INFORMATIQUE
TECHNIQUE

Expert

React

Développez le Front End de vos applications web et mobiles avec JavaScript

En téléchargement

le code source des exemples

+ QUIZ

Version en ligne **OFFERTE !** pendant 1 an

Sébastien CASTIEL

eni

Expert IT

React

Développez le Front End de vos applications web et mobiles avec JavaScript

Table des matières

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence ENI de l'ouvrage **EIREA** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

Chapitre 1
Découverte de React

Chapitre 2
Ajouter du comportement aux composants

Chapitre 3
Concevoir une application avec Redux

Chapitre 4
Gérer les effets de bord avec Redux-Saga

Chapitre 5
Développer pour le mobile avec React Native

Chapitre 6
Fonctionnalités avancées avec React Native

Chapitre 7
Gestion de formulaires et du routage

Chapitre 8
Sécurité et persistance avec Firebase

Chapitre 9
Connecter React à une API GraphQL

Chapitre 10
Écrire des composants réutilisables

Chapitre 11
Tester une application React

Conclusion

Annexes

Avant-propos

1. React

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. Pas de modèle-vue-contrôleur, pas d'injection de dépendance, pas de bibliothèque de fonctions utilitaires généralistes... Il avait ainsi un côté minimaliste qui le rendait apprécié de beaucoup de développeurs.

Mais React annonçait aussi l'arrivée de JSX, qui comme nous le verrons, permet en quelque sorte de décrire à l'aide d'une syntaxe proche du HTML comment un composant graphique doit être rendu. Et ce, 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.

2. À qui s'adresse ce livre ?

Ce livre s'adresse 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.

3. Que trouverez-vous dans ce livre ?

Les deux premiers chapitres couvriront ce que React propose nativement (sans bibliothèque externe ou presque). Nous écrirons nos premiers composants, les ferons communiquer entre eux, les stylerons avec du CSS...

Le troisième chapitre vous fera découvrir Redux, qui permet de structurer une application un peu plus conséquente afin de la rendre plus facile à maintenir et à faire évoluer. Le quatrième apportera en complément une présentation de Redux-Saga, bibliothèque très intéressante pour augmenter les possibilités d'une application Redux.

Les cinquième et sixième chapitres seront consacrés à React Native. Nous verrons comment l'écriture d'applications mobiles n'est finalement pas plus complexe que celle d'applications web.

Les septième et huitième chapitres aborderont des notions plus avancées comme le routage ou l'utilisation de Firebase pour l'authentification ou la persistance de données distantes. En complément, le neuvième chapitre montrera une alternative à Firebase pour ce qui est de l'appel à une API : GraphQL.

Le dixième chapitre présentera 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.

Enfin, le onzième et dernier chapitre vous donnera des pistes pour apprendre à tester une application React, les composants d'une part, et la partie Redux d'autre part.

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.

4. À propos des exemples

Comme tout livre consacré à du développement, celui-ci 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-même 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 est également disponible en téléchargement en accompagnement de ce livre. 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.

N'ayez crainte si cela vous semble mystérieux pour le moment, le premier chapitre expliquera tout cela en détail.

5. 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 (https://reactjs.org), et notamment sa section Blog, vous informera des nouvelles versions contenant des évolutions majeures.
- Le subreddit dédié à React (https://www.reddit.com/r/ReactJS) 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 (https://twitter.com/reactjs) officiel reprend les annonces du site, mais ceux des développeurs Dan Abramov (@dan_abramov), Sophie Alpert (@sophiebits) et Andrew Clark (@acdlite) vous donneront accès à l'actualité la plus récente.
- Le blog de Dan Abramov, *Overreacted* (https://overreacted.io/), est une mine d'informations pour en savoir plus sur le fonctionnement interne de React. Attendez-vous à des sujets techniques !
- La section React du site *Dev.to* (https://dev.to/t/react) propose beaucoup de tutoriels de tout niveau pour approfondir vos connaissances et donner des astuces.
- En français, le site Putain de code (https://putaindecode.io) propose beaucoup d'articles sur le développement front-end et notamment sur React.

De plus, de nombreux articles fleurissent sur le web à propos de React, notamment lorsqu'une nouveauté est annoncée ou rendue disponible. N'hésitez pas à les découvrir.

Chapitre 1
Découverte de React

1. Installation des outils requis

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*). 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.

1.2 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 de Microsoft (https://code.visualstudio.com/).

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.

1.3 Et ensuite ?

Bien évidemment, vous aurez besoin d'un navigateur web. Tout navigateur peut convenir, je recommande néanmoins d'utiliser Firefox ou Chrome en raison des outils de développement qu'ils proposent. De plus vous pouvez dès maintenant installer l'extension React Dev Tools (https://github.com/facebook/react-devtools) qui vous aidera à débuguer vos applications.

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, c'est-à-dire la transformation d'un langage vers un autre), 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 charge par tous les navigateurs, ou encore de découper notre application en fichiers comme bon nous semble. Parmi les plus connus, citons notamment Webpack très utilisé pour de très gros projets pour toutes les options et plugins qu'il propose.

Create-React-App (https://github.com/facebook/create-react-app) est également de plus en plus utilisé et permet de générer le squelette d'une application React en une seule commande.

Pour nos exemples, j'ai décidé d'utiliser un outil plus minimaliste : Parcel (https://parceljs.org/). 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 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 !* ». Félicitations, c'est votre première application React !

2.1 Explication du code

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

- Une balise `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 placer (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`. 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 du 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.

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 paramètre `props` : il s'agit d'un objet contenant les paramètres qui sont envoyés au composant. Ici, nous appelons `Greetings` ainsi :

```
<Greetings name="Sébastien"/>
```

Ainsi notre paramètre `props` vaudra `{ 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 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.

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. Ces éléments 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 règles 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 (https://reactjs.org/docs/dom-elements.html).

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 :

```
<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 ? Eh 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 composants, 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 compléterons 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.

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 créerons 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`, auquel sera passé en propriété un objet `task`, contenant les informations sur la tâche à afficher. Cet objet 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 de développement de votre navigateur et inspectez le contenu généré (au format de 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 ?

5. Ajouter du style

Jusque-là, nous nous sommes occupés du contenu à afficher, pas vraiment de l'aspect visuel. Il est bien évidemment possible d'utiliser du CSS avec React, voyons comment.

Tout d'abord, la première chose à savoir est que dans la mesure où le contenu généré n'est constitué que d'éléments HTML bien standards, il est tout à fait possible d'utiliser une feuille de styles CSS, comme avec du HTML classique.

Il suffit :

- de créer un fichier style.css dans le dossier *public* contenant nos styles ;
- dans le fichier index.html, d'ajouter une balise `<link>` pour faire référence à cette feuille de style ;
- dans nos composants d'ajouter les classes aux balises à l'aide de la propriété `className`.

Non seulement cette méthode fonctionne, mais elle est même utilisée par de nombreux développeurs, en raison de la similitude avec ce qui est fait depuis des années lorsqu'on écrit du HTML ou qu'on le génère avec du PHP par exemple.

Cependant la philosophie de React veut plutôt que l'on réfléchisse en termes de composants au maximum autonomes. Il serait donc dommage de séparer d'un composant les styles CSS qui le concernent.

La première option pour rapprocher un composant de ses styles consiste à définir une feuille de style par composant. Par exemple, pour le composant `Task`, à côté du fichier Task.js nous pouvons créer un fichier Task.css qui contiendrait ceci :

```
.task {
 color: blue;
}
```

Dans le fichier Task.js, il nous suffirait alors tout d'abord d'importer ce fichier par la directive `import` (ceci est permis par *Parcel* : tous les CSS importés seront ensuite rassemblés dans un fichier unique lors de la phase de construction de l'application), puis de définir un attribut `className` :

```
import React from 'react'
import './Task.css'

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

export default Task
```

Cette méthode a l'avantage d'être très simple à mettre en place, et on utilise bien du CSS tout à fait standard. Le CSS est à côté du composant, il est donc facile de naviguer de l'un à l'autre, ce qui est très bon pour la maintenabilité.

Il y a néanmoins des inconvénients à cette approche. Premièrement, le CSS que vous écrivez dans Task.css par exemple n'est pas limité au composant `Task`. Si dans un autre composant vous utilisez la classe `task`, notre CSS sera alors utilisé. Au mieux cela est voulu, et la maintenabilité ne sera pas aisée, car il sera difficile de retrouver le CSS correspondant à une classe. Au pire c'est une erreur, et il sera parfois laborieux de savoir pourquoi un composant ne s'affiche pas comme désiré.

React ne propose pas nativement de moyen plus élaboré de définir du CSS pour les composants, mais sachez qu'il existe une tendance dans la communauté : le *CSS-in-JS*. Il s'agit de définir les styles directement en JavaScript... pour le meilleur et pour le pire ! En effet, la méthode la plus radicale consiste à écrire les styles de manière *inline*, c'est-à-dire en passant par l'attribut `style` des composants :

```
const Task = ({ task }) =>
 <span style={{ color: blue }}>{task.label}</span>
```

Cela peut paraître effroyable à première vue, car lorsqu'on découvre le CSS on apprend rapidement à n'utiliser l'attribut `style` qu'en dernier recours. Mais on peut rendre ceci un peu plus élégant :

```
const styles = {
 task: { color: blue }
}
const Task = ({ task }) =>
 <span style={styles.task}>{task.label}</span>
```

C'est déjà mieux : en quelque sorte, nous définissons une classe `task`, même si ici nous n'avons pas de CSS à proprement parler. Mais il y a encore des problèmes associés à cette méthode. Notamment, il est impossible de définir qu'un style doit s'appliquer aux enfants de ce composant (là où en CSS nous aurions pu écrire `.task div` pour appliquer un style à toutes les `div` enfants par exemple). D'ailleurs, React recommande de ne pas utiliser cette méthode pour des raisons de performances.

Au final, à ce niveau, il serait recommandé d'utiliser la méthode précédente, c'est-à-dire les fichiers CSS pour chaque composant.

Mais sachez qu'il existe des bibliothèques dédiées, qui résolvent les problèmes déjà évoqués, en permettant :

- d'écrire du CSS parfaitement standard...
- ... mais qui ne s'applique qu'à un composant donné,
- et si on le souhaite, d'utiliser quelques syntaxes pratiques, comme ce que proposent Less ou Sass.

Parmi elles, l'une a ma préférence : *Styled Components* (https://www.styled-components.com/). Nous n'en parlerons pas plus ici, car pour l'utiliser il peut être nécessaire de comprendre certains concepts que l'on n'a pas encore abordés. Mais si cela vous intéresse, n'hésitez pas à consulter leur documentation et les nombreux exemples.

6. Prochaines étapes

Les notions que nous venons de voir vous permettront déjà d'écrire les premiers composants de vos premières applications React, et d'y afficher des données. Dans le chapitre suivant, qui complètera la découverte de React lui-même, sans autre bibliothèque, nous verrons comment ajouter du comportement à nos composants, en y gérant des données locales, en y faisant des requêtes Ajax, ou encore en réagissant aux entrées de l'utilisateur.

Chapitre 2
Ajouter du comportement aux composants

1. Conserver un état local dans le composant

Jusqu'alors, nous n'avons fait qu'afficher des données statiques : notre liste de tâches, définie dans le composant `App`. Bien évidemment, l'idée est de pouvoir modifier ces données, en ajoutant, modifiant et supprimant des tâches. Pour cela, il n'est pas possible de garder le tableau de tâches dans son état actuel, c'est-à-dire déclaré dans la fonction `App` :

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

Nous allons devoir transformer un peu notre composant. Nous avons vu dans le premier exemple que la manière la plus simple de définir un composant est de le faire avec une fonction, qui prend en paramètres les propriétés passées au composant.

Mais React nous donne un deuxième moyen de le faire : passer par une classe, héritant de la classe `Component`, déclarée dans le package `react`.

Voyons à quoi ressemble notre composant sous forme de classe :

```
// src/App.js

// Penser à importer la classe Component.
import React, { Component } from 'react'
import TaskList from './TaskList'

class App extends Component {
 render() {
   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
```

Comme vous pouvez le remarquer, ce n'est pas si différent : tout ce qui était dans la fonction App est à présent dans la méthode `render`. C'est en partie vrai, à une subtilité près qui n'est pas visible ici, car nous n'utilisons pas de propriétés. Mais si nous réécrivons le composant `Task` :

```
// src/Task.js
import React, { Component } from 'react'

class Task extends Component {
 render() {
   const { task } = this.props
   return <span>{task.label}</span>
 }
}

export default Task
```

Nous voyons ici que pour accéder aux propriétés nous devons utiliser `this.props` ; les propriétés ne sont pas passées en paramètres à la méthode `render`. Mais au final, en utilisant la déstructuration permise par JavaScript `const { att1, att2 } = obj`, ce n'est pas plus compliqué pour autant.

Très bien, mais quel est l'intérêt des *class-components* ? Il y en a plusieurs, et celui qui nous intéresse pour le moment est la possibilité offerte par React de garder au sein de la classe des données, qui peuvent être mises à jour par des actions de l'utilisateur, déclenchant la mise à jour du rendu du composant. Cela est appelé l'état du composant, ou plus couramment, son *state*. Le state d'un composant peut être de n'importe quel type, mais le plus souvent il s'agit d'un objet. Il est initialisé dans le constructeur :

```
class App extends Component {
 constructor(props) {
   super(props)
   this.state = {
     tasks: [
       { id: 1, label: 'Acheter du lait', isDone: true },
       { id: 2, label: 'Prendre des vacances', isDone: false }
     ]
   }
 }
 // ...
```

Notez que l'on appelle `super(props)` au tout début, car notre classe `App` hérite de `Component`. Si nous ne le faisons pas, React nous signale une erreur.

Point extrêmement important : le constructeur est le seul emplacement du composant où vous devrez définir le state ainsi (`this.state =...` ou `this.state.valeur =...`). Pour ce qui est de la mise à jour, nous utiliserons un autre moyen comme nous allons le voir. Si vous redéfinissez ou modifiez le state ainsi, React ne saura pas qu'il doit régénérer le contenu du composant, et vous risquez d'avoir des bugs difficiles à identifier.

Dans la méthode `render`, nous pouvons accéder au state grâce à `this.state` (en lecture seulement) :

```
render() {
 return (
   ...
   <TaskList tasks={this.state.tasks} />
```

```
    ...
  )
}
```

Très bien, nous avons un state, mais pour l'instant cela ne nous a rien apporté puisque nous ne le modifions nulle part. Ajoutons un bouton en dessous de la liste de tâches, qui permettra d'ajouter une nouvelle tâche :

```
return (
 <div>
   <h1>Tâches</h1>
   <TaskList tasks={this.state.tasks} />
   <button onClick={() => {}}>Ajouter une tâche</button>
 </div>
)
```

Le composant `button` permet de définir un paramètre `onClick` avec une fonction qui sera appelée lorsque le bouton sera cliqué. Cela est, en fait, permis par tous les composants issus du HTML : boutons, liens, etc., comme ce serait possible en HTML classique à l'aide de l'attribut `onclick`.

Que souhaite-t-on faire lorsque le bouton est cliqué ? Nous souhaitons ajouter une nouvelle tâche (dont le libellé sera pour l'instant constant et défini en dur). Pour cela, nous devons lui définir un ID qui n'est pas déjà celui d'une autre tâche. Nous pourrions utiliser la taille du tableau (`this.state.tasks.length`), mais cela ne fonctionnerait plus si l'on offrait la possibilité de supprimer une tâche.

Le plus judicieux est sûrement d'ajouter un nouvel élément à notre state : l'ID de la prochaine tâche à créer. Dans le constructeur, nous aurons donc :

```
this.state = {
 nextId: 3,
 tasks: [ ...
}
```

Nous devons donc créer une nouvelle tâche, l'ajouter au tableau des tâches, et incrémenter le `nextId`. Simple, non ?

Il y a tout de même un point important à prendre en compte : comme nous l'évoquions un peu plus haut, nous ne devons pas modifier directement le state.

Pour le mettre à jour tout de même, nous allons indiquer à React quelle doit être la nouvelle version du state, puis lui se chargera de le modifier et régénérer le rendu si nécessaire.

Nous créons donc une nouvelle tâche, et une nouvelle version du state. Et pour signaler à React le changement, nous utilisons `this.setState` :

```
const newTask = {
 id: this.state.nextId,
 label: 'Nouvelle tâche'
}
const newState = {
 nextId: this.state.nextId + 1,
 tasks: [...this.state.tasks, newTask]
}
this.setState(newState)
```

Il ne nous reste qu'à appeler ce code lorsque le bouton est cliqué, et le tour est joué ! Voici le code du bouton pour notre nouvelle version du composant App :

```
<button
 onClick={() => {
   const newTask = {
     id: this.state.nextId,
     label: 'Nouvelle tâche'
   }
   this.setState({
     nextId: this.state.nextId + 1,
     tasks: [...this.state.tasks, newTask]
   })
 }}
>
 Ajouter une tâche
</button>
```

Tout devrait bien fonctionner si vous lancez l'application : un clic sur le bouton devrait ajouter une nouvelle tâche dans la liste, avec pour libellé **Nouvelle tâche**.

Remarque

Notez que l'appel à `setState` ne déclenche pas immédiatement la mise à jour du state par React ; il s'agit d'un traitement asynchrone. Notamment React préfère exécuter les mises à jour du state par lot, ce qui signifie que deux appels à `setState` peuvent ne résulter qu'à une seule mise à jour intégrant les deux modifications. Cela signifie également que si vous récupérez `this.state` juste après un appel à `setState`, vous aurez vraisemblablement encore la version non mise à jour du state. Pour savoir quand est effectivement mis à jour le state, `setState` accepte en deuxième paramètre une fonction appelée à ce moment :

```
this.setState({ nextId: this.state.nextId + 1 }, newState => {
 console.log('State mis à jour :', newState)
})
```

Prochaine étape : nous souhaitons donner à l'utilisateur la possibilité de spécifier un libellé pour la tâche qu'il ajoute (c'est la moindre des choses !).

2. Réagir aux actions et entrées de l'utilisateur

Créer un champ texte se fait de la manière la plus intuitive :

```
<input type="text" />
```

Supposons que l'on ait un attribut de notre state qui stocke la valeur à afficher dans le champ texte, et que cet attribut s'appelle `taskLabel`. On peut alors écrire :

