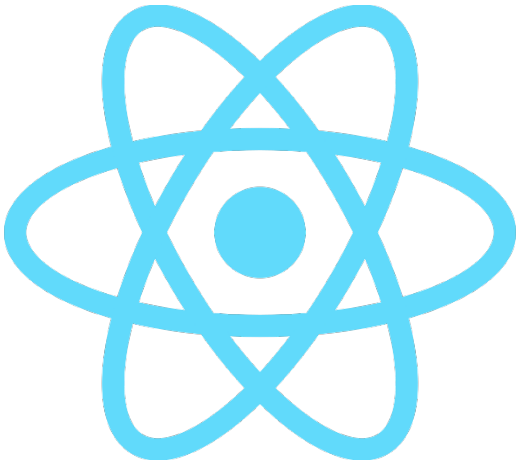
REACT

Objectif pédagogique :

- Etre capable d'initialiser et paramétrer un projet REACT
- Etre capable de réaliser des interfaces utilisateur
- Etre capable de comprendre les capacités et contraintes de REACT et REACT Router
- Etre capable de mettre en place un système de navigation ou de routing côté Front-end avec REACT Router
- Etre capable de gérer les requêtes http d'interaction côté Front-end avec REACT Router

REACT

Présentation de REACT



1. Historique

React est une bibliothèque JavaScript développée par Facebook, lancée en mars 2013. À l'origine, elle a été créée pour résoudre les problèmes liés à la complexité croissante des interfaces utilisateur (UI) des applications web. Son créateur, Jordan Walke, a voulu développer une solution qui permettrait de créer des interfaces de manière efficace et performante. Depuis sa sortie, React a gagné en popularité et est devenu un des outils les plus utilisés dans le développement web, soutenu par une large communauté et un écosystème florissant de bibliothèques et d'outils complémentaires.

2. Objectif

Le principal objectif de React est de simplifier la **création d'interfaces utilisateur** (UI) en rendant le code plus modulaire, prévisible et facile à maintenir. React permet aux développeurs de :

- Créer des interfaces réactives et interactives de manière efficace.
- Gérer facilement l'état de l'application, ce qui devient complexe à mesure que les applications évoluent.
- **Optimiser la performance** grâce à un rendu partiel des éléments modifiés dans l'UI (plutôt qu'un rechargement complet).

REACT

3. Fonctionnalités

Pour atteindre les objectifs précédents, React s'appuie sur plusieurs fonctionnalités :

- **Composants** : React repose sur le concept de composants, qui sont des blocs de construction autonomes et réutilisables. Chaque composant peut gérer son propre état et sa logique, facilitant ainsi la gestion de l'UI.

- **JSX** : React utilise une syntaxe appelée JSX (JavaScript XML) qui permet d'écrire des éléments UI de manière déclarative, en mélangeant JavaScript et HTML. Cela améliore la lisibilité et la simplicité du code.

- **État et Props** : Les composants React peuvent gérer un état interne (state) et recevoir des données (props) d'autres composants. Cela permet de créer des interfaces dynamiques et de gérer la logique d'affichage en fonction des interactions de l'utilisateur.

- **Virtual DOM** : React utilise un Virtual DOM pour optimiser les mises à jour de l'interface. Au lieu de manipuler le DOM directement, React crée une représentation virtuelle de celui-ci. Lorsqu'un changement se produit, React met à jour le Virtual DOM, compare les changements et ne met à jour que les éléments nécessaires dans le DOM réel, améliorant ainsi les performances.

- **Écosystème riche** : React bénéficie d'un écosystème large et varié, comprenant des outils de gestion d'état comme Redux, des bibliothèques pour le routage comme React Router, et bien plus encore. Cela permet aux développeurs de créer des applications complexes en utilisant des solutions éprouvées.

Lien vers le site officiel avec la documentation et tutoriel : https://fr.react.dev/

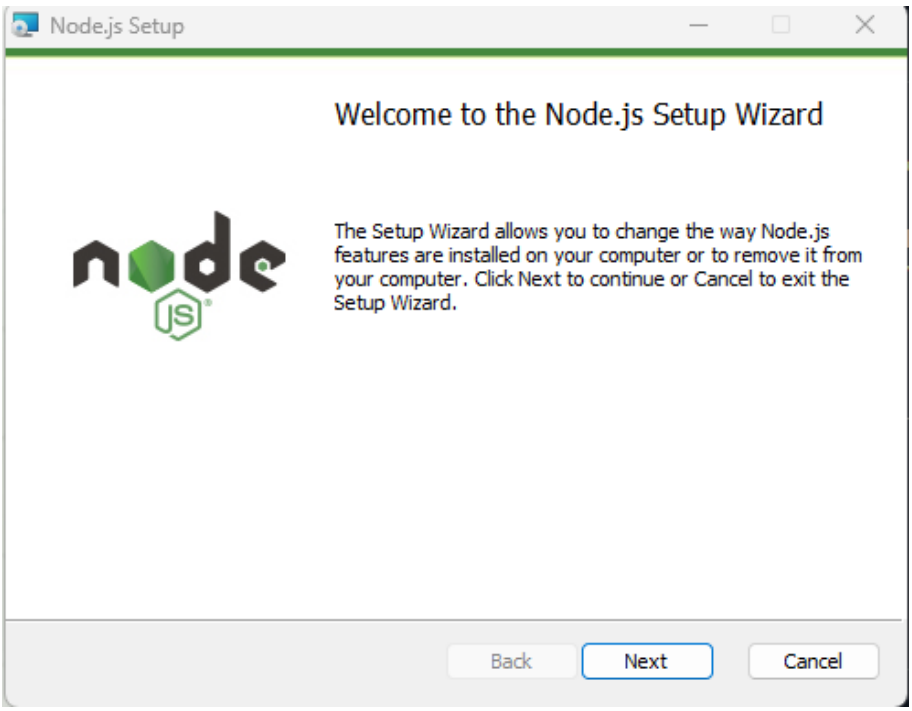
REACT

Installation de REACT

Avant de pourvoir installer REACT, vous aurez besoin d'installer Node.js.

Télécharger Node.js sur le site officiel (la dernière version TLS fait très bien l'affaire) : https://nodejs.org/fr

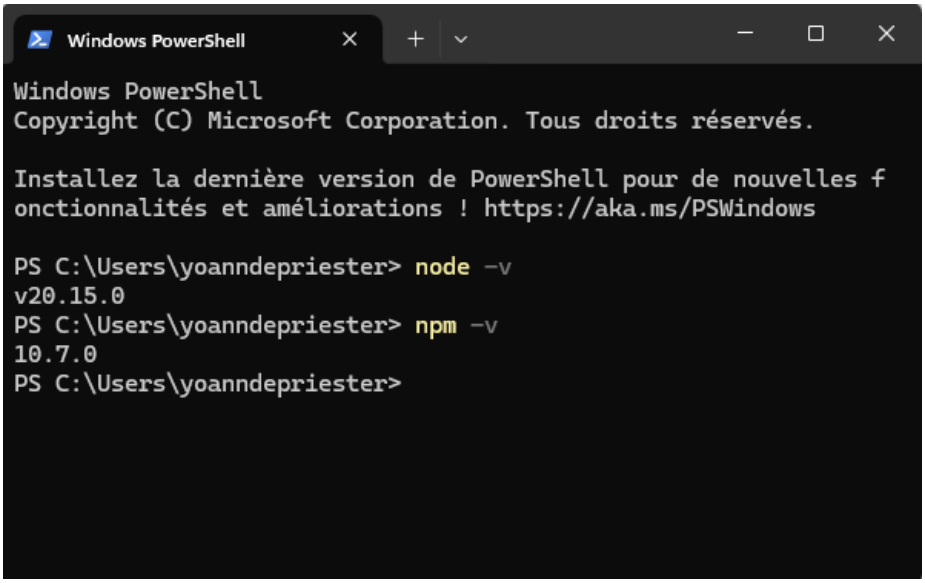
Puis exécuter le fichier pour installer Node.js



REACT

Une fois l'installation terminée, vérifions que nous possédons la bonne version de Node.js.

Ouvrez un terminal de commande et tapez la commande **node -v**. Si le numéro de version s'affiche, alors tout va bien. Faites de même pour vérifier la version en npm (le gestionnaire de paquet de Node.js) en tapant **npm -v**.



Maintenant vous pouvez installer REACT en tapant la commande **npm install react**

REACT

Vite.js

1. Présentation

Pour nous aider dans l'exploitation de REACT, nous allons utiliser Vite.js.

Vite est un outil de développement front-end créé par Evan You, le créateur de Vue.js. Cet outil vise à améliorer la rapidité de développement en offrant un serveur de développent rapide et une compilation optimisée des fichiers et modules Javascript.

