Maîtrisez un framework JavaScript parmi les plus utilisés dans l'industrie

SÉBASTIEN CASTIEL

DES APPLICATIONS MODERNES AVEC REACT

- Créez des applications React complètes
- Développez des applications web et mobiles
- Écrivez des composants réutilisables

Introduction

Lorsqu'en 2013 Facebook a annoncé la sortie de React, on peut dire que certains l'ont détesté, comme d'autres ont y vu un fantastique potentiel. En effet React n'était pas annoncé comme un nouveau framework JavaScript comme AngularJS ou Ember, mais comme une bibliothèque permettant de générer des composants dans le DOM. Il avait ainsi un côté minimaliste qui le rendait apprécié de beaucoup de développeurs.

Mais React annonçait aussi l'arrivé de JSX, qui comme nous le verrons permet en quelque sorte de décrire à l'aide d'une syntaxe proche de HTML comment un composant graphique doit être rendu. Directement dans du code JavaScript. Il était donc nécessaire de passer par une phase de transpilation permettant à partir de JavaScript + JSX de générer du JavaScript standard. Et cela n'a pas plu à tout le monde (et ne plaît toujours pas à tout le monde d'ailleurs).

D'autres dont je fais partie ont été séduits par l'opportunité d'écrire des composants réutilisables, gérant chacun les comportements qui leur sont associés. Séduits également par un écosystème qui s'est créé autour de React dès sa sortie : une multitude de composants disponibles, une communauté grandissante, mais surtout des mises à jour fréquentes de la part de Facebook.

Si vous vous apprêtez à lire ce livre c'est sans doute que cela fait quelque temps que vous entendez parler de React et que vous souhaitez mettre les mains dedans. J'espère d'abord par ce livre vous donner les éléments clés qui vous permettront de réaliser vos premières applications en React. Mais également vous faire découvrir une partie des nombreux outils fréquemment utilisés avec lui.

À qui s'adresse ce livre ?

À toute personne curieuse de découvrir React, ou à toute personne ayant suivi un tutoriel React et souhaitant aller plus loin. S'il n'est pas nécessaire d'avoir expérimenté React ou un framework JavaScript avant, il est toutefois préférable

d'avoir un minimum de connaissance du langage JavaScript, et si possible des nouveautés apportées par ES2015 : classes, *arrow functions*, etc. En effet React est beaucoup plus agréable à utiliser avec ces fonctionnalités.

Notez que ce livre ne couvre pas la partie serveur d'une application : API Rest, base de données, etc. Pour cela je vous encourage à vous documenter par exemple sur Node.js si vous aimez JavaScript. Ne seront pas abordées en profondeur non plus les problématiques de déploiement d'une application React, bien que ce point soit brièvement présenté en annexe B.

Que trouverez-vous dans ce livre ?

Le premier chapitre couvrira ce que React propose de base (sans bibliothèque externe ou presque). Nous écrirons nos premiers composants, les ferons communiquer entre eux, les stylerons avec du CSS… Il correspond également à ce qui est disponible dans React depuis le début, c'est-à-dire ce qui est le plus utilisé. Notamment, les dernières nouveautés de React 16 n'y seront pas présentées, mais rassurez-vous, le chapitre 5 en abordera certaines.

Le second chapitre vous fera découvrir Redux, qui permet de structurer une application un peu plus conséquente afin de la rendre plus facile à maintenir et faire évoluer. Et le troisième chapitre sera consacré à React Native. Nous verrons comment grâce à un outil formidable l'écriture d'applications mobiles n'est pas plus complexe que celle d'applications web.

Le quatrième chapitre abordera des notions plus avancés comme le routage ou l'utilisation de Firebase pour l'authentification ou le stockage de données distantes. Il s'agit sans doute du chapitre le plus technique.

Enfin le chapitre 5 présente quelques manières de rendre votre code React plus facile à maintenir et réutiliser, et cela passe par quelques possibilités introduites récemment dans React, comme les *hooks*.

La conclusion du livre propose des pistes pour aller un peu plus loin, par exemple à l'aide de bibliothèques de composants bien connues des développeurs React, ou encore comment générer un site statique à l'aide de React.