```
<input type="text" value={this.state.taskLabel} />
```

Jusque-là tout va bien ; sauf que si vous lancez l'application avec un tel champ texte, vous aurez beau essayer de modifier la valeur du champ, celle-ci ne changera pas. D'ailleurs, React vous affichera un bel avertissement dans la console, vous suggérant d'ajouter une propriété `onChange` au champ. Eh oui, lorsque nous utilisons l'attribut `value`, cela indique à React que, quel que soit le contexte, la valeur du champ texte doit être celle spécifiée. Nous devons donc mettre à jour `this.state.taskLabel` dès que l'utilisateur essaie de modifier la valeur.

Nous avons vu dans la section précédente comment faire en sorte qu'un code soit exécuté lorsqu'un bouton est cliqué par l'utilisateur. React utilise en fait les évènements classiques du DOM, dans cet exemple l'évènement `click`. Gérer la modification d'un champ texte se fait de manière similaire avec l'évènement `change`, et donc la propriété `onChange`.

Comme en JavaScript classique (sans React), la fonction appelée lorsque l'évènement est déclenché prend un paramètre contenant des informations sur l'évènement. Notamment, `event.target.value` permet d'obtenir la nouvelle valeur de l'`input` dans notre cas.

Pour mettre à jour `this.state.taskLabel` dès que l'évènement `change` est déclenché, nous pouvons donc écrire :

```
<input
 type="text"
 value={this.state.taskLabel}
 onChange={event => this.setState({
   taskLabel: event.target.value
 })}
/>
```

Afin d'organiser au mieux notre application, nous allons créer un nouveau composant qui contiendra le formulaire d'ajout de tâche, constitué d'un champ texte pour le libellé de la tâche, ainsi que d'un bouton. Ce composant ne sera pas chargé d'ajouter directement la nouvelle tâche au tableau `this.state.tasks`. Au lieu de cela, il acceptera une propriété `addTask`, qui contiendra une fonction à appeler à la validation du formulaire et permettant d'ajouter effectivement la tâche. Cette fonction sera écrite dans `App`.

Dans la mesure où notre formulaire – appelons-le `TaskForm` – aura besoin d'un state pour contenir la valeur du champ texte, nous devons le créer sous forme de classe :

```
// TaskForm.js
import React, { Component } from 'react'

class TaskForm extends Component {
 constructor(props) {
   super(props)
   this.state = { label: '' }
 }
```

```
  render() {
    return (
      <div>
        <input
          type="text"
          placeholder="Nouvelle tâche"
          value={this.state.label}
          onChange={event => {
            this.setState({ label: event.target.value })
          }}
        />
        <button
          onClick={() => {
            this.props.addTask(this.state.label)
            this.setState({ label: '' })
          }}
        >
          Ajouter
        </button>
      </div>
    )
  }
}

export default TaskForm
```

Notez que lorsque le bouton est cliqué, on appelle `this.props.addTask` afin de créer la tâche, mais on réinitialise également le champ texte avec une valeur vide.

Attardons-nous quelques instants sur les fonctions appelées au déclenchement d'évènements. Jusqu'ici, nous les avons définies directement dans le JSX, ce qui est parfaitement acceptable lorsque les fonctions sont très simples (pour simplement appeler `setState` par exemple). Mais lorsque ces fonctions font quelques lignes, cela peut rapidement surcharger le JSX ; il paraît alors judicieux de les extraire en dehors. Plusieurs solutions pour cela.

La solution la plus simple consiste à déclarer les fonctions juste au-dessus du JSX, au début de la méthode `render` :

```
render() {
 const handleInputChange = event => { /* ... */ };
 return (
```

```
    ...
    <input onChange={handleInputChange}/>
    ...
  )
}
```

Cela fonctionnerait très bien, néanmoins on aimerait les sortir complètement de `render`, afin de sépaerr clairemeent les problématiques de rendu et de comportement. Puisque l'on est dans une classe, il serait intéressant d'en faire des méthodes. Intuitivement, on écrirait :

```
handleInputChange(event) {
 this.setState(/* ... */)
}
render() {
 return (
   ...
   <input onChange={this.handleInputChange}/>
   ...
 )
}
```

Malheureusement, cela ne fonctionnera pas. Ceci est dû à JavaScript et non pas à React ; cela tient aux propriétés du mot-clé `this`. Nous n'entrerons pas dans les détails ici (je vous encourage à vous renseigner sur `this`), mais pour faire simple, lorsque l'on stocke `this.handleInputChange` dans une variable (ou ici la propriété d'un composant), le fait d'appeler la variable comme une fonction fait que dans le corps de la fonction `this` ne fait plus référence à l'objet initial. Si vous exécutez ce code, vous aurez donc une erreur ressemblant à « *this.setState is not a function* ».

Trois solutions simples pour résoudre cela. La première est de ne pas passer `this.handleInputChange` dans l'attribut `onChange`, mais une nouvelle fonction appelant `this.handleChange` :

```
<input onChange={event => this.handleInputChange(event)} />
```

La deuxième est, dans le constructeur, de remplacer la méthode `handleInputChange` de l'objet créé par une nouvelle version, forçant la valeur de `this` :

```
constructor() {
 this.handleInputChange = this.handleInputChange.bind(this)
}
```

Enfin, la troisième est une astuce arrivant quasiment au même résultat : déclarer `handleInputChange` non pas comme une méthode de la classe, mais comme un attribut de l'objet.

```
handleInputChange = event => {
 this.setState(/* ... */)
}
```

Cependant, cette dernière méthode vous imposera d'ajouter un plugin à Babel permettant de gérer les attributs de classe. La commande suivante permet de l'installer :

```
$ yarn add --dev babel-plugin-transform-class-properties
```

Pour utiliser ensuite ce plugin, il suffit de mettre à jour la section `babel` du fichier package.json :

```
"babel": {
 "presets": [/* ... */],
 "plugins": [
   "transform-class-properties"
 ]
}
```

Malgré cette contrainte, la troisième méthode est celle qui a ma préférence, car c'est celle qui me paraît la plus simple et la plus concise.

Mais depuis les quelques années que React existe, aucun consensus n'a été trouvé dans la communauté entre ces trois manières de faire (il en existe sûrement d'autres), chacune a ses avantages et inconvénients. Notamment, la première crée une nouvelle fonction à chaque fois que `render` est appelée, ce qui n'est pas idéal... Bref, c'est plus une question de préférence.

Revenons-en à notre application. Maintenant que notre composant `TaskForm` est prêt, il nous reste à l'intégrer dans le composant `App`. Pour cela nous allons remplacer le bouton que nous avions ajouté dans la section précédente :

```
addTask = label => {
 const newTask = { id: this.state.nextId, label }
 this.setState({
   nextId: this.state.nextId + 1,
   tasks: [...this.state.tasks, newTask]
 })
}
render() {
 return (
   <div>
     <h1>Tâches</h1>
     <TaskList tasks={this.state.tasks} />
     <TaskForm addTask={this.addTask} />
   </div>
 )
}
```

C'est terminé, notre application dispose maintenant d'un formulaire permettant d'ajouter une nouvelle tâche. Voici à présent un petit exercice qui vous permettra de vérifier que les concepts vus dans cette section (et les précédentes) sont bien acquis.

Exercice

Nous souhaiterions donner la possibilité à l'utilisateur de marquer une tâche comme effectuée. Pour cela, il faudrait qu'à côté de chaque tâche se trouve une case à cocher, qui est cochée si la tâche est effectuée.

Voici quelques suggestions et questions à vous poser pour y parvenir :

- Comment et où allez-vous stocker le fait qu'une tâche est effectuée ou non ?
- Où l'action permettant de marquer une tâche comme effectuée va-t-elle être déclarée ? Où va-t-elle être appelée ?
- Pour gérer la valeur d'une case à cocher (`<input type="checkbox"/>`), il faut utiliser l'attribut `checked` et non `value`. Il est de type booléen (`checked={true}`).

Essayez de faire cet exercice par vous-même. En effet, mis à part des subtilités évoquées ci-dessus, nous avons normalement vu tout ce que vous avez besoin de savoir pour réussir à implémenter cette nouvelle fonctionnalité. Voici néanmoins un exemple de correction.

Correction

Bien évidemment, comme c'est souvent le cas en développement, il existe autant de solutions possibles à un problème qu'il y a de développeurs. Vous avez peut-être trouvé une autre solution que celle décrite ici (elle est même peut-être meilleure).

Je propose de commencer par le composant le plus spécifique (`Task`) pour remonter progressivement vers le composant le plus général (`App`). Mais avant cela, répondons d'abord à la question concernant le modèle de données : où stockerons-nous le fait qu'une tâche est effectuée ? J'ai choisi de le faire dans la tâche elle-même, c'est-à-dire dans le state du composant `App`. Une tâche aura donc un attribut `isDone`, en plus des attributs `id` et `label`.

Dans ce composant `Task`, nous souhaitons tout d'abord afficher une case à cocher :

```
<input type="checkbox" />
```

Cette case doit être cochée si la tâche est effectuée :

```
<input type="checkbox" checked={task.isDone} />
```

Enfin, lorsque l'utilisateur clique sur la case, on souhaite mettre à jour la tâche pour la marquer effectuée. Les tâches ne sont pas stockées dans le state de ce composant (qui est d'ailleurs *stateless*, c'est-à-dire qu'il n'a pas d'état propre), mais la case à cocher est bien ici, c'est donc là qu'on va intercepter le clic. Nous allons donc supposer qu'une propriété `setTaskStatus` est passée au composant, celle-ci prenant en paramètre un booléen à mettre à *true* pour dire que la tâche est effectuée, *false* sinon :

```
const Task = ({ task, setTaskStatus }) => {
 return (
   <label>
     <input
       type="checkbox"
       checked={task.isDone}
```

```
        onChange={event => {
          const isDone = event.target.checked
          setTaskStatus(isDone)
        }}
      />
      {task.label}
    </label>
  )
}
```

Notre composant `Task` est prêt, passons à `TaskList`. Peu de choses à mettre à jour dans ce composant. Nous devons passer une propriété `setTaskStatus` à `Task`. Les tâches ne sont toujours pas stockées dans ce composant, nous allons donc encore une fois attendre que cette fonction nous soit passée par un composant parent.

Petite subtilité néanmoins : la fonction `setTaskStatus` que l'on doit passer à `Task` doit définir le statut d'une tâche bien définie, elle ne doit prendre en paramètre que le statut. Mais le composant `TaskList` affichant toutes les tâches, la fonction qui va lui être passée en paramètre va devoir prendre en paramètres l'ID d'une tâche et le nouveau statut de cette tâche.

Pour cela, on ne passe pas directement la fonction `setTaskStatus` fournie par le parent, mais une fonction appelant cette dernière avec un ID de tâche fixé :

```
const TaskList = ({ tasks, setTaskStatus }) => (
  <ul>
    {tasks.map(task => (
      <li key={task.id} className="task-item">
        <Task
          task={task}
          setTaskStatus={isDone => setTaskStatus(task.id, isDone)}
        />
      </li>
    ))}
  </ul>
)
```

Bien, il ne reste plus qu'à mettre à jour le composant App. Celui-ci va devoir définir une fonction `setTaskStatus` qui va être passée comme propriété à `TaskList` :

```
<TaskList
 tasks={this.state.tasks}
 setTaskStatus={this.setTaskStatus}
/>
```

Que doit faire `setTaskStatus` ? Réponse simple : elle doit mettre à jour l'attribut `isDone` de la tâche dont l'ID lui est passé en paramètre. Mais ce n'est pas aussi simple, car on ne doit pas modifier le state directement, il faut passer par `setState`. Nous allons donc devoir créer une nouvelle version du state, dans laquelle :

- la tâche à mettre à jour sera une copie de la tâche initiale (avec son attribut `isDone` mis à jour) ;
- le tableau des tâches sera une copie du tableau initial, avec la tâche copiée en lieu et place de la tâche initiale.

Pour cela, je propose le petit algorithme suivant :

- récupérer l'index de la tâche T concernée dans le tableau ;
- créer une copie de cette tâche T avec l'attribut `isDone` mis à jour ;
- récupérer les tâches Ts1 situées avant la tâche T dans le tableau, ainsi que celles situées après Ts2 ;
- créer un nouveau tableau de tâches en concaténant Ts1, T et Ts2.

Cela nous donne le code suivant :

```
setTaskStatus = (taskId, isDone) => {
 const { tasks } = this.state
 const taskIndex = tasks.findIndex(t => t.id === taskId)
 const tasksBefore = tasks.slice(0, taskIndex)
 const tasksAfter = tasks.slice(taskIndex + 1)
 const newTask = { ...tasks[taskIndex], isDone }
 this.setState({
   tasks: [...tasksBefore, newTask, ...tasksAfter]
 })
}
```

Vérifions que nous ne modifions pas le state actuel :

- `findIndex` ne modifie pas le tableau, il ne fait que renvoyer un index ;
- `slice` ne modifie pas le tableau, mais crée un nouveau tableau à partir des éléments correspondant aux paramètres (index du premier élément et nombre d'éléments) ;
- la tâche concernée n'est pas modifiée grâce à la syntaxe `newTask = { ...task, isDone }`. On crée bien un nouvel objet dans lequel les attributs de la tâche sont copiés, et auquel on force la valeur de `isDone`.

Tout est bon, on ne modifie pas le state, on ne fait qu'appeler `setState` avec un objet constitué d'objets nouveaux, ou bien des objets du state actuel qui ne sont pas modifiés (les autres tâches).

Cela clôt la correction de l'exercice ainsi que la section consacrée aux entrées utilisateur. Les concepts qui y ont été évoqués ne sont pas les plus simples à appréhender, n'hésitez pas à ajouter des fonctionnalités à cet exemple pour être sûrs de les maîtriser. Par exemple :

- Sauriez-vous afficher différemment (couleur...) une tâche selon qu'elle est effectuée ou non ? Peut-être qu'utiliser du CSS comme vu précédemment vous aidera...
- Sauriez-vous permettre la modification du libellé d'une tâche ?
- Sauriez-vous donner la possibilité de supprimer une tâche ?

3. Requêtes Ajax et cycle de vie des composants React

Jusque-là, nous n'avons gardé nos tâches que dans le state local. Nous allons voir dans cette section comment aller récupérer les tâches depuis une API externe. L'appel Ajax n'aura en fait rien à voir avec React, nous allons utiliser la fonction `fetch` désormais disponible sur tous les navigateurs récents. En revanche, il est pratique de savoir comment gérer cela au mieux avec un composant React.

Pour notre exemple, nous allons récupérer la liste de tâches depuis une API rendue disponible par *JSON Placeholder* (https://jsonplaceholder.typicode.com).

Celle-ci met simplement à disposition des ressources via une API REST, idéale pour tester des appels Ajax. L'URL que nous allons utiliser est la suivante : https://jsonplaceholder.typicode.com/users/10/todos

Celle-ci nous renvoie un résultat au format JSON :

```
[
 {
   "userId": 10,
   "id": 181,
   "title": "ut cupiditate sequi aliquam fuga maiores",
   "completed": false
 },
 {
   "userId": 10,
   "id": 182,
   "title": "inventore saepe cumque et aut illum enim",
   "completed": true
 }
]
```

La question est de savoir où dans notre application nous allons effectuer l'appel. De toute évidence, c'est le composant `App` qui va avoir cette responsabilité. La première idée pourrait être de faire cet appel dans le constructeur de la classe. Seulement, il se peut qu'une instance du composant soit créée, mais que ce composant ne soit pas pour autant ajouté au DOM. Ou pour suivre la terminologie React, que ce composant ne soit pas monté (*mounted*).

React nous met donc à disposition des méthodes que l'on peut redéfinir et qui seront appelées à différents instants du cycle de vie du composant. Celle qui nous intéressera ici est `componentDidMount`, appelée comme son nom l'indique dès que le composant est monté.

Dans notre composant `App`, la méthode `componentDidMount` va donc faire appel à l'API, puis appeler `setState` avec les tâches récupérées :

```
componentDidMount() {
 fetch('https://jsonplaceholder.typicode.com/users/10/todos')
   .then(res => res.json())
   .then(tasks => {
     this.setState({
       // On doit convertir les tâches depuis le format
       // reçu depuis l'API vers le format des tâches
```

```
        // que l'on gère.
        tasks: tasks.map(task => ({
          id: task.id,
          label: task.title,
          isDone: false
        })),
        // Pour le nextId on prend l'ID maximal auquel on
        // ajoute 1
        nextId: Math.max(...tasks.map(task => task.id)) + 1
      })
    })
  }
```

Ainsi nos tâches sont bien initialisées, problème résolu. Mais afin d'améliorer légèrement notre application, il serait pertinent de gérer les deux cas suivants :

- les tâches sont en cours de chargement ;
- une erreur s'est produite.

Nous disposons de la connaissance nécessaire pour cela, il nous suffit premièrement de déclarer deux nouveaux attributs dans le state initial :

```
constructor() {
 super()
 this.state = {
   nextId: null,
   tasks: null,
   isFetching: true,
   hasError: false
 }
}
```

Dans l'appel à l'API, nous pouvons à présent mettre à jour ces attributs selon le résultat :

```
.then(tasks => {
 this.setState({
   isFetching: false,
   tasks: /* ... */,
   nextId: /* ... */
 })
})
.catch(() => {
 this.setState({ isFetching: false, hasError: true })
```

```
})
```

Il ne nous reste donc plus qu'à adapter l'affichage du composant en fonction des attributs `isFetching` et `hasError` :

```
render() {
 if (this.state.hasError) {
   return <p>Oups, une erreur s'est produite...</p>
 }
 if (this.state.isFetching) {
   return <p>Chargement en cours...</p>
 }
 return (
   <div>
     <h1>Tâches</h1>
     ...
 )
}
```

Cette manière de gérer le chargement asynchrone de données est relativement courante, et nous en verrons d'autres applications plus loin dans le livre, notamment en utilisant Redux au chapitre suivant.

Nous avons vu la méthode `componentDidMount` du cycle de vie d'un composant ; sachez que d'autres méthodes dont disponibles, par exemple `componentWillUnmount`, exécutée lorsque le composant est retiré du DOM. L'endroit idéal pour libérer des ressources par exemple : fermer une connexion *Websocket*, supprimer un timer, etc. D'autres méthodes permettent de gérer la mise à jour d'un composant. Par exemple `getDerivedStateFromProps` permet de mettre à jour le state lorsque des propriétés passées au composant sont mises à jour. L'ensemble de ces méthodes est bien évidemment décrit en détail dans la documentation de React (https://reactjs.org/docs/react-component.html#the-component-lifecycle).

Cela clôt notre brève exploration du cycle de vie d'un composant. Voici un petit exercice qui vous permettra si le vous souhaitez de vérifier que vous êtes à l'aise avec ce que nous venons de voir.

Exercice

Nous avons vu comment récupérer des données depuis une API, qu'en est-il de l'envoi de données ? Vous sentez-vous capable d'ajouter un appel dès qu'une tâche est créée afin que cette tâche soit sauvegardée sur le serveur ? Qu'en est-il du rafraîchissement des tâches, au cas où un autre utilisateur en aurait ajouté une nouvelle ?

4. Simplifier les composants grâce aux hooks

Jusque-là, dans ce chapitre, nous avons vu que pour ajouter du comportement aux composants (état local, réaction aux actions de l'utilisateur ou gestion du cycle de vie) il était nécessaire d'avoir un composant écrit sous forme de classe. En réalité, fin 2018, à l'occasion de la React Conf, Facebook a annoncé l'ajout d'une fonction à React qui a séduit la plupart des développeurs : les *hooks*. Ceux-ci permettent de gérer l'intégralité des possibilités offertes aux composants dans une fonction, et non dans une classe.

Reprenons par exemple le composant `App` de notre application, et initialisons une nouvelle version, sous forme de fonction cette fois-ci. Naïvement, déclarons les éléments du state à l'aide de simples variables, de même que les fonctions `addTask` et `setTaskStatus`.