Ainsi, utiliser Vite dans un projet REACT apporte des avantages non négligeables :

- **Temps de démarrage et rechargement rapides**
- **Configuration prête à l'emploi pour React**
- **Support TypeScript et JSX sans effort**
- **Optimisation des dépendances**

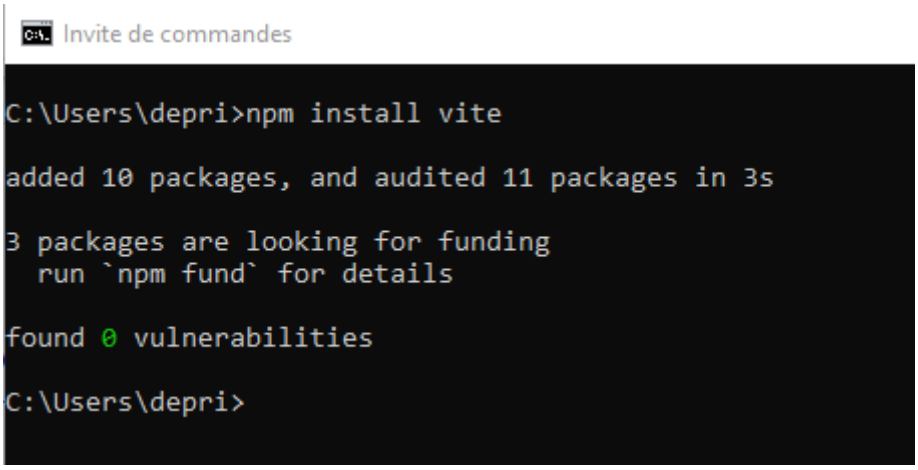
Lien vers le site officiel : https://vite.dev/

Lien vers une version française du site : https://vitejs.fr/

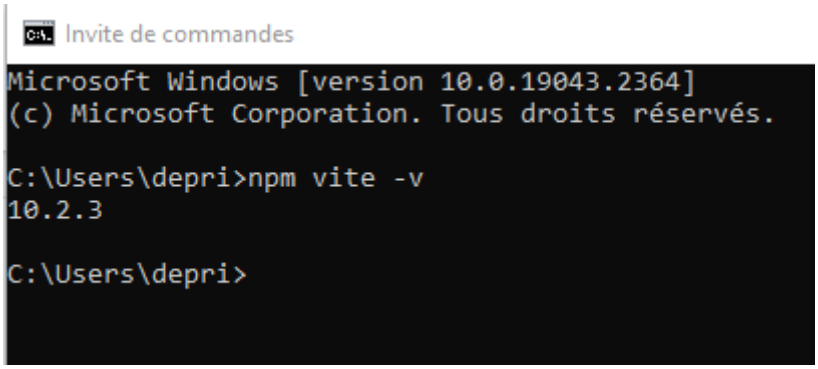
REACT

2. Installation de Vite.js

Pour installer Vite.js, nous allons ouvrir un terminal de commande et lancer la commande **npm install vite**



Une fois cela fait, nous pouvons vérifier le numéro de la version installée avec **npm vite -v**



Maintenant nous pouvons créer notre premier projet REACT.

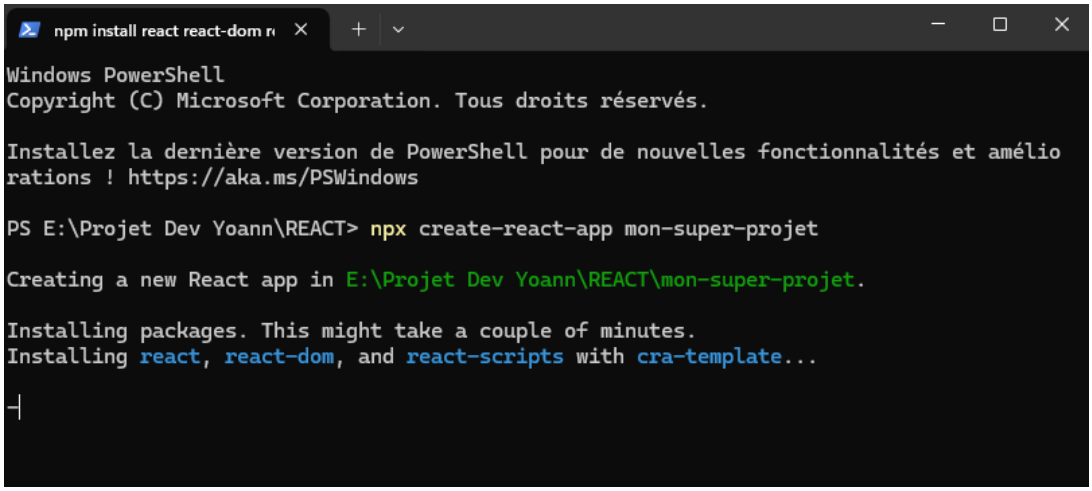
REACT

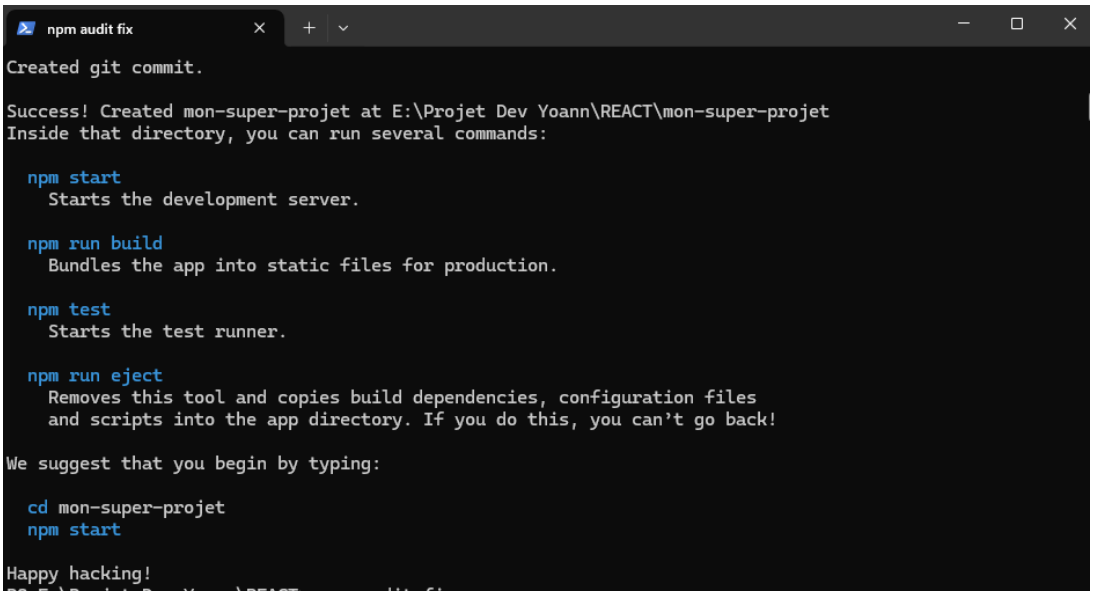
Créer un Projet REACT

1. Sans l'aide de Vite.js

Pour créer un projet REACT, ouvrez le terminal de commande, et déplacer vous dans le dossier où souhaitez créer votre projet.

Puis tapez la commande **npx create-react-app nom_de_votre_projet**





REACT

2. Avec Vite.js

Avec Vite.js, la commande à utiliser pour créer un projet REACT est **npm create vite@latest**

Après avoir valider l'installation du package demandé, Vite va vous demander le nom de votre projet. Il créera alors un dossier avec le nom entré.

```
PROBLEMS  OUTPUT  TERMINAL  PORTS  DEBUG CONSOLE

PS C:\Users\depri\Desktop\REACT> npm create vite@latest
Need to install the following packages:
create-vite@5.5.5
Ok to proceed? (y) y
? Project name: » my-first-project
```

Vous aurez ensuite à choisir le framework correspondant à votre projet.

```
? Select a framework: » - Use arrow-keys. Return to submit.
    Vanilla
    Vue
>   React
    Preact
    Lit
    Svelte
    Solid
    Qwik
    Angular
    Others
```

Puis, Vite vous demande quel langage sera utilisé pour votre projet. Nous allons choisir uniquement Javascript.

```
? Select a variant: » - Use arrow-keys. Return to submit.
    TypeScript
    TypeScript + SWC
>   JavaScript
    JavaScript + SWC
    Remix ↗
```

REACT

Il ne reste plus qu'à entrer dans le dossier de votre projet en tapant la commande **cd nom_projet**, puis à installe les modules nécessaires avec la commande **npm install.**

```
Scaffolding project in C:\Users\depri\Desktop\REACT\my-first-project...

Done. Now run:

  cd my-first-project
  npm install
  npm run dev

PS C:\Users\depri\Desktop\REACT> cd my-first-project
PS C:\Users\depri\Desktop\REACT\my-first-project> npm install
```

```
PS C:\Users\depri\Desktop\REACT> cd my-first-project
PS C:\Users\depri\Desktop\REACT\my-first-project> npm install

added 249 packages, and audited 250 packages in 9s

102 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\depri\Desktop\REACT\my-first-project>
```

