Node.js – React.js

Deux sources de documentation

- https://react.dev
- https://developer.mozilla.org/fr/docs/Web/JavaScript

1. Node.js

Node.js, créé en 2009, est un moteur javascript exécuté côté serveur. Alors que le JavaScript est traditionnellement utilisé du côté client pour créer des interfaces efficaces et conviviales, Node.js étend son utilisation au backend, permettant ainsi le développement complet d'applications web dynamiques.

1.1. Comment fonctionne Node.js – Comparaison PHP – Node.js

PHP	Node.js
- Mode de fonctionnement synchrone : le serveur reçoit une requête du client, exécute un script puis renvoie au client le résultat converti en html. Les appels au serveur sont bloquants pour le reste de la navigation. - Nécessite un serveur web externe - Surtout utilisé pour le développement web classique	- Mode de fonctionnement asynchrone car basé sur le moteur javascript interprétable par le client. Node.js élimine le blocage résultant de l'attente de la réponse d'un serveur. - Node.js est son propre serveur web - Peut être utilisé pour le développement d'applications mobiles multi-plateforme.

1.2. En particulier, que peut-on notamment faire avec Node.js ?

- Des Single Page Applications (SPA) : une SPA est caractérisée par l'utilisation d'une unique page HTML qui englobe toute l'application. Au démarrage, les éléments statiques tels que le header, le footer et un contenant vide sont chargés. Par la suite, la partie dynamique est généralement récupérée à l'aide d'API, ce qui améliore la fluidité de la navigation en évitant de recharger la page entière.

  Remarque : L'arborescence d'une application peut contenir plusieurs pages, mais chacune de celle-ci viendra dynamiquement meubler le contenant.

  Exemples de SPA : les applications de messagerie, réseaux sociaux, e-commerce

- Des Real-Time Applications (RTA) : le contenant est mis à jour en temps réel. Dans une même application, les RTA peuvent cohabiter avec des SPA ou des sites web classiques. Le fonctionnement d'une RTA repose sur un serveur qui surveille les mises à jour en temps réel et les communique au client. Le service peut être rendu par un serveur particulier, comme Node.js, où via des websockets indépendantes.
  Remarque : une websocket est une technologie utilisant un protocole de communication bidirectionnel en temps réel.
  Exemples de RTA : messageries, jeux multi-joueurs, vidéo-conférence

- Des API REST : communication de données entre applications web indépendantes.
  Contextes d'utilisation d'API REST : e-commerce, streaming, applications bancaires, etc.

1.3. Et React.js ?

Node.js est essentiel pour React.js car il offre des fonctionnalités JavaScript pour le développement, la gestion des dépendances (npm - Node Package Manager) et l'exécution des projets.

2. React.js – brève description

React.js est une librairie JavaScript créée par Facebook en 2013 pour le développement d'interfaces utilisateur(UI) fluides et rapides.

Une application React est construite à partir de "composants", qui sont des modules indépendants réutilisables, associés pour former l'application complète. On peut se représenter les composants comme des briques assemblées pour constituer un ensemble fonctionnel. Chaque composant est spécialisé dans une tâche particulière et se caractérise par son objet, sa logique, sa présentation, sa taille. Par exemple, un composant peut représenter un message, un formulaire, une liste dans un formulaire, une barre de navigation, un footer, une page entière voire un simple bouton.

React.js intègre d'autres librairies JavaScript, tierces ou propres, pour diverses tâches, telles que React Router, qui fournit les différents composants nécessaires à la navigation (le routage). Ces librairies ne sont pas toutes exclusives à React.js mais lui sont adjointes en fonction des besoins de l'application.

Les langages utilisés pour le développement d'une application React sont le HTML, le CSS, le JavaScript et le JSX (javascript XML).

3. Le JSX

Le JSX, ou JavaScript XML, n'est pas spécifique à React mais figure parmi les langages utilisés pour développer une application React.

Le JSX est une extension de la syntaxe javascript, qui simplifie l'intégration du HTML dans un fichier JavaScript. Une telle simplification d'écriture ne modifie pas le fonctionnement du HTML et du JavaScript, mais facilite le codage. On emploie d'ailleurs souvent l'expression de "sucre syntaxique". Le JSX ressemble à du HTML au niveau de sa forme, ou à du XML, qui lui donne sa dénomination.