```
const App = () => {
 const nextId = null
 const tasks = []
 const isFetching = true
 const hasError = false

 const setTaskStatus = (taskId, isDone) => {}

 const addTask = label => {}

 if (hasError) {
   return <p>Oups, une erreur s'est produite…</p>
 }
 if (isFetching) {
   return <p>Chargement en cours…</p>
 }
```

```
  return (
    <div>
      <h1>Tâches</h1>
      <TaskList tasks={tasks}
                setTaskStatus={setTaskStatus} />
      <TaskForm addTask={addTask} />
    </div>
  )
}
```

Dans cet état, notre application est assez inutile, puisque le state ne peut pas changer. Le message « Chargement en cours... » sera donc toujours affiché. Commençons par voir comment gérer le state à l'aide des hooks de React.

4.1 Gérer le state local avec useState

Afin de gérer le state local et permettre sa modification, React propose le hook `useState` (par convention, tous les hooks ont leur nom qui commence par « use »). Comme tous les hooks, il s'agit d'une fonction à appeler dans le composant. Celle-ci renvoie deux valeurs (dans un tableau) :

- La valeur actuelle de l'élément du state ;
- Une fonction permettant de mettre à jour cette valeur, déclenchant ainsi un nouveau rendu du composant.

`useState` prend en unique paramètre la valeur initiale du state. Nous pouvons donc déclarer notre state ainsi :

```
const [nextId, setNextId] = useState(null)
const [tasks, setTasks] = useState(null)
const [isFetching, setIsFetching] = useState(true)
const [hasError, setHasError] = useState(false)
```

Notez que pour coller à l'implémentation précédente nous aurions tout aussi bien pu n'appeler qu'une seule fois `useState` :

```
const [state, setState] = useState({ ... })
```

Ce serait plutôt une question de préférence, mais dans la mesure où rien n'empêche d'appeler autant de fois `useState` qu'on le souhaite, cela peut rendre le code plus lisible ainsi.

À présent, nous pouvons écrire le code des fonctions `setTaskStatus` et `addTask`, en utilisant les nouvelles fonctions de mise à jour du state :

```
const setTaskStatus = (taskId, isDone) => {
 const { tasks } = this.state
 const taskIndex = tasks.findIndex(t => t.id === taskId)
 const tasksBefore = tasks.slice(0, taskIndex)
 const tasksAfter = tasks.slice(taskIndex + 1)
 const newTask = { ...tasks[taskIndex], isDone }
 setTasks([...tasksBefore, newTask, ...tasksAfter])
}

const addTask = label => {
 const newTask = { id: nextId, label }
 setNextId(nextId + 1)
 setTasks([...tasks, newTask])
}
```

4.2 Optimiser le rendu avec useCallback

Sur le plan optimisation, il existe un inconvénient à déclarer des fonctions ainsi dans le corps d'un composant. En effet, à chaque fois que le composant sera rendu, une nouvelle fonction sera créée par JavaScript. On peut difficilement contrer cela, la solution serait de déclarer la fonction hors du composant, mais comme elle a besoin d'accéder à certains éléments déclarés dans le composant (hors de sa portée, donc). En revanche, lorsque React met à jour l'arbre des composants affichés, si la fonction est passée en propriété à un composant (comme c'est le cas pour les fonctions de callback), le fait que ce soit une nouvelle fonction peut déclencher inutilement un nouveau rendu du sous-arbre des composants.

Pour optimiser cela, React met à disposition un hook spécialement prévu pour les fonctions de callback (c'est-à-dire les fonctions que l'on passe en propriétés à un autre composant) : `useCallback`.

Celui-ci prend deux paramètres :

- la fonction proprement dite ;
- un tableau contenant les éléments dont la mise à jour doit créer une nouvelle fonction.

Supposons par exemple que l'on ait le code suivant :

```
const [name, setName] = useState('John')
const [age, setAge] = useState(42)
const logName = () => console.log(name)
```

Si, dans le composant, deux champs permettent de mettre à jour `name` et `age`, alors la mise à jour de n'importe laquelle de ces deux valeurs effectuera un nouveau rendu du composant, et créera donc une nouvelle fonction `logName`. Si cette fonction est passée en paramètre à un autre composant, celui-ci sera donc également rendu à nouveau, puisque l'une de ses propriétés aura changé. En l'occurrence, si l'âge a changé, ce composant sera à nouveau rendu inutilement, puisque la fonction ne dépend que du nom.

En déclarant `logName` avec `useCallback`, on optimise ce comportement :

```
const [name, setName] = useState('John')
const [age, setAge] = useState(42)
const logName = useCallback(
 () => console.log(name),
 [name],
)
```

À présent, d'un rendu sur l'autre, `logName` ne sera modifiée que si `name` est modifié. Cela peut paraître comme une micro-optimisation, dont on ne verrait les avantages que sur une très grosse application. En réalité cela peut optimiser drastiquement les performances à partir d'une application de taille moyenne, et cela est encore plus vrai sur mobile avec React Native.

Il est donc désormais considéré comme une bonne pratique de déclarer toute fonction de callback à l'aide de `useCallback`. Mais attention à bien fournir en second paramètre les valeurs à prendre en compte pour créer une nouvelle fonction. C'est en réalité simple : toute variable utilisée dans la fonction doit être déclarée dans le tableau en second paramètre.

```
const setTaskStatus = useCallback(
 (taskId, isDone) => {
   const taskIndex = tasks.findIndex(t => t.id === taskId)
   const tasksBefore = tasks.slice(0, taskIndex)
   const tasksAfter = tasks.slice(taskIndex + 1)
   const newTask = { ...tasks[taskIndex], isDone }
   setTasks([...tasksBefore, newTask, ...tasksAfter])
 },
```

```
  [tasks],
)

const addTask = useCallback(
 label => {
   const newTask = { id: nextId, label }
   setNextId(nextId + 1)
   setTasks([...tasks, newTask])
 },
 [nextId, tasks],
)
```

Si vous oubliez d'y mettre une variable (et cela vous arrivera sans aucun doute), vous verrez rapidement que le comportement observé n'est pas celui souhaité, puisque la fonction appelée est comme une ancienne version de celle que l'on souhaiterait appeler en réalité. Par exemple ici, si vous oubliez de fournir `tasks` au callback `setTaskStatus`, vous observerez que l'ajout d'une tâche remplace en réalité la tâche précédente !

4.3 Requête Ajax au démarrage avec useEffect

Le dernier hook que nous allons voir pour le moment permet de gérer en partie le cycle de vie du composant, en remplaçant notamment la méthode `componentDidMount`. Le hook `useEffect` prend en paramètre une fonction, qui sera appelée chaque fois que le composant est rendu. Cela peut paraître étrange, car dans notre cas, pour charger la liste de tâches, on ne souhaite effectuer cette opération qu'une seule fois. Cela peut facilement être réalisé en associant un state local contenant un attribut booléen (un flag) indiquant si le traitement a déjà été fait. Par exemple :

```
const [first, setFirst] = useState(false)
useEffect(() => {
 if (first) {
   // On peut faire la requête Ajax
   // ...
   setFirst(false)
 }
})
```

Pour notre exemple, créons un flag `hasFetchedFirst` qui sera à `true` si la requête Ajax a été lancée une première fois.

```
const [hasFetchedFirst, setHasFetchedFirst] = useState(false)

useEffect(() => {
 if (!hasFetchedFirst) {
   setHasFetchedFirst(true)
   setHasError(false)
   setIsFetching(true)
   fetch('https://jsonplaceholder.typicode.com/users/10/todos')
     .then(res => res.json())
     .then(newTasks => {
       setIsFetching(false)
       setTasks(
         newTasks.map(task => ({
           id: task.id,
           label: task.title,
           isDone: false,
         })),
       )
       setNextId(Math.max(...newTasks.map(task => task.id)) + 1)
     })
     .catch(() => setHasError(true))
 }
})
```

En utilisant cette nouvelle version du composant App, le comportement devrait être exactement le même que précédemment. Peut-être vous est-il difficile de voir l'intérêt des hooks pour le moment, et c'est bien normal. De manière globale, un composant React utilisant les hooks est plus simple et plus facile à maintenir que son équivalent utilisant une classe. De plus, comme nous le verrons au dernier chapitre de ce livre, il est possible d'écrire des hooks personnalisés, pour extraire du comportement d'un composant et le partager avec d'autres. D'ailleurs il existe d'autres hooks proposés par React que nous découvrirons au fil du livre.

Sachez cependant qu'à l'heure où la seconde édition de ce livre est écrite, les hooks sont encore une fonctionnalité récente. Déjà utilisés dans de nombreuses applications en production, leur stabilité n'est plus remise en cause. Cependant, les bibliothèques tierces ne les utilisent pas encore toutes.

Certaines n'ont pas tardé à faire évoluer leur API pour profiter des hooks, mais ce n'est pas encore une généralité.

De plus, React ne recommande pas (encore) de migrer toutes les applications des class-components aux hooks, ni de ne plus utiliser les class-components dans les nouvelles applications. Les hooks comportent leur lot de subtilité, et il est parfois préférable de s'assurer qu'on les comprend suffisamment avant d'envisager de les utiliser partout. Dans la suite du livre, la plupart du temps nous utiliserons les class-components pour les composants qui nécessitent un state local ou une gestion de leur cycle de vie.

4.4 Un point d'attention important sur les hooks

Une des subtilités des hooks mérite d'ailleurs d'être abordée. En effet, il ne s'agit en réalité pas de fonctions aussi simples qu'elles peuvent le paraître. Elles sont extrêmement liées au noyau de React, et c'est ce qui leur donne tout leur potentiel, mais cela vient avec une contrainte : dans un composant donné, quel que soit le contexte, ce sont toujours les mêmes hooks qui doivent être appelés et dans le même ordre.

Impossible donc d'écrire :

```
const App = props => {
 if (props.user) {
   useEffect(() => { ... });
 }
 const [user, setUser] = useState({})
 // ...
}
```

Cela ferait qu'en fonction de la propriété `user` on appellerait soit `useEffect` puis `useState`, soit simplement `useState`. En fait, React gère les hooks grâce à l'ordre dans lequel ils sont appelés dans le composant. Cela paraît contre-intuitif, mais c'est un choix qui a ses raisons, et est parfaitement assumé par les équipes développant React (https://reactjs.org/docs/hooks-rules.html#explanation).

Enfin, si cette contrainte paraît étrange, elle n'est absolument pas dérangeante lorsqu'on utilise les hooks, et il est toujours possible de s'en sortir autrement sans ajouter de complexité.

5. Déclarer et typer les propriétés des composants

Nous en avons quasiment terminé avec notre découverte de React, mais ce chapitre n'aurait pas été complet si je n'avais pas abordé les *PropTypes*. Il s'agit d'un moyen de déclarer quelles propriétés attend votre composant, ainsi que leur type.

Il existe plusieurs manières de réaliser ce contrôle. Certains développeurs utilisent le langage TypeScript (https://www.typescriptlang.org/docs/handbook/jsx.html) qui permet de typer les variables, paramètres des fonctions, et pour notre cas les propriétés des composants. D'autres utilisent Flow (https://flow.org/en/docs/react/components/) qui permet d'arriver au même résultat. Mais la méthode la plus simple est d'utiliser la bibliothèque PropTypes (https://reactjs.org/docs/typechecking-with-proptypes.html).

À l'origine, cette bibliothèque faisait partie intégrante de React. Ce n'est plus le cas, justement pour alléger React dans la mesure où certains développeurs ne l'utilisaient pas, au profit de TypeScript ou Flow. Il faut désormais l'installer à part : `yarn add prop-types`

Reprenons notre composant `Task`. Celui-ci attend deux propriétés :

- `task` : un objet correspondant à une tâche, comportant un ID, un libellé et un flag indiquant si la tâche est effectuée ;
- `setTaskStatus` : une fonction à appeler pour mettre à jour le statut d'une tâche.

De manière simple, nous pouvons déclarer ces propriétés ainsi :

```
import React, { Component } from 'react'
// Penser à importer PropTypes.
import PropTypes from 'prop-types'
import './Task.css'

const Task = ({ task, setTaskStatus }) => {
 // ...
```

```
}

Task.propTypes = {
 task: PropTypes.object,
 setTaskStatus: PropTypes.func
}

export default Task
```

Ainsi, au moment où le composant sera utilisé, les propriétés qui lui seront passées seront analysées afin de vérifier qu'elles correspondent bien au type attendu. Si ce n'est pas le cas, un avertissement sera affiché dans la console. Cela peut être très utile pour prévenir les bugs ; de plus, les *propTypes* sont une première et précieuse forme de documentation pour vos composants.

C'est une première étape pour le composant `Task`, mais il nous manque pas mal d'informations. En effet, cela n'aurait aucun sens d'utiliser notre composant sans lui fournir une tâche (dans l'implémentation actuelle, nous aurions même une erreur). De plus, on peut admettre qu'il n'est pas indispensable de donner une fonction `setTaskStatus` ; si aucune fonction n'est fournie, le clic sur la case à cocher ne fera juste rien.

Pour rendre une propriété obligatoire, on utilisera `isRequired`, et grâce à l'attribut `defaultProps` on donnera une valeur par défaut à `setTaskStatus`, en l'occurrence une fonction ne faisant rien :

```
Task.propTypes = {
 task: PropTypes.object.isRequired,
 setTaskStatus: PropTypes.func
}
Task.defaultProps = {
 setTaskStatus: () => {}
}
```

C'est mieux, mais il nous manque une information essentielle : à quoi doit ressembler l'objet `task` ? Pour décrire la structure attendue d'un objet, on utilisera `PropTypes.shape` :

```
Task.propTypes = {
 task: PropTypes.shape({
   id: PropTypes.number.isRequired,
   label: PropTypes.string,
   isDone: PropTypes.bool
```

```
  }).isRequired,
  setTaskStatus: PropTypes.func
}
Task.defaultProps = {
  setTaskStatus: () => {}
}
```

Comme nous aurons besoin de la structure d'une tâche dans les autres composants, je suggère de placer la `shape` dans un fichier shapes.js :

```
// shapes.js
import PropTypes from 'prop-types'

export const taskShape = PropTypes.shape({
  id: PropTypes.number.isRequired,
  label: PropTypes.string,
  isDone: PropTypes.bool
})

// Task.js
import PropTypes from 'prop-types'
import { taskShape } from './shapes'

// ...
Task.propTypes = {
  task: taskShape.isRequired,
  setTaskStatus: PropTypes.func
}
Task.defaultProps = {
  setTaskStatus: () => {}
}
// ...
```

Voilà qui est fait pour le composant `Task`. Pour le composant `TaskList`, ce n'est pas plus compliqué. Nous allons juste utiliser `PropTypes.arrayOf` afin de spécifier que la propriété `tasks` est un tableau d'objets correspondant à `taskShape` que nous venons de créer :

```
TaskList.propTypes = {
  tasks: PropTypes.arrayOf(taskShape).isRequired,
  setTaskStatus: PropTypes.func
}
TaskList.defaultProps = {
```

