

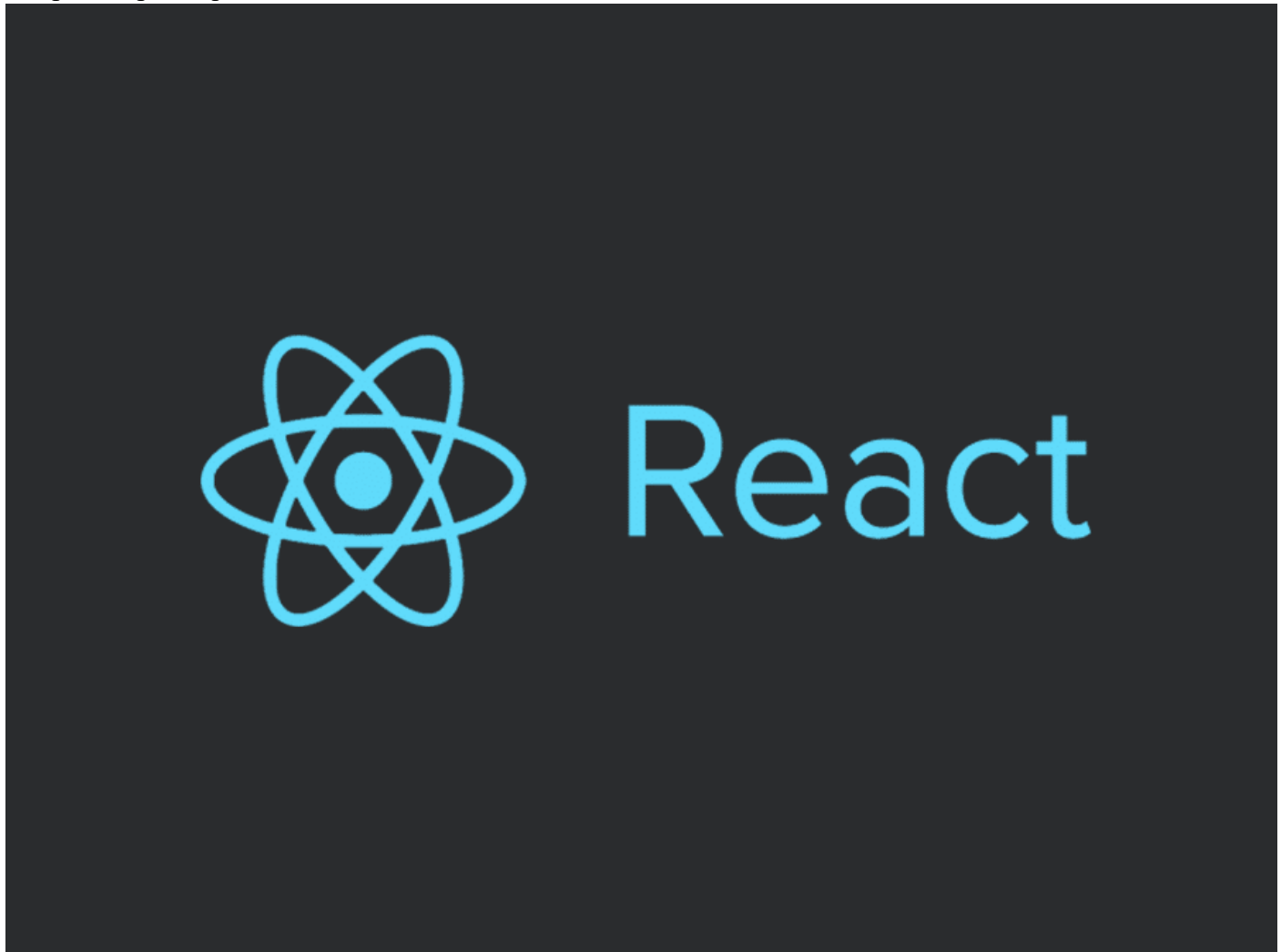
React.JS

I. Présentation de React

1. *Définition de React*

React est un framework JavaScript open-source développé par Facebook. Il est largement utilisé pour la création d'interfaces utilisateur interactives et dynamiques dans le développement web. Contrairement à un langage de programmation complet, React se concentre sur la construction de composants réutilisables pour construire des interfaces utilisateur modulaires. Ces composants sont des blocs de construction autonomes qui encapsulent le code HTML, CSS et JavaScript nécessaire pour afficher

une partie spécifique de l'interface utilisateur.



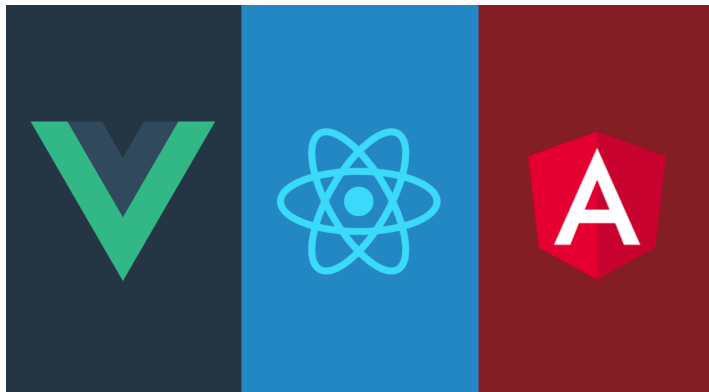
Grâce à son virtual DOM (Document Object Model), React permet de mettre à jour efficacement et rapidement les éléments de l'interface utilisateur en fonction des modifications de l'état de l'application.

2. **Rappel sur les framework et de son utilité pour les développeurs**

Un framework est un ensemble de bibliothèques, d'outils et de conventions de développement qui simplifient le processus de création d'applications. Il fournit une structure et des fonctionnalités prédéfinies, ce qui permet aux développeurs de se concentrer davantage sur la logique métier plutôt que de passer du



temps à résoudre des problèmes techniques courants.



En utilisant un framework tel que React, les développeurs peuvent bénéficier de nombreux avantages tels que la réutilisation de code, l'organisation et la modularité du code, ainsi que la collaboration facilitée avec d'autres développeurs.

Les frameworks permettent également de suivre des conventions établies, ce qui facilite la maintenance et la compréhension du code par l'équipe de développement.

3. *React Vs vue.js vs Angular*

React, **Vue.js** et **Angular** sont tous trois des frameworks ou des bibliothèques JavaScript populaires utilisés pour créer des applications web modernes. Ils ont chacun leurs forces et leurs particularités, voici une comparaison simple entre eux :

React (Bibliothèque JavaScript développée par Facebook)

- Favorise un modèle de programmation plus flexible et moins prescriptif.
- L'accent est mis sur la création de composants réutilisables.
- La bibliothèque elle-même est plus petite et plus légère que Angular.
- Utilisé par Facebook, Instagram, Airbnb, et d'autres.

Vue.js (Framework JavaScript développé par Evan You, ancien employé de Google)

- Syntaxe simple et facile à comprendre, excellent pour les débutants.
- Se situe entre React et Angular en termes de flexibilité et de convention.
- Permet une intégration facile dans les projets existants.
- Utilisé par Alibaba, Xiaomi, et d'autres.

Angular (Framework JavaScript développé par Google)

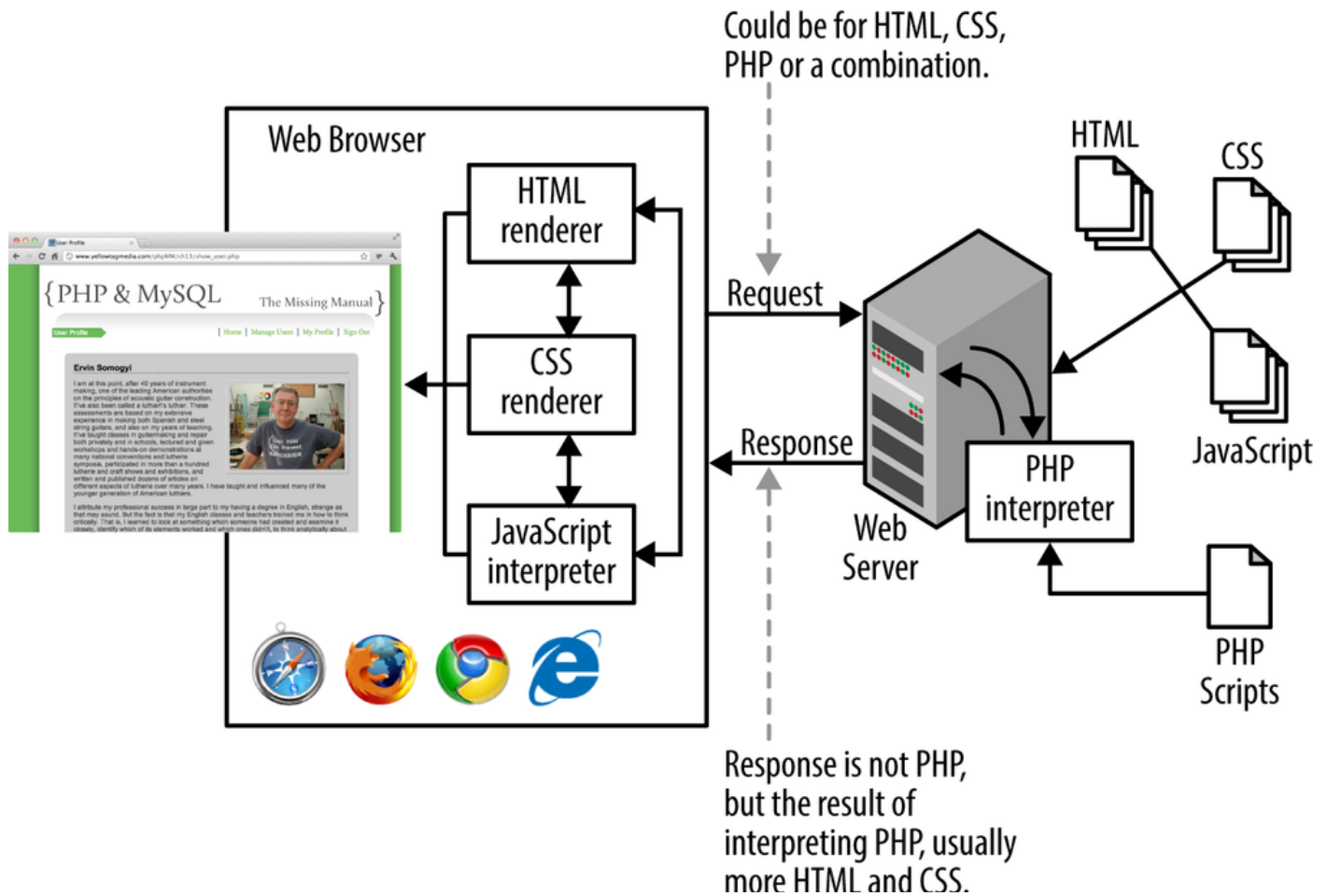
- Comprend un ensemble d'outils plus complet "out of the box" (gestion des formulaires, gestion de l'état, etc.).
- Suit une structure et des conventions strictes, ce qui peut faciliter la maintenance à grande échelle.
- Utilise le TypeScript pour le développement, ce qui peut améliorer la qualité du code et le débogage.
- Utilisé par Google, Microsoft, IBM, et d'autres.

4. *Site et application WEB*

La tendance actuelle consiste à séparer la partie cliente et la partie serveur d'un site web.

La partie cliente comprend les fichiers HTML, CSS et JavaScript interprétés par le navigateur, tandis que la partie serveur utilise des langages tels que PHP ou Java pour effectuer des requêtes aux bases de données.

Traditionnellement, un site web envoie des pages HTML au navigateur du client à chaque demande, ce qui nécessite des requêtes fréquentes au serveur lors de la navigation ou du remplissage de formulaires.



Cependant, grâce à JavaScript, il est possible de créer des applications web qui fonctionnent de manière autonome une fois que le serveur a envoyé une seule page contenant suffisamment de JavaScript. Cela permet une navigation plus fluide et plus rapide, sans avoir à recharger la page entière à chaque interaction. Cette approche, appelée architecture ESPA (Single Page Applications), présente de nombreux avantages, tels qu'une meilleure réactivité et une expérience utilisateur plus interactive.



II. Présentation de react

5. Pourquoi react ?

Il y a plusieurs raisons convaincantes de choisir React pour le développement de vos applications :

- 1. Composants réutilisables** : React est basé sur le concept de composants, ce qui permet de créer des éléments d'interface réutilisables. Vous pouvez développer des composants indépendants qui encapsulent leur propre logique et leur propre style, ce qui facilite la maintenance, la réutilisation et la modularité du code.
- 2. Virtual DOM** : React utilise un DOM virtuel, une représentation légère de la structure de l'interface utilisateur, qui lui permet de minimiser les manipulations directes du DOM réel. Cela améliore considérablement les performances en évitant les mises à jour coûteuses et en ne mettant à jour que les parties de l'interface qui ont besoin de l'être, ce qui rend les applications plus rapides et réactives.
- 3. Écosystème et communauté active** : React bénéficie d'une large adoption et d'une communauté de développeurs très active. Cela signifie qu'il existe une multitude de ressources, de bibliothèques tierces, de composants préfabriqués et d'outils disponibles pour faciliter le développement. Vous pouvez trouver de l'aide, des tutoriels, des exemples de code et des réponses à vos questions facilement sur Internet.
- 4. Réactivité** : Grâce à son architecture basée sur les composants et à la gestion efficace du DOM virtuel, React offre une expérience utilisateur réactive et fluide. Les mises à jour instantanées de l'interface utilisateur permettent de créer des applications interactives et dynamiques sans que les utilisateurs aient à recharger la page.
- 5. Support à long terme** : React est soutenu par Facebook, ce qui lui confère une stabilité et une pérennité à long terme. Facebook investit continuellement dans le développement et l'amélioration de React, ce qui garantit une évolution constante du framework et une compatibilité avec les versions futures.
- 6. Facilité d'apprentissage** : Bien que React puisse sembler complexe au premier abord, il dispose d'une courbe d'apprentissage relativement douce, en particulier si vous avez des connaissances préalables en HTML, CSS et JavaScript. De plus, la documentation officielle de React est très complète et bien organisée, ce qui facilite l'apprentissage et la prise en main du framework.

Ainsi choisir React vous permet de bénéficier de composants réutilisables, de performances optimisées grâce au DOM virtuel, d'un écosystème riche et d'une communauté active, de réactivité, d'un support à long terme et d'une courbe d'apprentissage accessible. Ces avantages font de React un choix populaire et solide pour le développement d'applications Front End.