REACT

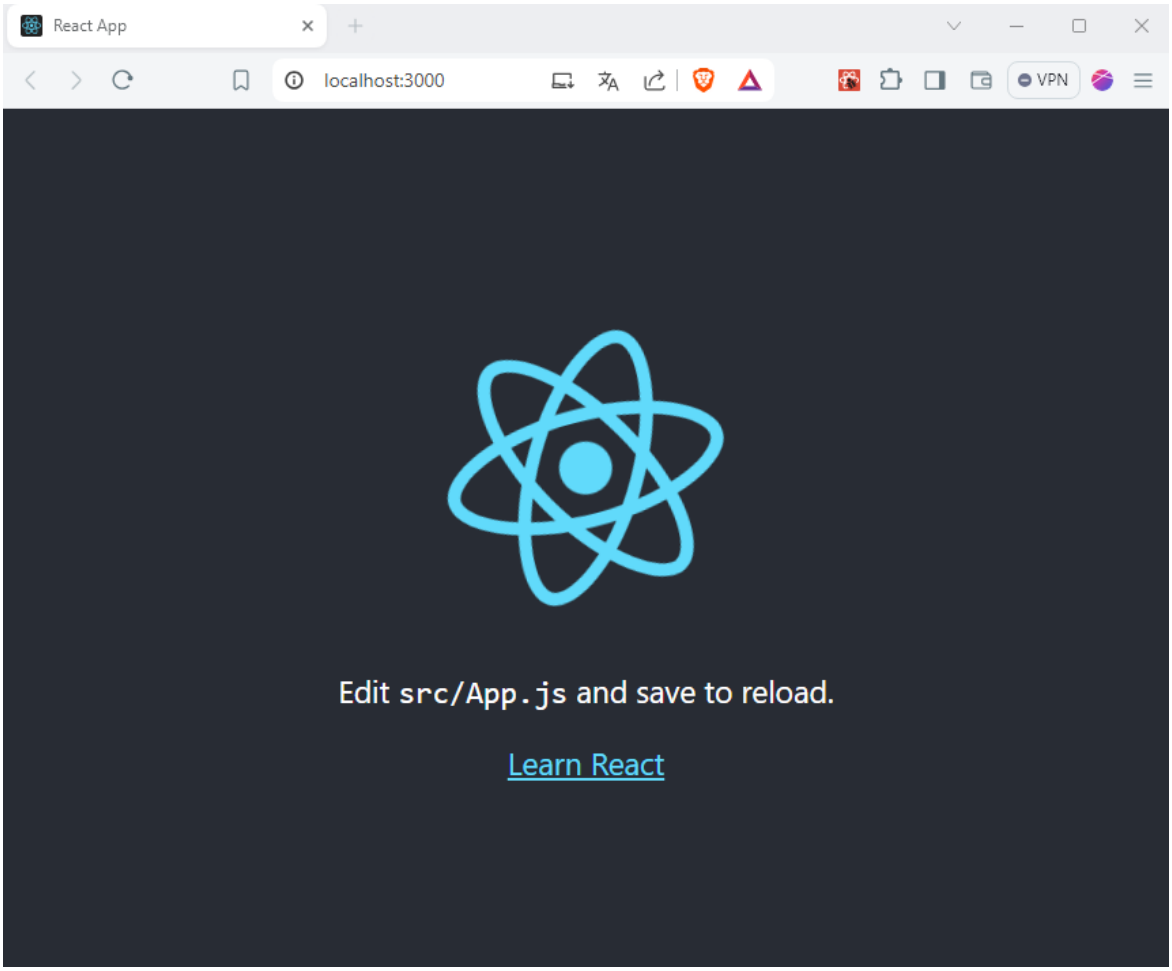
Lancer le Server REACT

1. Sans utiliser Vite.js

Afin de voir le résultat du développement de votre site/application web sans avoir à build votre projet, REACT embarque un server local permettant de le lancer.

Pour cela, au sein d'un terminal, allez dans le dossier racine de votre projet, et tapez la commande **npm start**.

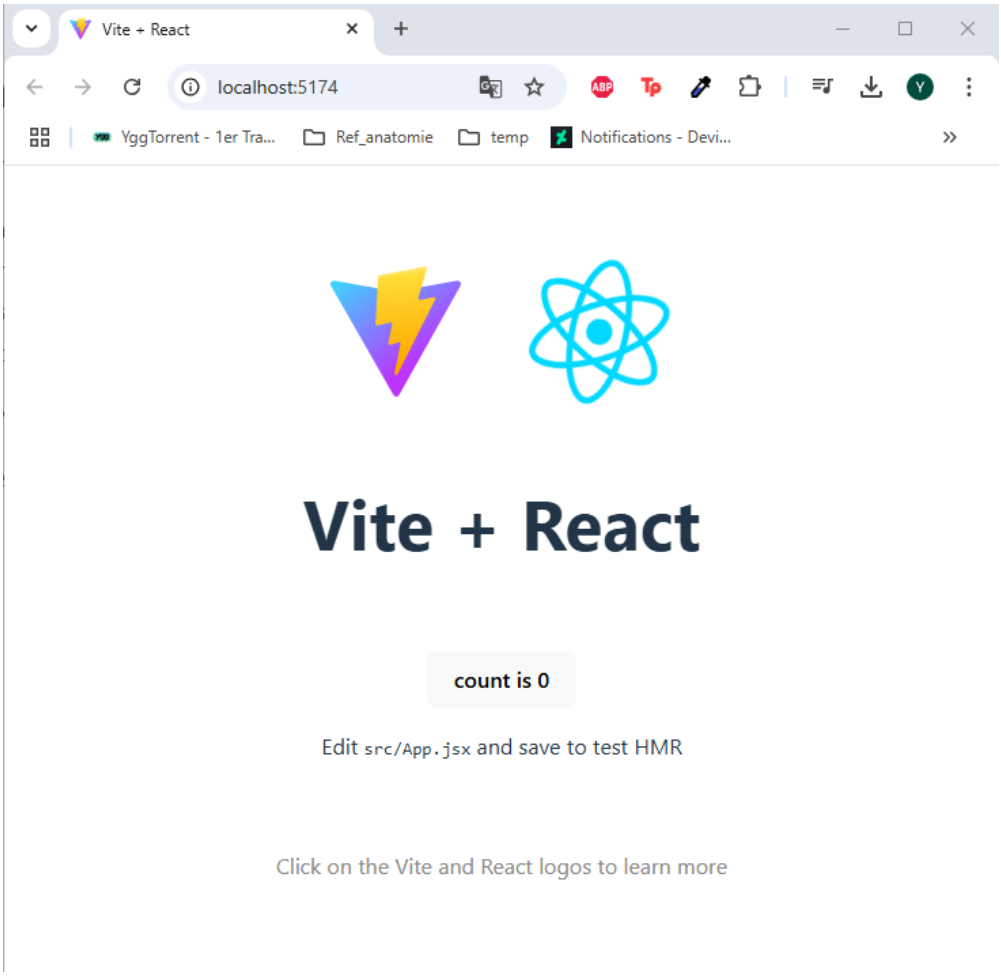
Une fois le server lancé, il ouvrira votre projet dans un nouvel onglet de votre navigateur web.



REACT

2. Avec Vite.js

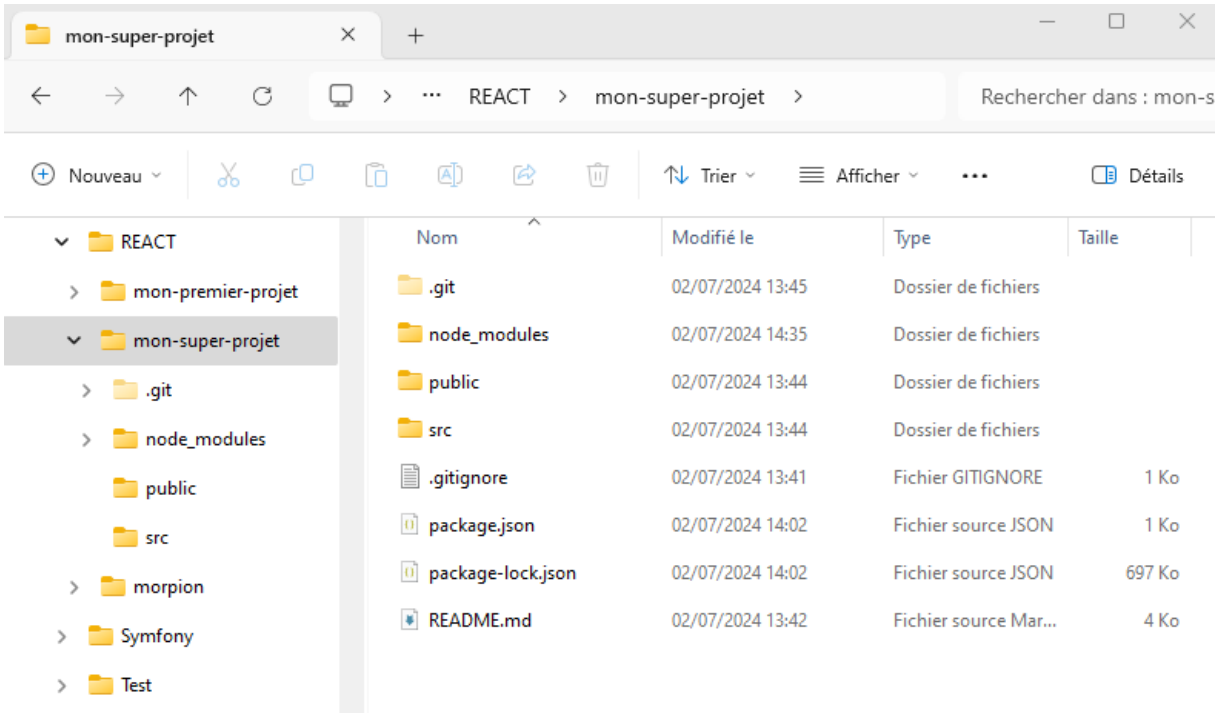
Toujours dans le dossier racine de votre projet, lancer la commande **npm run dev**



REACT

Structure du Dossier

En explorant pour la première fois le dossier de notre nouveau projet, voilà la structure sur laquelle on tombe.