```
  setTaskStatus: () => {}
}
```

Pour ce qui est du composant `App`, il n'attend aucune propriété. Je vous laisse comme exercice le soin de définir les `propTypes` pour le composant `TaskForm`.

Pour en savoir plus sur les `propTypes` et les possibilités offertes (elles sont nombreuses), vous pouvez vous référer à la page qui leur est consacrée dans la documentation de React.

6. En conclusion

Nous en avons terminé avec notre exploration des possibilités offertes par React seul, c'est-à-dire sans autre bibliothèque ou presque. Comme vous pouvez le constater, ces possibilités sont déjà nombreuses, et encore nous n'en avons vu qu'une partie. Les connaissances acquises dans ces deux premiers chapitres vous permettront déjà de développer vos premières applications web à l'aide de React, et j'espère que ce chapitre vous a donné envie de continuer avec React !

Sachez tout de même que la communauté React est très grande. Cela signifie premièrement que vous trouverez toujours quelqu'un pour vous aider sur des forums, Stack Overflow, Reddit, Twitter..., mais aussi que pour un même problème vous aurez sans doute beaucoup de solutions. Nul doute que si quelqu'un d'expérimenté lit ce livre, il verra des pratiques qu'il jugera impertinentes, voire mauvaises. Cela ne signifie pas que c'est la mauvaise manière de faire, mais peut-être juste qu'une manière plus optimisée existe. Peut-être même que cette nouvelle manière est possible depuis peu. React évolue très rapidement, et l'état de l'art devient rapidement obsolète.

Quoi qu'il en soit, ne prenez rien de ce que vous lirez dans ce livre pour argent comptant : pratiquez, écrivez des petites applications React, tenez-vous informés sur les nouveautés et nouvelles pratiques, et remettez en question (courtoisement et de manière constructive) tous les articles qui vous diront qu'une manière de faire est meilleure qu'une autre.

Dans le prochain chapitre, nous allons voir comment structurer une application un peu plus complexe grâce à Redux. En effet, dans l'exemple de ce chapitre, nous gardions les données de notre application dans le state du composant principal, mais cela peut rapidement devenir complexe lorsque l'application compte plusieurs dizaines de composants et que tous doivent se passer mutuellement des données et déclencher des actions permettant de les mettre à jour. Redux nous permettra de mettre un peu d'ordre dans tout ça.

Chapitre 3
Concevoir une application avec Redux

1. Introduction

Dans les deux premiers chapitres, nous avons vu les bases de React, qui vous permettent déjà de créer vos premières applications. En effet, nous avons vu comment créer des composants, qui stockent des données (le state), et se les transmettent par le biais des propriétés. Dans l'exemple de gestion de liste de tâches présenté, les données étaient stockées dans le composant principal de l'application (App), qui les fournissait aux composants qui en avaient besoin.

Ce fonctionnement est parfaitement acceptable dans une application ; il peut néanmoins devenir laborieux à gérer et maintenir lorsque l'application stocke beaucoup de données, qui seront utilisées par beaucoup de composants. C'est pour répondre à ce problème que Redux entre en jeu.

Créé en 2015, Redux se décrit comme un conteneur d'état prévisible (*predictable state container* en anglais). Il permet de stocker l'état d'une application, tout en donnant les moyens d'y accéder de manière globale, et de le mettre à jour. Il n'est pas spécifique à React, c'est la bibliothèque React-Redux qui fournit les éléments permettant de faire travailler React et Redux en collaboration.

Pourquoi Redux est-il particulièrement adapté à React ? Je dirais que c'est parce qu'ils partagent des valeurs clés, la plus importante étant la notion d'immutabilité.

En effet, je n'ai pas insisté sur ce point dans le premier chapitre, mais React encourage à ne pas modifier des objets une fois créés, c'est par exemple pourquoi on passe par `setState` pour mettre à jour l'état d'un composant. De la même manière, on ne modifiera jamais l'état stocké par Redux, nous allons voir comment nous en sortir autrement.

2. Découverte de Redux

Avant de nous lancer dans un exemple mettant en œuvre Redux, parcourons rapidement quelques concepts qu'il définit afin de comprendre comment il fonctionne. Il faut savoir que ce fonctionnement est en réalité très simple, mais comme souvent les choses les plus simples permettent, mises bout à bout, d'arriver aux solutions les plus élégantes.

2.1 Concepts de base

2.1.1 Le state

L'élément le plus important de Redux est le state. C'est en effet sa principale fonction : stocker l'état d'une application. Les données qu'il stocke peuvent être de tout type : objets, nombres, tableaux, fonctions, etc. Le state peut être lu (par un composant React ou autre) ; en revanche, il ne sera jamais modifié directement. Nous allons devoir indiquer à Redux comment en générer la version suivante.

2.1.2 Les actions

Afin de mettre à jour le state, la première étape sera de déclencher une action, par exemple au clic sur un bouton. On dira alors que l'action est *dispatchée*. Une action doit être un objet, constitué d'un type, et éventuellement d'autres données.

L'idéal est de concevoir une action comme un verbe, associé à des paramètres. Par exemple, si le state contient un compteur (exemple classique d'introduction à Redux), on pourrait avoir les actions *incrémenter* et *définirValeur(valeur)*, la seconde étant une action qui prend un paramètre *valeur*.

2.1.3 Le reducer

Le reducer est une fonction, prenant en paramètres un state et une action, et renvoyant un nouveau state. Par exemple, pour un compteur, à partir d'un state valant 5 et d'une action *incrémenter*, nous renverrions comme nouveau state la valeur 6. Nous ne mettons donc pas à jour le state, nous indiquons simplement à Redux une nouvelle version à prendre en compte.

Note importante : le reducer doit être une fonction pure, au sens défini par la programmation fonctionnelle, c'est-à-dire notamment :

- Que son comportement et sa valeur de retour doivent être déterministes : pour un ensemble de paramètres donnés, on doit toujours avoir le même retour.
- Qu'elle ne doit pas avoir d'effet de bord, c'est-à-dire modifier un état qui lui est extérieur. On ne pourra donc pas mettre à jour d'autres variables, dispatcher une autre action, ou encore déclencher un traitement asynchrone à base de promesses.

Cela peut paraître contraignant au premier abord, mais nous verrons comment Redux nous permet de nous en sortir tout de même de manière élégante. De plus ces contraintes sont à la base de ce qui rend une application Redux plus facile à maintenir.

2.1.4 Le store

Le store n'est autre que l'objet unifiant les notions que nous venons de voir. Nous l'initialisons au démarrage de l'application, en lui fournissant un reducer et un state initial. Puis nous pourrons :

- lire le state courant ;
- dispatcher des actions ;
- souscrire aux changements du state (c'est-à-dire appeler une fonction dès que le state est mis à jour).

Notez que pour un store nous n'aurons toujours qu'un seul reducer, mais qu'il est aisé comme nous le verrons de combiner des reducer pour n'en faire qu'un seul ; ce ne sera donc jamais une contrainte.

2.2 Premier exemple

Pour notre première utilisation de Redux nous n'utiliserons pas React. Il s'agira d'un simple compteur, avec un bouton permettant de l'incrémenter. L'exemple est ouvertement inspiré de l'exemple *counter-vanilla* de la documentation de Redux.

Comme dans le premier chapitre, nous utiliserons Parcel. Commençons donc par initialiser le projet et installer les dépendances nécessaires :

```
$ yarn init -y
$ yarn add --dev parcel-bundler babel-preset-env \
   babel-plugin-transform-object-rest-spread
$ yarn add redux
$ mkdir public src
```

Notre fichier public/index.html appelera le script *src/index.js*, mais affichera également le HTML de notre application, puisque pas de React ici :

```
<p>
 Compteur:
 <span id="counter"></span>
 <button onclick="handleIncrement()">+</button>
</p>
<script src="../src/index.js"></script>
```

Il n'y a pas encore de trace de Redux. Nous définissons un endroit où la valeur du compteur devra s'afficher (la balise `span` d'ID *counter*), et au clic sur le bouton nous appellerons une fonction `handleIncrement`, qui sera définie dans *src/index.js* par `window.handleIncrement =...`, car nous ne pouvons importer de fichier module JavaScript dans un fichier HTML.

Passons ensuite au fichier src/index.js. Commençons par définir le state initial de notre application. Nous stockerons uniquement un compteur pour le moment, ce pourrait donc être une simple valeur numérique : 0. Mais au cas où nous souhaiterions par la suite stocker d'autres informations, il est plus judicieux d'utiliser un objet :

```
// Initial state
const initialState = { counter: 0 }
```

Pour ce qui est des actions qui seront disponibles dans notre application, il n'y en aura qu'une seule, permettant d'incrémenter le compteur. Définissons donc le type associé à cette action :

```
// Action types
const INCREMENT = 'increment'
```

Afin de faciliter la création d'une action de ce type, définissons également une fonction `increment` :

```
// Action creators
const increment = () => ({
 type: INCREMENT
})
```

Pour avoir les éléments nécessaires pour créer notre store, il ne nous manque que le reducer. Il ne réagit qu'à un seul type d'action, `INCREMENT`, et dans ce cas renvoie comme compteur la valeur de l'attribut `counter` du state, incrémentée de 1 :

```
// Reducer
const reducer = (state = initialState, action) => {
 switch (action.type) {
   case INCREMENT:
     return { ...state, counter: state.counter + 1 }
   default:
     return state
 }
}
```

Quelques remarques sur les bonnes pratiques pour écrire un reducer :

- Le premier paramètre (le state courant), n'est pas défini au premier appel du reducer. Il est donc pratique d'utiliser notre `initialState` en valeur par défaut afin d'initialiser le state à ce moment.
- Structurer un reducer avec un bloc `switch` est la manière de faire la plus simple et la plus courante, mais n'a rien d'obligatoire.
- Utiliser la syntaxe `{ ...state, nouvellesValeurs }` permet d'une part de s'assurer qu'on ne modifie pas le state actuel, et d'autre part de ne mettre à jour que les attributs qui nous intéressent ici. En l'occurrence nous n'en avons qu'un seul (`counter`), mais la plupart du temps il y en aura d'autres, prenons donc dès maintenant de bonnes habitudes.

- Si l'action ne correspond à aucun type connu, nous devons tout de même renvoyer un state, nous renvoyons donc le state courant sans modification (cas `default` du `switch`). En effet Redux commence par dispatcher des actions à lui au démarrage.

Par ailleurs, remarquez que cette fonction est pure, comme définie dans la section précédente : nous ne modifions pas de valeur extérieure à la fonction (notamment le state), et elle n'entraîne pas d'effet de bord.

À présent, nous pouvons créer notre store. Commençons par importer (en haut du fichier) la fonction `createStore` de Redux :

```
import { createStore } from 'redux'
```

Puis créons donc ce store en lui donnant en paramètre le reducer :

```
// Store
const store = createStore(reducer)
```

Le store étant créé, nous pouvons à présent faire deux choses :

- lire son state grâce à `store.getState()` ;
- mettre à jour son state en dispatchant des actions grâce à `store.dispatch(...)`.

Pour afficher le compteur, créons une fonction `render`, dont le rôle sera d'afficher la valeur de l'attribut `counter` du state dans la balise `span` que nous avons définie dans le HTML :

```
// Render function
const render = () => {
 const { counter } = store.getState()
 document.getElementById('counter').innerHTML = counter
}
```

Nous souhaitons appeler cette fonction dès que le state est mis à jour. Pour cela, nous utiliserons `store.subscribe`, qui permet d'exécuter la fonction qui lui est passée en paramètre après chaque mise à jour, dans notre cas, `render` :

```
// Store subscriptions
store.subscribe(render)
```

De plus, afin d'afficher la valeur initiale du compteur, nous pouvons appeler une première fois `render` (par exemple en bas du fichier) :

```
// Initial rendering
render()
```

Il nous reste une chose à faire : définir la fonction `handleIncrement`, appelée dès que le bouton est cliqué. Dans cette fonction, nous créerons une action de type `INCREMENT` grâce à la fonction `increment` que nous avons écrite, puis nous dispatcherons cette action grâce à `store.dispatch(...)` :

```
// Event handlers
window.handleIncrement = () => {
 store.dispatch(increment())
}
```

Nous avons fini d'écrire notre application ! Lancez l'application avec `yarn start` puis observez que tout fonctionne comme prévu.

L'une des remarques que j'entends le plus souvent auprès de personnes utilisant Redux pour la première fois est que cela paraît compliqué, et un brin magique dans le sens où Redux agit comme une boîte noire dont on ne connaît pas le comportement interne. Pourtant, si vous prenez un peu de recul sur le code que nous venons d'écrire, vous pourrez voir que ce que nous savons de Redux à ce niveau nous permet d'en écrire une implémentation simplifiée, mais répondant à notre besoin.

Exercice

Comme premier exercice, je vous propose donc de compléter le fichier own-redux.js suivant, et de remplacer la ligne où l'on importe Redux dans notre application par cette version (`import { createStore } from './own-redux'`), le but étant évidemment que notre version soit compatible avec ce que nous avons déjà écrit :

```
export const createStore = reducer => ({
 // Membres privés
 _state: /* ??? */,
 _subscribers: [],
 // Méthodes publiques
 getState() { /* ??? */ },
 subscribe(subscriber) { /* ??? */ },
 dispatch(action) { /* ??? */ }
```

```
})
```

Correction

Tout d'abord, pour l'initialisation du state, nous avons vu que le reducer doit être capable de renvoyer la version initiale du state lorsque le premier paramètre lui étant passé est `undefined` :

```
_state: reducer(undefined, 'redux-init'),
```

La méthode `getState` est relativement simple, puisqu'il suffit de renvoyer l'attribut privé `_state`. De même pour `subscribe`, nous n'avons qu'à ajouter à `_subscribers` la fonction passée en paramètre :

```
getState() {
 return this._state
},
subscribe(subscriber) {
 this._subscribers.push(subscriber)
},
```

Enfin, dans `dispatch`, nous avons deux choses à faire : d'abord mettre à jour le state en appelant le reducer avec l'action passée en paramètre, puis appeler chaque `subscriber` pour le notifier de la mise à jour du state :

```
dispatch(action) {
 // On met à jour le state en appelant le reducer.
 this._state = reducer(this._state, action)

 // On appelle chaque subscriber.
 this._subscribers.forEach(subscriber => {
   subscriber()
 })
}
```

Notre implémentation minimaliste de Redux est complète. Vu comme cela, Redux n'est finalement pas bien compliqué. Bien évidemment, comme nous le verrons dans les sections suivantes, Redux propose beaucoup d'autres fonctionnalités, comme celle de combiner plusieurs reducer, ou encore d'utiliser des middlewares pour gérer des actions asynchrones comme des requêtes Ajax, etc.

Mais avant d'aller plus en profondeur dans Redux, voyons comment enfin l'utiliser dans une application React.

3. Utilisation avec React

Dans l'exemple précédent, vous avez dû penser qu'utiliser Redux était laborieux : beaucoup d'opérations sont nécessaires pour un résultat modeste, etc. Mais c'est en le couplant avec React (ou une autre bibliothèque comme Vue.js ou Angular) que tout son potentiel est visible. Il va nous permettre d'extraire un maximum de logique métier (mise à jour du state, appels à une API, etc.) des composants, pour n'y conserver que leur responsabilité originale : le rendu.

3.1 Découverte de React-Redux

Utiliser Redux comme nous venons de le voir avec un composant React serait tout à fait envisageable. On pourrait par exemple créer notre store, puis le passer en propriété de chaque composant de l'application, qui serait alors en mesure d'afficher des valeurs du state, et de dispatcher des actions. Il y aurait cependant certaines subtilités à gérer, et cela resterait laborieux.

Heureusement, la bibliothèque React-Redux est là pour nous faciliter grandement la tâche. En effet, nous n'aurons pas besoin de nous soucier du store, car elle nous permet de définir pour certains composants :

- à quels éléments du state nous souhaitons avoir accès (pour les afficher par exemple) ;
- quelles actions nous souhaitons être mesure de dispatcher.

Reprenons par exemple notre compteur. Créons un composant similaire à ce que nous avons vu dans la section précédente, mais bien en React cette fois-ci et non en HTML classique :

```
// Counter.js
const Counter = ({ counter, increment }) => (
 <p>
   Compteur:
   <span>{counter}</span>
   <button onClick={() => increment()}>+</button>
 </p>
)

Counter.propTypes = {
```

```
  counter: PropTypes.number.isRequired,
  increment: PropTypes.func.isRequired
}

export default Counter
```

Rien de nouveau dans ce composant ; on attend que deux propriétés lui soient passées : la valeur du compteur `counter` et une fonction `increment` permettant d'incrémenter le compteur. Pas encore de trace de Redux, nous y venons.

Où souhaitons-nous faire intervenir Redux ici ? Tout d'abord le résultat auquel nous souhaitons arriver est de pouvoir faire appel à notre composant ainsi :

```
<Counter />
```

C'est-à-dire sans lui passer manuellement les propriétés `counter` ni `increment`; nous aimerions indiquer par un quelconque moyen que `counter` doit être initialisé avec l'attribut `counter` du state, et que `increment` soit une fonction qui dispatcherait l'action `INCREMENT` dans notre store Redux afin d'en mettre à jour le state.

C'est exactement ce que fait React-Redux. Nous aurons ici deux fonctions à définir. Tout d'abord une fonction permettant de récupérer les éléments du state qui nous intéressent. Par convention, cette fonction est appelée `mapStateToProps` :

```
const mapStateToProps = state => ({
  counter: state.counter
})
```

Comme son nom l'indique, cette fonction va nous permettre d'indiquer à React-Redux quel *mapping* (correspondances) nous souhaitons effectuer entre les attributs du state et les propriétés du composant.

La seconde fonction est similaire et nous permet d'effectuer le mapping entre les actions que l'on veut dispatcher et les propriétés du composant ; de même par convention on l'appelle `mapDispatchToProps` :

```
const mapDispatchToProps = dispatch => ({
  increment: () => dispatch(increment())
})
```

Notez que React-Redux nous offre la possibilité d'écrire ce mapping d'une manière plus concise, comme un objet et non une fonction :

```
const mapDispatchToProps = {
 increment: () => increment()
}
```

Les deux formes sont totalement équivalentes ; je préfère personnellement la deuxième car plus concise.

Une fois ces deux fonctions définies, il nous reste à indiquer à React-Redux comment les utiliser. Cela se fait en utilisant la fonction `connect`, qui prend en paramètre les deux fonctions. La syntaxe de son utilisation peut paraître un peu déroutante, car elle renvoie une nouvelle fonction, à laquelle on passe en paramètre un composant pour obtenir en retour un nouveau composant, augmenté du lien avec Redux :

```
const CounterWithRedux = connect(
 mapStateToProps,
 mapDispatchToProps
)(Counter)
```

Nous verrons dans un prochain chapitre que la fonction renvoyée par `connect` est en réalité ce que l'on appelle un *high-order component* : c'est-à-dire une fonction renvoyant un composant à partir d'un autre composant.

Voici à quoi peut ressembler notre fichier counter.js maintenant avec l'utilisation de React-Redux :

```
// Counter.js
import React from 'react'
import PropTypes from 'prop-types'
import { connect } from 'react-redux'
import { actions } from './store'

const Counter = ({ counter, increment }) => (
 <p>
   Compteur:
   <span>{counter}</span>
   <button onClick={() => increment()}>+</button>
 </p>
)

Counter.propTypes = {
```

```
 counter: PropTypes.number.isRequired,
 increment: PropTypes.func.isRequired
}

const mapStateToProps = state => ({
 counter: state.counter
})

const mapDispatchToProps = {
 increment: () => actions.increment()
}

export default connect(
 mapStateToProps,
 mapDispatchToProps
)(Counter)
```

Notez qu'ici il est supposé que nous avions un fichier store.js, exportant notamment un objet `actions` contenant la fonction `increment` qui crée une action pour incrémenter notre compteur. Ce fichier peut ressembler comme deux gouttes d'eau à ce que nous avons vu dans l'exemple de la section précédente :

```
// store.js
import { createStore } from 'redux'

const initialState = { counter: 0 }

const actionTypes = {
 INCREMENT: 'increment'
}

export const actions = {
 increment: () => ({
   type: actionTypes.INCREMENT
 })
}

const reducer = (state = initialState, action) => {
 switch (action.type) {
   case actionTypes.INCREMENT:
     return { ...state, counter: state.counter + 1 }
   default:
     return state
```