À la fin du livre vous trouverez également en annexes la présentation d'outils comme les React Dev Tools facilitant le développement d'applications React, ou encore des indications sur comment déployer une application React.

À propos des exemples

Comme tout livre consacré à du développement ce livre est riche en exemples. Dans la plupart des chapitres, un exemple complet nous guidera dans les notions à découvrir. Je vous encourage à reproduire les exemples vous-mêmes au fur et à mesure de la lecture afin de bien appréhender les concepts. De plus si une erreur survient, il sera sans doute plus facile de savoir d'où elle vient si vous avez au même moment ce livre ouvert à la bonne page.

L'intégralité des exemples du livre sont également disponibles sur un dépôt GitHub dédié [1]. Afin de les exécuter, la procédure est sauf contre-indication toujours la même, il suffit d'installer les dépendances avec la commande `yarn`, puis de lancer l'application avec `yarn start`, le tout dans le répertoire de l'exemple que vous souhaitez lancer.

Rester informé et en savoir plus

Le monde de React et du développement web en général évolue très vite, bien trop vite en tout cas pour qu'un livre puisse suivre le rythme. Aussi voici quelques sources pour vous tenir au courant de ce qui se passe :

— Le site de React bien évidemment [2], et notamment sa section Blog, vous informera des nouvelles versions contenant des évolutions majeures.
— Le subreddit dédié à React [3] référence probablement tout bon article lié à React. Même si comme tout subreddit il nécessite de faire un peu le tri entre les informations qui y circulent...
— Le compte Twitter @reactjs [4] officiel reprend les annonces du site, mais ceux des développeurs Dan Abramov (@dan_abramov [5]), Sophie Alpert (@sophiebits [6]) et Andrew Clark (@acdlite [7]) vous donneront accès à l'actualité la plus récente.
— Le récent blog de Dan Abramov, *Overreacted* [8], est une mine d'informations pour en savoir plus sur le fonctionnement interne de React. Attendez-vous à des sujets techniques !

1. https://github.com/scastiel/livre-react-exemples/
2. https://reactjs.org
3. https://www.reddit.com/r/ReactJS
4. https://twitter.com/reactjs
5. https://twitter.com/dan_abramov
6. https://twitter.com/sophiebits
7. https://twitter.com/acdlite
8. https://overreacted.io/

— La section React du site *Dev.to*[9] propose beaucoup de tutoriels de tout niveau pour approfondir vos connaissances et donner des astuces.
— En français, le site *Putain de code*[10] propose beaucoup d'articles sur le développement front-end et notamment sur React.

Enfin, sachez que j'ai lancé en même temps que j'écrivais ce livre le site *MasterReact.io*[11] dont l'ambition est de fournir régulièrement des articles en complément au livre. Vous y trouverez des articles allant du niveau débutant à un niveau avancé, ainsi que quelques nouvelles du livre.

À présent plongeons au cœur du sujet, c'est parti pour l'écriture de votre première application React !

Bonne lecture.

9. https://dev.to/t/react
10. https://putaindecode.io/
11. https://www.masterreact.io

Chapitre 1

Découverte de React

Lançons-nous dès maintenant dans la mise en place de notre première application React. Tout d'abord : de quels outils avons-nous besoin ?

1.1 Installation des outils requis

1.1.1 NodeJS

Une application React n'a pas besoin de NodeJS pour fonctionner, mais pour générer une application interprétable par un navigateur à partir de plusieurs fichiers organisés, utilisant une syntaxe propre à React, NodeJS est sinon indispensable du moins fortement pratique !

La manière la plus simple à ce jour d'installer NodeJS est selon moi d'utiliser *NVM (Node version manager)*[1]. Mais vous pouvez aussi utiliser la distribution Node associée à votre système d'exploitation ou bien le programme d'installation officiel.

Avec Node sera automatiquement installé le gestionnaire de paquet *NPM*, mais pour ma part je préfère son alternative *Yarn*, que vous pouvez installer avec la commande `npm install -g yarn`. Lorsque je décrirai des commandes dans ce livre j'utiliserai Yarn, mais tout est également faisable avec NPM si vous préférez.