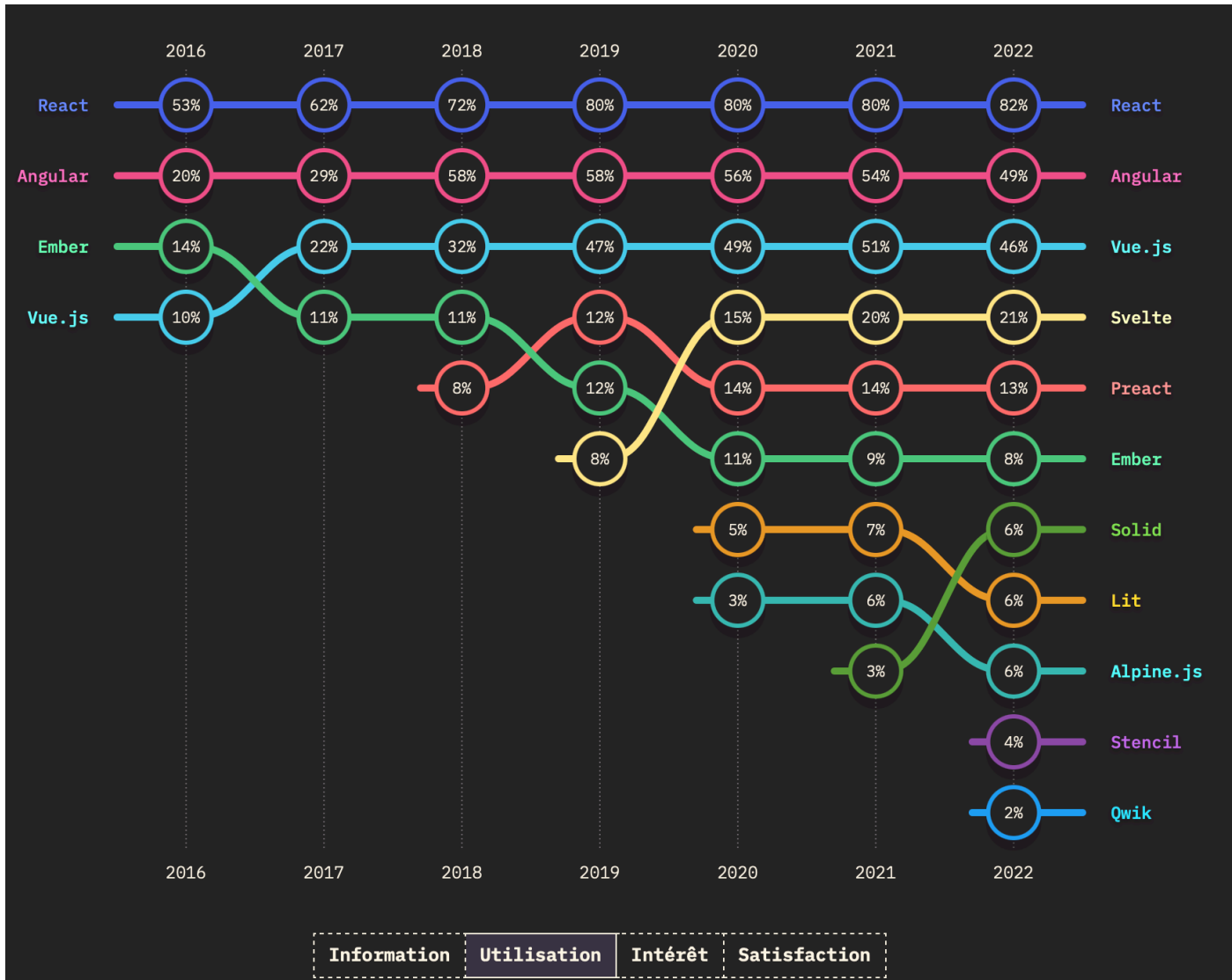


Figure 1 : utilisation des frameworks

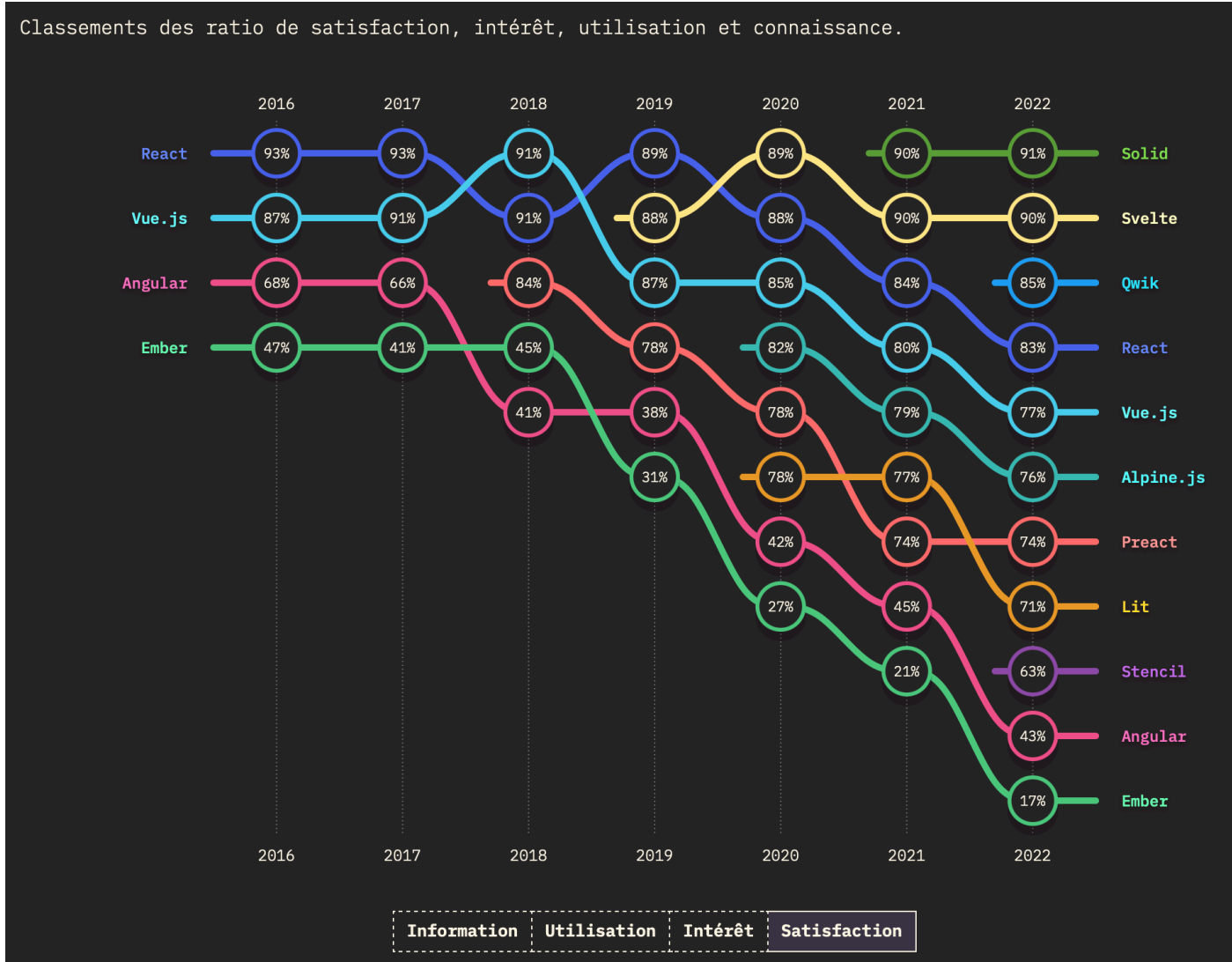
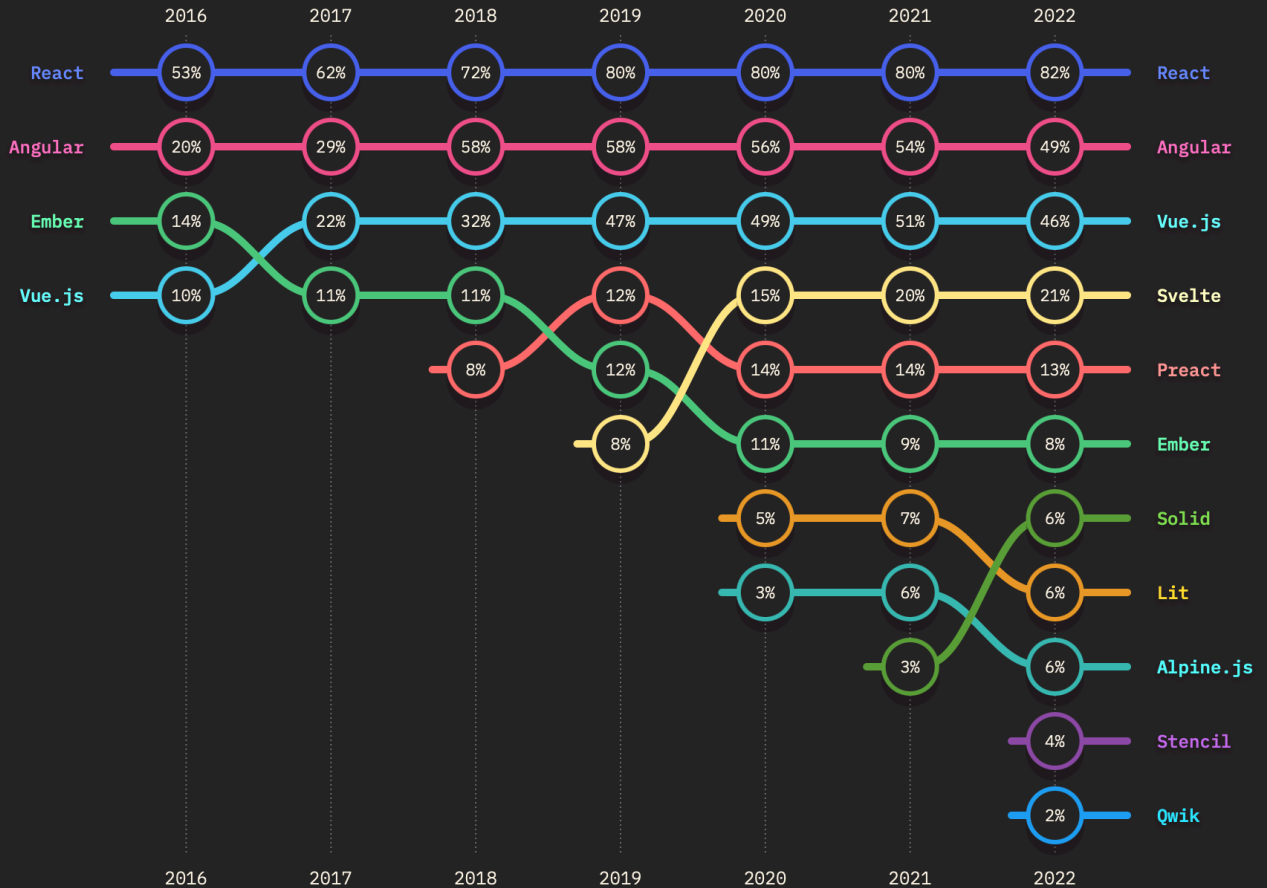


Figure 2 : satisfaction

Classements des ratio de satisfaction, intérêt, utilisation et connaissance.



Information Utilisation Intérêt Satisfaction

1. React est orienté composant

React est un framework orienté composants qui facilite le développement d'applications. L'approche consiste à créer de nombreux petits composants indépendants qui, une fois assemblés, forment une application complète.

Chaque composant représente une partie autonome de l'application, avec sa propre structure de code HTML, des règles de style CSS spécifiques et une fonction JavaScript pour implémenter un comportement particulier. Il est important de noter que les composants React s'appuient sur le standard des Web Components, qui découpe une page web en fonction de ses différentes fonctionnalités, telles que la barre de navigation, les boîtes de dialogue ou le contenu principal.

Bien que ce standard ne soit pas encore pris en charge par tous les navigateurs, il pourrait le devenir à l'avenir. React n'est pas le seul projet à s'intéresser à cette nouvelle norme, mais il est l'un des premiers à envisager sérieusement son intégration.

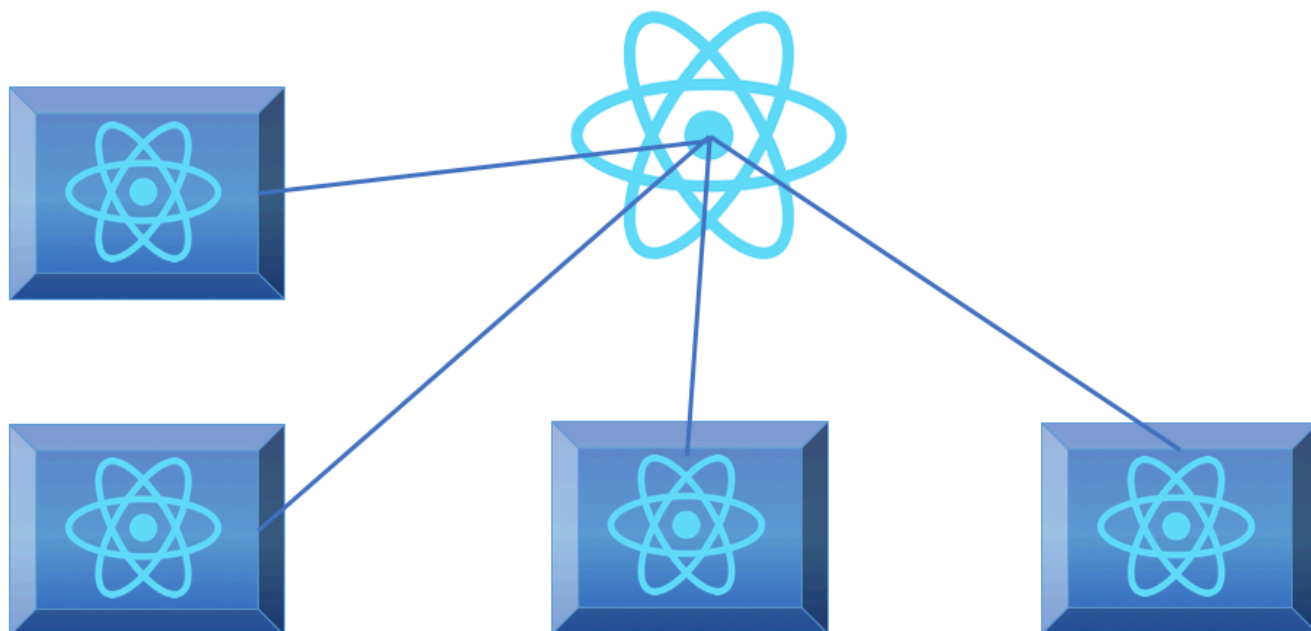


Figure 3 : React et ses composants

2. **ECMAScript 6 ?**

ECMAScript 6, également connu sous le nom ES6 ou ES2015, est une version majeure du langage de programmation JavaScript. Il a été publié en juin 2015 et apporte de nombreuses fonctionnalités et améliorations significatives par rapport aux versions précédentes.



Une des principales caractéristiques d'ES6 est l'introduction de nouvelles syntaxes et constructions qui permettent de développer du code JavaScript de manière plus concise et expressive.

Parmi ces fonctionnalités, on trouve les classes, qui permettent de créer des objets avec une syntaxe similaire à la programmation orientée objet (POO), les fonctions fléchées, qui offrent une syntaxe plus concise pour les fonctions anonymes, et les modules, qui facilitent la gestion des dépendances et l'organisation du code. ES6 introduit également de nouvelles fonctionnalités pour améliorer la manipulation des tableaux, comme les méthodes map, filter et reduce, qui permettent de travailler sur les éléments d'un tableau de manière plus efficace et expressive. Les promesses sont une autre nouveauté importante d'ES6, offrant une manière plus propre et plus simple de gérer les opérations asynchrones.

En plus de ces nouvelles fonctionnalités, ES6 propose également des améliorations au niveau de la gestion des variables avec l'introduction des mots-clés let et const, qui offrent un meilleur contrôle de la portée des variables, ainsi que la déstructuration d'objets et de tableaux, qui permet de simplifier l'accès aux valeurs des structures de données.

3. **TypeScript**

TypeScript est un langage de programmation open source développé par Microsoft. Il est conçu pour ajouter des fonctionnalités de typage statique optionnelles à JavaScript, ce qui permet de détecter et de prévenir les erreurs de typage avant l'exécution du code. TypeScript étend les fonctionnalités d'ES6 en ajoutant la possibilité de déclarer des types de variables, des interfaces, des classes, des modules, etc. TypeScript est

	Web Avance	
	REACT.JS	

ensuite compilé en JavaScript standard pour être exécuté dans un navigateur ou un environnement JavaScript.

4. **TypeScript et React ?**

React.js est une bibliothèque JavaScript populaire pour la construction d'interfaces utilisateur interactives. Il utilise le langage JavaScript pour définir la structure et la logique des composants. Cependant, comme JavaScript est un langage à typage dynamique, il peut être difficile de détecter les erreurs de typage avant l'exécution.