On peut constater grâce au dossier **.git** que React a initialisé le dossier en tant que dépôt local git. Il a aussi créé un fichier **.gitignore**, déjà rempli avec le nom de dossier et fichier qui seront ignorés par le dépôt distant.

Le dossier **node_modules** contient tous les modules importés par Node.js pour le bon fonctionnement de React.

Le dossier **public** contient les fichiers accessibles aux visiteurs.

Le dossier **src** contient les fichiers sources de votre site/appli comme les fichiers CSS ou JS.

REACT

Les Composants

Les Composants sont la base sur laquelle repose la librairie REACT.

Un Composant n'est rien d'autre qu'un bout d'UI (Interface Utilisateur) qui, assemblé avec d'autres Composants, va construire votre page. Ainsi, un header, un formulaire, un bouton, ou un menu de navigation sont autant de composant que vous développerez avec REACT pour créer les pages visibles par vos utilisateurs.

De plus, il faut garder en tête qu'un Composant peut lui-même être constitué d'un ou plusieurs autres Composants.

1. Définir un Composant

Un Composant REACT est toujours défini au sein d'un fichier **.jsx** dont le nom commence par une majuscule, au sein du dossier **src.**

Dans ce fichier, le Composant prendra la forme d'une fonction dont le nom **commence TOUJOURS par une majuscule**, et qui appellera la fonction **return** pour renvoyer des balises ou d'autres composants écrits en JSX.

Exemple :

Nous souhaitons créer un composant pour définir une liste d'ingrédient.

*Dans le dossier **src**, nous créons un fichier **List.js** (la majuscule est importante pour comprendre clairement qu'il s'agit d'un composant). Puis au sein de ce fichier, nous définissions notre fonction **List()** qui va **return** notre liste :*

```
function List() {
    return (
      <ul>
        <li>Salade</li>
        <li>Tomate</li>
        <li>Oignon</li>
        <li>Sauce Blanche</li>
      </ul>
    );
}
```

2. Exporter / Importer un Composant

Définir un Composant ne suffit pas à le rendre utilisable au sein de notre projet. Pour l'utiliser au sein d'une page, il faut l'**exporter**. Pour cela, nous devrons utiliser à la fin du fichier l'instruction **export default nom_du_composant** (sans les parenthèses de la fonction).

L'instruction **export default** définit quel composant sera exporté. Il ne peut y avoir qu'un seul et unique **export default** par fichier **.jsx**.

REACT

Voyons maintenant comment **importer** notre composant pour l'utiliser au sein de notre projet. Tout cela se passe au sein d'un fichiers **.jsx** où je souhaite utiliser mon composant. A l'intérieur, nous devons utiliser l'instruction **import Nom_composant from** '**chemin_relatif_du_composant**'.

Exemple :

*Exportons notre composant **List** pour l'utiliser au sein d'un composant visant à faire une **Card**.*

*Pour cela, **exportons** notre composant :*

```
function List() {
    return (
      <ul>
        <li>Salade</li>
        <li>Tomate</li>
        <li>Oignon</li>
        <li>Sauce Blanche</li>
      </ul>
    );
}

export default List;
```

*Maintenant, dans un fichier **Card.jsx**, créons un composant **Card** et importons notre **List.** Nous utiliserons le langage **JSX** pour intégrer notre **List** au sein de notre **Card** :*

```
import List from "./List";

function Card(){
    return (
        <List />
    );
}

export default Card;
```

*Nous avons dorénavant un composant **Card** qui affichera notre **List** lorsqu'on l'intégrera dans une page. **Card** imbriquant le composant **List**, cela fait de **Card** le composant **parent**, et de **List** son composant **enfant**. Mais comment intégrer ce composant **Card** dans une page justement ? Pour cela, ça se passe dans le fichier **index.jsx**.*

REACT

3. Intégrer un composant dans une page : les fichiers index.html et index.js

Lorsque vous lancez le server REACT, ce dernier va générer une page à partir du fichier **index.html** (qui se trouve dans le dossier **public**) qui sert de template, et en utilisant le fichiser **index.js** (du dossier **src**) pour définir le contenu.

Voyons donc à quoi ressemble ces 2 fichiers :

Fichier index.html :

public > index.html > html > body > h1

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
      user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the build.
      Only files inside the `public` folder can be referenced from the HTML.

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-root public URL.
      Learn how to configure a non-root public URL by running `npm run build`.
    -->
    <title>React App</title>
  </head>
```

Sans surprise, la partie **<head>** du fichier contient les paramètres et informations de la page.

Intéressons-nous plutôt au **<body>** :

REACT

public > index.html > html > body > noscript

```
1   <!DOCTYPE html>
2   <html lang="en">
3   >  <head> ···
28     </head>
29     <body>
30       <noscript>You need to enable JavaScript to run this app.</noscript>
31       <div id="root"></div>
32       <!--
33         This HTML file is a template.
34         If you open it directly in the browser, you will see an empty page.
35
36         You can add webfonts, meta tags, or analytics to this file.
37         The build step will place the bundled scripts into the <body> tag.
38
39         To begin the development, run `npm start` or `yarn start`.
40         To create a production bundle, use `npm run build` or `yarn build`.
41       -->
42     </body>
43   </html>
44
```

Dans cette partie, à la ligne 31, nous trouvons une ** ** possédant l'**id root**. C'est cette balise que le fichier **index.js** va viser pour inclure le contenu de la page.

Fichier index.js :

src > JS index.js > ...