1. https://github.com/creationix/nvm

1.1.2 Un éditeur de texte

Tout éditeur de texte peut être utilisé bien évidemment, du plus simple (bloc-notes, VI) à l'IDE le plus complexe comme WebStorm ou Eclipse. Pour ma part je pense que le meilleur compromis est d'utiliser un éditeur avancé mais léger, et mon choix s'est porté sur *VS Code*[2] de Microsoft.

Il est disponible sur les principaux systèmes d'exploitation, gère nativement la syntaxe JSX pour React, et propose pour les utilisateurs avancés des extensions liées à des outils facilitant le développement : ESLint, Prettier, etc.

1.1.3 Et ensuite ?

Bien évidemment vous aurez besoin d'un navigateur web. Tout navigateur peut convenir, je recommanderais néanmoins d'utiliser Firefox ou Chrome en raison des outils de développement qu'ils proposent. De plus vous pouvez dés maintenant installer l'extension React Dev Tools[3] qui vous aidera à débugguer vos applications. Pour cela vous pourrez aller voir l'annexe A qui vous présentera les possibilités offertes par cet outil.

1.2 Création du premier projet

Une fois que les outils nécessaires sont installés, commençons sans plus tarder. Créons un dossier, par exemple *hello-react*, et ouvrons un terminal dans ce dossier. Sans rentrer dans les détails pour le moment, sachez que React utilise une syntaxe qui lui est propre pour écrire les composants (un ajout au langage JavaScript), et donc qu'il est nécessaire de passer par une phase de compilation (en fait, de transpilation), pour obtenir un code JavaScript que les navigateurs savent interpréter.

De nombreux outils existent afin de faire cette transformation et au passage de permettre par exemple de profiter des dernières nouveautés de JavaScript non prises en charges par tous les navigateurs, ou encore de découper notre application en fichiers comme bon nous semble. Parmi les plus connus, citons notamment Webpack[4] très utilisé pour des très gros projets pour toutes les options et plugins qu'il propose.

Create-React-App[5] est également de plus en plus utilisé et permet de générer le squelette d'une application React en une seule commande.

2. https://code.visualstudio.com
3. https://github.com/facebook/react-devtools
4. https://webpack.js.org
5. https://github.com/facebook/create-react-app

Pour nos exemples, j'ai décidé d'utiliser un outil plus minimaliste : Parcel [6]. Pour l'installer nous utiliserons *Yarn* (ou *NPM*). Initialisons donc notre projet, et installons Parcel et les bibliothèques et outils qui nous seront utiles :

```
$ yarn init -y
$ yarn add --dev parcel-bundler babel-preset-env babel-preset-react
$ yarn add react react-dom
```

Une fois tout cela installé, nous allons modifier le fichier *package.json* (généré par Yarn), afin d'y ajouter les deux sections suivantes (avant l'accolade fermante `}` à la fin) :

```
{
  // ...
  "scripts": {
    "start": "parcel public/index.html"
  },
  "babel": {
    "presets": ["env", "react"]
  }
  // ...
}
```

La section `scripts` va nous permettre de définir ce qui doit être fait lorsque nous lançons la commande `yarn start` ; ici nous lançons donc `parcel`. Et la section `babel` nous permet d'indiquer que notre code utilise du JSX, et qu'il faut donc utiliser le plugin Babel permettant de gérer cette syntaxe pour la convertir en code JavaScript standard.

Il ne reste qu'à écrire le code ! Créons deux dossiers *public* et *src*, et deux fichiers *public/index.html* et *src/index.js* :

```
// src/index.js
import React from 'react'
import ReactDOM from 'react-dom'

const content = <div>Hello!</div>
const div = document.getElementById('app')
ReactDOM.render(content, div)
```

```
<!-- index.html -->
<div id="app" />
<script src="../src/index.js"></script>
```

Avant d'entrer dans l'explication de ce code, essayons de lancer notre projet avec la commande `yarn start`. Si tout va bien, votre console devrait afficher quelque chose

6. https://parceljs.org/

comme ceci :

```
yarn run v1.5.1
$ parcel public/index.html
Server running at http://localhost:1234
Built in 242ms.
```

Et en ouvrant votre navigateur à l'URL `http://localhost:1234`, vous devriez voir le texte « Hello ! ». Voilà, félicitations, c'est votre première application React :).