C'est là que TypeScript entre en jeu. TypeScript permet d'ajouter des fonctionnalités de typage statique à JavaScript, ce qui facilite la détection précoce des erreurs de typage dans le code React.js. En utilisant TypeScript avec React.js, les développeurs peuvent bénéficier d'une meilleure vérification des types, d'une meilleure documentation du code, d'une meilleure autocomplétion et d'une refactoring plus sûr.



De plus, TypeScript offre une prise en charge native pour les fonctionnalités avancées d'ES6 et des versions ultérieures, ce qui permet aux développeurs React.js d'utiliser des fonctionnalités telles que les modules, les classes, les interfaces et bien d'autres.

III. Premier pas

1. **Installer l'environnement**

a) **Visual Studio**

A installer si ce n'est déjà fait.

b) **Démarrage dans VS**

Il existe plusieurs méthodes pour démarrer un projet React, certaines étant plus rapides que d'autres.

Par exemple, le projet Create React App permet de configurer rapidement un nouveau projet React en une seule ligne de commande.

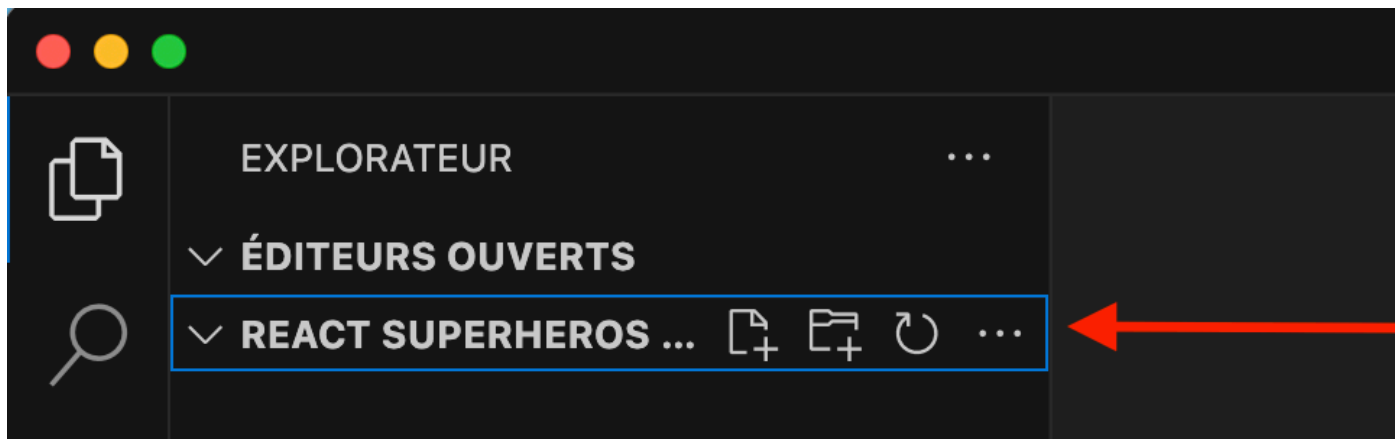
Cependant, il est préférable de commencer avec un dossier vide afin de comprendre le processus complet de création d'une nouvelle application React à partir de zéro.

Pour démarrer, il est nécessaire d'ajouter certains fichiers de configuration dans le dossier principal du projet, dont le rôle sera détaillé ultérieurement. Un nouveau dossier vide doit être créé pour servir de racine au projet. Il est important de noter que ce qui sera développé constitue la base de l'ensemble du cours.

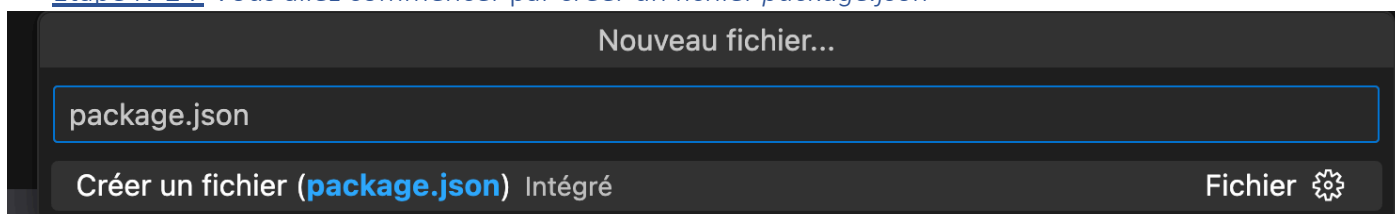
En d'autres termes, le projet débute dès maintenant, en abordant les fondamentaux. Il convient de créer un dossier appelé "React SuperHeros App" et de l'ouvrir dans un éditeur de texte. Le choix de ce nom est dû au fait que tout au long du cours, une application de gestion de Super Héros sera développée. À partir du dossier vide, l'objectif est d'aboutir à une application finale prête pour la production, offrant une vision globale du processus de développement d'une application React. Bien entendu, il est possible de donner un autre nom au dossier, car cela n'a pas d'importance pour la suite.

2. TD démarrer son projet React

Étape N°1 : Lancez Visual Code et créez dans un répertoire *Reac SuperHeros App* et ouvrez-le dans votre VS



Étape N°2 : Vous allez commencer par créer un fichier *package.json*



Copiez/coller ce code

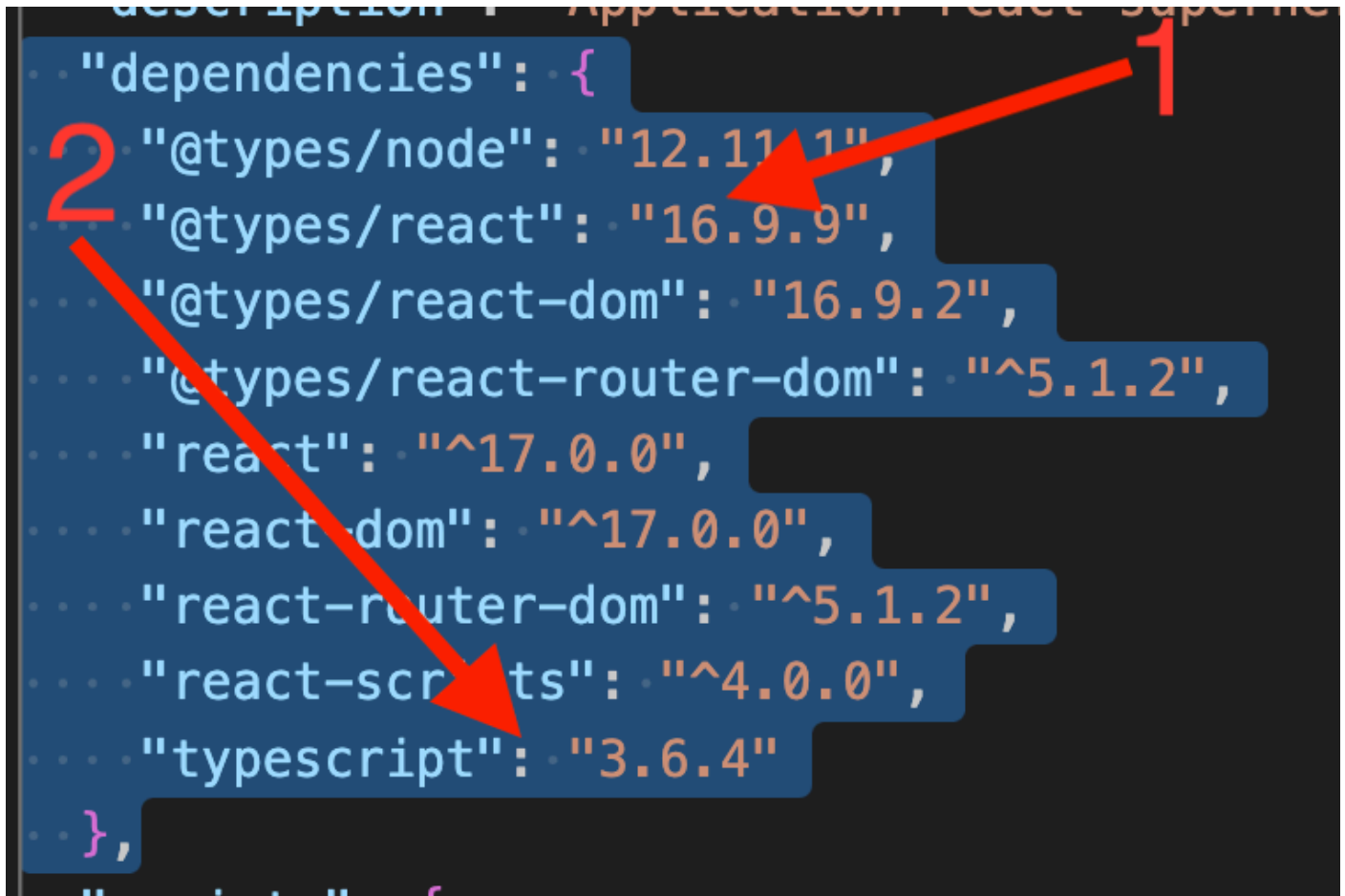
```
{
  "name": "superheros-app",
  "version": "1.0.0",
  "description": "Une application de superhéros",
```

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
"dependencies": {
  "react": "^17.0.2",
  "react-dom": "^17.0.2"
},
"devDependencies": {
  "react-scripts": "^5.0.0"
},
"browserslist": {
  "production": [
    ">0.2%",
    "not dead",
    "not op_mini all"
  ],
  "development": [
    "last 1 chrome version",
    "last 1 firefox version",
    "last 1 safari version"
  ]
}
```

a) Remarques dépendances

Il y a une liste de dépendance à partir de

```
"dependencies": {
```



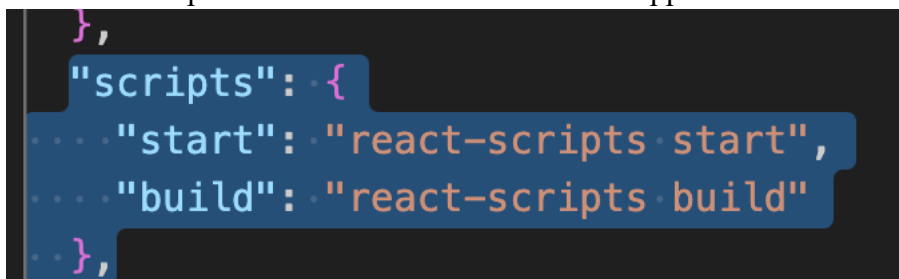
```

    "dependencies": {
      "@types/node": "12.11.1",
      "@types/react": "16.9.9",
      "@types/react-dom": "16.9.2",
      "@types/react-router-dom": "^5.1.2",
      "react": "^17.0.0",
      "react-dom": "^17.0.0",
      "react-router-dom": "^5.1.2",
      "react-scripts": "^4.0.0",
      "typescript": "3.6.4"
    },
  
```

1. La version de React
2. Celle de TypeScript (pas présente dans la nouvelle version)

b) Remarques scripts

- Le script *start* : Met en place un environnement de développement pour tester l'application React dans un navigateur.
- Le script *build* : Crée le livrable final de l'application une fois le développement terminé.



```

    "scripts": {
      "start": "react-scripts start",
      "build": "react-scripts build"
    },
  
```

c) Remarques fin du fichier

- *eslintConfig* : Configuration pour ESLint, un outil d'analyse statique du code JavaScript.
- *browserlist* : Liste des navigateurs pris en charge pour la compatibilité du code.

```
"eslintConfig": {
  "extends": "react-app",
},
"browserslist": {
  "production": [
    ">0.2%",
    "not dead",
    "not op_mini all"
  ],
}
```

Étape N°3 : Créez un nouveau fichier appelé *tsconfig.json* et y mettez ce code :

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "esnext",
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "jsx": "react",
    "sourceMap": true,
    "esModuleInterop": true,
    "strict": true,
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "noImplicitAny": false,
    "noImplicitReturns": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true
  },
  "include": ["src"],
  "exclude": ["node_modules", "build", "dist", "public"]
}
```

d) Que fait tsconfig.json ?

Le fichier **tsconfig.json** est utilisé pour configurer le compilateur TypeScript pour un projet TypeScript. Il spécifie les options du compilateur à utiliser lors de la compilation du code TypeScript.

e) la version de javascript

Cette ligne indique que notre code va être « compilé » en javascript ES6 :

```
3 "target": "es6",
```

ES6, également connu sous le nom de ECMAScript 2015, est la sixième édition de la spécification ECMAScript, qui est la spécification standard sur laquelle JavaScript est basé.

f) option de « compilation »

```
20      "jsx": "react",
```

Dans un fichier **tsconfig.json** de TypeScript, l'option **"jsx": "react"** indique au compilateur TypeScript de transformer chaque balise JSX en un appel à la fonction **React.createElement**.

Le JSX est une syntaxe étendue de JavaScript utilisée par des bibliothèques comme React pour définir des structures de composants similaires au HTML. Cependant, cette syntaxe n'est pas native de JavaScript ou TypeScript, c'est pourquoi nous avons besoin de cette option pour indiquer à TypeScript comment la traiter.

Pour terminer, nos fichiers écrits avec react devront avoir comme type TSX qui est un mélange de TypeScript et JSX.

g) Où se trouve les fichiers TSX ?

L'option *include* indique le répertoire dans lequel se trouve ces fichiers : uniquement dans le répertoire *src*