```
1   import React from 'react';
2   import ReactDOM from 'react-dom/client';
3   import './index.css';
4   import App from './App';
5   import reportWebVitals from './reportWebVitals';
6
7   const root = ReactDOM.createRoot(document.getElementById('root'));
8   root.render(
9     <React.StrictMode>
10      <App />
11    </React.StrictMode>
12  );
13
14  // If you want to start measuring performance in your app, pass a function
15  // to log results (for example: reportWebVitals(console.log))
16  // or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
17  reportWebVitals();
18
```

REACT

Tout d'abord, ce fichier importe les composant **React** et **ReactDOM** nécessaires au bon fonctionnement de REACT.

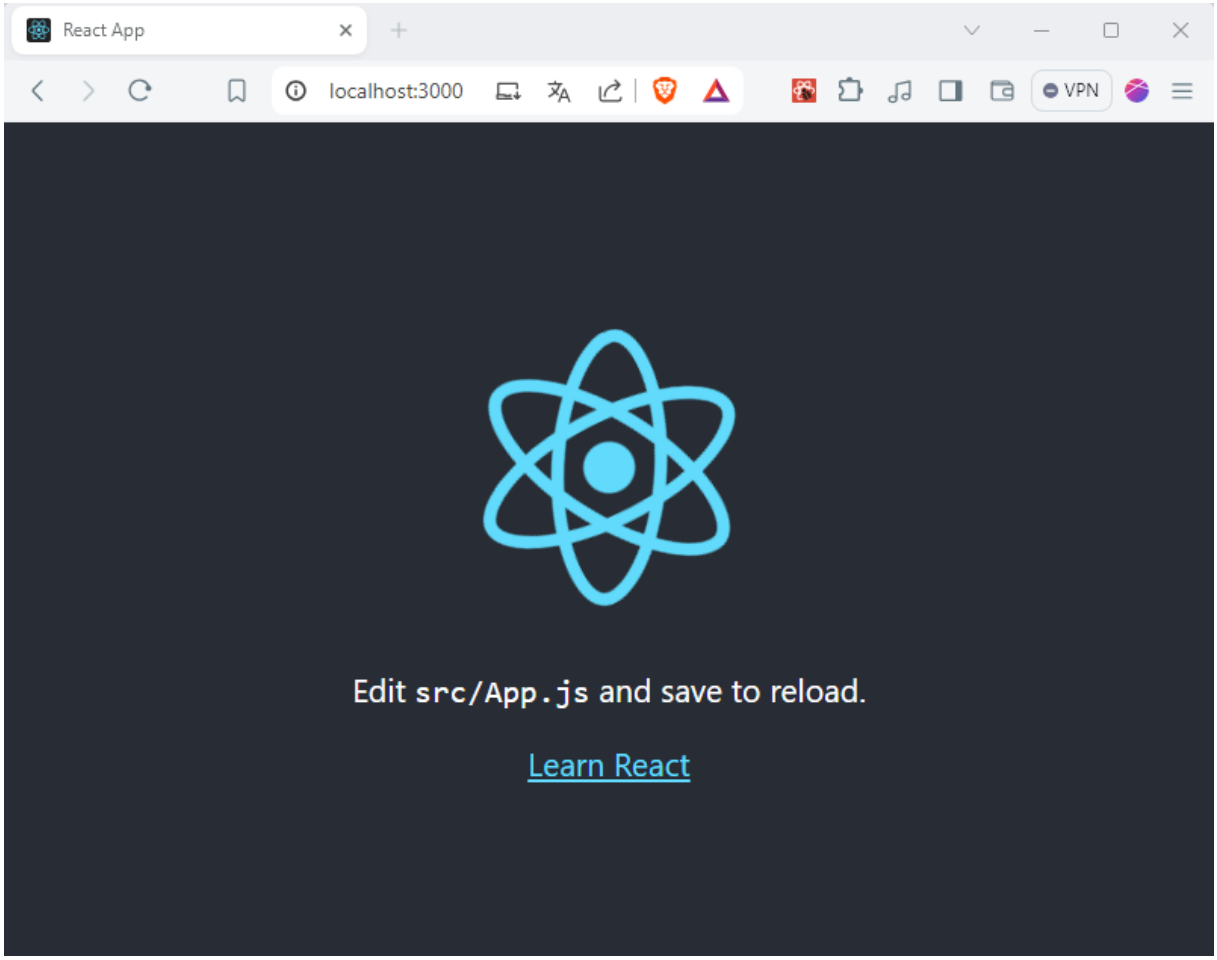
Puis il importe le fichier **index.css** qui permet la mise en forme de la page.

Note : pour styliser un composant, il sera préférable de créer un fichier css qui lui est dédié, et importer ce fichier au sein du fichier du composant.

Le fichier importe aussi le composant **App** à partir du fichier **App.jsx** pour l'utiliser au sein de la page. Vous l'aurez compris, ce composant correspond à ce qu'affiche la page de votre projet la première fois que vous lancez le server REACT.

Enfin, c'est à partir de la ligne 7 que la magie opère. La constante **root** est définie pour viser l'élément HTML possédant l'**id root**. Puis la méthode **render()** est appelé sur cette constante pour faire le rendu des composants appelé en son sein.

Au départ, cela donne donc le rendu suivant :



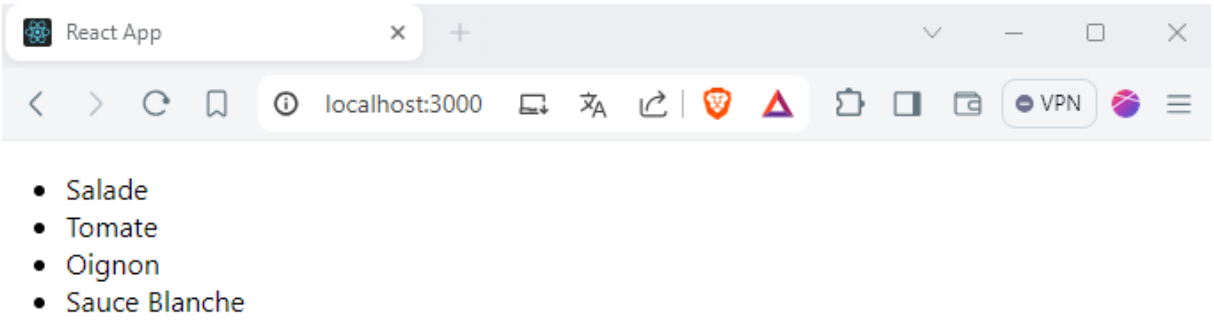
REACT

Exemple :

*Intégrons maintenant notre **Card** au rendu de la page. Nous devons commencer par l'**importer,** puis l'appeler dans la méthode **render()** :*

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

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <List />
  </React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Voici ce que nous obtenons :



*Note : il est probable que vous vous demandiez comment créer une nouvelle page REACT pour utiliser nos composants. Pour cela, nous avons besoin d'un système de **routage**, mais patience on y reviendra plus tard.*

REACT

4. Contrainte : Renvoyer un seul élément racine

L'une des contraintes à respecter lorsque l'on définit un Composant, est que ce dernier ne peut pas retourner plusieurs éléments en même temps. A la place, il est nécessaire de les imbriquer dans un unique élément parent.

Pour réaliser cela, imbriquez vos éléments au sein d'une autre balise, comme la balise ** **, ou bien utilisez un **Fragment**. Un **Fragment** est tout simplement une balise vide **<> </>** qui ne sera pas interprété par le navigateur et n'entraînera dont pas la création d'un élément HTML au sein du DOM.

Exemple :

Notre liste d'ingrédients est bien belle, mais cela manque d'un titre pour ajouter du contexte.

*Si nous essayons d'ajouter directement un titre **<h1></h1>** à notre composant **Card**, nous avons une erreur au sein de notre éditeur de code :*

src > JS Card.js > default

```
1  import List from "./List";
2
   Codiumate: Options | Test this function
3  function Card(){
4      return (
5          <h1>Le Parfait Kebab</h1>
6          <List />
7      );
8  }
9
10 export default Card;
```

*Pour corriger cette erreur, nous devons envelopper notre **<h1>** et notre **List** d'une autre balise. Par exemple une balise **<article> </article>** :*

src > JS Card.js > default

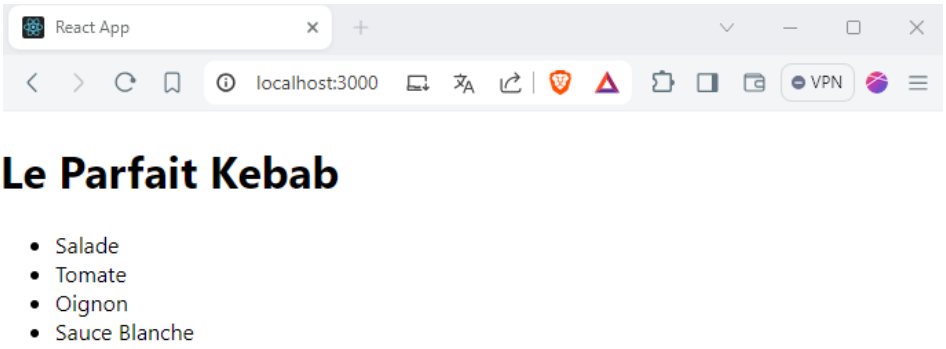
```
1  import List from "./List";
2
   Codiumate: Options | Test this function
3  function Card(){
4      return (
5          <article>
6              <h1>Le Parfait Kebab</h1>
7              <List />
8          </article>
9      );
10 }
11
12 export default Card;
```

REACT

*Nous pouvons aussi utiliser un **Fragment <> </>** à la place :*