En React, un composant JSX est une fonction javascript dans laquelle la logique, la structuration et la présentation se trouvent réunies. Le composant est de ce fait autonome et réutilisable puisque complet. Un exemple de composant JSX :

```
export default function Accueil() {
  const lib = "React";
  return (
    <div>
      <p>Bienvenue !</p>
      <p>{
        lib === "React" ? "Je suis une application React " : "Je ne suis pas une application React"
      }</p>
    </div>
  )
}
```

Règles du JSX

Il existe des convertisseurs html – JSX.

1. Le nom d'un composant commence par une majuscule

2. Bonne pratique : un composant par fichier; nommer le fichier comme le composant

3. Recommandation React : toutes les balises *devraient* être fermées de manière séparée, y compris les balises sans fermeture en HTML comme img ou input :

   ```
   <input className="une_classe" value="une_valeur" onChange={...}> <input>
   ```

4. Les attributs sont écrits en *camelCase*. Les mots réservés dans le langage JavaScript (class, for) change de dénomination ou de forme :

```
<input className="une_classe" value="une_valeur" onChange={...}><input>
<label htmlFor="nom">Nom : </label>
```

5. Un composant JSX **retourne un seul élément parent**, dont les enfants sont le code du composant. Dans l'exemple ci-dessus, l'élément retourné est le externe.

   On peut remplacer le parent par des balises vides **<> </>**

6. Insérer du JavaScript dans le JSX : **accolades**

   - Insérer des valeurs dynamiques

```
const titre = "Le langage ….";
const src="../images/image.jpg";
return (
  <div>
    <p>{titre}</p>
   <img src={src} alt="Illustration"></img>
  </div>
)
```

   - Utiliser des méthodes, par exemple la méthode itérative map()

```
 const tab = ["Jules", "Emma", "Louis", "Charline"];
 tab.sort();
return (
 <div>
   <p>
     { tab.map( (elt) => (<p>- {elt}</p>) ) }
   </p>
 </div>
)
```

A propos des fonctions fléchées et classiques à l'intérieur d'un composant : une fonction fléchée retourne directement l'élément à afficher; une fonction classique doit explicitement retourner la valeur à afficher :

```
{tab.map((elt) => (<p>- {elt}</p>))}<p> </p>
```

```
{tab.map(function(elt){ return <p>{elt}</p> })}
```

4. Le virtual DOM

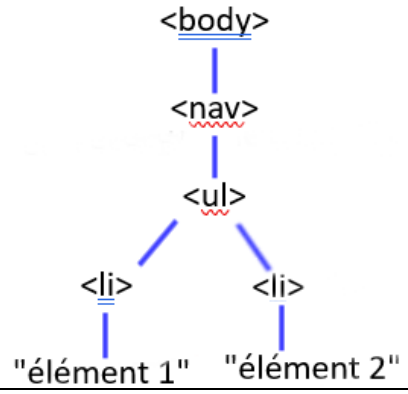
Le DOM virtuel est l'élément central de React.

Rappel du fonctionnement du HTML : le DOM, Document Object Model, est la représentation hiérarchique des objets d'une page web, telle qu'elle est rendue dans un navigateur. Tout objet de la page web est repris dans la hiérarchie, depuis la balise racine <html> jusqu'au contenu texte entre deux balises.

Un élément de l'arbre DOM s'appelle un nœud (node); chaque nœud contient une référence à son nœud enfant, de manière que la position de chaque nœud dans l'arbre soit clairement identifiée :

<table>
<tr><th>Code HTML</th><th>Représentation DOM sous forme d'arbre</th><th>Rendu navigateur</th></tr>
<tr>
<td>

```
<body>
  <nav>
    <ul>
      <li>élément 1</li>
      <li>élément 2</li>
    </ul>
  </nav>
</body>
```

</td>
<td> 
</td>
<td>. élément 1
. élément 2</td>
</tr>
<tr><td colspan="3">JavaScript : atteindre l'élément</td></tr>
<tr><td colspan="3">

```
let li1 = document.querySelector('nav ul li:nth-child(1)').textContent; //ou 'nav ul li'
let li2 = document.querySelector('nav ul li:nth-child(2)').textContent;
```

➔ Affichera le nœud élément 1 pour *li1* et le nœud élément 2 pour *li2*

</td></tr>
</table>

Lorsqu'une modification est apportée à un élément du DOM est modifié, cela implique le rechargement complet du DOM, ce qui est coûteux en termes de temps de chargement. React résout ce problème par la technique du Virtual Dom.