1.2.1 Explication du code

Commençons par le fichier *index.html*. Vous pouvez voir qu'il contient deux choses :

- une `div` vide qui a pour ID « app ». C'est l'élément de la page dans lequel nous allons injecter notre application React. Il n'y a aucune contrainte sur cet élément : ce peut être n'importe quel élément HTML tant que vous pouvez le retrouver en JavaScript (il lui faut donc généralement un ID ou une classe, à moins que ce ne soit le seul élément de la page) ;
- une balise `script` qui charge notre fichier *index.js*.

En quelque sorte ce fichier HTML est le point d'entrée de notre application puisque c'est lui qui est affiché lorsque l'utilisateur ouvre l'application dans son navigateur. Si vous souhaitez donner un titre à la page ou y ajouter par exemple des scripts supplémentaires (Google Analytics, etc.), c'est ici qu'il faut les mettre (comme dans un fichier HTML classique).

Notez que nous faisons référence à notre fichier JavaScript grâce à son chemin relatif (*../src/index.js*) ; c'est Parcel qui se chargera notamment de remplacer dans le fichier HTML généré ce chemin par l'URL du fichier JavaScript, comme nous allons le voir un peu plus loin.

Passons ensuite au fichier *index.js* :

```
import React from 'react'
import ReactDOM from 'react-dom'
```

Tout d'abord nous importons `react` et `react-dom`. Vous pouvez avoir l'impression qu'importer React ne sert à rien ici étant donné que nulle part nous n'utilisons la variable `React`. Nous allons voir dans quelques instants que l'usage de React est en fait masqué par le fait d'utiliser du JSX.

Nous importons ensuite `react-dom`, qui va nous donner accès à la méthode `render`. Sans rentrer trop dans le détail pour le moment, sachez que si historiquement React était fait pour le web, React Native s'est progressivement imposé et l'équipe en

charge de React a décidé de conserver dans React uniquement ce qui était générique aux deux librairies (web et natif, le cœur de React donc), et d'extraire dans *React DOM* ce qui concernait le web.

```
const content = <div>Hello!</div>
const div = document.getElementById('app')
ReactDOM.render(content, div)
```

Nous créons ensuite le contenu de notre application à l'aide de la syntaxe JSX. Enfin nous y arrivons ; quelle est donc cette syntaxe qui ressemble comme deux gouttes d'eau à du HTML ? En réalité ce n'est pas du HTML, mais plutôt une manière élégante de créer des nœuds dans le DOM.

Pour simplifier, imaginez que cela revient en fait à écrire ceci :

```
const content = document.createElement('DIV')
content.innerHTML = 'Hello!'
const div = document.getElementById('app')
content.appendTo(div)
```

En réalité React gère le JSX bien mieux que cela (en gardant en mémoire un DOM virtuel notamment), mais l'idée reste la même. Nous allons voir plus loin quelques différences entre le JSX et le HTML au niveau de la syntaxe.

Pour ce qui est du reste du fichier, nous récupérons la `div` principale de notre application grâce à son ID, puis nous demandons à `ReactDOM.render` de générer le rendu de l'application dans cette `div`.

Afin d'en apprendre un peu plus sur le JSX, ajoutons un tout petit peu de logique à notre application.

1.2.2 Composants et propriétés

Voici la nouvelle version du fichier *index.js* :

```
import React from 'react'
import ReactDOM from 'react-dom'

const Greetings = props => {
  return (
    <span>
      Bonjour <strong>{props.name}</strong> !
    </span>
  )
}

const App = () => <Greetings name="Sébastien" />
```

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

Cette fois-ci nous créons deux fonctions `Greetings` et `App`. Ces deux fonctions renvoient du JSX : ce sont des *composants* React. En effet il s'agit de la première des deux manières classiques de déclarer un composant.