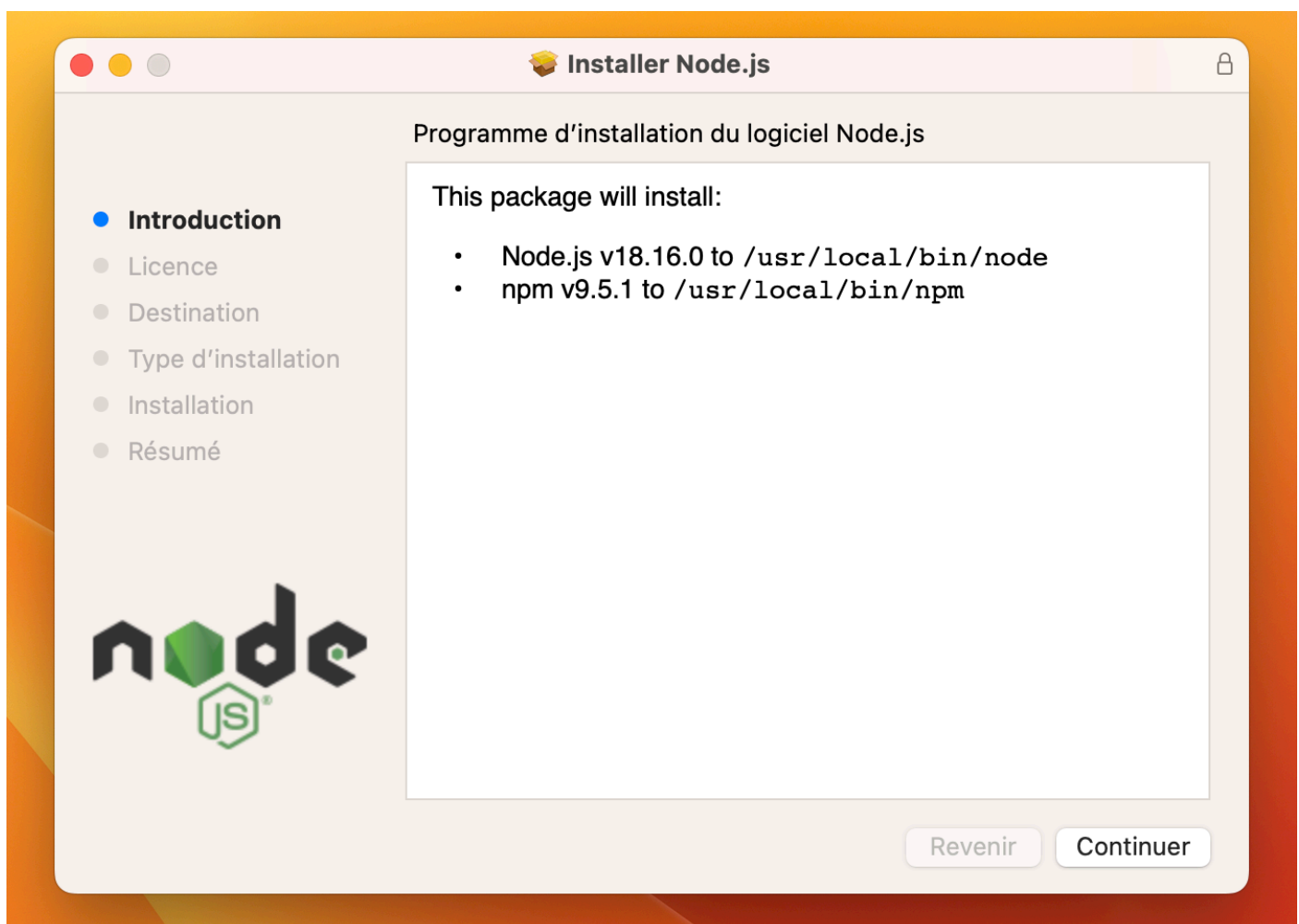
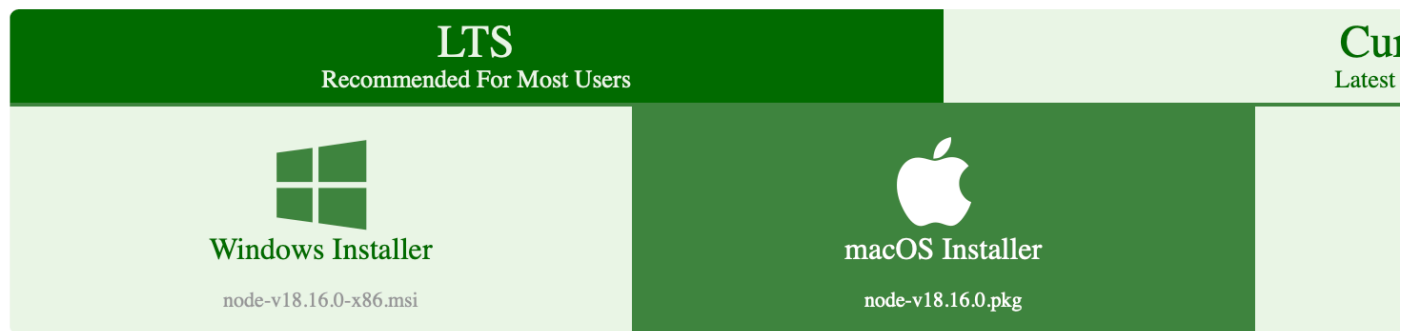
```
23      "include": [
24          |   "src"
25      ]
```

3. installer node.js

Afin de mettre en place les dépendances requises pour notre projet React, indiquées dans le fichier **package.json**, nous nous appuierons sur un utilitaire très utile, appelé NPM (Node Package Manager). Pour y parvenir, l'installation de NodeJS est un prérequis. Bien que NodeJS, qui exécute du code JavaScript côté serveur, ne soit pas utilisé directement dans le cadre de cette formation, son installation est nécessaire pour la mise en place de NPM. Cela nous donnera la possibilité d'exécuter des commandes "npm ..." sur notre système.

Étape N°4 : Allez sur <https://nodejs.org/en/download> puis installez node.js

Pour MAC :



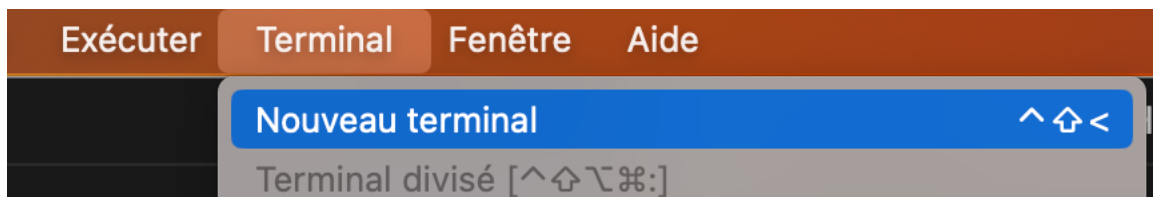
4. Installer les dépendances

a) Explications

C'est une étape essentielle pour configurer un projet React : la commande **npm install** est utilisée pour installer toutes les dépendances nécessaires pour un projet React, ou plus généralement pour un projet JavaScript. Ces dépendances sont définies dans un fichier **package.json** à la racine du projet.

Quand vous exécutez **npm install** dans votre terminal ou invite de commande, npm (Node Package Manager) cherche dans le fichier **package.json** et télécharge toutes les dépendances spécifiées dans la section "**dependencies**" ainsi que dans la section "**devDependencies**" (qui contient généralement des outils nécessaires pendant le développement mais pas lors de l'exécution du code) du fichier. Ces dépendances sont téléchargées dans un dossier **node_modules** à la racine de votre projet.

Étape N°5 : Dans VS exécutez la commande `npm install`



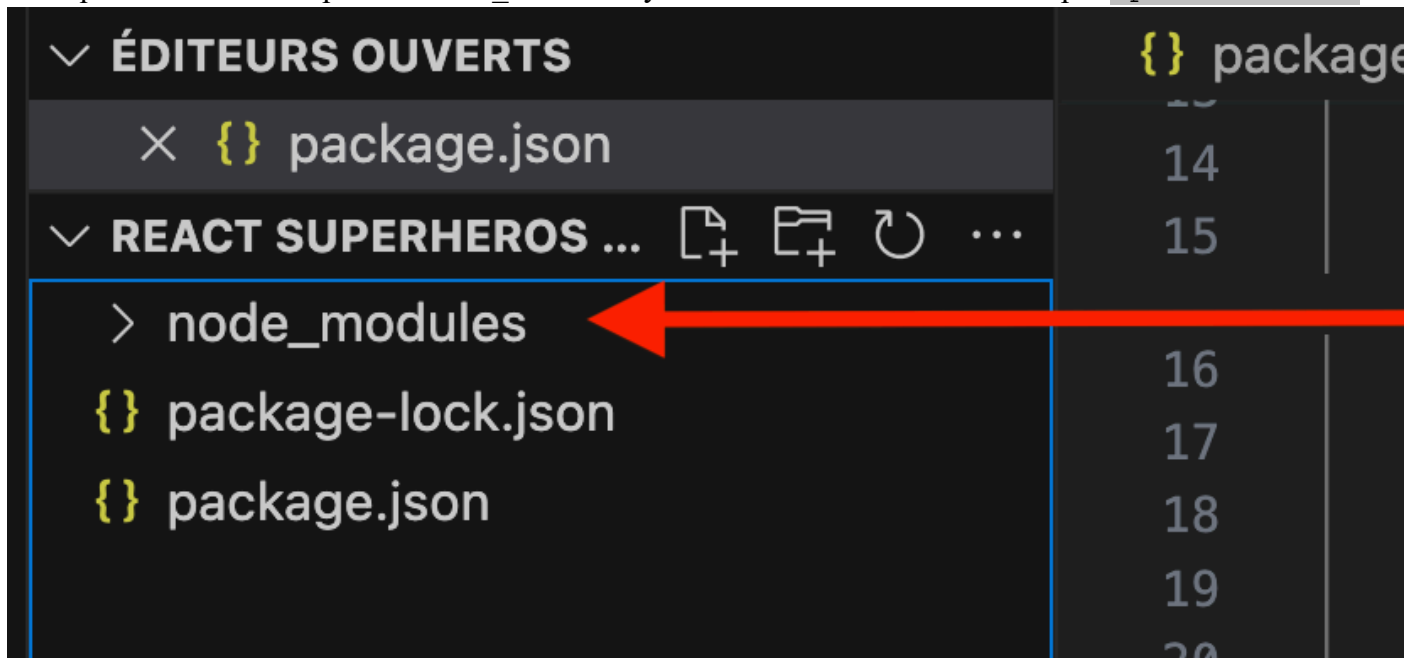
Puis

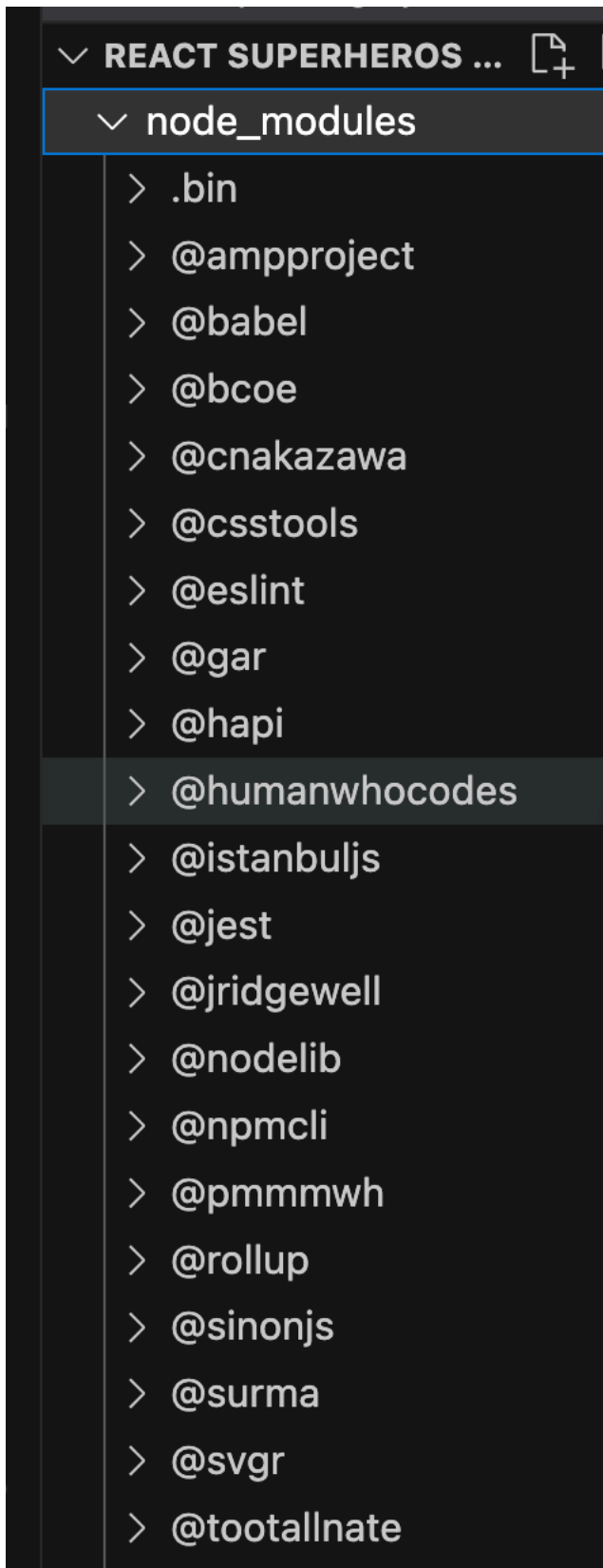
```
roberttomczak@macpro-rtomczak React SuperHeros App % npm install
```

Cela peut prendre plusieurs minutes.

```
roberttomczak@macpro-rtomczak React SuperHeros App % npm install
(#####) :: idealTree:terser-webpack-plugin: sill fetch manifest ms@2.1.2
```

Vous pouvez ouvrir le répertoire **node_modules** et y voir tous les modules installés par `npm install` :





b) Remarques sur les Erreurs ou warning

```
up to date, audited 1930 packages in 6s

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

31 vulnerabilities (1 low, 1 moderate, 20 high, 9 critical)

To address issues that do not require attention, run:
  npm audit fix

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

Et si vous exécutez un `npm audit`

Cette commande affiche une liste détaillée des vulnérabilités, ainsi que des conseils sur la manière de les résoudre.

Il y en a de deux types

1. **Packages looking for funding:** Certains packages open-source cherchent du financement pour aider à leur développement. En exécutant **npm fund**, vous verrez plus de détails sur comment vous pouvez soutenir ces projets.
2. **Vulnerabilities:** Cela signifie que certains des paquets que vous avez installés ont des vulnérabilités connues. Elles sont classées en plusieurs niveaux de gravité : faible, modérée, élevée et critique. C'est une pratique courante de résoudre ces vulnérabilités, surtout celles de haute et critique gravité.

Il est important de noter que la résolution des vulnérabilités doit être faite avec prudence. La mise à jour des paquets peut parfois entraîner des problèmes de compatibilité avec d'autres parties de votre code. Si vous utilisez **npm audit fix --force**, assurez-vous de tester votre application soigneusement après.

Étape N°6 : Exécutez `npm audit`

Étape N°7 : Exécutez `npm audit fix --force` puis relancez `npm audit` pour vérifier

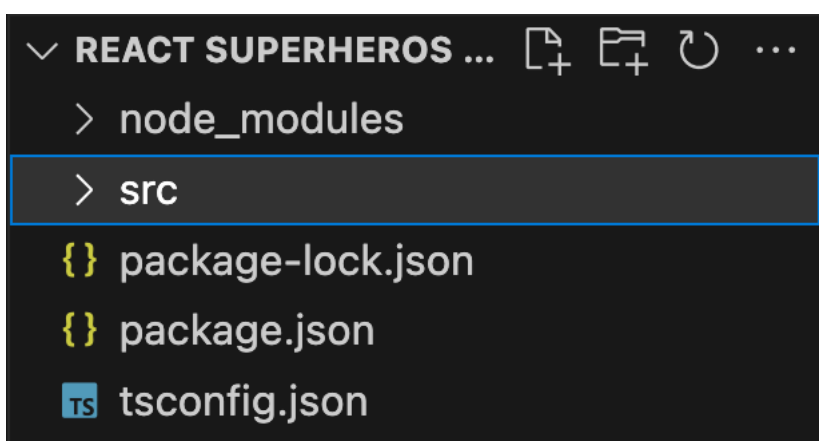
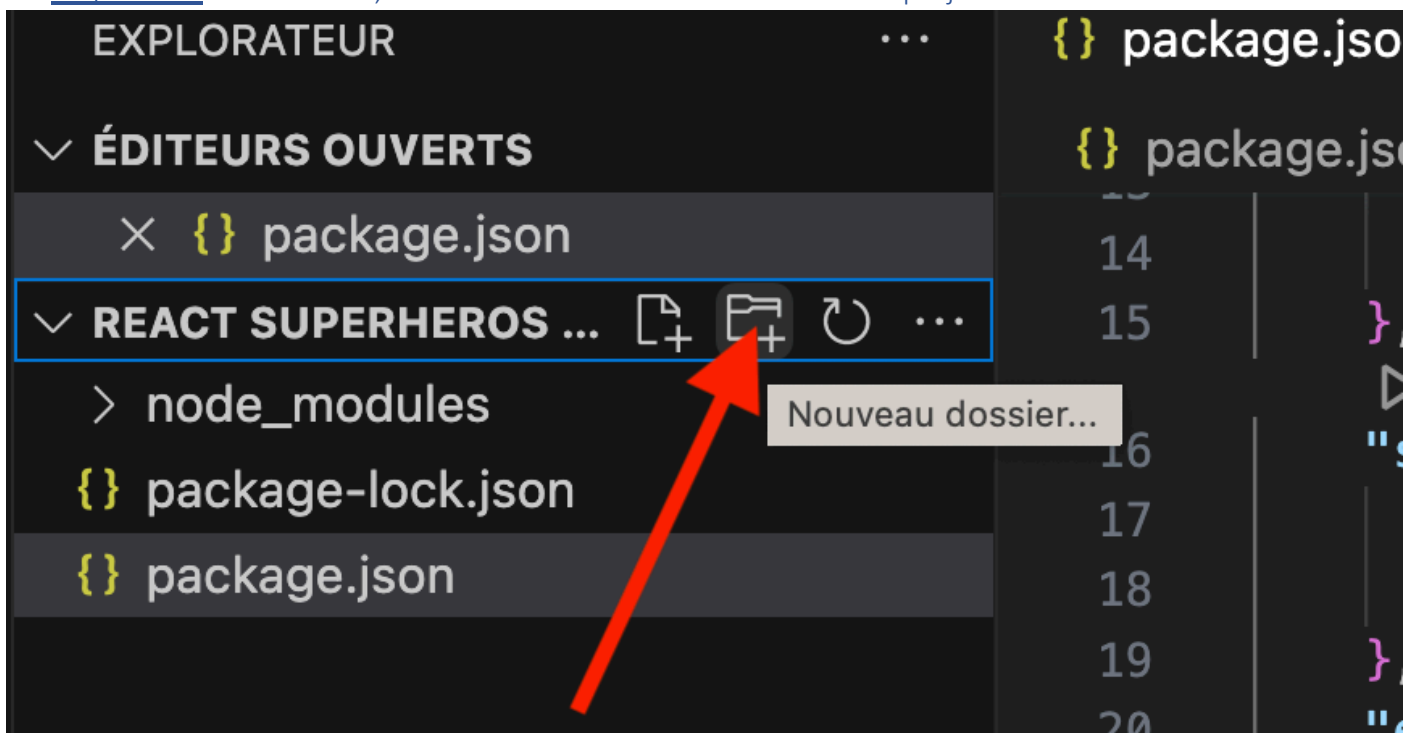
5. Création d'un composant

Plutôt que mettre tous les fichiers dans le même répertoire, nous allons les structurer à l'aide de dossier. Dans le fichier, `tsconfig.json` le répertoire déclaré est `src`