```
src > JS Card.js > default
1   import List from "./List";
2
    Codiumate: Options | Test this function
3   function Card(){
4       return (
5           <>
6               <h1>Le Parfait Kebab</h1>
7               <List />
8           </>
9       );
10  }
11
12  export default Card;
```

*Maintenant, voici le rendu de notre composant **Card** (n'oubliez pas d'importer votre composant **Card** dans le fichier **index.jsx** et d'intégrer la balise **JSX <Card />** dans la méthode **render()**) :*



REACT

Les Classes HTML

Lorsque nous écrivons nos Composants, nous souhaitons souvent leur attribuer une ou plusieurs classes HTML. En HTML, l'attribut **class** est fait pour ça, mais en Javascript, il s'agit d'un terme réservé. A la place, vous devrez utiliser l'attribut **className** qui sera traduit par la méthode **render()** en attribut **class**.

*Note : même si l'utilisation de l'attribut **class** semble fonctionner, il ne faut pas l'utiliser.*

Exemple :

*Ajoutons une **class** card à la balise **<article>** de notre composant **Card.** Profitons-en pour créer un fichier **Card.css** pour le styliser notre composant et l'importer dans notre fichier **.jsx** :*

Fichier Card.jsx :

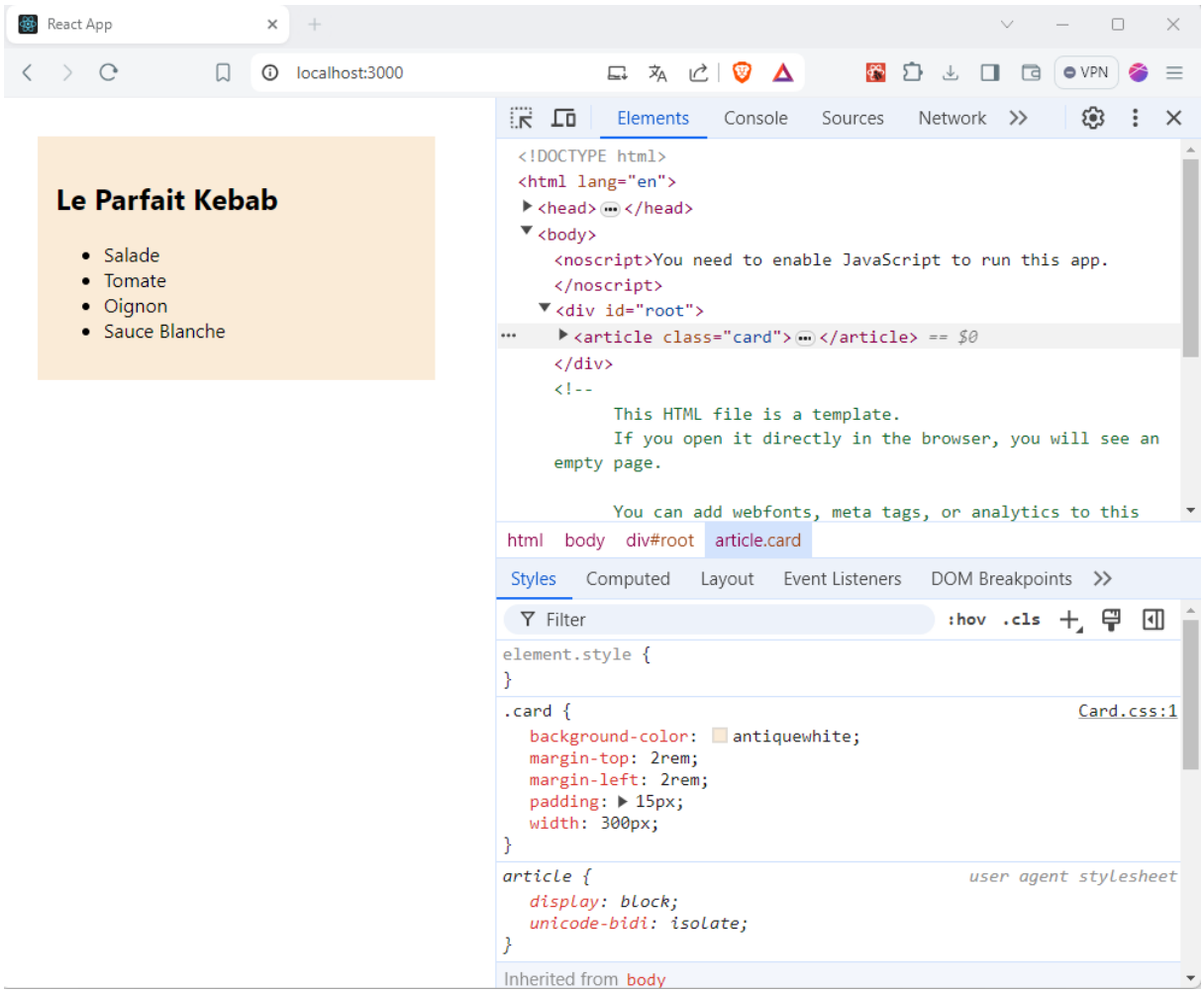
```
import List from "./List";
import './Card.css';

function Card(){
    return (
        <article className="card">
            <h1>Le Parfait Kebab</h1>
            <List />
        </article>
    );
}

export default Card;
```

Fichier Card.css :

```
.card {
    background-color: antiquewhite;
    margin-top:2rem ;
    margin-left: 2rem;
    padding:15px;
    width: 30vw;
}
```

REACT

Rendu :



REACT

EXERCICES

Exercice 1 : ToDo List partie 1

1) Avec Vite.js, initialiser un nouveau projet REACT javascript nommé ToDoList (npm create vite@latest)
2) Lancer le server Node avec Vite.js pour afficher votre projet dans un navigateur (npm run dev)
3) Dans le fichier App.jsx du dossier **/src**, créer une fonction ToDo() qui retourne une balise
4) Créer dans la balise plusieurs contenant chacun le nom d'une tache à faire
5) Dans la fonction App(), effacer le code déjà. Puis, créer une constante DATE qui instancie un new Date().
6) Faire en sorte que le Composant retourne un titre <h1>, un titre <h2> qui affiche la date du jour en appliquant la méthode toLocaleString() à la constante DATE , puis le Composant ToDo

Exercice 2 : Card User partie 1

1) Avec Vite.js, initialiser un nouveau projet REACT javascript nommé CardUser (npm create vite@latest)
2) Lancer le server Node avec Vite.js pour afficher votre projet dans un navigateur (npm run dev)
3) Dans le dossier **/src**, créer un fichier Card.jsx et un fichier Card.css
4) Dans le fichier Card.jsx, importer le fichier CSS, et créer une fonction Card() qui retourne une balise <article></article>
5) Au sein de la balise <article>, créer le profil d'un utilisateur. Il aura :
   - Une image de profil (utiliser un générateur d'image, ou lorem picsum comme source)
   - Un Pseudo dans un balise h2
   - Une Email dans une balise h3
   - Une description dans une balise p
6) Donner à la balise <article> la class « card »
7) Dans le fichier CSS, styliser votre Card en utilisant au moins une fois la classe « card »
8) Dans le fichier App.jsx, importer le composant Card, puis effacer le code au sein de la fonction App()
9) Faire en sorte que la fonction App() retourne un fragment <></>
10) Au sein du fragment, afficher une balise <h1> ayant pour contenu « Liste des Utilisateurs »
11) En dessous du <h1> afficher votre composant <Card />

Les Props

En REACT, les Composants parents peuvent passer des données à leurs Composants enfants. Pour cela, les Composants utilisent des **Props**.

Un **Props** est comme un attribut HTML ou un argument d'une fonction Javascript, et peut prendre n'importe quel type de valeur JS : un nombre, une chaîne de caractère, un tableau, un objet, ou même une fonction, ...

Les **Props** que vous pouvez passer à une balise sont bien définis (REACT respectant le standard HTML). Par exemple, une balise **<img>** aura comme **Props** : ***className, src, alt, width et height***. Néanmoins, lors que vous créez vos propres Composants, vous pouvez passer les **Props** de votre choix.

1. Passer des Props au composant enfant

Au moment où vous appelez votre composant, vous pouvez lui passer des **Props** en utilisant la syntaxe suivante :

```
<VotreComposant nom_props={valeur} />
```

Faites bien attention à utiliser les **accolades JSX** pour passer vos données.

*Note : dans l'absolu, il n'est pas nécessaire d'utiliser les **{}** pour assigner des chaînes de caractère.*

Exemple :

*Plutôt que d'écrire nos éléments de liste **<li>** en dur dans notre composant **List**, nous allons créer un Composant **ListElement** dans le fichier **List.jsx**, ce qui rendra nous permettra d'avoir un code réutilisable et paramétrable pour créer des **<li>** :*

```
function ListElement(){
    return <li></li>
}
```

*Note : il est possible de définir plusieurs Composants dans un même fichier. Mais au final, un seul d'entre eux sera exporté avec **export default**.*

*Puis on imbrique nos éléments de liste au sein de notre Composant **List**, et on en profite pour leur passer des **Props** : de quoi leur donner un **index** et un nom d'**ingredient**. On assigne à chaque **Props** une valeur en la passant entre accolade **{} :***

```
function List() {
    return (
      <ul>
        <ListElement index={0} ingredient={'Salade'}/>
        <ListElement index={1} ingredient={'Tomate'}/>
        <ListElement index={2} ingredient={'Oignon'}/>
        <ListElement index={3} ingredient={'Sauce Blanche'}/>
      </ul>
    );
}
```

2. Lire les Props dans le composant enfant

Pour que les **Props** passés à un Composant lors de son appel soient fonctionnels, il faut dire au Composant comment les lire. Pour cela, il faut tout d'abord les passer en arguments de la fonction définissant le Composant, en les encadrant par des accolades **{}**, chaque **Props** étant séparé par une virgule :

```
function VotreComposant({nom_props1, nom_props2}){}
```

*Note : les **Props** ont le même rôle que les arguments des fonctions. Ils sont même les seuls arguments que peuvent avoir un Composant.*

Puis il est nécessaire d'indiquer comment le Composant les utilise :

```
function VotreComposant({nom_props1, nom_props2}){
  return <div className={nom_props1}>{nom_props2}</div>
}
```

Exemple :

*Complétons notre Composant **ListElement** pour lui dire comment lire ses **Props index** et **ingredient** :*