```
  }
}

export default createStore(reducer)
```

Ici nous exportons notamment les actions afin de les rendre disponibles dans notre composant. De plus, comme vous le voyez nous exportons également le store, car nous allons devoir le connecter à React-Redux (il faut bien que `connect` soit capable de savoir à quel store se référer pour récupérer le state et dispatcher les actions).

Pour cela, React-Redux nous offre, en plus de la fonction `connect`, un composant appelé `Provider`, auquel nous passons en propriété le store, et dans lequel nous englobons toute notre application, ou du moins toute la partie à laquelle nous souhaitons donner accès au store. J'ai choisi ici de faire cela dans notre fichier principal index.js :

```
// index.js
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'
import Counter from './Counter'
import store from './store'

ReactDOM.render(
 <Provider store={store}>
   <Counter />
 </Provider>,
 document.getElementById('app')
)
```

L'avantage de `Provider`, c'est qu'il n'est nécessaire de l'utiliser qu'une seule fois : tous les composants inclus à l'intérieur, y compris les composants appelés par d'autres composants, auront accès au store via l'utilisation de `connect`. Cela peut vous paraître un brin magique ; en effet jusque-là nous n'avons jamais pu accéder à des données depuis un composant autrement que par ses propriétés. En fait, React-Redux utilise une fonctionnalité de React que nous n'avons pas encore vue : les contextes. Nous aurons l'occasion d'en reparler dans les prochains chapitres.

Nous avons maintenant tous les éléments qui nous permettent de lancer notre application. Pensez juste à ajouter comme dépendance React-Redux (`yarn add react-redux`) en plus des dépendances habituelles, et vous devriez obtenir un résultat similaire à l'exemple de la section précédente, mais avec React cette fois-ci !

3.2 Connecter un composant à Redux à l'aide des hooks

Dans le chapitre précédent, nous avons découvert les hooks, arrivés dans React il y a peu, et facilitant le développement de composants ayant une certaine logique à gérer, comme garder un state local (`useState`) ou effectuer une opération à leur chargement (`useEffect`). Je vous avais promis que la plupart des grandes bibliothèques se mettaient à leur tour à proposer des hooks pour simplifier leur API, et bien bonne nouvelle, React-Redux en fait partie.

Les hooks vont nous permettre de nous affranchir de définir les fonctions `mapStateToProps` et `mapDispatchToProps`. Nous n'aurons plus non plus à appeler la fonction `connect` au moment d'exporter le composant. Du point de vue extérieur, notre composant sera tout ce qu'il y a de plus basique.

Concrètement, React-Redux met à disposition les deux hooks suivants :

- `useSelector` prend en paramètre un sélecteur, c'est-à-dire une fonction qui à partir du state renvoie la valeur que l'on souhaite. Cette valeur est à son tour renvoyée par le hook.
- `useDispatch` renvoie simplement une fonction permettant de dispatcher une action.

Voici à quoi ressemble le composant `Counter` lorsqu'on utilise des hooks pour le connecter à Redux :

```
import React, { useCallback } from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { actions } from './store'

const Counter = () => {
 const counter = useSelector(state => state.counter)
 const dispatch = useDispatch()
```

```
  const increment = useCallback(() => {
    dispatch(actions.increment())
  }))

  return (
    <p>
      Compteur:
      <span>{counter}</span>
      <button onClick={increment}>+</button>
    </p>
  )
}

export default Counter
```

Tout d'abord, nous récupérons la valeur de l'attribut `counter` du state en appelant `useSelector`, puis nous créons une fonction `dispatch` en appelant `useDispatch`. Nous utilisons ensuite le hook `useCallback` vu au chapitre précédent. Pour rappel, celui-ci est utilisé pour créer des fonctions qui seront passées en propriété à un composant, et éviter des nouveaux rendus inutiles. Dans le callback `increment` ainsi créé, nous dispatchons une action `increment`.

Notez que la philosophie pour dispatcher une action est quelque peu différente de la méthode classique en utilisant `mapDispatchToProps` et non le hook `useDispatch` :

- Avec `mapDispatchToProps` et `connect`, on fournit le créateur d'action (ici `actions.increment`), et `connect` fait en sorte que le composant reçoive une propriété nommée également `increment`, permettant de dispatcher l'action.
- Avec `useDispatch`, on appelle explicitement le créateur d'action pour créer une action, que l'on dispatche nous-mêmes.

La raison de ce changement est qu'avec `mapDispatchToProps`, il pouvait être difficile de comprendre que la fonction que l'on recevait dans le composant n'était pas le créateur d'action, mais bien une fonction créée par React-Redux. Il était donc facile pour un débutant de faire l'erreur d'appeler le créateur d'action plutôt que cette fonction, et ainsi de ne pas comprendre pourquoi l'application ne donnait pas le résultat escompté.

Avec le hook `useDispatch`, c'est beaucoup plus explicite, car il laisse moins de place à une mauvaise compréhension.

Vous pouvez constater que le composant paraît moins complexe lorsqu'il est écrit avec des hooks. Ou du moins, que le fichier est moins verbeux. Finalement, le résultat est le même, ainsi que la complexité réelle. C'est plus une question de préférence qui fera que l'on utilisera les hooks ou le traditionnel `connect`.

Au moment de l'écriture de ce livre, les hooks de React-Redux sont encore très récents, et la plupart des projets utilisent encore l'approche sans les hooks, mais cela tend à évoluer rapidement. En tout cas, dans la suite de ce chapitre, je continuerai de présenter les exemples avec `connect`, `mapStateToProps` et `mapDispatchToProps`, mais un très bon exercice pourra consister à faire évoluer les composants pour utiliser les hooks à la place.

4. Actions complexes et asynchrones

Jusque-là, notre manière d'utiliser Redux est restée relativement simple :

- dans le composant nous dispatchons une action ;
- l'action passe dans le reducer, qui met à jour le state ;
- le composant est réaffiché pour tenir compte du nouveau state.

Si ce processus convient déjà à beaucoup de besoins, certains vont nous poser quelques problèmes : comment peut-on indiquer qu'une action doit à son tour dispatcher une autre action ? Comment peut-on faire en sorte qu'une action déclenche une requête HTTP (asynchrone donc) ?

Pour répondre à ces problèmes, il existe une bibliothèque qui va venir se greffer à Redux : Redux-Thunk (https://github.com/reduxjs/redux-thunk). Celle-ci se présente sous forme d'un middleware pour Redux, c'est-à-dire que l'on va indiquer à Redux que l'on souhaite l'utiliser, et cela suffira à ajouter de nouvelles possibilités que nous allons voir.

Afin de présenter Redux-Thunk, nous allons créer un chronomètre minimaliste, qui offrira les fonctionnalités suivantes :

- être démarré, déclenchant ainsi l'affichage du nombre de secondes écoulées à partir de ce moment ;
- être mis en pause momentanément ;
- être remis à zéro.

Cet exemple ne pose aucune difficulté technique, d'ailleurs grâce au premier chapitre de ce livre, en utilisant un state local pour le composant, vous n'auriez sans doute aucun problème pour le réaliser. Mais ici nous souhaitons utiliser Redux !

Commençons par initialiser un projet semblable à ceux que nous avons créés jusque-là. Pour cela, vous pouvez récupérer les fichiers package.json et public/index.html d'un exemple précédent, comme celui associé à la section sur React-Redux.

Nous allons tout d'abord créer notre store grâce à ce que nous savons déjà faire. Le state sera composé d'un attribut `time` contenant le temps écoulé en millisecondes, ainsi qu'un booléen `isRunning`, vrai si le chronomètre a été démarré et n'a pas été mis en pause :

```
// store.js
import { createStore } from 'redux'

const initialState = {
 time: 0,
 isRunning: false
}
```

Nous disposerons de trois actions `start`, `pause` et `reset` pour interagir avec le chronomètre :

```
const actionTypes = {
 START: 'start',
 PAUSE: 'pause',
 RESET: 'reset'
}

export const actions = {
 start: () => ({
```

```
    type: actionTypes.START
  }),
  pause: () => ({
    type: actionTypes.PAUSE
  }),
  reset: () => ({
    type: actionTypes.RESET
  })
}
```

Le reducer est pour le moment extrêmement simple :

```
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case actionTypes.START:
      return { ...state, isRunning: true }
    case actionTypes.PAUSE:
      return { ...state, isRunning: false }
    case actionTypes.RESET:
      return { ...state, time: 0 }
    default:
      return state
  }
}

export default createStore(reducer)
```

Pour ce qui est du composant, rien de nouveau non plus : nous affichons le temps écoulé (avec un formatage afin d'afficher le temps en secondes avec trois décimales), et trois boutons pour dispatcher les actions. Bien évidemment, nous utilisons React-Redux pour accéder au state et aux actions du store :

```
// Stopwatch.js
import React from 'react'
import { connect } from 'react-redux'
import { actions } from './store'

const formatTime = time =>
  (time / 1000).toLocaleString(undefined, {
minimumFractionDigits: 3 })

const Stopwatch = ({ time, isRunning, start, pause, reset }) => (
  <div style={{ fontFamily: 'monospace' }}>
```

```
    {formatTime(time)}
    {isRunning ? (
      <button onClick={() => pause()}>Pause</button>
    ) : (
      <button onClick={() => start()}>Start</button>
    )}
    <button onClick={() => reset()}>Reset</button>
  </div>
)

const mapStateToProps = ({ time, isRunning }) => ({ time,
isRunning })

const mapDispatchToProps = { ...actions }

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Stopwatch)
```

Enfin, le fichier index.js qui permet de rassembler le tout :

```
// index.js
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'
import Counter from './Counter'
import store from './store'

ReactDOM.render(
  <Provider store={store}>
    <Counter />
  </Provider>,
  document.getElementById('app')
)
```

Si vous lancez l'application, vous verrez bien affiché « 0.000 », mais les boutons n'auront pas beaucoup d'effet si ce n'est de changer le bouton *Start* par le bouton *Pause* et vice-versa. Et pour cause, à aucun moment nous n'avons indiqué comment déclencher le chronomètre, c'est-à-dire incrémenter le temps écoulé.

Ce que nous souhaiterions faire consiste, lorsque l'action START est dispatchée, à incrémenter à chaque milliseconde le compteur, et donc pour cela dispatcher une nouvelle action à chaque incrément (ici nous utiliserons un incrément de 100, toutes les 100 millisecondes donc).

Commençons par créer l'action :

```
const actionTypes = {
 // ...
 INCREMENT: 'increment'
}

export const actions = {
 // ...
 increment: () => ({
   type: actionTypes.INCREMENT
 })
}

const reducer = (state = initialState, action) => {
 switch (action.type) {
   // ...
   case actionTypes.INCREMENT:
     return { ...state, time: state.time + 100 }
   // ...
 }
}
```

Il ne reste qu'à appeler cette action ; mais comment déclencher une action plusieurs fois et à une certaine fréquence ? Nous pourrions envisager de la déclencher dans le composant, mais en utilisant Redux nous souhaitons justement éviter de donner cette responsabilité aux composants (et qu'en serait-il si plusieurs composants doivent pouvoir gérer le chronomètre ?). Et il est bien évidemment exclu par Redux de déclencher une action dans le reducer. Il reste un endroit où nous pourrions déclencher l'action, il s'agit de la fonction permettant de créer une action. Et c'est exactement ce que Redux-Thunk nous permet de faire !

En effet, jusque-là, nous n'avons toujours renvoyé qu'un objet contenant le type et les paramètres de l'action :

```
export const actions = {
 start: () => ({
   type: actionTypes.START
 })
 // ...
}
```

Redux-Thunk va nous permettre de renvoyer un autre type d'objet, en l'occurrence une fonction qui sera appelée avec comme paramètre une fonction `dispatch` que nous pourrons appeler pour dispatcher n'importe quelles actions, autant de fois que nous le souhaitons, et de manière asynchrone si nécessaire.

Nous pouvons donc, lorsqu'une action `START` est créée, appeler `setInterval` pour dispatcher à une fréquence donnée notre action `INCREMENT` :

```
export const actions = {
 start: () => dispatch => {
   setInterval(() => dispatch(actions.increment()), 100)
   dispatch({ type: actionTypes.START })
 }
 // ...
}
```

Remarquez que nous dispatchons toujours explicitement une action `START` (qui n'est donc plus exactement créée par l'appel à `start()`) qui, elle, arrivera au reducer afin de mettre à jour l'attribut `isRunning`. En effet, notez bien que si une action est dispatchée et que celle-ci est une fonction, alors c'est Redux-Thunk qui la prendra en charge, et elle n'arrivera donc jamais jusqu'au reducer.

Il ne reste qu'à ajouter le middleware Redux-Thunk à notre store. Pour cela nous devons également importer la fonction `applyMiddleware` de Redux :

```
import { createStore, applyMiddleware } from 'redux'
import thunk from 'redux-thunk'

// ...

export default createStore(reducer, applyMiddleware(thunk))
```

À présent, vous devriez voir votre chronomètre afficher le temps écoulé à partir du clic sur le bouton *Start*. Toutefois, le bouton *Pause* ne le stoppe pas encore. Pour cela nous avons plusieurs possibilités. Nous pouvons faire en sorte que le temps ne soit incrémenté que si l'attribut `isRunning` du state est à `true`. La méthode choisie ici consiste plutôt à profiter du fait qu'il est possible de stopper un appel à `setInterval` grâce à `clearInterval`. Nous allons devoir pour cela stocker l'ID renvoyé par `setInterval` dans le state.

```
export const actions = {
 start: () => dispatch => {
   const intervalId = setInterval(() =>
dispatch(actions.increment()), 100)
   dispatch({ type: actionTypes.START, intervalId })
 }
 // ...
}

const reducer = (state = initialState, action) => {
 switch (action.type) {
   case actionTypes.START:
     return { ...state, isRunning: true, intervalId:
action.intervalId }
   // ...
 }
}
```

Pour ce qui est de l'action `PAUSE`, nous allons également utiliser ce que propose Redux-Thunk, car nous allons devoir faire un appel à `clearInterval` en lui passant l'ID stocké dans le state. Pour cela, à la fonction que nous renvoyons à la création de l'action, Redux-Thunk passe en plus de `dispatch` une autre fonction `getState`, qui comme son nom l'indique nous permet de récupérer des valeurs stockées dans le state :

```
export const actions = {
 // ...
 pause: () => (dispatch, getState) => {
   const { intervalId } = getState()
   clearInterval(intervalId)
   dispatch({ type: actionTypes.PAUSE })
 }
 // ...
}
```

```
const reducer = (state = initialState, action) => {
 switch (action.type) {
   // ...
   case actionTypes.PAUSE:
     return { ...state, isRunning: false, intervalId: null }
   // ...
 }
}
```

Ainsi, le bouton *Pause* devrait à présent avoir le comportement attendu ! Comme vous pouvez le voir, créer des actions déclenchant des traitements asynchrones ou dispatchant elles-mêmes des actions n'est pas si compliqué. Il s'agit d'un pattern commun, dont l'un des principaux cas d'utilisation est évidemment d'appeler une API à l'aide d'une requête HTTP.

Nous allons voir dans la dernière section de ce chapitre comment mettre en place ce cas d'usage à l'aide d'un exemple complet, ce qui nous permettra non seulement de résumer ce que nous avons vu dans ce chapitre, mais également de voir comment concevoir une application React-Redux légèrement plus complexe, en séparant la logique du store en plusieurs modules.

Mais avant cela, si le cœur vous en dit, voici un petit exercice ayant rapport à l'exemple de cette section.

Exercice

Le chronomètre que nous avons développé dans cette section n'est en réalité pas très précis. En effet, même si nous demandons d'incrémenter le compteur toutes les 100 millisecondes, rien de garantit que Redux dispatchera les bonnes actions au bout d'exactement 100 millisecondes. Cela est d'ailleurs plus flagrant si vous décidez d'incrémenter le compteur à chaque milliseconde : on a alors l'impression que le temps est ralenti !

Sauriez-vous revoir notre chronomètre, afin non pas d'incrémenter le compteur à une fréquence donnée, mais plutôt de stocker dans le state l'heure exacte de démarrage du chronomètre (son *timestamp* : `Date.now()`), et à cette fréquence donnée effectuer une soustraction entre le *timestamp* actuel et la date de démarrage ? Sauriez-vous comment gérer la pause dans ce cas ?

5. Un exemple complet

À présent que nous avons vu les principales fonctionnalités de Redux, voyons comment mettre tout cela en pratique avec un exemple plus complet. Dans cette section nous allons réaliser une application minimaliste permettant d'afficher la liste des sujets postés sur le *subreddit* consacré à React. Lorsque l'utilisateur cliquera sur le titre d'un sujet, nous afficherons alors le texte associé à ce sujet.

Concernant l'appel à l'API de Reddit, c'est relativement simple puisque la plupart du temps il suffit, à partir d'une page Reddit (par exemple `https://www.reddit.com/r/reactjs`), d'ajouter le suffixe « .json » pour obtenir les mêmes données que la page, mais au format JSON : `https://www.reddit.com/r/reactjs.json`. Cette URL est la première que nous appellerons pour obtenir la liste des sujets postés. Pour obtenir les détails sur un sujet (et notamment le texte associé), l'attribut `permalink` dans le JSON nous permettra de construire l'URL à appeler pour obtenir le JSON.

Comme cet exemple est plus complexe que ce que nous avons vu précédemment, nous allons voir comment scinder notre store en différents modules, que nous appellerons *services*. Cette terminologie n'a rien d'un standard, et il se peut que vous entendiez un autre terme à la place. Pour ma part, j'entends par service un module exportant des actions et un reducer.

Cet exemple comportant plus de code qu'habituellement, il ne sera peut-être pas facile de le réaliser par vous-même à la simple lecture de ce livre. Aussi vous pouvez vous rendre directement sur le dépôt GitHub des exemples du livre afin de le récupérer et le faire tourner.

5.1 Les services

La partie store de notre application sera constituée de trois services :

- `posts` s'occupera de la récupération des sujets du subreddit React.
- `postsDetails` ira chercher les détails pour les sujets (lorsqu'ils seront cliqués).

- `routing` se chargera du cycle de vie de l'application, c'est-à-dire d'afficher un sujet spécifique dès qu'il est cliqué, et de réafficher la liste lorsque le bouton retour est cliqué.

Chacun de ces services est relativement simple, et pris chacun indépendamment, ils ne présentent pas de nouveauté par rapport à ce que nous avons vu. Commençons par le service `posts` :

```
// services/posts/actions.js
export const FETCH_POSTS = 'FETCH_POSTS'
export const FETCH_POSTS_BEGIN = 'FETCH_POSTS_BEGIN'
export const FETCH_POSTS_SUCCESS = 'FETCH_POSTS_SUCCESS'
export const FETCH_POSTS_ERROR = 'FETCH_POSTS_ERROR'

export const fetchPosts = () => async dispatch => {
 dispatch(fetchPostsBegin())
 try {
   const res = await
     fetch('https://www.reddit.com/r/reactjs.json')
   const posts = (await res.json()).data.children
     .map(child => child.data)
   dispatch(fetchPostsSuccess(posts))
 } catch (err) {
   dispatch(fetchPostsError(err))
 }
}

export const fetchPostsBegin = () => ({
 type: FETCH_POSTS_BEGIN
})

export const fetchPostsSuccess = posts => ({
 type: FETCH_POSTS_SUCCESS,
 posts
})

export const fetchPostsError = error => ({
 type: FETCH_POSTS_ERROR,
 error
})

//
```

```
services/posts/reducer.js
import {
 FETCH_POSTS_BEGIN,
 FETCH_POSTS_SUCCESS,
 FETCH_POSTS_ERROR
} from './actions'

const initialState = {
 posts: [],
 error: null,
 isFetching: false
}

export default (state = initialState, action) => {
 switch (action.type) {
   case FETCH_POSTS_BEGIN:
     return { ...state, isFetching: true, error: null }
   case FETCH_POSTS_SUCCESS:
     return { ...state, isFetching: false,
       posts: action.posts }
   case FETCH_POSTS_ERROR:
     return { ...state, isFetching: false,
       error: action.error }
   default:
     return state
 }
}
```

Comme nous l'avions déjà fait dans la section précédente d'introduction à Redux-Thunk, nous avons une première action `fetchPosts` qui va effectuer une action asynchrone (en l'occurrence un appel HTTP à l'API de Reddit), et en fonction du résultat de cette-ci appeler une action `success` ou une action `error`.

Le service `postsDetails` est relativement similaire. Ici nous ne souhaitons pas garder les détails que pour un seul sujet, c'est pourquoi notre state est un objet dont les clés seront les ID des sujets dont on a au moins demandé la récupération des détails (et dont la récupération peut être en cours, terminée ou avoir déclenché une erreur).