```

22     },
23     "include": [
24       "src"
25     ]
26   }
  
```

Étape N°8 : Pour ce faire, créez un dossier `src` à la racine de votre projet



Étape N°9 : Nous allons créer le fichier qui va contenir le code de votre composant. On va l'appeler simple *App.tsx*. Créez ce fichier puis mettez-y ce code

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

const App: FunctionComponent = () => {
  const name: String = 'React';

  return (
    <h1>Avec React : Hello {name} !</h1>
  )
}

export default App;
```

a) Explications du code

Ce code est un composant fonctionnel de base en React écrit en TypeScript.

Le résultat ? Ce code définit un composant fonctionnel React simple qui affiche un titre "Hello, React !" sur la page.

Voici le détail de chaque partie :

1. Importation de React et de FunctionComponent:

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

Cela importe la bibliothèque React ainsi que le type **FunctionComponent** depuis le module 'react'. Le type **FunctionComponent** (ou FC en abrégé) est un type générique (c'est-à-dire un type qui peut prendre un autre type comme argument) utilisé pour typer les composants fonctionnels en React.

2. Définition :

```
const App: FunctionComponent = () => {
  const name: String = 'React';

  return (
    <h1>Avec React : Hello {name} !</h1>
  )
}
```

Ici, **App** est défini comme un composant fonctionnel qui retourne une balise **h1** avec le texte "Hello, React !". **name** est une constante de type **String** définie à l'intérieur de la fonction.

- **const App:** Cela déclare une constante nommée **App**. En JavaScript et TypeScript, une constante est une variable qui ne peut pas être réaffectée après sa déclaration. Ici, **App** est utilisé pour stocker le composant fonctionnel React.
- **: FunctionComponent:** C'est une annotation de type TypeScript. Elle indique que la constante **App** est du type **FunctionComponent** qui est un type spécifique fourni par le package **react** pour représenter un composant fonctionnel React.

C'est une manière de dire à TypeScript (et à tout développeur qui lit le code) que **App** est un composant fonctionnel React.

- `= () => {}`: C'est la définition de la fonction. En JavaScript et TypeScript, `() => {}` définit une fonction flèche anonyme. Ici, elle est utilisée pour définir le composant fonctionnel. Le composant ne prend aucun prop (ce qui est indiqué par les parenthèses vides `()`), et actuellement, il ne renvoie rien (ce qui est indiqué par les accolades vides `{}`).

3. Rendu dans le DOM du composant APP

```
return (
  <h1>Avec React : Hello {name} !</h1>
)
```

C'est ce qui sera rendu dans le DOM lorsque ce composant est utilisé dans une application React. Voici ce que fait chaque partie :

- **return**: En JavaScript et TypeScript, **return** est utilisé pour spécifier la valeur de sortie d'une fonction. Dans un composant fonctionnel React, ce que vous retournez est ce qui sera rendu à l'écran lorsque le composant est utilisé.
- **<h1>Hello, {name} !</h1>**: Ceci est du JSX, qui est une extension de la syntaxe JavaScript qui ressemble à du HTML. Il définit une balise **<h1>** (qui est utilisée pour les titres en HTML) avec le texte "Hello, {name} !".
- **{name}**: Dans le JSX, les accolades `{}` sont utilisées pour intégrer des expressions JavaScript. Ici, **{name}** insère la valeur de la variable **name** dans le JSX. Dans votre code, **name** est défini comme **'React'**, donc la balise **<h1>** complète sera rendue comme "Hello, React !".

En résumé, ce morceau de code fait en sorte que lorsque le composant **App** est rendu, il affiche un titre "Hello, React !".

La syntaxe **{name}** dans le JSX est utilisée pour intégrer des expressions JavaScript à l'intérieur du JSX.

4. Exportation du composant App:

```
export default App;
```

Cette ligne exporte le composant **App** comme export par défaut du module. Cela signifie qu'il peut être importé dans un autre fichier avec l'instruction **import App from './App'**; (en supposant que le fichier s'appelle 'App.tsx').

b) Les fonctions fléchées

Les fonctions fléchées, ou *arrow functions* a comme structure :

nomDeLaFonction (paramètres) => { corps de la fonction }.

Une fonction fléchée est une nouvelle syntaxe pour définir les fonctions en JavaScript qui a été introduite avec ES6 (ECMAScript 2015). Elles sont aussi appelées "fonctions lambda" dans d'autres langages de programmation.

Ce type de fonctions ont deux avantages principaux par rapport aux fonctions traditionnelles définies avec le mot-clé **function** :

1. **Syntaxe plus concise** : Les fonctions fléchées permettent d'écrire des fonctions plus courtes et plus lisibles.
2. **Ne crée pas son propre contexte this** : Contrairement aux fonctions normales, les fonctions fléchées ne lient pas leur propre **this**. Elles héritent du **this** du contexte englobant. Cela est très utile lorsqu'on travaille avec des fonctions qui sont utilisées comme callbacks ou dans les classes de composants React.

c) Remarque sur JSX

JSX est une extension de syntaxe pour JavaScript qui ressemble beaucoup au HTML. Elle est utilisée avec React pour décrire à quoi devrait ressembler l'interface utilisateur. En d'autres termes, elle est utilisée pour décrire la structure de l'interface utilisateur en utilisant des composants React.

Enfin, pour que le navigateur puisse interpréter le code JSX, celui-ci doit être transpilé en JavaScript standard, généralement à l'aide d'un outil comme Babel.

"Transpiler" est un terme qui vient de la combinaison des mots "traduire" et "compiler". Un transpileur prend le code source écrit dans un langage de programmation et le produit dans un autre langage de programmation de niveau similaire.

Dans le contexte de JavaScript et de JSX, Babel est souvent utilisé comme un transpileur. Il prend le code JavaScript (ou JSX) que vous écrivez, qui peut inclure des caractéristiques plus récentes du langage qui ne sont pas encore supportées par tous les navigateurs, et le transforme en une version de JavaScript qui peut être interprétée par la majorité des navigateurs.

Dans le cas du JSX, qui est une extension de syntaxe utilisée avec React, Babel le prend et le transforme en JavaScript standard, car les navigateurs ne comprennent pas le JSX par défaut.

L'option **"jsx": "react"** indique au compilateur TypeScript de transformer chaque balise JSX en un appel à la fonction **React.createElement**. C'est essentiellement ce que Babel ferait aussi avec une configuration JSX pour React.

Voici comment cela fonctionne :

Si vous avez du code JSX, comme ceci :

```
const element = <h1>Hello, world!</h1>;
```

Après la transpilation avec l'option **"jsx": "react"**, cela devient :

```
const element = React.createElement("h1", null, "Hello, world!");
```

6. Le lien avec notre page web

Jusqu'à présent nous avons créé un composant mais il ne s'affiche pas dans le DOM.

Étape N°10 : Pour cela, nous allons déjà créer un fichier *index.tsx* (et non pas html) toujours dans le répertoire *src*

Étape N°11 : En vous inspirant des importations dans *App.tsx* :

- importez React
- importez ReactDOM de "react-dom"
- importez App que vous avez défini dans *App.tsx* (from "./App")

Étape N°12 : Ajoutez à la fin du fichier ceci

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

ReactDOM.render ?

ReactDOM.render(<App />, document.getElementById('root')); rend le composant React **App** à l'intérieur de l'élément DOM avec l'id '**root**'.

En pratique, c'est souvent la manière dont l'application React est initiée, avec un unique appel à **ReactDOM.render()** pour rendre le composant racine de l'application.

Voici le détail :

- **ReactDOM.render()**: Cette fonction est fournie par la bibliothèque ReactDOM, qui s'occupe de la mise à jour et du rendu des composants React dans le DOM.
- La fonction **ReactDOM.render()** prend deux arguments : l'élément React à rendre (dans ce cas, le composant **<App />**), et le nœud DOM où le composant React doit être rendu.
- **<App />**: Ceci est le composant React que vous souhaitez rendre. Dans votre exemple, c'est le composant **App** que vous avez défini.
- **document.getElementById('root')**: Ceci est une fonction JavaScript standard qui sélectionne un élément du DOM par son id. Dans cet exemple, elle sélectionne l'élément avec l'id '**root**'. C'est là que votre composant React sera rendu.

7. Démarrer votre application React

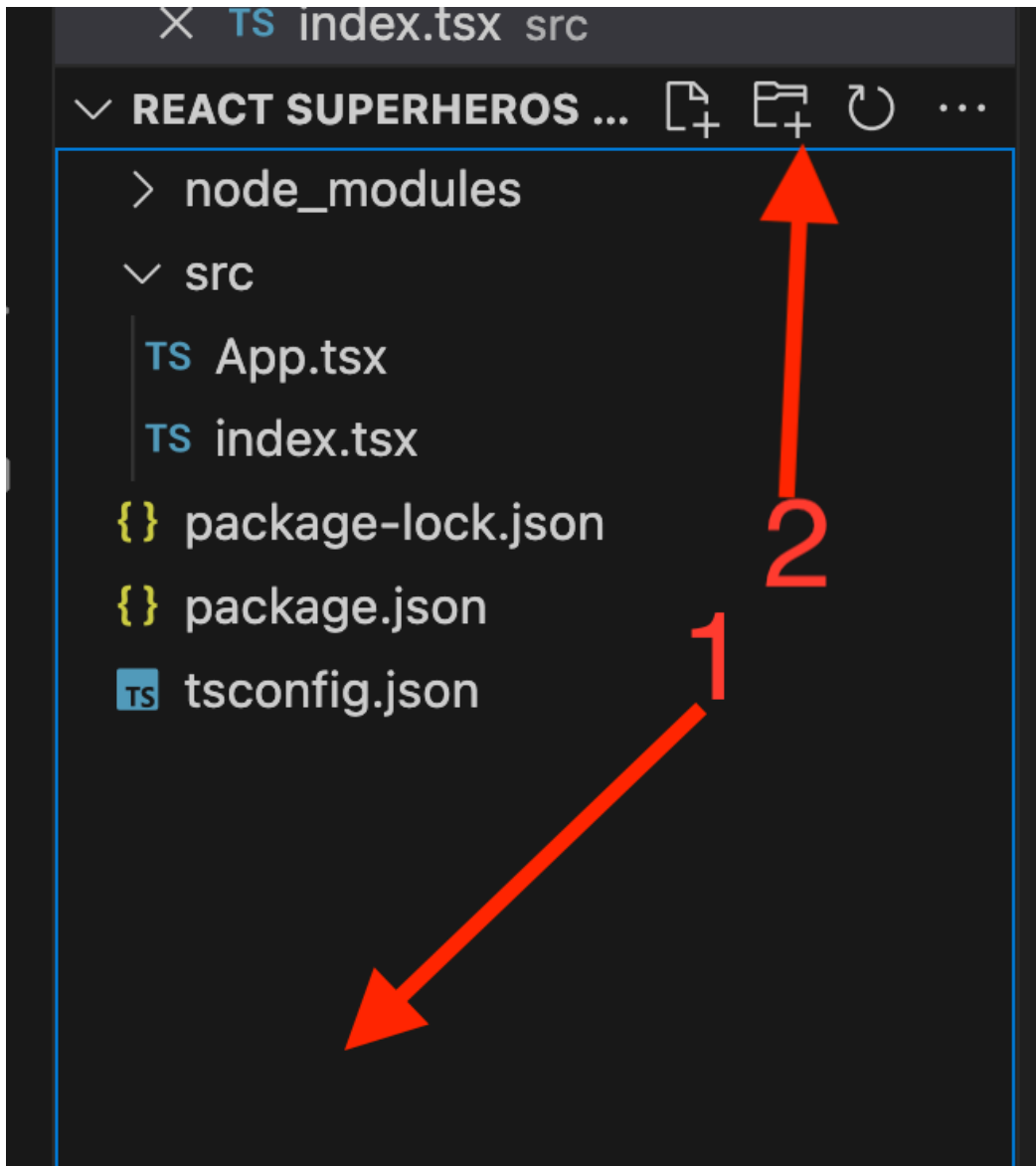
Maintenant que nous avons un composant, il faut l'afficher dans une page HTML.

Étape N°13 : Commencez par créer un répertoire *public* dans lequel vous allez y mettre un fichier *index.html*

Étape N°14 : Écrivez-y un document HTML simple avec uniquement *div* dont l'identifiant est *root*

Pour créer un répertoire à la racine il faut :

1. Cliquer en dehors de la liste des fichiers/répertoires
2. Créer un dossier



Pourquoi un répertoire *public* ?

Tous les fichiers dans le répertoire **public** seront copiés tels quels dans le build final de votre application. Ils ne passeront pas par le processus de bundling/transpilation comme le reste de votre code React.

Cela signifie que vous devez faire attention à ne pas y placer de code qui nécessite une transpilation ES6+ ou JSX, par exemple.

On peut donc y mettre :


un fichier HTML. Cela peut inclure des fichiers tels que :

- **index.html** : C'est souvent le seul fichier HTML de votre application React, qui sert de point d'entrée pour votre application. Il contient généralement une div avec un id de "root" ou similaire, qui est l'endroit où votre application React est montée par **ReactDOM.render()**.
- **Images et icônes**

- **Fichiers CSS et JavaScript statiques** : Bien que la plupart de vos fichiers CSS et JS soient probablement importés par des composants React vous pouvez avoir des fichiers CSS ou JS qui sont liés directement à partir de votre **index.html**.

Étape N°15 : Notre composant React *App* sera injecté à partir de cette ligne

```
<body>
  <div id="root">
    Première application avec React
  </div>
```



Étape N°16 : Vérifiez que vous avez bien cette structure

```

✓ REACT SUPERHEROS ... [Icons]
  > node_modules
  ✓ public
    <> index.html
  ✓ src
    TS App.tsx
    TS index.tsx
    {} package-lock.json
    {} package.json
    TS tsconfig.json
```

Étape N°17 : Pour « démarrer » l'application, entrez `npm start` dans un terminal

SORTIE CONSOLE DE DÉBOGAGE TERMINAL

```
○ roberttomczak@macpro-rtomczak React SuperHeros App % npm start
```

Et si vous n'avez pas d'erreur, une page HTML s'ouvre automatiquement

Avec React : Hello Word !