Puis afin d'utiliser un composant déjà créé, on utilise la même syntaxe que s'il s'agissait d'un élément HTML. C'est ce qui est fait dans le composant `App`, où nous appelons le composant `Greetings`.

Vous avez remarqué que la fonction `Greetings` prend un paramètres `props` : il s'agit d'un objet contenant les paramètres qui sont envoyés au composant. Ici nous appelons `Greetings` ainsi : `<Greetings name="Sébastien"/>`, notre paramètre `props` vaudra donc `{ name: 'Sébastien' }`. D'où l'utilisation de `props.name`.

Nous pourrions rendre le code du composant `Greetings` encore plus concis en utilisant l'interpolation des paramètres de JavaScript :

```
const Greetings = ({ name }) => (
  <span>
    Bonjour <strong>{name}</strong> !
  </span>
)
```

Dernière chose : pour placer le contenu d'une variable dans du JSX, il suffit de l'entourer d'accolades : `Bonjour {name} !`. Cela vaut aussi pour les propriétés des composants, nous aurions pu écrire :

```
const App = () => {
  const name = 'Sébastien'
  return <Greetings name={name} />
}
```

Vous l'avez compris, l'application affiche désormais « Bonjour Sébastien ». Cette première application est terminée, mais vous vous demandez maintenant comment la rendre disponible sur Internet. Bon peut-être pas celle-ci qui ne fait rien d'intéressant, mais sans doute une de vos prochaines réalisations :). Pour cela je vous renvoie vers l'annexe B à la fin du livre qui vous guidera dans la marche à suivre.

À présent attardons-nous le temps d'une section sur le langage JSX. Il est très pratique à utiliser, mais il comporte son lot de subtilités et pièges.

1.3 Le langage JSX

Au fur et à mesure vous verrez que le JSX est un langage très intuitif à utiliser. Voici deux propriétés de base pour ce qui est des balises utilisées :

— Toute balise commençant par une minuscule (`div`, `span`, `label`, etc.) est réservée aux éléments HTML. Ils sont déclarés par React DOM, et vous obtiendrez une erreur si vous utilisez un élément inexistant.
— Toute balise commençant par une majuscule (`Greetings`, `App`, etc.) doit être déclarée explicitement, ce doit donc être un élément du scope courant : fonction déjà déclarée, composant importé d'une bibliothèque ou d'un autre fichier... Cela veut aussi dire que tout composant que vous créerez devra avoir son nom commençant par une majuscule.

Pour ce qui est des propriétés :

— Une chaîne de caractères constante peut être passée comme en HTML, entre simples ou doubles quotes : `name="Sébastien"` ou `name='Sébastien'`.
— Toute valeur (code JavaScript) peut être passée entre accolades : `prop={1}`, `prop={true}`, `prop={name}`, `prop={'Sébastien'}` (ce dernier exemple étant exactement équivalent à `prop="Sébastien"`). Pour les objets, tableaux, fonctions, même principe : `prop={{ a: 1, b: 2 }}`, `prop={['a', 'b']}`, `prop={x => 2 * x}`.

Pour les composants comme pour les propriétés, les régles de nommage sont les mêmes que pour une variable JavaScript : caractères alphanumériques, underscore, etc. Pas de tiret par exemple, ni d'espace ou d'autres caractères spéciaux. De plus les propriétés sont sensibles à la casse.

Point important : la plupart du temps pour spécifier un attribut HTML, la propriété JSX a le même nom (pour `id` par exemple). Ce n'est cependant pas toujours le cas : l'exemple plus courant étant l'attribut `class` qui devient `className` en JSX, pour qu'il n'y ait pas de confusion avec le mot-clé `class` de JavaScript. Vous vous ferez souvent avoir au début, heureusement React vous affichera un petit avertissement dans la console de votre navigateur. Le détail des éléments concernés est bien évidemment disponible dans la documentation de React [7].