```
// services/postsDetails/actions.js
export const FETCH_POST_DETAILS = 'FETCH_POST_DETAILS'
export const FETCH_POST_DETAILS_BEGIN =
 'FETCH_POST_DETAILS_BEGIN'
export const FETCH_POST_DETAILS_SUCCESS =
 'FETCH_POST_DETAILS_SUCCESS'
export const FETCH_POST_DETAILS_ERROR =
 'FETCH_POST_DETAILS_ERROR'

export const fetchPostDetails = post => async dispatch => {
 dispatch(fetchPostDetailsBegin(post))
 try {
   const res = await
     fetch(`https://www.reddit.com${post.permalink}.json`)
   const details = (await res.json())[0].data.children[0].data
   dispatch(fetchPostDetailsSuccess(post, details))
 } catch (err) {
   dispatch(fetchPostDetailsError(post, err))
 }
}

export const fetchPostDetailsBegin = post => ({
 type: FETCH_POST_DETAILS_BEGIN,
 post
})

export const fetchPostDetailsSuccess = (post, details) => ({
 type: FETCH_POST_DETAILS_SUCCESS,
 post,
 details
})

export const fetchPostDetailsError = (post, error) => ({
 type: FETCH_POST_DETAILS_ERROR,
 post,
 error
})
```

```
// services/postsDetails/reducer.js
import {
 FETCH_POST_DETAILS_BEGIN,
 FETCH_POST_DETAILS_SUCCESS,
 FETCH_POST_DETAILS_ERROR
} from './actions'

const initialState = {}

export default (state = initialState, action) => {
 switch (action.type) {
   case FETCH_POST_DETAILS_BEGIN:
     return {
       ...state,
       [action.post.id]: {
         ...state[action.post.id],
         isFetching: true,
         error: null
       }
     }
   case FETCH_POST_DETAILS_SUCCESS:
     return {
       ...state,
       [action.post.id]: {
         ...state[action.post.id],
         isFetching: false,
         details: action.details
       }
     }
   case FETCH_POST_DETAILS_ERROR:
     return {
       ...state,
       [action.post.id]: {
         ...state[action.post.id],
         isFetching: false,
         error: action.error
       }
     }
   default:
     return state
 }
}
```

Enfin, le dernier service est beaucoup plus simple puisqu'il n'y a pas d'action asynchrone à déclencher. Simplement deux actions mettant à jour le state via le reducer :

```
// services/routing/actions.js
export const OPEN_POST = 'OPEN_POST'
export const CLOSE_POST = 'CLOSE_POST'

export const openPost = post => ({ type: OPEN_POST, post })

export const closePost = () => ({ type: CLOSE_POST })
```

```
// services/routing/reducer.js
import { OPEN_POST, CLOSE_POST } from './actions'

const initialState = { openedPost: null }

export default (state = initialState, action) => {
 switch (action.type) {
   case OPEN_POST:
     return { ...state, openedPost: action.post }
   case CLOSE_POST:
     return { ...state, openedPost: null }
   default:
     return state
 }
}
```

Maintenant que nos trois reducer sont écrits, comment les intégrer au sein d'un seul et même store, qui lui ne peut avoir qu'un seul reducer ? Pour cela nous allons utiliser la fonction `combineReducer` offerte par Redux. Comme son nom l'indique, cette méthode permet de combiner plusieurs reducer pour n'en faire qu'un seul :

```
// services/store.js
import { createStore, applyMiddleware, combineReducers }
 from 'redux'
import thunk from 'redux-thunk'
import postsDetailsReducer from './postsDetails/reducer'
import postsReducer from './posts/reducer'
import routingReducer from './routing/reducer'
```

```
const reducer = combineReducers({
 postsDetails: postsDetailsReducer,
 posts: postsReducer,
 routing: routingReducer
 })

 export default createStore(reducer, applyMiddleware(thunk))
```

Nous passons donc un objet en paramètre de `combineReducers`. Les clés de cet objet seront des clés du state de notre application. C'est-à-dire que dans notre cas, pour accéder aux sujets stockés dans l'attribut `posts` du state déclaré par le service `posts` (donc ici le reducer `postsReducer`), nous utiliserons `state.posts.posts` (dans `mapStateToProps` notamment).

5.2 Les composants

Notre application sera constituée de deux pages : la première affichant la liste des sujets, la seconde les détails d'un sujet donné. C'est le composant principal App qui affichera le bon composant en fonction du state :

```
// components/App.js
import React from 'react'
import PropTypes from 'prop-types'
import { connect } from 'react-redux'
import Posts from './Posts'
import SinglePost from './SinglePost'

const App = ({ openedPost, closePost }) =>
 openedPost ? <SinglePost /> : <Posts />

const mapStateToProps = state => ({
 openedPost: state.routing.openedPost
})

export default connect(mapStateToProps)(App)
```

Si l'attribut `openedPost` du state du service `routing` est défini, on affichera le composant `SinglePost`, sinon on affichera la liste des sujets avec le composant `Posts`.

Le code des composants `Posts` et `SinglePost` n'apportant pas grand-chose de nouveau par rapport à ce que nous avons vu jusque-là, je ne mettrai pas l'intégralité du code source ici. Voici cependant les parties du code qui nous permettent d'utiliser notre state :

```
// components/Posts.js

// ...
const mapStateToProps = state => ({
 posts: state.posts.posts,
 error: state.posts.error,
 isFetching: state.posts.isFetching
})

const mapDispatchToProps = { fetchPosts, openPost }

export default connect(
 mapStateToProps,
 mapDispatchToProps
)(Posts)
```

```
// components/SinglePost.js

// ...
const mapStateToProps = state => ({
 openedPost: state.routing.openedPost,
 postsDetails: state.postsDetails
})

const mapDispatchToProps = { closePost, fetchPostDetails }

export default connect(
 mapStateToProps,
 mapDispatchToProps
)(SinglePost)
```

Notez que dans `mapStateToProps` nous avons accès à l'ensemble du state et pas simplement à celui d'un service. Rien n'oblige à avoir un service par composant par exemple, bien qu'assez souvent dans une application il y ait tout de même une ressemblance importante entre l'organisation des composants et celle des services.

Il est même de plus en plus courant de regrouper les composants et services d'une même fonctionnalité dans le même répertoire, voire dans un module distinct (un paquet NPM par exemple).

Petit élément intéressant, c'est dans les composants que l'on déclenche la récupération des données, à savoir la liste des sujets pour `Posts` et les détails d'un sujet pour `SinglePost` :

```
// components/Posts.js
// ...
componentDidMount() {
 const { fetchPosts, posts } = this.props
 if (posts.length === 0) {
   fetchPosts()
 }
}
// ...
```

```
// components/SinglePost.js
// …
componentDidMount() {
 const { openedPost, postsDetails, fetchPostDetails } = this.props
 if (!postsDetails[openedPost.id]) {
   fetchPostDetails(openedPost)
 }
}
// ...
```

Une méthode alternative serait de déclencher l'action `fetchPosts` à l'initialisation du store, ou encore d'y déclencher une action d'initialisation, qui elle se chargerait de dispatcher les bonnes actions.

Bien que cet exemple reste relativement simple, il illustre la plupart des notions que nous avons vues dans ce chapitre. Je vous encourage vivement à le faire tourner sur votre machine, et à essayer de le modifier pour observer le comportement de l'application.

Par exemple, à titre d'exercice, vous pouvez essayer d'ajouter le nom de l'utilisateur ayant posté un sujet, et lorsque l'utilisateur clique sur celui-ci, afficher une nouvelle page contenant des informations sur l'utilisateur.

Cela vous donnera l'occasion de créer un nouveau service.

6. Conclusion

Voilà qui est fait pour notre exploration de Redux. J'espère que vous aurez apprécié les avantages qu'il procure par rapport à l'utilisation d'un state local à un composant, notamment en termes de maintenabilité et d'architecture. En guise de conclusion, laissez-moi évoquer l'une des questions qui pose encore du souci à de nombreux développeurs React et divise parfois la communauté : **Redux ou state local ?**

Si vous avez déjà essayé de créer une grosse application, que ce soit avec React ou non, stocker les données de l'application de manière à ce qu'elles soient accessibles de partout, tout en gérant efficacement leur mise à jour, vous est sans doute déjà apparu comme un problème conséquent. Redux apporte une solution à ce problème, avec ses avantages et ses inconvénients. Pour l'avoir utilisé intensément, je lui attribue les avantages suivants :

- Les données sont facilement accessibles depuis n'importe quel composant, sans avoir à passer des dizaines de propriétés aux composants intermédiaires dans la hiérarchie.
- La mise à jour des données dans un mécanisme ultrasimple car totalement déterministe et sans effet de bord (les reducer) permet d'éviter de nombreux problèmes liés à des traitements asynchrones (appels à une API par exemple).
- La logique métier de l'application (les services) est clairement séparée de l'affichage des données (les composants).

Il serait tout à fait possible d'arriver à ces avantages en ayant recours à une autre bibliothèque que Redux, et même sans bibliothèque externe. Dans son article *You might not need Redux* (https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367), Dan Abramov – pourtant co-créateur de Redux – explique pourquoi dans beaucoup d'applications Redux n'est pas nécessaire, et comment s'en passer en gardant toutefois la logique des reducer par exemple.

Donc faut-il utiliser tout le temps Redux et jamais de state local ? Clairement, non. Cela complexifie grandement le code sans nécessité. Malheureusement il n'existe pas de méthode universelle pour savoir si une donnée doit être stockée dans un state global (celui de Redux par exemple) ou un state local. Voici néanmoins quelques conseils :

- Si une donnée doit être accessible depuis plusieurs composants qui ne sont pas directement liés entre eux (relation parent-enfant par exemple), il y a de grandes chances que la mettre dans Redux soit une bonne idée.
- Si au contraire une donnée n'a de sens que pour un composant donné, inutile de la mettre dans Redux. Par exemple, un formulaire n'a pas besoin de stocker la valeur de chaque champ dans Redux. Un state local peut suffire, puis peut-être que la validation du formulaire devra extraire les données nécessaires issues du formulaire dans le state.

Dans le chapitre suivant, nous allons découvrir ensemble une autre bibliothèque qui vient compléter Redux pour gérer les effets de bord comme les traitements asynchrones : Redux-Saga. Elle est certes plus complexe à appréhender que Redux-Thunk, mais permet des patterns beaucoup plus évolués. Et lorsqu'on a pris un peu l'habitude, elle est selon moi plus agréable à utiliser au quotidien.

Chapitre 4
Gérer les effets de bord avec Redux-Saga

1. Introduction

Dans le chapitre précédent, nous avons vu comment gérer l'état et le cycle de vie d'une application à l'aide de Redux. Nous avons également vu comment gérer des actions asynchrones (requêtes HTTP par exemple) à l'aide de Redux-Thunk. Il existe plusieurs moyens de gérer les effets de bord d'une application avec Redux, et Redux-Thunk est probablement le plus simple à mettre en œuvre. Dans ce chapitre, nous allons en voir un autre permettant de mettre en place des patterns plus évolués : Redux-Saga (https://github.com/redux-saga/redux-saga).

La première chose permettant de dire que Redux-Saga diffère de Redux-Thunk est qu'il ne vient pas s'intégrer dans le cycle de vie de base de Redux (tel que décrit dans le chapitre précédent : actions, reducer, etc.). Le principe sera de définir des sagas qui seront déclenchées sur des actions données et qui auront la possibilité d'effectuer des traitements, comme lire le state, appeler des fonctions asynchrones, ou déclencher de nouvelles actions. Tout cela s'intégrera dans notre store Redux à l'aide d'un middleware.

L'une des principales difficultés à l'utilisation de Redux-Saga est qu'il repose sur une fonctionnalité de JavaScript assez peu utilisée au quotidien : les générateurs ou fonctions génératrices. Mais c'est aussi ce qui fait sa force, car les générateurs permettent d'appliquer les principes à la base de Redux-Saga, c'est-à-dire déclencher des effets.

Voyons dans un premier temps ce qui se cache derrière les générateurs, après quoi nous verrons en quoi ils sont utiles à la compréhension et à l'utilisation des sagas.

2. Les générateurs

Dit simplement, un générateur est une fonction renvoyant un itérateur. Commençons donc par présenter ce qu'est un itérateur. Il s'agit d'un objet possédant une interface permettant de :

- connaître sa valeur courante ;
- se déplacer à la valeur suivante ;
- savoir si l'on a atteint la fin de l'itérateur.

Plusieurs implémentations sont possibles ; voici par exemple une fonction permettant, à partir d'un tableau donné, d'obtenir un itérateur pour le parcourir :

```
const getArrayIterator = array => {
 let index = -1
 return {
   next() {
     index++
     return {
       value: array[index],
       done: index === array.length,
     }
   },
 }
}
```

Ici, les trois actions propres à l'itérateur sont faites au sein d'une seule fonction `next`, qui renvoie un objet contenant l'attribut `value` avec la valeur courante et l'attribut `done` qui est à `true` si on est au bout.

La manière la plus naturelle d'utiliser cet itérateur est par une boucle `while` :

```
let res
do {
 res = it.next()
 console.log(res)
} while (!res.done)
```

Notez qu'un itérateur n'est pas nécessairement fini. Il peut par exemple être pratique d'obtenir un itérateur parcourant une suite infinie de nombres, comme la suite de Fibonnacci :

```
const getFibonacciIterator = () => {
 let previous = 0
 let current = 1
 return {
   next() {
     const next = current + previous
     previous = current
     current = next
     return { value: previous, done: false }
   }
 }
}
```

Parcourir cet itérateur avec une boucle `while` causerait bien évidemment quelques problèmes, mais on peut parcourir les valeurs une à une, par exemple dans la console de votre navigateur :

```
> const fibIt = getFibonacciIterator()
{next: ƒ}
> fibIt.next()
{value: 1, done: false}
> fibIt.next()
{value: 1, done: false}
> fibIt.next()
{value: 2, done: false}
> fibIt.next()
{value: 3, done: false}
> fibIt.next()
{value: 5, done: false}
```

Dans ces deux exemples, les fonctions `getArrayIterator` et `getFibonacciIterator` renvoient un nouvel itérateur. JavaScript nous offre donc la possibilité de les définir d'une manière différente, sous forme de générateur, grâce au mot-clé `function*` :

```
function* getArrayIterator(array) {
 let index = 0
 while (index !== array.length) {
   yield array[index]
   index++
 }
}
```

```
function* getFibonacciIterator() {
 let previous = 1
 let current = 1
 while (true) {
   const next = current + previous
   previous = current
   current = next
   yield next
 }
}
```

Ces deux fonctions génératrices ont exactement le même comportement que celles que nous avons définies précédemment. Elles renvoient des itérateurs qui peuvent être parcourus exactement de la même manière. Et pourtant leur syntaxe peut paraître déroutante.

D'abord, le fait d'utiliser une boucle `while` : cela donne l'impression que l'on va parcourir tout le tableau dans le premier cas, ou que l'on déclenche une boucle infinie dans le second. En réalité, il n'en est rien, car l'utilisation du mot-clé `yield` stoppe en quelque sorte l'itération courante, et se met en attente de la suivante.

Pour mieux comprendre les générateurs, on peut donc imaginer qu'une instruction `yield` retourne une valeur, stoppe la fonction, et fait en sorte qu'au prochain appel on reprenne l'exécution à cet endroit exact, là où l'on s'était arrêté. Notez que ce n'est qu'une vision simplifiée des choses, même si elle est très pratique pour appréhender les générateurs au quotidien.

La réalité est que ce n'est qu'une syntaxe pour créer un itérateur comme nous l'avons fait au début de cette section.

Nous avons vu des générateurs permettant de parcourir des tableaux (finis ou « infinis »), mais juste avant de voir comment ils sont utilisés par Redux-Saga, il me semble important de présenter un autre cas d'utilisation plus proche de celui des sagas.

Imaginez que l'on souhaite définir un moteur d'exécution, permettant d'exécuter des petits programmes pour effectuer des opérations, demander des informations à l'utilisateur et lui afficher des résultats.

Par exemple, le programme suivant, en pseudo-code :