En React, le DOM est représenté sous forme d'un objet JavaScript; voici une représentation simplifiée de cet objet :

```
{
 type: 'body',
 children: [
  {
   type: 'nav',
   children: [
    {
     type: 'ul',
     children: [
      {
       type: 'li',
       children: ['élément 1']
```

```
        },
        {
          type: 'li',
          children: ['élément 2']
        }
      ]
    }
  ]
 }
]
}
```

Le composant React-DOM, qui appartient à la librairie React, crée une copie du DOM réel et la stocke en mémoire en tant qu'objet (comme ci-dessus). Ensuite, il compare cette copie avec l'objet d'origine (ci-dessus). Lorsqu'il détecte une différence, il repère sa position et effectue la mise à jour en mémoire. Ensuite, l'information est transmise au DOM réel, qui se met également à jour. Dans tous les cas, c'est uniquement l'objet concerné qui subira cette mise à jour.

Avantages

- Ce processus de différenciation / synchronisation (appelé *reconciling*) est particulièrement rapide puisqu'il s'effectue en mémoire.
- On évite les chargements complets parfois incessants du DOM.

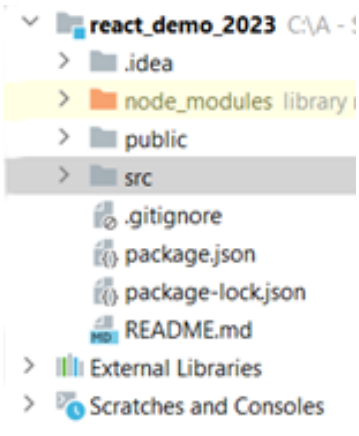
5. Un projet React de type SPA

5.1. Dossiers et fichiers par défaut

Installez Node.js sur https://nodejs.org (**choisir LTS**) puis vérifiez-en la bonne installation en entrant les commandes de vérification dans une fenêtre cmd : node -v et npm -v. Ces commandes afficheront la version du serveur et du manager de dépendances.

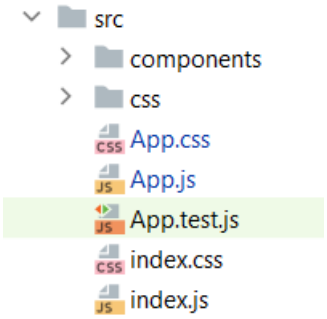
Dans Intellij, créez un projet React, dont le nom ne contiendra aucune majuscule. Stockez ce projet dans vos documents ou dans un autre dossier. Un serveur web indépendant n'est pas nécessaire.
Le projet démo affichera des offres d'emplois et permettra de rechercher les offres situées dans une ville.

Passons en revue l'arborescence et les codes par défaut.



Le dossier public contient les fichiers accessibles sans passer par le moteur de rendu de react, notamment le fichier index.html, point de départ de l'application.
Ce fichier contient la racine de l'application, qui sera utilisée dans index.js : `<div id="root"></div>`

Le fichier package.json liste notamment les dépendances et configuration des scripts de l'application; on y retrouvera notamment le package de routage parmi les dépendances, et le script start.



Les fichiers sources de l'application sont stockés dans le dossier src.
Le fichier public/index.html ne contient de balise <script> pour exécuter index.js, qui est le coeur de l'application. Lors de la création du projet, c'est npm qui charge un script "build" pour configurer l'application et appeler le script index.js.

Les scripts build et start sont stockés dans le dossier node_modules/react-scripts/scripts.

Dans le dossier src, créez les dossiers components et css.

Recommandations React :

- chaque composant devrait être unique dans son fichier
- chaque composant devrait être associé à son propre fichier css, de manière à ce que les styles ne concernent que le composant concerné.

Le respect de ces recommandations garantit une meilleure maintenance du projet, a fortiori lorsque celui-ci est développé en équipe.

Le fichier src/index.js

Il contient le premier composant, App.js, qui fournit l'affichage par défaut d'une application React.