Il n'est pas la peine de rafraîchir la page, tout se fait automatiquement
Si vous n'avez pas d'erreurs, passé à l'étape 10 : modification du code React

8. Que faire en cas d'erreur :

Par exemple :

```
os App/node_modules/webpack/lib/NormalModule.js:386:16)
  at /Users/roberttomczak/Dropbox/Acer/Cours INSA/01-Javascript/Cours/React/React SuperHeros App/node_modules/webpack/lib/NormalModule.js:418:10
  at /Users/roberttomczak/Dropbox/Acer/Cours INSA/01-Javascript/Cours/React/React SuperHeros App/node_modules/webpack/lib/NormalModule.js:293:13
  at /Users/roberttomczak/Dropbox/Acer/Cours INSA/01-Javascript/Cours/React/React SuperHeros App/node_modules/loader-runner/lib/LoaderRunner.js:367:11
  at /Users/roberttomczak/Dropbox/Acer/Cours INSA/01-Javascript/Cours/React/React SuperHeros App/node_modules/loader-runner/lib/LoaderRunner.js:233:18
  at context.callback (/Users/roberttomczak/Dropbox/Acer/Cours INSA/01-Javascript/Cours/React/React SuperHeros App/node_modules/loader-runner/lib/LoaderRunner.js:111:13)
  at /Users/roberttomczak/Dropbox/Acer/Cours INSA/01-Javascript/Cours/React/React SuperHeros App/node_modules/babel-loader/lib/index.js:51:103 {
  opensslErrorStack: [ 'error:03000086:digital envelope routines::initialization error' ],
  library: 'digital envelope routines',
  reason: 'unsupported',
  code: 'ERR_OSSL_EVP_UNSUPPORTED'
}
```

Essayez ceci :

- **npm cache clean --force** : cette commande force la suppression sans afficher le message d'information ni le chemin du répertoire de cache.
- **Supprimez** le dossier `node_modules`
- **Relancez un npm install**
- Et essayez à nouveau un **npm start**

9. Création automatique d'un projet React NE PAS FAIRE SI TOUT EST OK

Dans les paragraphes précédents, nous avons voulu montrer la structure d'un projet React et expliquer le code.

Cependant, il y a plus rapide :

Étape Ouvrez votre terminal ou votre invite de commandes.

Étape Exécutez la commande suivante en remplaçant **<nom-du-projet>** par le nom que vous souhaitez donner à votre projet React :

```
npx create-react-app <nom-du-projet>
```

Par exemple, si vous souhaitez nommer votre projet "superheros-app", la commande serait :

```
npx create-react-app superheros-app
```

Étape Attendez que la création du projet soit terminée. Cela peut prendre quelques instants car **create-react-app** télécharge et configure toutes les dépendances nécessaires.

Étape Une fois que la création est terminée, accédez au répertoire de votre nouveau projet :

```
cd <nom-du-projet>
```

Dans notre exemple :

```
cd superheros-app
```

Étape Vous pouvez maintenant exécuter votre application React en utilisant la commande :

```
npm start
```

Cette commande démarre le serveur de développement et ouvre automatiquement votre application dans votre navigateur par défaut.

Cela créera automatiquement un nouveau projet React avec une structure de fichiers de base et les dépendances nécessaires pour commencer à développer votre application. Vous pouvez ensuite éditer les fichiers dans le répertoire de votre projet et voir les modifications en temps réel dans votre navigateur lors du développement.

10. Modification du code react

Étape N°18 : Modifiez le fichier *App.tsx*, pour avoir :

- un h2
- Enlever "Avec React :"
- modifiez la valeur de *name* avec votre prénom

Étape N°19 : Observez que toutes vos modifications sont prises en compte sans avoir à rafraîchir la page

IV. Les hooks

1. Présentation

Les Hooks sont une fonctionnalité introduite dans React 16.8 qui permet d'utiliser l'état et d'autres fonctionnalités de React sans avoir à écrire une classe. En d'autres termes, ils permettent aux développeurs d'utiliser les fonctionnalités de React dans les composants fonctionnels.

Il existe plusieurs types de Hooks dans React, mais les plus couramment utilisés sont **useState** et **useEffect**.

a) useState

Le Hook **useState** est une fonction qui vous permet d'ajouter l'état React à un composant fonctionnel. Voici un exemple simple :

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

function Compter() {
  // Déclare une nouvelle variable d'état, que nous appellerons "compteur"
  const [compteur, setCompteur] = useState(0);

  return (
    <div>
      <p>Vous avez cliqué {compteur} fois</p>
      <button onClick={() => setCompteur(compteur + 1)}>
        Cliquez moi
      </button>
    </div>
  );
}

export default Compter;
```

Dans cet exemple, **useState** est utilisé pour déclarer une variable **compteur** qui est initialement définie à 0. Lorsque l'utilisateur clique sur le bouton, la fonction **setCompteur** est appelée avec la nouvelle valeur de **count**, qui est la valeur actuelle de **compteur** plus 1.

b) useEffect

Le Hook **useEffect** vous permet d'effectuer des effets secondaires dans les composants fonctionnels. Vous pouvez l'utiliser pour des opérations qui nécessitent un nettoyage, comme les abonnements ou les timers. Voici un exemple :

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

function Exemple() {
  const [compteur, setCompteur] = useState(0);

  // Similaire à componentDidMount et componentDidUpdate :
  useEffect(() => {
    // Mettre à jour le titre du document en utilisant l'API du navigateur
    document.title = `Vous avez cliqué ${compteur} fois`;

    // Vous pouvez renvoyer une fonction de nettoyage pour désabonnement, par exemple
    return function nettoyage() {
      document.title = `React App`;
    };
  }, [compteur]); // N'exécute l'effet que si compteur change

  return (
    <div>
      <p>Vous avez cliqué {compteur} fois</p>
    </div>
  );
}
```

```

    <button onClick={() => setCompteur(compteur + 1)}>
      Cliquez moi
    </button>
  </div>
);
}

export default Exemple;

```

Dans cet exemple, **useEffect** est utilisé pour mettre à jour le titre du document chaque fois que le compteur **count** est mis à jour. L'effet est déclenché uniquement lorsque la valeur de **count** change, comme indiqué par le deuxième argument à **useEffect**, qui est un tableau de dépendances. Le retour de nettoyage est une fonction qui sera exécutée avant le prochain appel de l'effet ou lorsque le composant est démonté.

Par défaut, le Hook **useEffect** est appelé après le premier rendu du composant, c'est-à-dire au démarrage de l'application ou plus précisément lorsque le composant est monté pour la première fois.

2. Application sur le projet

Dans notre code

1. La variable **name** est un chaîne de caractères
2. Elle vaut **Word**
3. Cette variable ne peut pas être modifiée, et il n'est pas possible de faire autrement

Ainsi, tout le long de l'application, **name** vaudra toujours **World** sauf si on modifie le fichier tsx. I Or, il faudrait pouvoir modifier sa valeur dynamique.

Les hooks et plus précisément ici, **useState** va pouvoir modifier cette valeur au cours de l'exécution.

a) La première ligne

Au lieu de

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

Nous avons :

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

- Explication : **useState** est l'un des Hooks fournis par React est une fonction qui vous permet d'ajouter l'état local à un composant fonctionnel.

b) La troisième ligne

Au lieu de

```
const name: String = 'React';
```

Nous avons :

```
const [name, setName] = useState('World');
```

Explication : Le Hook **useState** retourne un tableau contenant deux éléments : la valeur actuelle de l'état et une fonction pour modifier cette valeur

3. Manipulations

Étape N°20 : Mettez en place le hook **useState** et deux boutons. Le premier va ajouter à **name** un point d'interrogation et le second va le remettre à sa valeur initiale

N'oubliez pas que le Hook **useState** retourne un tableau contenant deux éléments : la valeur actuelle de l'état et une fonction pour modifier cette valeur

Exemple après trois appuis sur **Ajoutez un !**

Avec React : Hello World!!!!

Ajoutez un !

Remettre World

Après le clic sur **Remettre World** :

Avec React : Hello World!

Ajoutez un !

Remettre World

DDSD

V. Affichage d'informations sur les SuperHeros

1. Affichage nombre de SuperHeros

Étape N°21 : Créez un fichier SuperHeros.json et copiez ce texte :

```
[
  {
    "id": 1,
    "name": "Superman",
    "id-api": 644,
    "slug": "644-superman"
  },
  {
    "id": 2,
    "name": "Batman",
    "id-api": 69,
    "slug": "69-batman"
  },
  {
    "id": 3,
    "name": "Spider-Man",
    "id-api": 620,
    "slug": "620-spider-man"
  },
  {
    "id": 4,
    "name": "Wonder Woman",
    "id-api": 720,
    "slug": "720-wonder-woman"
  },
  {
    "id": 5,
    "name": "Iron Man",
    "id-api": 346,
    "slug": "346-iron-man"
  },
  {
    "id": 6,
    "name": "Thor",
    "id-api": 123,
    "slug": "123-thor"
  },
  {
    "id": 7,
    "name": "Hulk",
    "id-api": 456,
    "slug": "456-hulk"
  },
  {
    "id": 8,
    "name": "Captain America",
    "id-api": 789,
    "slug": "789-captain-america"
  },
  {
    "id": 9,
    "name": "Black Widow",
    "id-api": 101,
    "slug": "101-black-widow"
  },
  {
    "id": 10,
    "name": "Ant-Man",
    "id-api": 234,
    "slug": "234-ant-man"
  }
]
```



```
"id": 6,  
"name": "Captain America",  
"id-api": 149,  
"slug": "149-captain-america"  
},  
{  
  "id": 7,  
  "name": "Hulk",  
  "id-api": 332,  
  "slug": "332-hulk"  
},  
{  
  "id": 8,  
  "name": "Thor",  
  "id-api": 659,  
  "slug": "659-thor"  
},  
{  
  "id": 9,  
  "name": "Wolverine",  
  "id-api": 717,  
  "slug": "717-wolverine"  
},  
{  
  "id": 10,  
  "name": "Black Widow",  
  "id-api": 107,  
  "slug": "107-black-widow"  
},  
{  
  "id": 11,  
  "name": "The Flash",  
  "id-api": 267,  
  "slug": "267-flash-iv"  
},  
{  
  "id": 12,  
  "name": "Silver Surfer",  
  "id-api": 598,  
  "slug": "598-silver-surfer"  
},  
{  
  "id": 13,  
  "name": "Aquaman",  
  "id-api": 38,  
  "slug": "38-aquaman"  
},  
{
```

```

    "id": 14,
    "name": "Black Panther",
    "id-api": 106,
    "slug": "106-black-panther"
  },
  {
    "id": 15,
    "name": "Doctor Strange",
    "id-api": 226,
    "slug": "226-doctor-strange"
  },
  {
    "id": 16,
    "name": "Supergirl",
    "id-api": 643,
    "slug": "643-supergirl"
  }
]

```

2. Une classe de SuperHeros

Soit par exemple une classe Livre le code typescript est le suivant :

```

// Livres.ts
export class Livre {
  id: number;
  titre: string;
  idApi: number;
  presentation: string ;

  constructor(id: number, titre: string, idApi: number, presentation: string) {
    this.id = id;
    this.titre = titre;
    this.idApi = idApi;
    this.presentation = presentation;
  }
}

```

Étape N°22 : Écrivez une classe pour les SuperHeros et incorporez la dans votre projet

3. Le composant

Étape N°23 : Complétez le code de App.tsx correspondant à la classe qui vient d'être créée

```

// App.tsx
import React, { _____ } from 'react';
import _____ from _____; // Importation de notre classe SuperHero
import SuperHerosData from '_____ // Importation du fichier JSON

export const App = () => {

```

```
// Déclaration d'un état local 'heroes' pour stocker notre tableau de super-héros.
// 'setHeroes' est la fonction pour mettre à jour cet état.
const [heroes, setHeroes] = _____<SuperHero[]>([]);

// 'useEffect' est utilisé pour effectuer des effets de bord dans les composants
fonctionnels.
// Dans ce cas, nous l'utilisons pour initialiser notre tableau de super-héros à partir du
fichier JSON lorsque le composant est monté.
_____ (() => {
    // Nous convertissons chaque objet du fichier JSON en une instance de SuperHero et
    mettons à jour notre état.
    const heroesFromData = _____.map((heroData: any) => new SuperHero(heroData.id,
    heroData.name, heroData['id-api'], heroData.slug));
    setHeroes(heroesFromData);
}, []); // Le tableau vide en deuxième argument signifie que cet effet ne s'exécute qu'une
fois, lors du montage du composant.

// Rendu de notre composant : ici, nous affichons simplement le nombre de super-héros
chargés à partir du fichier JSON.
return (
    <div>
        <h1>Super Heroes App</h1>
        <p>Nombre de super-héros chargés: {heroes. _____}</p>
    </div>
);
}
```

4. Lister toutes les caractéristiques des SuperHeros

Dans le code suivant, on utilise la méthode **map** pour boucler sur notre tableau **heroes** et pour chaque élément (super-héros), on retourne un bloc **div** qui affiche les informations du super-héros. Chaque bloc **div** a une clé **key** qui est l'**id** du super-héros, ce qui est nécessaire lorsque vous mappez sur un tableau pour créer des éléments JSX en React.

Étape N°24 : Complétez ce code et le mettre dans le bon fichier

```
{heroes.map((hero: SuperHero) => (
    <div key={_____}>
        <h2>{_____}</h2>
        <p>Id: _____</p>
        _____
        _____
    </div>
))}
```

5. Avec les images

Toutes les images se trouvent à l'adresse suivante :

<https://cdn.jsdelivr.net/gh/rtomczak/superhero-api@0.3.0/api/images/sm/>

suivi du champ slug.

Par exemple pour l'image de Superman, l'URL est :

<https://cdn.jsdelivr.net/gh/rtomczak/superhero-api@0.3.0/api/images/sm/644-superman.jpg>



Étape N°25 : Ajoutez au fichier précédent l'affichage des images

VI. Points d'étapes

Maintenant que vous avez fait vos premiers pas avec React.JS, nous allons faire un point d'étape sur plusieurs éléments.

1. Javascript ou Typescript

a) Avantages/Inconvénients

Le choix entre JavaScript et TypeScript pour un projet React dépend de plusieurs facteurs, notamment les préférences de l'équipe, les exigences du projet et les ressources disponibles.

Voici quelques points à considérer :

TypeScript :

- **Avantages :**
 - **Sûreté des types :** TypeScript est un langage à typage statique, ce qui peut aider à prévenir certains types d'erreurs pendant le développement.
 - **Meilleure autocomplétion et refactoring :** Les outils de développement peuvent utiliser les informations de type pour fournir une meilleure autocomplétion et des outils de refactoring plus robustes.
 - **Documentation du code :** Les types peuvent servir de documentation pour le code, ce qui peut rendre le code plus facile à comprendre pour les nouveaux venus sur le projet.
- **Inconvénients :**
 - **Courbe d'apprentissage :** TypeScript peut être plus difficile à apprendre pour les développeurs qui n'ont pas d'expérience avec les langages à typage statique.
 - **Complexité supplémentaire :** TypeScript ajoute une étape de compilation supplémentaire et peut rendre le code plus complexe.

JavaScript :

- **Avantages :**
 - **Simplicité :** JavaScript est plus simple à utiliser car il n'y a pas de système de types à gérer.
 - **Popularité :** JavaScript est un langage plus populaire, donc il peut être plus facile de trouver des développeurs JavaScript.
- **Inconvénients :**
 - **Moins sûr :** Sans le système de types de TypeScript, il peut être plus facile de commettre certaines erreurs.

b) Comment savoir si je travaille en TS ou JS ?