```
name = prompt "What's your name?"
greetings = "Hello " + name + "!"
show greetings
```

Nous souhaitons que notre programme puisse être écrit en JavaScript, mais les fonctions du langage de haut niveau `prompt` et `show` ne sont pas connues. Il se peut qu'elles affichent une nouvelle fenêtre à l'utilisateur, ou qu'elles fassent appel à une API, ou encore qu'elles aillent lire des informations dans des fichiers. Elles ne sont peut-être même pas synchrones. En bref, il s'agit d'effets de bord du programme.

Afin de gérer ces effets de bord, commençons par définir deux fonctions utilitaires (en JavaScript cette fois) :

```
const prompt = name => ({ type: 'prompt', name })
const print = message => ({ type: 'print', message })
```

Notez que ces fonctions ne « font » rien, si ce n'est nous aider à définir les effets de bord possibles de notre moteur d'exécution. Mais en utilisant les générateurs de JavaScript, on peut à présent écrire notre programme ainsi :

```
function* myProgram() {
 const name = yield prompt('What's your name?')
 const greetings = `Hello ${name}!`
 yield print(greetings)
}
```

Notre programme est maintenant bien défini, et déclare les effets de bord qu'il déclenche. Bien évidemment, il n'est pas encore possible de l'exécuter à proprement parler puisque nous n'avons pas écrit le moteur d'exécution qui en est responsable. En revanche, on peut simuler son exécution manuellement, en parcourant l'itérateur qu'il renvoie :

```
> it = myProgram()
...
> it.next()
Object { value: {type: "prompt", name: "What's your name?"},
         done: false }
> it.next("John")
Object { value: {type: "print", message: "Hello John!"},
         done: false}
> it.next()
Object {value: undefined, done: true}
```

Ici, c'est nous qui avons simulé le moteur d'exécution du programme, en appelant à plusieurs reprises la méthode `next` de l'itérateur :

- Tout d'abord, l'itérateur nous a renvoyé un effet de type « prompt ». Nous savons donc que l'on doit demander à l'utilisateur une information (avec le message « *What's your name?* »). Pour l'itération suivante, nous envoyons donc l'information demandée : « John ».
- À l'itération suivante, le programme reçoit la valeur demandée, il s'agit du retour de l'instruction `yield`. Il continue donc son exécution, et déclenche l'effet suivant, de type « prompt », avec le message « *Hello John!* ». Nous affichons (virtuellement) ce message à l'utilisateur), et appelons l'itération suivante.
- Le programme est terminé, et met donc fin à l'itérateur.

À présent que nous comprenons le mécanisme d'exécution du programme, nous disposons des éléments nécessaires pour écrire la fonction qui permettra d'exécuter tout programme utilisant les effets `prompt` et `print`.

```
function executeProgram(program) {
 const it = program()
 let res = it.next()
 while (!res.done) {
   const effect = res.value
   switch (effect.type) {
```

```
      case 'prompt':
        const input = window.prompt(effect.desc)
        res = it.next(input)
        break
      case 'print':
        window.alert(effect.message)
        res = it.next()
        break
      default:
        throw new Error(`Invalid effect type: ${effect.type}`)
    }
  }
}
```

Vous reconnaissez le principe de parcours d'un itérateur à l'aide d'une boucle `while`, comme on l'a vu précédemment. Dans le corps de la boucle (c'est-à-dire pour chaque effet déclenché par le programme), un `switch/case` nous permet de décider quelle est l'action à appliquer. Si c'est un effet de type « prompt », on utilise `window.prompt` pour demander une valeur à l'utilisateur. S'il est de type « *print* », on affiche le message grâce à `window.alert`.

À ce niveau, il est possible que vous vous demandiez quels sont les avantages d'une telle approche. En effet, nous aurions probablement pu réaliser cela de manière beaucoup plus simple, sans même aborder le sujet de générateurs. C'est tout à fait vrai ; néanmoins, utiliser des fonctions génératrices et le principe des effets nous apporte ceci :

- Notre programme est défini par une fonction *pure*, c'est-à-dire que pour des paramètres donnés, l'exécution et le résultat retourné seront toujours les mêmes.
- Notre programme est beaucoup plus facilement testable : en environnement réel, il sera interprété par un moteur d'exécution, mais en test il peut l'être par un autre, qui par exemple simulerait les appels à une API si cela est nécessaire.
- La syntaxe du programme est somme toute relativement simple, une fois que l'on appréhende les `function*` et les `yield`.

Le concept d'*effets* que j'ai décrit ici est celui utilisé par Redux-Saga pour gérer les effets de bord, comme nous allons le voir immédiatement. Si vous n'avez pas compris l'ensemble de cette section, ne vous en faites pas.

Il est selon moi intéressant de comprendre ce qui se cache *sous le capot* mais cela n'est pas indispensable pour utiliser les sagas. Ce qu'il faut retenir au final c'est que rien n'est magique. Redux-Saga consiste finalement en un autre moteur d'exécution, plus complexe que celui que j'ai présenté, permettant de déclarer d'autres effets de bord.

3. Les effets de Redux-Saga

Le but principal de Redux-Saga est de proposer une manière de gérer des effets de bord dans une application utilisant Redux. Pour cela, de la même manière que dans notre exemple précédent où nous déclenchions des effets pour demander une saisie de l'utilisateur ou lui afficher une valeur, nous allons déclencher des effets de Redux-Saga, cette fois-ci pour interagir avec Redux ou déclencher d'autres effets de bord.

Les effets qu'il est possible de déclencher dans Redux-Saga sont de plusieurs types. L'effet `call` par exemple permet d'appeler une fonction, éventuellement asynchrone, et d'en récupérer le résultat. Typiquement, il s'agit d'une fonction ayant justement des effets de bord, comme une requête à une API.

Des effets permettent de manipuler le store de Redux :

- `select` lit une valeur dans le store grâce à un sélecteur, de manière semblable au hook `useSelector`.
- `put` dispatche une action.
- `take` attend l'arrivée d'une action d'un type donné.

Enfin, des effets permettent de manipuler l'exécution de la saga elle-même : `fork` pour dupliquer l'exécution de la saga courante, `delay` pour attendre une certaine durée, etc.

Afin de mieux comprendre ces effets et voir comment les utiliser, démarrons un exemple simple. Celui-ci ne fera que créer un store Redux, pas de React pour le moment.

Initialisons l'application :

```
$ yarn init
$ yarn add -D parcel-bundler
$ yarn add redux redux-saga core-js regenerator-runtime
```

Notez que nous installons les dépendances `core-js` et `regenerator-runtime`, ce qui va permettre d'utiliser les fonctions génératrices nécessaires à Redux-Saga (tous les navigateurs ne les supportent pas encore).

Créons le fichier public/index.html avec juste le contenu suivant :

```
<script src="../src/index.js"></script>
```

N'oublions pas d'ajouter le script `start` au package.json :

```
"scripts": {
 "start": "parcel public/index.html"
},
```

Commençons ensuite par créer notre store Redux, dans un fichier store.js. Le principe sera de récupérer depuis une API des informations sur un utilisateur, dont l'ID sera stocké dans le state. Définissons donc deux actions, la première pour demander la récupération de l'utilisateur, la seconde pour mettre le résultat dans le state :

```
// src/store.js

import createSagaMiddleware from 'redux-saga'
import { createStore, applyMiddleware } from 'redux'

// Actions
const getUser = () => ({ type: 'getUser' })
const getUserSuccess = user => ({
 type: 'getUserSuccess',
 payload: user
})
```

Le reducer ne réagira donc qu'à l'action `getUserSuccess` :

```
// Reducer
const initialState = { userId: 2, user: undefined }
const reducer = (state = initialState, action) => {
 switch (action.type) {
   case 'getUserSuccess':
```

```
      return { ...state, user: action.payload }
    default:
      return state
  }
}
```

Nous fournissons une fonction pour récupérer depuis le state l'ID de l'utilisateur dont nous souhaitons obtenir les informations :

```
// Selectors
const selectUserId = state => state.userId
```

Il ne reste qu'à créer le store, en y ajoutant le middleware qui permettra à Redux-Saga de s'exécuter. Pour cela, créons d'abord une saga vide. Il s'agit de la saga racine (*root saga*), qui comme nous le verrons ensuite se divisera en plusieurs sagas.

```
function* rootSaga() {}

const sagaMiddleware = createSagaMiddleware()
const store = createStore(
 reducer,
 applyMiddleware(sagaMiddleware)
)
sagaMiddleware.run(rootSaga)
```

Notez la façon de procéder pour exécuter la saga : on crée d'abord un middleware, puis le store auquel on ajoute ce middleware, et enfin on demande au middleware d'exécuter la saga.

Exportons le store, les actions et les sélecteurs de notre *store.js*, et créons le fichier index.js qui placera notre store dans l'objet `window`, afin que nous puissions jouer avec dans la console du navigateur.

```
// src/store.js
export { store, getUser, getUserSuccess, selectUserId }
```

```
// src/index.js
import 'regenerator-runtime/runtime'
import { store, getUser } from './store'

window.getUser = getUser
window.store = store
```

Vous pouvez à présent lancer l'application (`yarn start`), mais évidemment il ne se passera pas grand-chose lorsque vous dispatcherez l'action `getUser`, puisque nous n'avons rien défini dans ce cas.

```
> store.getState()
{userId: 2, user: undefined}
> store.dispatch(getUser())
> store.getState()
{userId: 2, user: undefined}
```

Il est alors temps d'écrire le contenu de notre saga. Créons un nouveau fichier saga.js, et mettons-y notre saga racine :

```
export function* rootSaga() {}
```

Nous allons devoir surveiller les actions dont le type est « getUser », et lorsqu'une d'entre elles se présente, exécuter un traitement spécifique. Pour cela, nous allons déclencher un effet `takeEvery`, avec en paramètre le type d'action à surveiller, ainsi qu'une saga à exécuter dans ce cas.

```
import { takeEvery } from '@redux-saga/core/effects'

function* getUserSaga(action) {}

export function* rootSaga() {
 yield takeEvery('getUser', getUserSaga)
}
```

Ainsi, à chaque fois qu'une action de type « getUser » passera dans le store, la saga `getUserSaga` sera exécutée, et l'action lui sera passée en paramètre. Notre saga racine se divisera alors en deux sagas : l'une continuant son exécution de `rootSaga`, et l'autre exécutant `getUserSaga` sur chaque action « getUser ».

Il ne nous reste qu'à écrire le contenu de `getUserSaga`. Nous allons vouloir effectuer trois opérations, chacune correspondant à un effet déclenché :

- D'abord nous récupérons l'ID de l'utilisateur depuis le state, au moyen de l'effet `select`.
- Puis on appelle une fonction externe asynchrone se chargeant de l'appel à l'API, grâce à l'effet `call`.
- Enfin on dispatche une action `getUserSuccess` grâce à l'effet `put`.

Cela donne le code complet suivant pour notre fichier saga.js :

```
import { select, call, put, takeEvery }
 from '@redux-saga/core/effects'
import { selectUserId, getUserSuccess } from './store'
import { getUserFromApi } from './api'

function* getUserSaga() {
 const userId = yield select(selectUserId)
 const user = yield call(getUserFromApi, userId)
 console.log('User:', user)
 yield put(getUserSuccess(user))
}

export function* rootSaga() {
 yield takeEvery('getUser', getUserSaga)
}
```

Notez que la fonction `getUserFromApi` est définie dans un autre fichier afin de ne pas surcharger notre saga, mais cela n'a rien d'obligatoire. Il est cependant généralement préférable de sortir des sagas ce qui n'est pas directement lié au cycle de vie de notre store Redux. Voici le code de la fonction `getUserFromApi` :

```
export const getUserFromApi = async userId => {
 const res = await fetch(
   `https://jsonplaceholder.typicode.com/users/${userId}`,
 )
 const user = await res.json()
 return user
}
```

Nous aurions très bien pu intégrer ce traitement dans la saga au moyen de deux effets `call`, mais cela n'a que peu d'intérêt. De plus, définie ainsi en renvoyant une promesse, la fonction est réutilisable dans un autre contexte que Redux-Saga si nécessaire.

Remarque

Pour être extrêmement rigoureux, l'appel à `console.log` devrait lui aussi être encapsulé dans un effet `call` dans la mesure où écrire dans la console représente un effet de bord : `call(console.log, "User:", user)`. Mais étant donné qu'il est présent pour des raisons évidentes de débogage et ne devrait pas rester dans le code final d'une application en production, il est plus pratique de l'utiliser ainsi.

À présent, si vous exécutez les mêmes instructions que précédemment dans votre console, vous devriez récupérer les informations de l'utilisateur depuis l'API de JSONPlaceholder :

```
> store.getState()
{userId: 2, user: undefined}
> store.dispatch(getUser())
{type: "getUser"}
User: {id: 2, name: "Ervin Howell", ...}
> store.getState()
{userId: 2, user: {id: 2, name: "Ervin Howell", ...}}
```

Cela conclut notre premier exemple d'utilisation de Redux-Saga. Dans la section suivante, nous allons mettre en place pas à pas une petite application complète, ce qui nous donnera l'occasion d'explorer des patterns communément utilisés avec les sagas.

4. Un exemple plus complet

Nous allons réaliser une petite application permettant d'interroger une API de recherche d'adresse proposée par l'administration française : adresse.data.gouv.fr. Celle-ci permet à partir d'une requête (un texte libre) de récupérer une liste de résultats correspondant à des adresses réelles, avec des informations comme leur géolocalisation exacte.

Les cas d'utilisations possibles sont nombreux : autocomplétion sur un champ de saisie d'adresse, recherche d'adresse sur une carte, etc. L'avantage pour notre exemple est qu'à ce jour aucune clé n'est nécessaire pour l'utiliser l'API, elle est donc très simple à appeler (pas d'inscription nécessaire notamment).

Notre application présentera un champ de saisie pour la requête, un bouton permettant de lancer la recherche, et une liste de résultats. Elle devra gérer un affichage spécifique pour les cas suivants :

- la recherche n'a pas encore été lancée
- la recherche est en cours
- une erreur s'est produite
- aucun résultat n'est renvoyé
- au moins un résultat est renvoyé

Notre recherche d'adresse en action

L'intégralité de l'état de notre application sera stockée dans le state de Redux (et notamment le contenu du champ de saisie), et cela va de soi, nous utiliserons Redux-Saga pour exécuter la recherche.

Pour initialiser l'application, je vous suggère de dupliquer l'exemple précédent, et d'y ajouter les dépendances `react`, `react-dom` et `react-redux`.

Commençons par écrire la partie Redux de notre application. Comme dans le chapitre précédent, organisons cela en *services* pour plus de clarté, même si pour notre exemple nous n'en aurons qu'un seul : le *search service*.

Notre service proposera une action pour mettre à jour la requête de recherche (ce qui sera affiché dans le champ de saisie), et des actions pour mettre à jour l'état de la recherche. Jusque-là, cela est très similaire à ce que nous faisions avec Redux-Thunk dans le chapitre précédent.

```
// src/services/search/actions.js
export const updateQuery = query =>
 ({ type: 'updateQuery', payload: query })
export const search = () => ({ type: 'search' })
export const searchSuccess = results => ({
 type: 'searchSuccess',
 payload: results,
})
export const searchFailure = error => ({
 type: 'searchFailure',
 payload: error,
})
```

Pour ce qui est du state, nous aurons naturellement un attribut pour la requête, ainsi qu'un autre pour les résultats de recherche. De plus, nous aurons un flag indiquant si la requête est en cours, et un autre indiquant si une erreur s'est produite. Notez que l'attribut `results` est initialisé à `undefined` plutôt qu'à un tableau vide. Cela nous permettra d'afficher un message spécifique si la recherche n'a pas encore été effectuée.

```
// src/services/search/reducer.js
const initialState = {
 query: '',
 isPending: false,
 hasError: false,
 results: undefined,
}

export const reducer = (state = initialState, action) => {
 switch (action.type) {
   case 'updateQuery':
     return { ...state, query: action.payload }
   case 'search':
     return { ...state, isPending: true, hasError: false }
```

```
    case 'searchSuccess':
      return { ...state, isPending: false,
               results: action.payload }
    case 'searchFailure':
      return { ...state, isPending: false, hasError: true }
    default:
      return state
  }
}
```

Afin de faciliter la lecture depuis le state dans les composants et dans la saga, nous fournissons également des *sélecteurs*. Il s'agit de fonctions prenant en paramètre le state, et renvoyant la valeur demandée. L'avantage de telles fonctions est qu'il n'est pas nécessaire de connaître l'organisation interne des données dans le state pour les utiliser. Si l'implémentation du service change, il ne sera nécessaire de modifier que les sélecteurs, et non chaque endroit où ils sont utilisés.

De plus, comme notre reducer sera utilisé dans un `combineReducer`, les données de notre service ne seront pas à la racine du state, c'est pourquoi nous exportons une constante `namespace` depuis l'*index.js* du service, que nous utilisons dans les sélecteurs, et que nous utiliserons également au moment de construire le reducer global de l'application.

```
// src/services/search/index.js
export const namespace = 'search'
```

```
// src/services/search/selectors.js
import { namespace } from '.'

export const selectQuery = state =>
 state[namespace].query
export const selectSearchIsPending = state =>
 state[namespace].isPending
export const selectSearchHasError = state =>
 state[namespace].hasError
export const selectSearchResults = state =>
 state[namespace].results
```

Passons à la saga. Nous souhaitons réagir aux actions de type « search », déclenchées lorsque l'utilisateur clique sur le bouton « Go ». À ce moment, nous devons récupérer la requête dans le state (en utilisant le sélecteur `selectQuery`), puis appeler la fonction de recherche `searchAddresses` avec la requête en paramètre, et enfin dispatcher une action `searchSuccess` avec les résultats. Dans le cas où une erreur s'est produite (par exemple, pas de connexion au réseau), on dispatchera une action `searchFailure`.