Observons les différentes lignes, qui nous donnent déjà quelques informations sur la manière dont les fichiers de l'application sont intégrés dans d'autres fichiers :

```
1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3  import './index.css';
4  import App from './App';
```

```
5  const root = ReactDOM.createRoot(document.getElementById('root'));
6  root.render(
7    <React.StrictMode>
8      <App />
9    </React.StrictMode>
   );
```

Pour pouvoir utiliser les ressources React, que ce soit la librairie ou les fichiers sources, il faut les importer dans le fichier qui en fera usage. Les lignes 1 et 2 importent des fichiers de la librairie React.
La ligne 4 importe le composant par défaut App en précisant sa localisation dans le projet.
La ligne 5 récupère la div root définie dans index.html, pour en faire la racine de la page HTML.
Ensuite, la méthode render() assure le rendu du composant <App/> dans le navigateur.
Les lignes 7 à 9 sont écrites en JSX.

Comment React convertit-il le JSX en HTML ?
React compile le JSX pour en faire du JavaScript. Par exemple, une balise HTML sera convertie en JavaScript par la méthode paramétrée createElement(…). Ensuite, React crée ou recrée l'arbre DOM virtuel dont il a été question plus haut et met à jour le DOM réel.

Ajouter Bootstrap

Ouvrez le terminal intellij et tapez la commande :

npm i bootstrap

Dans le composant App.js, importez la librairie de cette manière, ainsi que le(s) fichier(s) css utiles :

```
import "bootstrap/dist/css/bootstrap.min.css";
import '../css/fichier.css';
```

5.2. Fichiers propres à l'application

Créer un dossier *css* dans le dossier src, ainsi qu'un dossier *composants*. Vous placerez les styles personnels et customisés dans le dossier css, et les composants dans le second dossier. Pour rappel, les composants consistent en la logique, structuration et présentation d'une fonctionnalité ou, le plus souvent, d'une partie de fonctionnalité. Leur nom commencent toujours par une majuscule.

Dans le dossier composants, créez les fichiers JavaScript Emploi.js et Recherche.js.

5.3. Routage

Le routage permet d'utiliser un menu de liens vers les différentes fonctionnalités ou composants.

La bibliothèque la plus courante est React Router.

Installez le système de routage à partir du terminal :

npm i react-router-dom

La dépendance installée sera visible dans le fichier package.json.

Le fichier App.js est le point de départ pour le code de l'application elle-même. Il contiendra le menu et le routage.

Remarque : le menu peut faire l'objet d'un composant séparé autonome.

Dans le composant App.js, commencez par supprimer les lignes inutiles. Vous obtenez :

```
import React from "react";
import "bootstrap/dist/css/bootstrap.min.css";

export default function App() {
  return (
    <>

    </>
  );
}
```

Ajoutez le composant de routage <BrowserRoute> et définissez les routes elles-mêmes. On importera également les composants utilisés dans le routage. Dans Intellij, vous pouvez cliquer droit sur les composants <BrowerRouter>, <Routes> et <Route> pour les importer en tête de fichier.

```
import React from "react";
import './App.css';
import "bootstrap/dist/css/bootstrap.min.css";
import Emplois from "./components/Emplois";
import Recherche from "./components/Recherche";
import {BrowserRouter, Routes, Route} from "react-router-dom";

export default function App() {
  return (
    <>
      <BrowserRouter>
          <Routes>
              <Route path="/emplois" element={<Emplois/>}></Route>
              <Route path="/recherche" element={<Recherche/>}></Route>
          </Routes>
      </BrowserRouter>
    </>
```

```
  );
}
```

Remarquez la manière de définir une route en React :

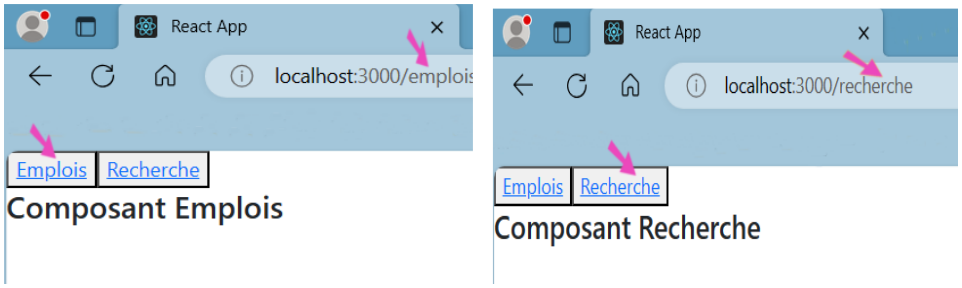
```
<Route path="/emplois" element={<Emplois/>}></Route>
```

- ➔ L'attribut path crée l'URL de la page, le lien lui-même vers cette page restant à définir.
- ➔ L'attribut element reçoit une expression JavaScript dont la valeur est le nom du composant.