```
function ListElement({index,ingredient}){
    return <li index={index}>{ingredient}</li>
}
```

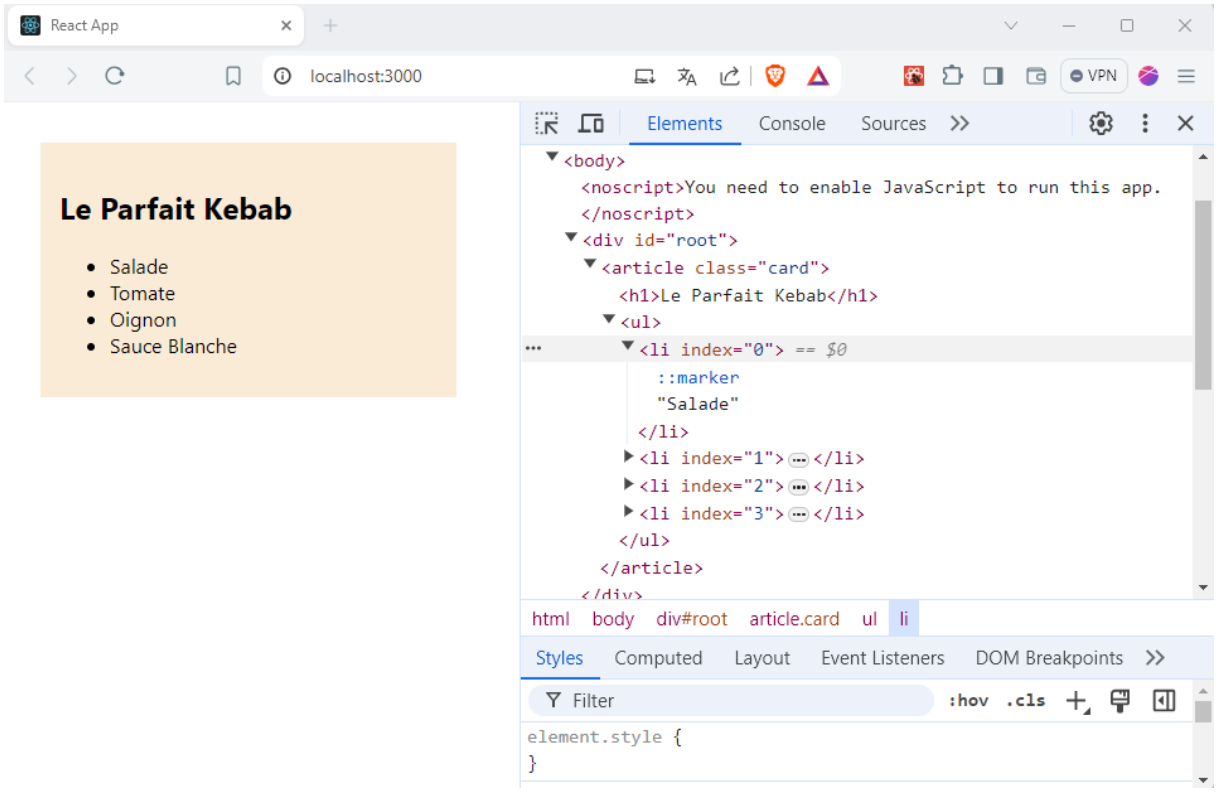
Ainsi, nous obtenons un Composant réutilisable et paramétrable avec des données à chaque fois qu'on l'appellera.

REACT

Code entier du composant List :

```
function ListElement({index,ingredient}){
    return <li index={index}>{ingredient}</li>
}

function List() {
    return (
      <ul>
        <ListElement index={0} ingredient={'Salade'}/>
        <ListElement index={1} ingredient={'Tomate'}/>
        <ListElement index={2} ingredient={'Oignon'}/>
        <ListElement index={3} ingredient={'Sauce Blanche'}/>
      </ul>
    );
}

export default List;
```

Rendu :



3. Valeur par défaut

Comme pour les arguments d'une fonction, il est possible d'assigner une valeur par défaut à un **Props**. Le Composant utilisera alors cette valeur lorsqu'elle ne sera pas spécifiée pour le **Props** au moment de l'appel du Composant :

```
function VotreComposant({nom_props1, nom_props2="valeur_par_défaut"}){
  return <div className={nom_props1}>{nom_props2}</div>
}
```

Exemple :

*Donnons à notre **Props ingredient** une valeur par défaut. Ce sera l'ingrédient mystère de notre Super Kebab :*

```
function ListElement({index,ingredient="L'Ingrédient Mystère"}){
    return <li index={index}>{ingredient}</li>
}
```

*Maintenant, si on imbrique un **ListElement** au sein de notre **List** sans définir le nom de l'ingrédient, notre Composant **ListElement** prendra la valeur par défaut pour ce **Props :***

```
function List() {
    return (
      <ul>
        <ListElement index={0} ingredient={'Salade'}/>
        <ListElement index={1} ingredient={'Tomate'}/>
        <ListElement index={2} ingredient={'Oignon'}/>
        <ListElement index={3} ingredient={'Sauce Blanche'}/>
        <ListElement index={4}/>
      </ul>
    );
}
```

Le Parfait Kebab

- Salade
- Tomate
- Oignon
- Sauce Blanche
- L'Ingrédient Mystère

4. Le Props CHILDREN

Il existe un **Props** natif qui s'appelle **children** dont le but est de réserver un endroit au sein du **Composant** pour en insérer d'autres.

Pour utiliser le **Props children**, il suffit de l'implémenter comme n'importe quel autre **Props**. Cependant, lors de l'utilisation du **Composant**, il n'y pas besoin de passer une valeur au **Props children**. A la place, on appel le **Composant** à l'aide d'une **balise ouvrante** et d'une **balise fermante**, et on passe à l'intérieur les **Composants** que l'on souhaite y insérer.

Exemple :

Créons une liste non ordonnée ayant un gros titre et pouvant accueillir n'importe quel **Composant** :

```
function ExempleList({children}){
return <>
            <h1>Ma Liste</h1>
            <ul>
                {children}
            </ul>
        </>
}
```

Et créons maintenant un **Composant** pour nos éléments de liste :

```
function ElementList({element}){
    return <li>
                {element}
            </li>
}
```

REACT

Voilà maintenant comment importer les deux :

```
root.render(
  <React.StrictMode>
    <ExempleList>
        <ElementList element={'Premier Element'}/>
        <ElementList element={'Deuxième Element'}/>
        <ElementList element={'Troisième Element'}/>
    </ExempleList>
  </React.StrictMode>
);
```

Et voici le rendu :

REACT

EXERCICES

Exercice 1 : ToDo List partie 2

1) Reprendre l'exercice ToDo List précédent
2) Modifier le Composant ToDo pour qu'il prenne un Props **todo** et un Props **date**. Ce Composant renverra uniquement un <li> qui affichera le **todo** suivi de la **data**
3) Modifier le Composant App pour qu'à la place du composant ToDo, il renvoit maintenant un <ul> qui possède 3 Composants ToDo (chacun avec un **todo** et une **date** différente)

Exercice 2 : Card User partie 2

1) Reprendre l'exercice Card User précédent
2) Dans le fichier Card.jsx, enlever la source de l'image, le pseudo, l'email et la description (conserver les balises malgré tout)
3) A la place, utiliser une props :
   - **image** pour la source de l'image,
   - **pseudo** pour le pseudo de l'utilisateur
   - **email** pour l'email de l'utilisateur
   - **description** pour la description de l'utilisateur
4) Au début du fichier App.jsx, créer 3 constantes USER différents (avec un nom différent). Chaque constante est un objet possédant une **image**, un **pseudo**, un **email**, et une **description** (choisir les valeurs pour chaque objet)
5) Faire ne sorte que le Composant App affiche chaque objet USER en utilisant le composant Card

Tableau d'éléments

Une situation qui arrive fréquemment est l'affichage automatisée d'une liste d'élément requérant le même Composant, comme une todo-list ou bien, et c'est notre cas avec notre exemple, une liste d'ingrédient pour une recette.

Dans ce cas, ces éléments seront récupérés au sein de tableau, comme ça peut être le cas lorsque l'on récupère une série de donnée auprès d'une API. Comme il ne sera pas possible d'écrire les Composants un à un dans le code, il faudra passer par une boucle pour les générer automatiquement.

Nous avons 2 moyens de faire ça :

- Ecrire une boucle avant le **return** pour générer un tableau de nos Composants à afficher
- Ecrire une boucle dans le **return** pour générer chaque Composant à afficher

Voyons chacun de ces 2 cas avec notre exemple.

1. Tableau de Composant

Commençons par réécrire le code. Dans le fichier **App.jsx**, nous allons avoir un tableau avec la liste de nos ingrédients :

```
1  import './App.css'
2
3  const INGREDIENTS = ['Salade','Tomate','Oignon','Sauce Blanche']
4
```

Puis écrivons à la suite notre Composant ListElement. Cette fois-ci nous ne lui passeront pas de Props index :