Voici comment vous pouvez déterminer si un projet React est écrit en TypeScript ou en JavaScript :

1. Extensions de fichier :

- Les fichiers JavaScript auront une extension `.js` ou `.jsx` pour les fichiers qui contiennent du JSX.
- Les fichiers TypeScript auront une extension `.ts` pour les fichiers de code général et `.tsx` pour les fichiers qui contiennent du JSX.

2. Syntaxe du code :

- TypeScript est un sur-ensemble de JavaScript qui ajoute des types statiques. Si vous voyez des annotations de type dans le code, comme par exemple `let x: number = 3;` ou `function (x: string): string { ... }`, alors le code est écrit en TypeScript.
- Si le code ne contient pas d'annotations de type, alors il est probablement écrit en JavaScript. Cependant, gardez à l'esprit que TypeScript est un sur-ensemble de JavaScript, donc du code JavaScript valide est aussi du TypeScript valide. Par conséquent, l'absence d'annotations de type ne signifie pas nécessairement que le code est du JavaScript.

3. Fichiers de configuration :

- Un projet TypeScript aura un fichier `tsconfig.json` à la racine du projet. Ce fichier contient les paramètres de configuration de TypeScript pour le projet.

Si vous trouvez des fichiers avec des extensions `.ts` ou `.tsx`, des annotations de type dans le code, et un fichier `tsconfig.json` dans le projet, alors il s'agit très probablement d'un projet React écrit en TypeScript.

c) Pourquoi Typescript a été préféré à Javascript dans ce projet ? Langage pour l'avenir ?

TypeScript gagne en popularité et en adoption dans l'industrie du développement logiciel. Selon le sondage annuel de Stack Overflow en 2020, TypeScript est maintenant l'un des langages de programmation les plus aimés et recherchés. Cependant, prévoir si TypeScript est "l'avenir" est un peu plus complexe. Voici quelques points à considérer :

1. **Popularité croissante** : TypeScript a gagné en popularité parmi les développeurs JavaScript pour plusieurs raisons, notamment pour sa sécurité de type statique, ses outils de développement améliorés, et sa compatibilité avec les nouvelles fonctionnalités de JavaScript.
2. **Adoption par l'industrie** : De grandes entreprises, comme Microsoft (qui a développé TypeScript), Google, Airbnb, et Slack, ont adopté TypeScript pour certains de leurs projets. En outre, des projets open-source importants, comme Angular et Deno, utilisent également TypeScript.
3. **Favorisé par certains cadres** : Certains cadres de développement populaires, comme Angular, favorisent ou encouragent l'utilisation de TypeScript. Cela pourrait inciter davantage de développeurs à apprendre et à utiliser TypeScript.

Cependant, cela ne signifie pas que JavaScript est en voie de disparition ou sera remplacé par TypeScript. TypeScript est un sur-ensemble de JavaScript, donc tout ce qui peut être fait en JavaScript peut également être fait en TypeScript. JavaScript reste un langage très populaire et largement utilisé.

d) Exemples JavaScript <-> TypeScript

Voici quelques exemples de code JavaScript et de leurs équivalents TypeScript :

i. Exemple 1 : Déclaration de variable

JavaScript :

```
let x = 5;
```

TypeScript :

```
let x: number = 5;
```

Dans TypeScript, vous pouvez spécifier le type de variable lors de sa déclaration.

ii. Exemple 2 : les tableaux

JavaScript :

```
let desNombres = [1, 2, 3, 4, 5];
```

TypeScript :

```
let desNombres: number[] = [1, 2, 3, 4, 5];
```

iii. Exemple 3 : Fonction

JavaScript :

```
function add(a, b) {
  return a + b;
}
```

TypeScript :

```
function add(a: number, b: number)
  : number {
  return a + b;
}
```

Dans TypeScript, vous pouvez spécifier les types des paramètres et le type de retour de la fonction.

iv. Exemple 4 : Objet

JavaScript :

```
let user = {
  name: "John",
  age: 30
};
```

TypeScript :

```
let user: {
  name: string, age:
  number
}
=
{
  name: "John",
  age: 30
};
```

Dans TypeScript, vous pouvez spécifier les types des propriétés de l'objet.

v. Exemple 5 : Classes :

JavaScript :

```
class Car {
  constructor(marque, modele) {
    this.marque = marque;
    this.modele = modele;
  }
  startEngine() { return 'Vroom!'; }
}
```

TypeScript :

```
class Car {
  marque: string;
  modele: string;
```

```
constructor(marque string, modele: string) {
  this.marque = marque;
  this.modele = modele;
}

startEngine(): string {
  return 'Vroom!';
}
}
```

Notez que TypeScript est un sur-ensemble strict de JavaScript, ce qui signifie que chaque programme JavaScript valide est également un programme TypeScript valide. Les ajouts de TypeScript, tels que les annotations de type et les interfaces, fournissent un typage statique optionnel pour JavaScript, ce qui peut aider à la détection des erreurs et à l'autocomplétion dans les éditeurs de code.

2. Configuration du projet

a) tsconfig.json

Le fichier **tsconfig.json** est utilisé pour configurer le compilateur TypeScript pour un projet TypeScript. Il spécifie les options du compilateur à utiliser lors de la compilation du code TypeScript.

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "esnext",
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "jsx": "react",
    "sourceMap": true,
    "esModuleInterop": true,
    "strict": true,
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "noImplicitAny": false,
    "noImplicitReturns": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true
  },
  "include": ["src"],
  "exclude": ["node_modules", "build", "dist", "public"]
}
```

b) la version de javascript

Cette ligne indique que notre code va être « compilé » en javascript ES6 :

```
3      "target": "es6",
```


c) option de « compilation »

```
20 "jsx": "react",
```

Dans un fichier **tsconfig.json** de TypeScript, l'option **"jsx": "react"** indique au compilateur TypeScript de transformer chaque balise JSX en un appel à la fonction **React.createElement**.

d) Les répertoires

```
"include": ["src"],
"exclude": ["node_modules", "build", "dist", "public"]
}
```

vi. L'option include

indique le répertoire dans lequel se trouve ces fichiers : uniquement dans le répertoire *src*

i. L'option exclude

L'option **exclude** dans **tsconfig.json** est utilisée pour spécifier les fichiers ou les dossiers que le compilateur TypeScript doit ignorer.

e) Node.JS

Node.js est une plateforme logicielle open-source qui exécute du JavaScript côté serveur. C'est une nécessité pour le développement avec React pour plusieurs raisons :

1. **Gestion des paquets** : Node.js vient avec npm (node package manager), qui est utilisé pour gérer les dépendances de votre projet. Les bibliothèques et les frameworks comme React sont installés en tant que dépendances via npm.
2. **Outils de développement** : Beaucoup d'outils de développement pour React, comme Create React App, Next.js ou Gatsby, sont construits avec Node.js. Ces outils nécessitent Node.js pour fonctionner correctement.
3. **Environnement de développement** : Lorsque vous développez une application avec React, vous aurez probablement besoin d'un serveur de développement local pour tester votre application. Node.js peut être utilisé pour créer et exécuter ce serveur.
4. **Rendu côté serveur** : Si vous choisissez de faire du rendu côté serveur avec React (ce qui peut améliorer les performances et le référencement), vous aurez besoin d'un environnement côté serveur capable d'exécuter JavaScript. Node.js est parfait pour cela.

VII. Points d'étapes : Les fichiers

1. **App.tsx**

Dans une application React, **App.tsx** est généralement le point d'entrée principal de l'application ou le composant racine. C'est ici que les composants de niveau supérieur de votre application sont généralement rendus.

Le fichier **App.tsx** est particulièrement important dans une application créée avec "Create React App" qui utilise TypeScript. Le **.tsx** signifie qu'il s'agit d'un fichier JSX qui est écrit en TypeScript.

Première version

Ce code crée un composant React qui affiche "Avec React : Hello React !" dans un élément h1.

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

const App: FunctionComponent = () => {
  const name: String = 'React';

  return (
    <h1>Avec React : Hello {name} !</h1>
  )
}

export default App;
```

Voici une explication ligne par ligne :

1. **import React, { FunctionComponent } from 'react';**

Cette ligne importe le module React et le type **FunctionComponent** du module 'react'.

FunctionComponent est un type générique pour décrire un composant fonctionnel en React.

2. **const App: FunctionComponent = () => {**

Cette ligne déclare un composant appelé **App**. Il utilise le type **FunctionComponent** pour indiquer que c'est un composant fonctionnel. En TypeScript, **:** est utilisé pour indiquer le type d'une variable ou d'une constante.

3. **const name: String = 'React';**

Cette ligne déclare une constante appelée **name** de type **String** et lui assigne la valeur 'React'.

4. **return (<h1>Avec React : Hello {name} !</h1>)**

Cette ligne est le rendu du composant. Elle renvoie un élément JSX qui est un titre (h1) contenant du texte et la valeur de la constante **name**. Les accolades **{}** sont utilisées en JSX pour insérer des expressions JavaScript.

5. **}**

Cette accolade ferme la fonction du composant.

6. **export default App;**

Cette ligne exporte le composant **App** comme exportation par défaut du module. Cela signifie que lorsque vous importez ce module dans un autre fichier sans spécifier une exportation nommée, c'est le composant **App** qui sera importé.

2. Les hooks

Les Hooks sont une fonctionnalité ajoutée dans React 16.8 qui permettent d'utiliser l'état et d'autres fonctionnalités de React dans les composants fonctionnels, qui étaient auparavant limités aux composants de classe.

Les Hooks sont des fonctions qui "se connectent" aux fonctionnalités internes de React. Il existe plusieurs Hooks fournis par React, chacun ayant un but différent :

- **useState** : C'est le Hook le plus couramment utilisé. Il permet d'ajouter un état local dans un composant fonctionnel. Avant les Hooks, seul les composants de classe pouvaient avoir un état.

- **useEffect** : Ce Hook est utilisé pour effectuer des effets de bord (appels de fonction qui interagissent avec le monde extérieur, comme les requêtes réseau, les manipulations du DOM, etc.) dans les composants fonctionnels.
- **useContext** : Ce Hook permet d'accéder à la valeur actuelle d'un contexte React. C'est utile pour partager des données globales entre plusieurs composants.
- **useReducer** : Ce Hook est une alternative à **useState** qui accepte un réducteur de type **(state, action) => newState** et renvoie l'état actuel avec une méthode **dispatch**. Il est préféré lorsque la logique de mise à jour de l'état est complexe.
- **useRef** : Ce Hook est utilisé pour créer des références mutables.
- **useMemo** et **useCallback** : Ces Hooks sont utilisés pour optimiser les performances en évitant les calculs ou les créations de fonction inutiles.

Il est également possible de créer vos propres Hooks personnalisés pour réutiliser la logique de l'état entre différents composants.

a) Hook **useState** : chaque fois que l'état change sa valeur est modifiée

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

function Compter() {
  // Déclare une nouvelle variable d'état, que nous appellerons "compteur"
  const [compteur, setCompteur] = useState(0);

  return (
    <div>
      <p>Vous avez cliqué {compteur} fois</p>
      <button onClick={() => setCompteur(compteur + 1)}>
        Cliquez moi
      </button>
    </div>
  );
}

export default Compter;
```

useState permet d'ajouter un état local dans un composant fonctionnel et dans cet exemple, le Hook **useState** est utilisé pour ajouter un état local au composant fonctionnel **Compter**.

L'état est une valeur qui est maintenue par le composant et qui peut changer au fil du temps. Chaque fois que l'état change, le composant est rerendu avec la nouvelle valeur de l'état.

Voici comment le Hook **useState** fonctionne dans cet exemple :

- **const [compteur, setCompteur] = useState(0);** : **useState** est appelé avec la valeur initiale **0**. Cette fonction renvoie un tableau de deux éléments. Le premier élément est la valeur actuelle de l'état (**compteur**), et le deuxième élément est une fonction (**setCompteur**) qui permet de mettre à jour cet état.

Dans ce cas, **compteur** est la valeur actuelle de l'état et **setCompteur** est la fonction qui permet de mettre à jour cet état. C'est donc React qui fournit et définit **setCompteur** lorsque vous appelez **useState**.

La fonction **setCompteur** peut être appelée avec une nouvelle valeur pour mettre à jour l'état **compteur**.

Lorsqu'elle est appelée, React mettra à jour l'état **compteur** avec la nouvelle valeur et réexécutera le composant avec la nouvelle valeur de l'état.

- **<p>Vous avez cliqué {compteur} fois</p>** : Ici, la valeur actuelle de **compteur** est affichée dans le rendu du composant. Initialement, cela affichera "Vous avez cliqué 0 fois".
- **<button onClick={() => setCompteur(count + 1)}>Cliquez moi</button>** : Ce bouton a un gestionnaire d'événements **onClick** qui appelle la fonction **setCompteur** avec la valeur actuelle de **compteur** plus 1. Cela mettra à jour l'état **compteur** et forcera le composant à être rerendu avec la nouvelle valeur de **compteur**.

b) **useEffect** : effet de bord après chaque rendu

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

function Exemple() {
  const [compteur, setCompteur] = useState(0);

  // Similaire à componentDidMount et componentDidUpdate :
  useEffect(() => {
    // Mettre à jour le titre du document en utilisant l'API du navigateur
    document.title = `Vous avez cliqué ${compteur} fois`;
  }, [compteur]); // N'exécute l'effet que si compteur change

  return (
    <div>
      <p>Vous avez cliqué {compteur} fois</p>
      <button onClick={() => setCompteur(compteur + 1)}>
        Cliquez moi
      </button>
    </div>
  );
}

export default Exemple;
```

useEffect est utilisé pour effectuer un effet de bord après chaque rendu.

Dans cet exemple l'effet de bord, dans ce cas, est de changer le titre du document pour indiquer combien de fois vous avez cliqué.

Le Hook **useEffect** fonctionne de manière similaire à **componentDidMount** et **componentDidUpdate** dans les composants de classe.

Voici comment cela fonctionne dans cet exemple :

- **useEffect(() => {...}, [compteur]);**

Le premier argument de **useEffect** est une fonction qui contient le code de l'effet à exécuter. Dans cet exemple, l'effet est de mettre à jour le titre du document pour afficher le nombre de fois que l'utilisateur a cliqué sur le bouton.

Le deuxième argument de **useEffect** est un tableau de dépendances. C'est une liste de valeurs que l'effet utilise et pour lesquelles il doit être réexécuté lorsqu'elles changent. Dans cet exemple, l'effet dépend de la valeur de **compteur**. Donc chaque fois que **compteur** change (c'est-à-dire que l'utilisateur clique sur le bouton), l'effet est réexécuté et le titre du document est mis à jour.

c) useEffect est appelé après le premier rendu du composant

Par défaut, le Hook **useEffect** est appelé après le premier rendu du composant, c'est-à-dire au démarrage de l'application ou plus précisément lorsque le composant est monté pour la première fois.

C'est pourquoi il est souvent utilisé pour des actions similaires à ce qui serait mis dans **componentDidMount** dans un composant de classe.

Ensuite, **useEffect** est également appelé après chaque mise à jour du composant (par exemple, après un changement d'état ou de prop), sauf si vous fournissez un tableau de dépendances comme deuxième argument.

Si un tableau de dépendances est fourni, **useEffect** ne sera exécuté qu'après le premier rendu et lorsque l'une des dépendances change. Si le tableau de dépendances est vide (`[]`), **useEffect** ne s'exécute qu'une fois après le premier rendu, simulant ainsi un comportement **componentDidMount**.