Entre les balises <BrowserRouter>, ajoutez un menu que vous stylerez selon votre goût. Un menu Bootstrap est bien entendu approprié. Ajoutez l'import du composant **Link** de react-router-dom :

```
import React from "react";
import "bootstrap/dist/css/bootstrap.min.css";
import Emplois from "./components/Emplois";
import Recherche from "./components/Recherche";
import {BrowserRouter, Routes, Route, Link} from "react-router-dom";

export default function App() {
  return (
    <>
      <BrowserRouter>
        <nav>
          <button className=""><Link to={"/emplois"}>Emplois</Link></button>
          <button className=""><Link to={"/recherche"}>Recherche</Link></button>
        </nav>
        <Routes>
          <Route path="/emplois" element={<Emplois/>}></Route>
          <Route path="/recherche" element={<Recherche/>}></Route>
        </Routes>

      </BrowserRouter>
    </>
  );
}
```

Votre routage est prêt :



5.4. L'API fetch

Fetch est une API JavaScript standard (native) qui propose des fonctionnalités simples pour récupérer de manière **asynchrone** des données, que ce soit à partir d'un fichier interne ou hébergées à une adresse externe, comme une base de données quelconque. L'API offre toutes les fonctionnalités CRUD de gestion des données.

LA PROGRAMMATION ASYNCHRONE

1. Synchrone / asynchrone

La programmation est synchrone lorsque toutes les parties d'un programme s'exécutent séquentiellement les unes après les autres de sorte qu'un code en cours d'exécution bloque nécessairement l'exécution des autres blocs de codes. La programmation procédurale en C est synchrone.
Dans la programmation asynchrone, l'ordre séquentiel n'existe plus car certaines tâches peuvent être exécutées en arrière-plan, laissant le reste du code s'exécuter sans entrave. En JavaScript, les opérations asynchrones sont gérées par un mécanisme composé de callbacks, d'async/await (JavaScript 2017) et de *promesses*.

2. L'objet Promise

Un objet *Promise* représente le résultat réussi ou échoué d'une requête asynchrone. Une réussite entraîne l'appel du callback *resolve*, un échec l'appel du callback *reject*. Pour chacun de ces deux cas, un traitement spécifique est prévu. Par exemple, on peut afficher les données récupérées avec succès ou afficher un message d'erreur en cas de problème.

Le résultat de la requête s'appelle Promesse parce que l'objet s'engage à fournir ultérieurement (asynchronisme oblige) un résultat. En JavaScript les API axios (librairie tiers) et fetch (native), par exemple, se basent sur l'objet Promise pour leurs fonctionnalités asynchrones.

Structure élémentaire d'une promesse en JavaScript :

Le code d'une promesse se fait en deux parties :

- La création de l'objet Promise et l'appel du callback resolve() si la promesse est tenue ou du callback reject() si elle est rompue.
- L'utilisation de la promesse, quelle que soit la situation de réussite ou d'échec. L'utilisation fera appel à la méthode then(callback) si resolve a été appelé, ou à la méthode catch(callback) si reject a été exécuté.

```
//création de l'objet
const promesse = new Promise((resolve,reject)=> {
    //Ici, code d'envoi d'une requête

    const ok = true; //simulation de réussite de la requête

    if(ok === true){
        resolve();  //appel du callback resolve
    }else {
```

```
            reject();    //appel du callback reject
        }
})

//utilisation de l'objet
promesse
        .then(() => {
            console.log('les données ont été récupérées')
        })
        .catch(() => {
            console.log('Pas de données')
        });
```

Récupérer des données au format json

Soit le fichier produits.json stockés dans le dossier public, situé sur la racine de l'application.

```
let div = document.querySelector('#data');
let ch="";
fetch('./data/produits.json')
```