```
1  import './App.css'
2
3  const INGREDIENTS = ['Salade','Tomate','Oignon','Sauce Blanche']
4
5  function ListElement({ingredient}){
6    return <li>{ingredient}</li>
7  }
```

REACT

Passons maintenant à notre Composant List. A l'intérieur nous déclarons un tableau vide dont le but est de récupérer chaque Composant ListElement généré par une boucle forEach() sur nos INGREDIENTS.

Une fois fait, nous n'aurons plus qu'à afficher la liste de composant en appelant le tableau au sein de notre return :

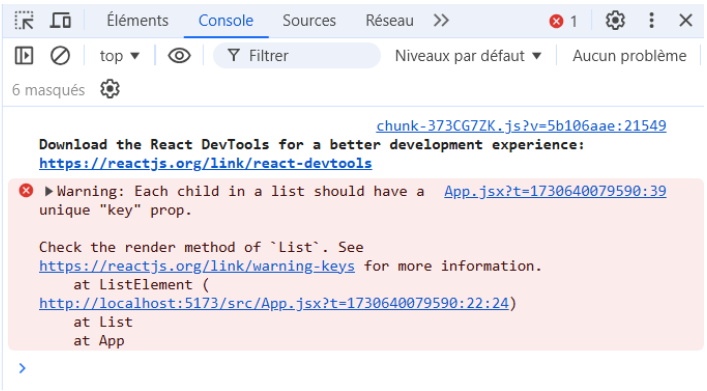
```
10  function List() {
11    const LIST_INGREDIENT = []
12
13    INGREDIENTS.forEach(ingredient => {
14      LIST_INGREDIENT.push(<ListElement ingredient={ingredient}/>)
15    })
16
17    return (
18      <ul>
19        {LIST_INGREDIENT}
20      </ul>
21    );
22  }
```

Nous n'avons plus qu'à importer notre List dans le Composant App (qui sera importé lui-même dans notre index) :

```
24
25  function App() {
26
27    return (
28      <>
29        <List />
30      </>
31    )
32  }
33
34  export default App
```

Le rendu :

- Salade
- Tomate
- Oignon
- Sauce Blanche



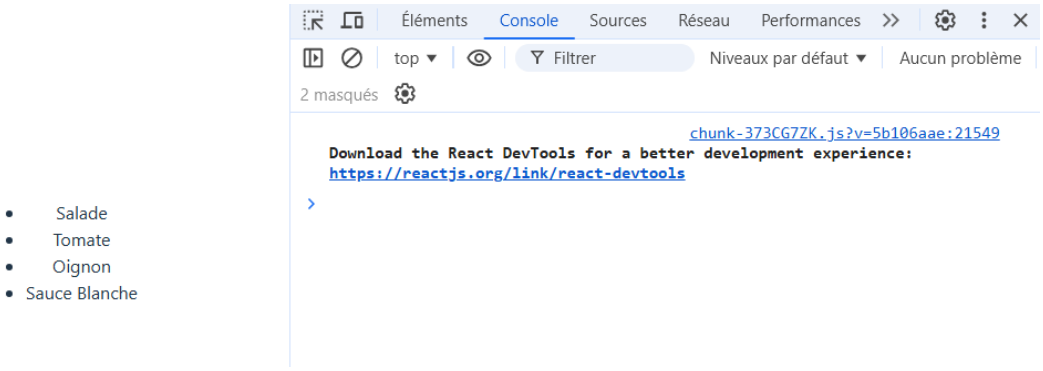
REACT

Outre l'affichage de notre liste d'ingrédient, nous observons toutefois une erreur. En effet, lorsque l'on génère une liste de Composant manière automatisée, REACT souhaite que l'on passe une **key** unique à chacun d'entre pour pouvoir les reconnaître. Ce sera le cas pour des éléments de liste comme ici, mais aussi pour une série de card <article> généré de la même manière.

Modifions alors la boucle au sein de notre Composant List pour qu'elle applique une **key** à nos ListElement. On donnera à chaque **key** l'index de notre ingrédient au sein de son tableau :

```
10  function List() {
11    const LIST_INGREDIENT = []
12
13    INGREDIENTS.forEach(ingredient => {
14      LIST_INGREDIENT.push(<ListElement key={INGREDIENTS.indexOf(ingredient)} ingredient={ingredient}/>)
15    })
16
17    return (
18      <ul>
19        {LIST_INGREDIENT}
20      </ul>
21    );
22  }
```

Nous pouvons voir maintenant, que lors du rendu, nous n'avons plus d'erreur :



REACT

2. Boucle d'Affichage

Pour la seconde manière de générer une liste d'élément, nous allons devoir écrire une boucle **map()** au sein du **return** de notre Composant List. Attention toutefois, contrairement à d'habitude, la fonction de callback de la boucle **map()** ne s'écrit pas entre accolade **{}** mais entre parenthèse **()** :

```
10  function List() {
11    return (
12      <ul>
13        {INGREDIENTS.map(ingredient=>(
14          <ListElement key={INGREDIENTS.indexOf(ingredient)} ingredient={ingredient}/>
15        ))}
16      </ul>
17    );
18  }
```

Cette manière de faire est généralement favorisé par les développeurs. Elle demande moins de ligne de code à écrire, et est aussi plus facile à lire.

REACT

Pour la suite, recréons notre Card dans le Composant **App** et continuons avec le code suivant :

```
1   import './App.css'
2
3   const INGREDIENTS = ['Salade','Tomate','Oignon','Sauce Blanche']
4
5   function ListElement({ingredient}){
6     return <li>{ingredient}</li>
7   }
8
9   function List() {
10    return (
11      <ul>
12        {INGREDIENTS.map(ingredient=>(
13          <ListElement key={INGREDIENTS.indexOf(ingredient)} ingredient={ingredient}/>
14        ))}
15      </ul>
16    );
17  }
18
19  function App() {
20    return (
21      <article className='card'>
22        <h1>Le Parfait Kebab</h1>
23        <List />
24      </article>
25    )
26  }
27
28  export default App
29
```

REACT

EXERCICES

Exercice 1 : ToDo List partie 3

Reprendre l'exercice ToDo List précédent.

1) Au début du fichier App.jsx, créer une constante TODOS qui est un tableau contenant plusieurs objets. Chaque objet possède une attribut **todo** et un attribut **date** (choisir les valeurs des attributs pour chacun des objets)

2) Au sein du Composant App, retourner, grâce une boucle, l'affichage de chaque ToDo sous forme de composant ToDo (ainsi, chaque ToDo ne sera plus affichée de manière individuelle). Ne pas oublier d'inclure une **key** à chaque ToDo.

Exercice 1 : Card User partie 3

Reprendre l'exercice Card User.

1) Au début du fichier App.jsx, réunir chaque objet USER dans un tableau d'objet.

2) Au sein du Composant App, retourner, grâce une boucle, l'affichage de chaque utilisateur sous forme de composant Card (ainsi, chaque Card ne sera plus affichée de manière individuelle). Ne pas oublier d'inclure une **key** à chaque Card.

Exercice 2 : Calendrier

1) Avec Vite.js, initialiser un nouveau projet REACT javascript nommé Calendar (npm create vite@latest)

2) Lancer le server Node avec Vite.js pour afficher votre projet dans un navigateur (npm run dev)

3) Dans le fichier App.jsx, avant la fonction App(), créer une constante MONTH qui sera un tableau répertoriant les mois de l'année

```
const MONTH = ['Janvier','Février','Mars','Avril','Mai',
  'Juin','Juillet','Août','Septembre','Octobre','Novembre','Décembre']
```

4) Créer une constante DAY qui sera un tableau répertoriant les jours de la semaine

```
const DAY = ['Lundi','Mardi','Mercredi','Jeudi','Vendredi',
'Samedi','Dimanche']
```

REACT

5) Dans une constante DAY_LETTER, à l'aide d'une boucle (map par exemple par exemple) appliquée à la constante DAY, récupérer au sein d'un tableau la première lettre de chaque jour de la semaine

6) Créer un Composant **Day**, qui aura les Props **jour** et **className**. Ce Composant retournera une ** ** qui aura un attribut **className** (qui récupère la valeur du Props **className**), et aura pour contenu la valeur du Props **jour**

7) Dans le Composant **App**, créer une constante ROW qui sera un tableau vide. Puis, à l'aide d'une boucle, remplit ce tableau de 31 Composants **Day** qui afficheront le numéros des jours de 1 à 31, et qui auront tous la class **backgroundOrange**.

8) Faire que le Composant **App** retourne un **<article>** avec la class **card**. Cet article contiendra :
- un Composant **Day** qui aura en Props **jour** l'index 9 du tableau MONTH, et en Props **className** la valeur '**month'**
- une <section> ayant comme class '**grid-7**'. Cette <section> sera constitué de chaque élément du tableau DAY_LETTER, chacun affiché avec le composant **Day** (utiliser une boucle map())
- une <section> ayant comme class '**grid-7 days**'. Cette <section> sera constitué de chaque élément du tableau ROW

9) Dans le fichier App.css, styliser votre rendu en exploitant les classes ajoutées :
- **backgroundOrange** : au survol de ces éléments, la couleur de fond passe en orange
- **month** : définit une couleur de fond orange et une taille de police de 2rem
- **grid-7** : définit un display grid avec un template de 7 colmuns de taille égale
- les éléments enfant de **grid-7** doivent avoir une taille de 50x50 px
- les éléments enfant de **days** doivent avoir un margin de 3px et un bordure grise de 1px