Pour placer du contenu dynamique dans le corps même d'un élément JSX, les règles sont en fait les mêmes que pour les propriétés. Vous pouvez donc écrire par exemple :

```
<span>Carrés :{[1, 2, 3, 4, 5].map(x => x * x).join(', ')}</span>
```

Cependant il n'est possible que de passer des *expressions* dans du JSX. Cela exclut donc de mettre des `if` ou des `for`. Pourtant il serait très tentant d'écrire :

7. https://reactjs.org/docs/dom-elements.html

```
<span>
  {
    // Cela ne compilera pas!
    if (test) { return 'Oui' }
    else { return 'Non' }
  }
</span>
```

Heureusement il existe une alternative très intéressante : l'utilisation de l'opérateur ternaire `? :` :

```
<span>{test ? 'Oui' : 'Non'}</span>
```

Si vous souhaitez ne rien afficher dans le cas où une condition est fausse, vous pouvez également utiliser l'opérateur `&&` :

```
<span>{test && 'Oui'}</span>
```

Dans le cas où les conditions ou résultats sont plus complexes, ou que l'on a plus de deux cas à gérer, il est également possible (et même recommandé pour la lisibilité) de passer par une fonction intermédiaire :

```
const renderResult = () => {
  if (condition1) {
    return 'Oui'
  } else if (condition2) {
    return <em>Peut-être...</em>
  } else {
    return 'Non'
  }
}
return <span>Resultat : {renderResult()}</span>
```

Car oui, il n'y a pas que les composants qui peuvent renvoyer du JSX. N'importe quelle fonction en a le droit.

Qu'en est-il des boucles ? Et bien non, pas de boucle `for` ou `while` dans du JSX, mais il est par contre très pratique d'utiliser la méthode `map` des tableaux. Supposons par exemple que l'on souhaite afficher une liste de fruits :

```
const fruits = ['Pomme', 'Pêche', 'Poire', 'Abricot']
```

Nous pouvons commencer par écrire une fonction qui va nous générer le JSX pour un fruit donné :

```
const renderFruit = fruit => <li>{fruit}</li>
```

Puis utiliser `map` pour obtenir un tableau contenant le rendu pour chaque fruit :

```
const renderedFruits = fruits.map(renderFruit)
```

Et comme React est très bien fait, il nous permet d'afficher directement un tableau de composants. Il va simplement afficher les composants les uns après les autres comme on pourrait s'y attendre.

```
return <ul>{renderedFruits}</ul>
```

Tout devrait bien s'afficher, néanmoins React va vous afficher une erreur dans la console, comme *Warning : Each child in an array or iterator should have a unique « key » prop.* (le message exact peut avoir changé depuis l'écriture de ce chapitre). En effet lorsque React affiche un tableau de composant, il a besoin d'une propriété `key` sur chacun de ses composants, lui permettant lorsqu'il doit réafficher le tableau (avec de nouveaux éléments ou un nouvel ordre), de savoir quel composant affiché correspond à quel élément du tableau.

Modifions donc notre fonction `renderFruit` afin d'ajouter la propriété `key` :

```
const renderFruit = <li key={fruit}>{fruit}</li>
```

Nous pouvons utiliser n'importe quelle valeur comme `key`, du moment que celle-ci est unique dans le tableau. Idéalement une clé donnée doit identifier un élément donné du tableau. Ce peut donc être un ID par exemple, et en dernier recours l'index dans le tableau (à éviter cependant car cela ne sera pas optimisé dans le cas où des éléments du tableau sont réordonnés).

En plus concis, voici à quoi peut ressembler notre composant affichant des fruits :

```
const Fruits = ({ fruits }) => (
  <ul>
    {fruits.map(fruit => <li key={fruit}>{fruit}</li>)}
  </ul>
)

// Utilisation :
return <Fruits fruits={['Pomme', 'Pêche', 'Poire', 'Abricot']} />
```

Après avoir créé nos premiers composants, nous allons voir dans la section suivante quelques unes des possibilités offertes par React afin de leur ajouter de la logique. Nous partirons d'un exemple simple que nous rafinerons au fur et à mesure. Il s'agira d'une application de gestion de liste de tâches, permettant d'ajouter des nouvelles tâches et de les marquer comme effectuées.

1.4 Un composant par fichier

Dans les exemples que nous venons de voir, il n'y avait qu'un seul fichier JavaScript où l'on déclarait nos composants. Pour des raisons évidentes il serait intéressant de séparer nos composants en plusieurs fichiers.