➔ Création de la promesse et retour d'une réponse

```
  .then(data => data.json())
```

➔ .then : appel du callback **resolve** : data.json() convertit la réponse data en json

```
  .then((data) => {
        console.log(data);
        data.produit.forEach((elt)=>{
           console.log(elt.nom);

        })
        div.innerHTML= ch;
    }
  )
  .catch(() => {
     console.log("Echec")
  });
```

➔ .then : la promesse donnée par la conversion en json est également résolue et le traitement peut être appliqué aux données

➔ Le callback **reject** a été exécuté

5.5. Les hooks de react

Un hook est une fonction qui gère le cycle de vie des données internes d'un composant. Ces données représentent **l'état** (state) du composant. Le nom d'un hook commencent toujours par **use-** .
Dans les exemples ci-dessous, 2 hooks seront utilisés : useState() et useEffect().
Avant utilisation, les hooks devront importés dans le fichier du composant :

```
import React, {useEffect, useState} from "react";
```

5.5.1. Le hook useState()

Le hook useState définit l'état initial d'une variable; il est toujours exécuté dès le lancement du composant. Il retourne un tableau dont les éléments sont deux callbacks :

```
const [bien,setBien]= useState(null);
```

- Le premier élément, bien , est un getter
- Le second élément, setBien , est un setter
- Le paramètre null (dans le cas de données objets encore à venir) est l'initialisation de la donnée bien. Le useState peut être initialisé avec une chaîne vide ou le nombre 0 selon le types de données utilisées.

Lorsque les données sont récupérées par l'API fetch, le callback setBien est appelé et le callback bien reçoit les données.

5.5.2. Le hook useEffect()

Ce hook est **EXÉCUTÉ APRES L'EXÉCUTION DU COMPOSANT**. Il a pour objectif de gérer ce qu'on appelle les effets secondaires (side effects), c'est-à-dire tout ce qui résulte d'une action à l'intérieur du composant.

Structure de ce hook :

```
useEffect(()=> {
  //code à exécuter

},[]); // tableau des dépendances (données nécessaires)
```

Le hook useEffect est donc une fonction prenant deux paramètres : un **callback** et un **tableau des dépendances** qu'une

Le tableau des dépendances est essentiel puisqu'il dit au hook ce qu'il doit écouter. Supposons qu'on ait une propriété "bien" contenant des objets "biens", on pourra demander au hook d'*écouter* toutes les interactions qui auront lieu sur la propriété :

```
useEffect(()=> {
   //code à exécuter

},[bien]); // surveiller et réagir à toute modification sur l'objet bien
```

Un exemple pratique

```
import React, { useEffect, useState } from "react";

export default function ExempleHooks() {
   const [variable1, setVariable1] = useState('1');

   useEffect(() => {
      console.log("valeur de variable1 dans useEffect : ", variable1);
   }, [variable1]);

   const handleClick = () => {
      setVariable1('2'); // Changer la valeur de variable1 après le rendu initial
   };

   return (
      <div>
         <p>Dans return : {variable1}</p>
         <button onClick={handleClick}>Changer la valeur</button>
      </div>
   );
}
```

Au départ s'affiche la valeur 1 (celle du return) ainsi que le bouton permettant de modifier la valeur de la variable. Ensuite, la fonction handleClick s'exécute : elle appelle le setter setVariable1 pour mettre la variable à jour et le useEffect(), prévu pour écouter tout changement sur cette variable, s'exécute aussi pour afficher son message.