3. Version plus élaborée de APP.TSX

```
// App.tsx
import React, { useState, useEffect } from 'react';
import { SuperHero } from './SuperHero'; // Importation de notre classe SuperHero
import SuperHerosData from './SuperHeros.json'; // Importation du fichier JSON

export const App = () => {
  // Déclaration d'un état local 'heroes' pour stocker notre tableau de super-héros.
  // 'setHeroes' est la fonction pour mettre à jour cet état.
  const [heroes, setHeroes] = useState<SuperHero[]>([]);

  // 'useEffect' est utilisé pour effectuer des effets de bord dans les composants
  fonctionnels.
  // Dans ce cas, nous l'utilisons pour initialiser notre tableau de super-héros à partir du
  fichier JSON lorsque le composant est monté.
  useEffect(() => {
    // Nous convertissons chaque objet du fichier JSON en une instance de SuperHero et
    mettons à jour notre état.
    const heroesFromData = SuperHerosData.map((heroData: any) => new SuperHero(heroData.id,
    heroData.name, heroData['id-api'], heroData.slug));
    setHeroes(heroesFromData);
  }, []); // Le tableau vide en deuxième argument signifie que cet effet ne s'exécute qu'une
  fois, lors du montage du composant.
```

```
// Rendu de notre composant : ici, nous affichons simplement le nombre de super-héros
chargés à partir du fichier JSON.
return (
  <div>
    <h1>Super Heroes App</h1>
    <p>Nombre de super-héros chargés: {heroes.length}</p>
    {heroes.map((hero: SuperHero) => (
      <div key={hero.id}>
        <h2>{hero.name}</h2>
        <p>Id: {hero.id}</p>
        <p>Id API: {hero.idApi}</p>
        <p>Slug: {hero.slug}</p>
        {
          <img src={`https://cdn.jsdelivr.net/gh/rtomczak/superhero-
api@0.3.0/api/images/sm/${hero.slug}.jpg`} alt={hero.name} />
        }
      </div>
    ))}
  </div>
);
export default App;
```

Ce script React TypeScript définit le composant **App**, qui utilise un tableau de données **SuperHero** pour afficher une liste de super-héros.

Chaque super-héros est défini par une classe **SuperHero** et les données de ces super-héros sont chargées à partir d'un fichier JSON.

a) `const [heroes, setHeroes] = useState<SuperHero[]>([]);`

Ici, un nouvel état local **heroes** est déclaré à l'aide du Hook **useState**. Cet état représente un tableau d'objets **SuperHero**. Initialement, cet état est défini comme un tableau vide. **setHeroes** est la fonction qui sera utilisée pour mettre à jour cet état.

b) `useEffect(() => {...}, []);`

Le Hook **useEffect** est utilisé pour effectuer une action après le rendu du composant. Dans notre cas, il est utilisé pour effectuer une action (charger les données de super-héros à partir du fichier JSON) lorsque le composant est monté pour la première fois.

Ici, il est utilisé pour initialiser l'état **heroes** avec les données du fichier JSON.

c) `const heroesFromData = SuperHerosData.map((heroData: any) => new`

Cette ligne crée un nouveau tableau d'objets **SuperHero** à partir des données dans **SuperHerosData**.

La fonction **map** est utilisée pour convertir chaque objet du fichier JSON en une instance de **SuperHero**.

Cette fonction de haut niveau **parcourt** chaque élément du tableau et **applique** une fonction à chaque élément, puis **retourne** un nouveau tableau avec les résultats.

La fonction passée à **map** est une fonction fléchée qui prend un élément **heroData** du tableau **SuperHerosData** comme argument et retourne une nouvelle instance de la classe **SuperHero**, initialisée avec **heroData.id**, **heroData.name**, **heroData['id-api']** et **heroData.slug**.

Voici une explication détaillée de chaque partie de cette ligne :

- **SuperHerosData.map(...)** : cela appelle la méthode **map** sur **SuperHerosData**.
- **(heroData: any) => ...** : ceci définit une fonction fléchée qui prend un argument **heroData**. **any** est le type de cet argument, signifiant qu'il peut être n'importe quel type en TypeScript.
- **new SuperHero(heroData.id, heroData.name, heroData['id-api'], heroData.slug)** : ceci crée une nouvelle instance de la classe **SuperHero** en utilisant les propriétés de **heroData**.
- **const heroesFromData = ...** : ceci assigne le résultat de l'opération **map** à la nouvelle constante **heroesFromData**.

Ensuite, **setHeroes** est appelé avec le tableau nouvellement créé pour mettre à jour l'état **heroes**. Le tableau vide en deuxième argument signifie que cet effet ne s'exécute qu'une fois, lors du montage du composant.

- **return (...)**

Le code dans la fonction **return** est ce qui est effectivement rendu par le composant. Ici, il affiche un titre, le nombre de super-héros chargés à partir du fichier JSON, puis une liste de ces super-héros. Chaque super-héros est affiché dans un bloc **<div>** avec son nom, son id, son idApi, son slug et une image. Notez que la fonction **map** est utilisée pour créer un nouvel élément pour chaque super-héros dans l'état **heroes**.

- **export default App;**

Cette ligne exporte le composant **App** en tant qu'exportation par défaut du module.

4. Clic.tsx

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

const App: FunctionComponent = () => {
  const [name, setName] = useState('World');
  return (
    <div>
      <h1>Avec React : Hello {name}!</h1>
      <button onClick={() => setName(name + "!")}>
        Ajoutez un !
      </button>
      <button onClick={() => setName("World")}>
        Remettre World
      </button>
    </div>
  )
}

export default App;
```


Ce code définit un composant React, **App**, qui affiche un message de bienvenue et deux boutons. Le composant utilise l'état **name** pour stocker le texte affiché dans le message de bienvenue. Initialement, **name** est défini comme 'World', donc le message affiché est "Avec React : Hello World!". Le premier bouton, lorsqu'il est cliqué, ajoute un "!" à la fin de **name** et met à jour le message affiché. Le deuxième bouton réinitialise **name** à "World", ce qui réinitialise le message à son état original. Ces modifications d'état sont effectuées en utilisant la fonction **setName**, qui est liée à l'état **name** via le Hook **useState** de React

5. SuperHero.ts

Les fichiers **.ts** et **.tsx** sont tous deux des fichiers TypeScript, mais il y a une différence clé entre eux :

- Les fichiers **.ts** sont des fichiers TypeScript standard. Ils peuvent contenir n'importe quel code TypeScript valide.
- Les fichiers **.tsx** sont des fichiers TypeScript qui peuvent également contenir du JSX. JSX est une extension de syntaxe pour JavaScript qui ressemble à XML ou HTML. Il est couramment utilisé avec des bibliothèques comme React pour décrire la structure de l'interface utilisateur.

Donc, la principale différence entre **.ts** et **.tsx** est que **.tsx** peut contenir du JSX, tandis que **.ts** ne le peut pas. Si vous travaillez sur un projet React avec TypeScript, vous utiliserez probablement des fichiers **.tsx** pour vos composants React afin de pouvoir utiliser la syntaxe JSX.

```
// SuperHeros.ts

// On définit une classe 'SuperHero' qui va représenter chaque super-héros de notre fichier JSON.
export class SuperHero {
  // 'id' est un nombre qui représente l'identifiant unique de chaque super-héros.
  id: number;

  // 'name' est une chaîne de caractères qui représente le nom du super-héros.
  name: string;

  // 'idApi' est un nombre qui représente l'identifiant unique du super-héros dans l'API.
  idApi: number;

  // 'slug' est une chaîne de caractères qui représente le slug du super-héros.
  // Il est optionnel car certains super-héros dans notre fichier JSON n'ont pas de slug.
  slug: string | undefined;

  // Le constructeur est une méthode spéciale qui est utilisée pour créer et initialiser un objet créé à partir d'une classe.
  // Ici, le constructeur prend l'id, le nom et l'idApi comme arguments obligatoires, et le slug comme argument optionnel.
  constructor(id: number, name: string, idApi: number, slug?: string) {
    // On attribue les valeurs reçues par le constructeur à nos attributs de classe.
    this.id = id;
```



```

    this.name = name;
    this.idApi = idApi;
    this.slug = slug;
  }
}

```

Dans cette classe **SuperHero**, il y a quatre propriétés : **id**, **name**, **idApi**, et **slug**. Chacune de ces propriétés a un type spécifié : **id** et **idApi** sont des nombres (**number**), **name** et **slug** sont des chaînes de caractères (**string**). Le type **slug** est légèrement différent car il peut aussi être **undefined**, indiquant que **slug** est une propriété optionnelle.

Ensuite, le constructeur est une méthode spéciale dans la classe, utilisée pour créer et initialiser un nouvel objet. Quand un nouvel objet **SuperHero** est créé, le constructeur est appelé avec les valeurs **id**, **name**, **idApi**, et **slug** fournies, qui sont ensuite assignées aux propriétés correspondantes de l'objet. Notez que le paramètre **slug** dans le constructeur est marqué comme optionnel avec le symbole **?**, signifiant qu'il n'a pas besoin d'être fourni lors de la création d'un nouvel objet **SuperHero**.

Enfin, cette classe est exportée avec le mot-clé **export**, ce qui signifie qu'elle peut être importée et utilisée dans d'autres fichiers TypeScript dans le même projet.

6. *Index.stx*

```

import React from "react";
import ReactDOM from "react-dom";
import App from './App'

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

```

Ce fichier est généralement le point d'entrée principal d'une application React.

Tout d'abord, il importe les bibliothèques nécessaires. **React** est importé de "react", qui est la bibliothèque principale de React. **ReactDOM** est importé de "react-dom", qui est une bibliothèque permettant de manipuler le DOM (Document Object Model) avec React. Enfin, le composant **App** est importé du fichier **App**.

Ensuite, la méthode **ReactDOM.render** est utilisée pour rendre le composant **App** dans l'élément DOM avec l'id 'root'. L'appel à **ReactDOM.render** prend deux arguments : le premier est ce que vous voulez rendre (dans ce cas, le composant **App**), et le deuxième est où vous voulez le rendre (dans ce cas, l'élément DOM avec l'id 'root').

La ligne **ReactDOM.render(<App />, document.getElementById('root'))** signifie donc "Rendez le composant **App** à l'intérieur de l'élément DOM ayant l'id 'root'". En pratique, cela signifie que votre application React sera affichée à l'intérieur de cet élément 'root'.

VIII. Suite du TD/TP

1. Filtrage des super-héros :

Ajoutez un champ de recherche qui permet à l'utilisateur de filtrer la liste des super-héros par nom. Vous devrez ajouter un nouvel état pour le texte de recherche et utiliser cette valeur pour filtrer la liste des super-héros dans le rendu.

Pour cela :

Étape N°26 : Ajouter un état pour le texte de recherche

Vous devrez ajouter un nouvel état pour stocker le texte de recherche actuel. Vous pouvez le faire en utilisant le Hook **useState** de React et l'état initiale est une chaîne vide.

Étape N°27 : Ajouter un champ de recherche à l'interface utilisateur

Vous devez ensuite ajouter un champ de recherche à votre interface utilisateur. Ce champ de recherche doit mettre à jour le texte de recherche lorsque l'utilisateur tape. (utilisez `event.target.value`)

Étape N°28 : Filtrer la liste des super-héros en fonction du texte de recherche

Maintenant que vous avez un moyen de saisir et de stocker le texte de recherche, vous pouvez l'utiliser pour filtrer la liste des super-héros (nom = **filteredHeroes**)

Vous pouvez le faire en utilisant la méthode **filter** de JavaScript sur votre tableau de super-héros.

filter est une méthode qui crée un nouveau tableau avec tous les éléments qui passent le test implémenté par la fonction fournie. Dans ce cas, le test est de savoir si le nom du super-héros inclut le texte de recherche. Vous pouvez convertir à la fois le nom et le texte de recherche en minuscules pour faire une comparaison insensible à la casse à l'aide de la méthode `toLowerCase()`

Étape N°29 : Afficher la liste filtrée des super-héros

Enfin, vous devez modifier le rendu de votre liste de super-héros pour utiliser **filteredHeroes** au lieu de **heroes**.

Vous avez maintenant une application qui permet à l'utilisateur de filtrer la liste des super-héros en

tapant dans un champ de recherche.

Rechercher un super-héros

2. Trier les super-héros :

Ajoutez une fonctionnalité qui permet à l'utilisateur de trier la liste des super-héros par nom ou par id. Vous pourriez implémenter cela avec des boutons ou un menu déroulant qui change l'ordre de la liste.

Pour ajouter cette fonctionnalité de tri des super-héros, vous pouvez suivre ces étapes :

Étape N°30 : Ajouter un nouvel état pour le critère de tri

Ajoutez un nouvel état à l'aide du Hook **useState** pour stocker le critère de tri actuel. Par exemple, vous pouvez le nommer **sortKey** et utiliser "name" comme valeur par défaut :

Étape N°31 : Ajouter une fonction pour trier les super-héros

Expliquez ce code, puis mettez le dans le fichier :

```
const sortedHeroes = filteredHeroes.sort((a: SuperHero, b: SuperHero) => {
  if(sortOption === "name") {
    return a.name.localeCompare(b.name);
  } else {
    return a.id - b.id;
  }
});
```

Étape N°32 : Ajouter une interface utilisateur pour changer la clé de tri

Ajoutez des boutons ou un menu déroulant qui permet à l'utilisateur de changer la clé de tri. Lorsqu'un utilisateur clique sur un bouton ou sélectionne une option dans le menu déroulant, utilisez la fonction **setSortKey** pour mettre à jour la clé de tri :

Étape N°33 : Utiliser le tableau trié dans le rendu

Remplacez l'utilisation de **heroes** dans le rendu par **sortedHeroes** pour afficher la liste des super-héros triée :

Avec ces changements, votre application doit maintenant permettre à l'utilisateur de trier la liste des

Super Heroes App

Rechercher un super-héros Trier par nom

super-héros par nom ou par ID.

3. Affichage détaillé :

Actuellement, l'application affiche tous les détails de chaque super-héros directement dans la liste. Modifiez l'application pour n'afficher que le nom de chaque super-héros dans la liste, et ajoutez une fonctionnalité qui permet à l'utilisateur de cliquer sur un super-héros pour voir plus de détails. Vous pourriez implémenter cela en ajoutant un nouvel état qui traque le super-héros actuellement sélectionné.

Étape N°34 : Ajoutez un nouvel état pour le super-héros sélectionné

Vous aurez besoin d'un état pour garder une trace du super-héros actuellement sélectionné. Vous pouvez l'ajouter à votre composant **App** comme ceci :

```
const [selectedHero, setSelectedHero] = useState<SuperHero | null>(null); // Nouvel état
pour suivre le super-héros sélectionné
```

Étape N°35 : Modifier la fonction de rendu pour n'afficher que le nom des super-héros

Dans votre fonction de rendu, modifiez la carte de super-héros pour n'afficher que le nom du super-héros. Ajoutez également un gestionnaire de clic qui appelle **setSelectedHero** avec le super-héros cliqué :

Étape N°36 : Ajouter un rendu conditionnel pour l'affichage détaillé

Après la liste des super-héros, ajoutez un rendu conditionnel qui affiche les détails du super-héros sélectionné si **selectedHero** n'est pas **null**.

Dans JavaScript et TypeScript, **&&** est un opérateur logique ET. Il est souvent utilisé dans React pour conditionnellement rendre (afficher) des composants ou des éléments.

Lorsqu'on écrit **{selectedHero && (/* du JSX ici */)}**, cela signifie "si **selectedHero** est vrai (ou truthy), alors rendre le JSX qui suit".

Étape N°37 : Testez votre application

Assurez-vous de tester votre application après avoir apporté ces modifications. Vérifiez que la liste des super-héros affiche uniquement les noms, que cliquer sur un nom affiche les détails du super-héros correspondant et que le bouton "Retour à la liste" vous ramène à la vue de la liste.