Commençons à concevoir notre application. Dans sa première version, nous définirons trois composants :

— un composant `Task` permettant d'afficher une tâche ;
— un autre composant `TaskList` permettant d'afficher une liste de tâches (nous utiliserons le premier composant `Task`) ;
— et enfin notre composant `App` qui affichera un titre ainsi que la liste de tâches via le composant `TaskList`.

De manière assez intuitive nous créeons donc trois fichiers, un pour chaque composant. Commençons avec le composant `Task` :

```
// src/Task.js
import React from 'react'

const Task = ({ task }) => <span>{task.label}</span>

export default Task
```

Rien de bien nouveau ici : nous créons un composant `Task`, dont nous extrairons des propriétés qui lui sont passées un paramètre `task` qui contiendra les informations sur la tâche à afficher. Cette tâche aura un attribut `label`, que nous afficherons dans une balise `span`.

Comme notre fichier contient notre composant, nous souhaitons l'importer dans d'autres fichiers ; il est donc exporté par l'instruction `export default Task`.

Le composant `TaskList` est un brin plus complexe, mais ne comporte pas de nouveautés par rapport à ce que nous avons vu précédemment :

```
// src/TaskList.js
import React from 'react'
import Task from './Task'

const TaskList = ({ tasks }) => (
  <ul>
    {tasks.map(task => (
      <li key={task.id}>
        <Task task={task} />
      </li>
    ))}
  </ul>
)
```

```
export default TaskList
```

En propriété de ce composant nous attendons un tableau de tâches `tasks`. Pour chacune de ces tâches nous affichons un élément `li`, au sein duquel nous appelons notre composant `Task` en lui passant la tâche en paramètre.

Notez que lorsque nous créons un élément `li` pour chaque tâche (grâce à `map`), nous fournissons à l'attribut `key` l'ID de la tâche, unique dans le tableau, comme cela est demandé par React.

Nous aurions pu intégrer l'élément `li` dans le composant `Task`, mais d'un point de vue conception il me semble plus pertinent de rendre le composant `Task` le plus générique possible ; peut-être aurons-nous à un moment l'envie de l'afficher ailleurs que dans une liste. Mais techniquement cela ne poserait aucun problème, nous aurions écrit :

```
tasks.map(task => <Task key={task.id} task={task} />)
```

Dans ce cas, il est indispensable que l'attribut `key` soit spécifié ici ; il ne peut pas l'être dans le composant `Task`. En effet React en a besoin avant même de générer le contenu de `Task`. D'ailleurs autre point intéressant, la propriété `key` n'est pas transmise au composant `Task`. C'est un attribut un peu spécial et traité uniquement par React.

Enfin notre dernier composant `App` :

```
// src/App.js
import React from 'react'
import TaskList from './TaskList'

const App = () => {
  const tasks = [
    { id: 1, label: 'Acheter du lait', isDone: true },
    { id: 2, label: 'Prendre des vacances', isDone: false }
  ]
  return (
    <div>
      <h1>Tâches</h1>
      <TaskList tasks={tasks} />
    </div>
  )
}

export default App
```

Rien de nouveau ici non plus, ce composant déclare un tableau de tâches (nos données de test), et affiche un titre ainsi que le composant `TaskList` en lui passant

en paramètre le tableau `tasks`. Il s'agit du composant principal de l'application.

Il ne reste plus qu'à utiliser ce composant dans le fichier *index.js* :

```
// src/index.js
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App'

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

Pour ce qui est du reste du code, nous reprendrons le même squelette que pour la première application que nous avons créée.

Lorsque nous lançons l'application, nous pouvons admirer notre liste de tâches ! Si vous ouvrez les outils développement de votre navigateur et inspectez le contenu généré (au format HTML), vous aurez probablement quelque chose qui ressemble à ceci :

```
<div id="app">
  <div>
    <h1>Tâches</h1>
    <ul>
      <li><span>Acheter du lait</span></li>
      <li><span>Prendre des vacances</span></li>
    </ul>
  </div>
</div>
```

C'est assez similaire à ce que l'on attendait, non ? :)