5.5.3. Application → Avec un fichier biens.json

- Stocker le fichier ci-dessous dans le dossier public
- Créer un lien de menu vers une page composant Biens.js
- Créer le composant Biens.js


```
{
  "biens": [
    {
      "type": "Maison",
      "chambres": 3,
      "situation": "Mons",
      "prix": 250000,
      "chauffage": {
        "energie": "gaz",
        "peb": "C"
      }
    },
    {
      "type": "Maison",
      "chambres": 4,
      "situation": "Mons",
      "prix": 290000,
      "chauffage": {
        "energie": "gaz",
        "peb": "D"
      }
    },
    {
      "type": "Maison",
      "chambres": 3,
      "situation": "Tournai",
      "prix": 350000,
      "chauffage": {
        "energie": "Mazout",
        "peb": "F"
      }
    },
    {
      "type": "Appartement",
      "chambres": 3,
      "situation": "Mons",
      "prix": 1950000,
      "chauffage": {
        "energie": "gaz",
        "peb": "G"
      }
    },
    {
      "type": "Maison",
      "chambres": 2,
      "situation": "Tournai",
      "prix": 210000,
      "chauffage": {
        "energie": "gaz",
        "peb": "G"
      }
    },
```

```
        {
          "type": "Maison",
          "chambres": 2,
          "situation": "Mons",
          "prix": 205000,
          "chauffage": {
            "energie": "mazout",
            "peb": "E"
          }
        },
        {
          "type": "Appartement",
          "chambres": 2,
          "situation": "Mons",
          "prix": 165000,
          "chauffage": {
            "energie": "mazout",
            "peb": "E"
          }
        },
        {
          "type": "Maison",
          "chambres": 2,
          "situation": "Mons",
          "prix": 205000,
          "chauffage": {
            "energie": "mazout",
            "peb": "E"
          }
        }
      ]
    }
```

Ce composant sera chargé de récupérer (de manière asynchrone) les biens avec l'API fetch et mettre les données à disposition du composant. Ces données seront gérées par les **hooks** useState() et useEffect().

Important : le fichier de données est stocké dans le dossier public.

Composants/Biens.js

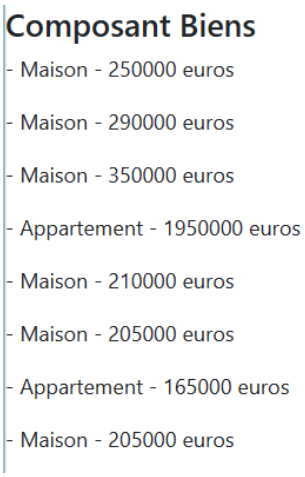
```
import React, {useEffect, useState} from "react";
import ListeBiens from "./ListeBiens";

export default function Biens() {
  const [bien,setBien]= useState(null);

  useEffect(()=>{
    //récupération des biens
    fetch('./biens.json')
      .then((data) => data.json())
      .then(data => {
        setBien(data)
```

```
        })
        .catch(error=>{
           console.log("problème ",error)
        })
  },[])

  useEffect(()=>{
     console.log("Tableau : ",bien)
  },[bien])


return (
    <div>
        <div><h4>Composant Biens</h4></div>
        <div>
            {console.log("Les biens sous l'appel : ", bien)}
            {
                bien ? ( bien.biens.map((elt, index) => (
                    <p key={index}> - {elt.type} - {elt.prix} euros</p> ))
                ) : ( <p>Chargement en cours...</p> )
            }
        </div>
    </div>
  );
}
```

Une explication

- bien ? → si le tableau des biens contient des données
- (bien.biens.map((… → boucle d'affichage (map requiert une clé pour chaque élément utilisé)
- (Chargement en cours...) → car les données arrivent de manière asynchrone

Le composant doit afficher :



Un composant peut faire appel à un composant en lui envoyant des paramètres : les *props*.

5.5.4. Les props

Il s'agit de **prop**riétés envoyées **du composant parent vers le composant enfant** (jamais dans l'autre sens). Le composant enfant est importé dans le composant parent, puis appelé avec communication d'une ou plusieurs variables.

Exemple avec le fichier json emplois.js, à stocker dans le dossier public

Le composant parent va récupérer les données du fichier, les communiquer par props au composant enfant. Ce dernier les affichera et donnera le détail des données de la ligne qui aura été cliquée.
Le composant parent délègue ainsi la gestion de l'affichage et ses suites à un composant enfant.

Le fichier emplois.js


```
{
 "emplois": [
  {
   "id": 1,
   "entreprise": "Entreprise 1",
   "localisation": "Courtrai",
   "fonction": "Développeur fullstack JavaScript",
   "profil": "Maîtrise des langages web HTML/CSS/JavaScript, la connaissance d'une librairie est un atout"
  },
  {
   "id": 2,
   "entreprise": "Entreprise 2",
   "localisation": "Mons",
   "fonction": "Développeur backend Node.js",
   "profil": "Minimum 3 ans d'expérience dans le développement"
  },
  {
   "id": 3,
   "entreprise": "Entreprise 3",
   "localisation": "Bruxelles",
   "fonction": "Développeur junior frontend",
   "profil": "Débutants - expérimenté"
  },
  {
   "id": 4,
   "entreprise": "Entreprise 4",
   "localisation": "Mons",
   "fonction": "Développeur Java",
   "profil": "Minimum bachelier"
  },
  {
   "id": 3,
```

```
    "entreprise": "Entreprise 5",
    "localisation": "Bruxelles",
    "fonction": "Développeur C#",
    "profil": "bachelier - Master"
  }
 ]
}
```

Le composant enfant :

Composants/ListeEmplois.js

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

export default function Emploi({ data }) {
  const [emploiSelectionne, setEmploiSelectionne] = useState(null);

  const afficherDetails = (index) => {
    setEmploiSelectionne(index);
  };

  return (
    <div>
      <ul>
        {data.emplois.map((elt, index) => (
          <li key={index}>
            <a onClick={() => afficherDetails(index)}>{elt.fonction}</a><br />
          </li>
        ))}
      </ul>
      <div id="detail">
        {emploiSelectionne !== null && (
          <div>
            <p className="txtGras">{data.emplois[emploiSelectionne].localisation}</p>
            <p> --> {data.emplois[emploiSelectionne].profil}<br/>
            {data.emplois[emploiSelectionne].entreprise}<br />
            {data.emplois[emploiSelectionne].localisation}<br /></p>
          </div>
        )}
      </div>
    </div>
  );
}
```

Une explication

- La fonction onClick a pour valeur une fonction fléchée dont le code consiste à appeler une fonction afficherDetails en lui communiquant l'indice de l'élément courant.

- La fonction afficherDetails appelle le setter setEmploiSelectionne pour mettre à jour le getter emploiSelectionne
- La detail vérifie que emploiSelectionne contient bien une valeur d'indice. Si c'est le cas, elle affiche diverses propriétés de l'objet sélectionné.

N.B. l'opérateur ternaire est également valable :

```
<div id="detail">
  {emploiSelectionne !== null ? (
    <div>
      <p className="txtGras"> {data.emplois[emploiSelectionne].localisation}
</p>
      <p>
        {' --> ' + data.emplois[emploiSelectionne].profil}<br />
        {data.emplois[emploiSelectionne].entreprise}<br />
        {data.emplois[emploiSelectionne].localisation}<br />
      </p>
    </div>
  ) : null}
</div>
```

Le composant parent :

```
import React, {useEffect, useState} from "react";
import ListeEmplois from "./ListeEmplois";

export default function Emplois() {
  const [emplois, setEmplois] = useState(null);

  useEffect(()=> { //récupération des données
    fetch('./emplois.json')
      .then(response => response.json()) // Retourner response.json()
      .then(data => { //si la conversion en js s'est bien passée
        setEmplois(data); // Mettre à jour l'état avec les données converties
        console.log('then = null car asynchrone ',emplois)
      })
      .catch(error => {
        console.error("Impossible de récupérer les données ", error);
      });
  },[]);

  useEffect(() => {
    console.log('Emplois mis à jour : ', emplois);
  }, [emplois]); //écoute la dépendance emplois et réagit à toute interaction sur cette variable

  return (

    <div>
```

```
        <div><h4>Composant Emplois</h4></div>
        {console.log("les emplois : ",emplois)}
        {emplois ? (<ListeEmplois data={emplois}/>) : (null)}
      </div>
    )
}
```

Une explication

{emplois ? (<ListeEmplois data={emplois}/>) : (null)}

- emplois ? → si emplois est pourvu de données
- (<ListeEmplois data={emplois}/>) → appel du composant enfant. Le props est la variable data dont la valeur est le tableau des emplois

N.B. Les parenthèses améliorent la lisibilité du code.

Dans le navigateur :

Composant Emplois

- Développeur fullstack JavaScript
- Développeur backend Node.js
- Développeur junior frontend
- Développeur Java
- Développeur C#

Clic sur le premier lien :

Composant Emplois

- Développeur fullstack JavaScript
- Développeur backend Node.js
- Développeur junior frontend
- Développeur Java
- Développeur C#

Courtrai

--> Maîtrise des langages web HTML/CSS/JavaScript, la connaissance d'une librairie est un atout
Entreprise 1
Courtrai