```
// src/services/search/saga.js
import { takeEvery, select, call, put }
 from '@redux-saga/core/effects'
import { searchAddresses } from './api'
import { selectQuery } from './selectors'
import { searchSuccess, searchFailure } from './actions'

function* searchSaga() {
 const query = yield select(selectQuery)
 try {
   const results = yield call(searchAddresses, query)
   yield put(searchSuccess(results))
 } catch (err) {
   yield put(searchFailure(err))
 }
}

export function* rootSaga() {
 yield takeEvery('search', searchSaga)
}
```

La fonction `searchAddresses` est très similaire à la fonction `getUserFromApi` de l'exemple précédent :

```
// src/services/search/api.js
export const searchAddresses = async query => {
 const res = await fetch(
   'https://api-adresse.data.gouv.fr/search/?q='
     + encodeURIComponent(query),
 )
 const { features } = await res.json()
 return features
}
```

Notre service est complet, voyons maintenant comment créer le store de l'application en incluant ce service. Dans le chapitre précédent, inclure un service dans le store consistait à inclure son reducer dans un `combineReducer`. À présent, il est nécessaire de faire quelque chose de similaire pour inclure la saga du service.

Pour cela, la saga racine de l'application va se diviser en autant de sagas que nécessaire pour exécuter les sagas de chaque service inclus :

```
function* rootSaga() {
 yield fork(firstServiceSaga)
 yield fork(secondServiceSaga)
 // ...
}
```

Voici le code complet du fichier construisant le store de l'application :

```
// src/store.js
import createSagaMiddleware from '@redux-saga/core'
import { fork } from '@redux-saga/core/effects'
import { createStore, applyMiddleware, combineReducers }
 from 'redux'

// Search service
import { namespace as searchNamespace } from './services/search'
import { rootSaga as searchSaga } from './services/search/saga'
import { reducer as searchReducer }
 from './services/search/reducer'

const rootReducer = combineReducers({
 [searchNamespace]: searchReducer,
})

function* rootSaga() {
 yield fork(searchSaga)
}

const sagaMiddleware = createSagaMiddleware()
export const store = createStore(
 rootReducer,
 applyMiddleware(sagaMiddleware)
)
sagaMiddleware.run(rootSaga)
```

Il ne reste plus qu'à réaliser la partie présentation de notre application. Pour cela, créons un unique composant App qui contient le champ de saisie, le bouton, et les résultats :

```
// src/components/App.js
import React from 'react'
import './App.css'
// ...

const App = ({
 query, isPending, hasError,
 results, updateQuery, search
}) => {
 const handleFormSubmit = event => {
   event.preventDefault()
   search()
 }
 const handleQueryChange = event =>
   updateQuery(event.target.value)

 return (
   <div className="app">
     <form onSubmit={handleFormSubmit}>
       <input
         type="text"
         value={query}
         onChange={handleQueryChange}
         placeholder="Enter an address…"
         required
       />
       <button type="submit">Go!</button>
     </form>
     {results !== undefined && (
       <>
         <h2>Results</h2>
         {isPending && <p className="info">Loading…</p>}
         {hasError &&
           <p className="info error">An error occurred.</p>}
         {results.length > 0 ? (
           <ul>
             {results.map(result => (
               <li key={result.properties.id}>
                 {result.properties.label}
               </li>
```

```
            ))}
          </ul>
        ) : (
          <p className="info">Search returned no result.</p>
        )}
      </>
    )}
  </div>
)
}
```

Connectons ce composant à Redux comme nous savons si bien le faire à l'aide de `connect` :

```
// ...
import { connect } from 'react-redux'
import {
 selectQuery,
 selectSearchIsPending,
 selectSearchHasError,
 selectSearchResults,
} from '../services/search/selectors'
import { updateQuery, search } from '../services/search/actions'

// ...

const mapStateToProps = state => ({
 query: selectQuery(state),
 isPending: selectSearchIsPending(state),
 hasError: selectSearchHasError(state),
 results: selectSearchResults(state),
})

const mapDispatchToProps = {
 updateQuery,
 search,
}

export default connect(
 mapStateToProps,
 mapDispatchToProps,
)(App)
```

Notre application est quasiment complète, tout ce qu'il reste à faire est d'afficher le composant App dans l'application. Pour cela vous pouvez vous baser sur les exemples précédents, ou vous référer aux exemples téléchargeables accompagnant le livre. Si vous exécutez l'application, vous pourrez effectuer des recherches d'adresses, vérifier le bon affichage des messages en cas d'erreur ou si aucun résultat n'est renvoyé.

Avant de conclure cet exemple et ce chapitre, je vous propose une petite amélioration sur notre application. Et si nous faisions en sorte que la recherche soit lancée non pas lorsque l'utilisateur clique sur le bouton, mais plutôt chaque fois qu'il tape un caractère dans le champ ?

Plus précisément, à la manière des autocomplétion que l'on trouve sur le Web, nous souhaitons effectuer la recherche chaque fois qu'un caractère est tapé, mais en ajoutant un délai de sorte qu'il n'y ait pas trop de requêtes inutiles à l'API. Pour cela, un effet de Redux-Saga va nous aider : `throttle`.

Il fonctionne exactement comme `takeEvery`, mais en ajoutant ce délai avant de déclencher la saga. Les sagas de notre service peuvent désormais ressembler à cela :

```
function* searchSaga() {
 // ...
}

function* updateQuerySaga() {
 const query = yield select(selectQuery)
 if (query.length > 3) {
   yield put(search())
 }
}

export function* rootSaga() {
 yield takeEvery('search', searchSaga)
 yield throttle(1000, 'updateQuery', updateQuerySaga)
}
```

Ainsi, il est même possible désormais de supprimer le bouton du composant App, puisque la recherche est lancée automatiquement. Pratique, non ?

5. Conclusion

Dans le chapitre précédent, nous avions vu comment architecturer une application React autour de Redux, et gérer les effets de bord avec Redux-Thunk. Dans ce chapitre nous avons découvert une méthode plus puissante bien que plus complexe pour gérer ces effets : Redux-Saga. Ce dernier permet des patterns beaucoup plus avancés, et nous n'en avons vu qu'une petite partie ici. Il est possible de faire des « courses » (*race*) entre plusieurs sagas, d'en annuler certaines, ou encore de surveiller autre chose que le store de Redux.

N'ayez crainte, ce que nous avons vu ensemble devrait déjà correspondre à la plupart des besoins que vous rencontrerez dans votre utilisation de Redux. Et déjà avec ces notions vous êtes capables de mettre en place des traitements évolués.

Si vous vous retrouvez à utiliser Redux-Saga dans le cadre d'une application professionnelle qu'il est nécessaire de tester unitairement, vous pourrez découvrir dans le chapitre Tester une application React de ce livre la bibliothèque Redux Saga Test Plan (http://redux-saga-test-plan.jeremyfairbank.com/), permettant d'écrire des tests très lisibles et pertinents sous forme de « plan de test ». Cela découle directement du modèle adopté par Redux-Saga et de ses effets.

Voilà maintenant venu le moment de détourner notre attention du Web pour faire un tour du côté du mobile. Pas de panique si le développement mobile ne vous a jamais intéressé, ou si vous avez été effrayé par le développement iOS ou Android avec Objective C, Swift, Java ou Kotlin. Grâce à React, nous allons voir que développer des applications mobiles natives n'est pas plus compliqué que développer des applications web. En route !

Chapitre 5
Développer pour le mobile avec React Native

1. Présentation de React Native

À présent que nous avons vu les bases de React et comment concevoir une application grâce à Redux, vous disposez des connaissances nécessaires pour écrire des applications relativement complexes. Mais jusque-là, nous sommes restés dans le monde du Web, c'est-à-dire que les applications que nous avons écrites étaient exécutées dans un navigateur. Dans ce chapitre nous allons voir comment aller un peu plus loin en appliquant ce que nous avons vu pour créer des applications mobiles.

1.1 Un peu d'historique

Jusqu'à récemment, on pouvait dire que le développement pouvait être compartimenté en plusieurs types :

- le développement d'applications « bureau » (desktop) ;
- le développement d'applications web ;
- le développement d'applications mobiles.

En tant que développeur, au moment de concevoir une application, la première question à se poser était de savoir quelle était la plateforme cible. Est-ce que je souhaite créer une application web accessible de n'importe quel navigateur, en faisant un compromis sur la réactivité de l'interface, notamment sur mobile ? Ou bien une application mobile, même si cela implique de développer une application par OS mobile : iOS, Android, Windows... ?

En supposant que l'on souhaitait développer pour le mobile, cette question était d'autant plus importante étant donné l'essor incontestable des smartphones. En 2009, un outil a été créé promettant de faire disparaître ce problème. Phonegap, devenu depuis Apache Cordova (https://cordova.apache.org/), permettait de créer des applications mobiles universelles à l'aide de technologies web. Pour faire simple, l'idée était de créer une application web, puis l'outil intégrait cette application dans un mini-navigateur au sein d'une application native. En 2013, le framework Ionic (https://ionicframework.com/) a ajouté une surcouche à Cordova permettant de créer facilement des applications mobiles à l'aide du framework Angular.

Si Phonegap/Cordova a eu beaucoup de succès et est toujours très utilisé, force est de constater que certaines applications mobiles nécessitaient malgré tout une réactivité au niveau de l'interface qu'une application web « cachée » dans une application native n'était pas en mesure d'offrir.

En 2015, un nouvel acteur entre en scène. Dans la période de succès de React, Facebook annonce React Native, une surcouche à React permettant de développer des applications mobiles natives. Par native, j'entends que les éléments rendus par l'application (textes, boutons, etc.) sont bien des éléments graphiques spécifiques à la plateforme mobile, et non pas des éléments web dans un navigateur intégré.

Le succès a une fois de plus été au rendez-vous. Il n'était plus nécessaire de choisir entre du code JavaScript universel et une application mobile native à l'interface fluide. Il était désormais possible de réutiliser la même technologie pour développer sur le Web et sur mobile !

Notez que depuis son annonce, React Native n'a jamais eu pour ambition de permettre la création d'applications universelles pouvant être exécutées sur navigateur et sur mobile. Le but était plutôt d'unifier les technologies utilisées.

Autrement dit, plutôt que « une seule application pour toutes les plateformes », React Native a préféré se positionner sur la philosophie « une seule technologie pour toutes les plateformes ».

Terminons notre rapide historique par l'entrée en scène d'un outil formidable que nous allons utiliser dans ce chapitre : Expo (https://expo.io/). Sorti en 2015 sous le nom d'Exponent, Expo facilite grandement le développement d'applications avec React Native car il gère tout le processus de génération de l'application mobile (en développement comme pour la production), pour iOS et Android. Depuis l'été 2019, il est même possible de générer une application web, grâce au projet React Native Web (https://github.com/necolas/react-native-web).

Nous ne nous occupons que du code JavaScript, et Expo nous permet de lancer l'application sur mobile ou dans le simulateur de notre choix. Il génère même les binaires de l'application qu'il n'y a plus qu'à soumettre à l'AppStore et Google Play. Autant dire que nous aurions tort de ne pas profiter de tous ces avantages pour notre apprentissage de React Native !

1.2 Outils utilisés

Pour résumer, voici les outils que nous utiliserons dans ce chapitre :

- React, bien entendu !
- React Native, qui est au mobile ce que React DOM (la bibliothèque que nous avions en dépendance dans nos applications jusqu'à maintenant) est au Web.
- Expo, pour rendre le développement plus facile en nous permettant de nous concentrer sur l'application elle-même et non sur les problématiques natives inhérentes à chaque plateforme (génération des binaires dans XCode ou Android Studio...).

Remarque

L'utilisation de React Native et Expo ne nécessite pas de compte développeur Apple ou Google pour développer et tester l'application. En revanche pour ce qui est de la distribution, les règles sont les mêmes que pour les applications mobiles traditionnelles. Pour diffuser une application iOS, vous aurez besoin d'un compte développeur Apple (100 $US par an), et il en sera de même pour une application Android si vous souhaitez voir votre application sur Google Play (25 $US).

Notez aussi que pour utiliser le simulateur iOS vous devrez travailler sur Mac, mais il vous sera tout de même possible de tester votre application à l'aide d'un vrai iPhone ou iPad, sans que celui-ci ait besoin d'être enregistré sur un compte développeur Apple.

La bonne nouvelle étant que vous pouvez tout à fait apprendre à développer avec React Native sans compte développeur, et ainsi créer vos premières applications. Si l'une de vos créations vous rend suffisamment fier au point de vouloir la diffuser au monde, peut-être qu'investir dans un compte développeur semblera plus facile.

C'est parti, attaquons sans plus tarder la mise en place de notre première application mobile.

2. Une première application

2.1 Génération de l'application et premier lancement

Pour initialiser une application, le seul élément à installer est `expo-cli`, paquet NPM à installer de manière globale et contenant les outils en ligne de commande d'Expo :

```
$ npm i -g expo-cli
```

Une fois le module installé, rendez-vous dans le répertoire où vous souhaitez créer votre application afin d'initialiser le projet :

```
$ expo init ma-premiere-app-mobile
```

Au jour où ces lignes sont écrites (cela change fréquemment), il est demandé si vous souhaitez utiliser le template « Managed » ou « Bare », si vous souhaitez utiliser TypeScript ou non, et si l'installation doit utiliser Yarn ou NPM. Dans tous les cas, prenez le choix par défaut.

Remarque

Depuis la sortie d'Expo 33 au printemps 2019, il est possible d'utiliser les fonctions d'Expo (notamment pour accéder à certaines fonctions natives de l'OS) même dans une application React Native ne reposant pas sur Expo. Expo distingue donc depuis les applications « managed », reposant sur Expo, des applications « bare » qui incluent Expo en tant que dépendance. Pour notre cas nous utiliserons le mode « managed ».

La commande crée le répertoire ma-premiere-app-mobile et l'initialise avec les fichiers nécessaires pour lancer une application minimaliste avec Expo. Les fichiers principaux sont les suivants :

- package.json : il est initialisé avec les dépendances nécessaires et les commandes qui nous faciliteront le développement.
- app.json : contient des métadonnées de l'application pour Expo, notamment la version d'Expo à utiliser. À terme, il contiendra aussi des données nécessaires pour générer le binaire de l'application.
- App.js : le fichier principal de l'application.

Allons jeter un œil au fichier App.js afin de voir de quoi il retourne. Notez que depuis que j'ai écrit ce chapitre il se peut qu'Expo ait été mis à jour au point de changer légèrement le contenu des fichiers générés. Je n'ai cependant aucun mal à croire que les principes décrits ici resteront les mêmes.

Tout d'abord sont importées les bibliothèques React, et sans surprise quelques composants de React Native :

```
import React from 'react'
import { StyleSheet, Text, View } from 'react-native'
```

Rien de vraiment nouveau ici. Le composant principal de l'application, `App` se présente comme une classe héritant de `Component` comme nous en avons déjà vues.

Cette classe ne contient qu'une méthode `render`, renvoyant trois éléments (composant `Text`), intégrés dans une `View` :

```
export default class App extends React.Component {
 render() {
   return (
     <View style={styles.container}>
       <Text>Open up App.js to start working on your app!</Text>
       <Text>Changes you make will automatically reload.</Text>
       <Text>Shake your phone to open the developer menu.</Text>
     </View>
   )
 }
}
```

Cette `View` définit un attribut `style`, qui fait référence à la déclaration de feuille de style déclarée en dessous à l'aide de `StyleSheet.create` :

```
const styles = StyleSheet.create({
 container: {
   flex: 1,
   backgroundColor: '#fff',
   alignItems: 'center',
   justifyContent: 'center'
 }
})
```

Nous verrons un peu plus loin ce que représentent les composants `View` et `Text` ainsi que les feuilles de style. Pour le moment, contentons-nous de lancer notre application. Mais d'abord, vous aurez besoin d'un simulateur iOS ou Android en cours d'exécution :

- Pour iOS vous devrez (sur Mac) d'abord vous assurer que Xcode est installé (gratuit sur l'AppStore), ainsi que les outils en ligne de commande associés (il suffit normalement de lancer Xcode pour ce soit suggéré et installé). Le simulateur sera ensuite disponible par le menu **Xcode - Open Developer Tool - Simulator**.
- Pour Android, il faudra installer Android Studio puis créer un projet (le projet de base proposé convient très bien), et enfin créer un simulateur grâce au menu **Tools - Android - AVD Manager** (il se peut que ce menu soit grisé pendant que le projet est totalement initialisé, soit quelques minutes).

Vous pourrez ensuite créer un simulateur avec la version d'Android de votre choix, je vous suggère de sélectionner la plus récente.

Une fois que le simulateur de votre choix tourne, lancez-y l'application. Pour cela rien de plus simple, il suffit de lancer la commande `yarn start` (ou `npm start`). Un serveur de développement va être lancé, nous laissant ensuite la main pour décider de ce que l'on veut faire ensuite. Pour exécuter l'application sur le simulateur iOS, appuyez sur **i**, et sur **a** pour Android. Vous pouvez aussi si vous le souhaitez lancer sur un vrai téléphone ; dans ce cas, je vous encourage à suivre les instructions proposées par Expo dans la mesure où la marche à suivre change fréquemment. Ici nous supposerons que vous utilisez le simulateur.

L'application dans les simulateurs iOS et Android

À présent que l'application est lancée dans un simulateur, vous devriez voir s'afficher les trois lignes de texte que nous avons vues dans le fichier App.js. Félicitations, vous venez de créer votre première application mobile !

Voyons d'un peu plus près comment fonctionnent les composants sur mobile, en créant une petite application simple.

2.2 Premiers composants

Dans cette section, nous allons modifier l'application que nous venons de lancer. Le but sera de créer une application de gestion de tâches, qui devrait vous rappeler celle créée dans le second chapitre de ce livre. Cela nous donnera l'occasion de découvrir quelques fonctionnalités de React Native.

Je vous propose tout d'abord de supprimer le contenu du fichier App.js pour le remplacer par le contenu suivant :

```
// App.js
import App from './src/components/App'
export default App
```

En effet, Expo impose que le point d'entrée de l'application soit ce fichier App.js, mais je trouve plus propre d'avoir un répertoire contenant les sources du projet plutôt que de les avoir à la racine.

Le fichier App.js de notre application (celui dans src/components) va définir un composant, utilisant un state local contenant les tâches dans l'attribut `todos`. Pour le moment, il se contentera d'afficher un composant `TodoList` que nous créerons dans quelques instants.

```
// src/components/App.js
import React, { Component } from 'react'
import { View, Text, StyleSheet } from 'react-native'
import TodoList from './TodoList'

class App extends Component {
 state = {
   todos: [
     { id: 1, label: 'Buy some milk' },
     { id: 2, label: 'Learn some React' }
   ]
 }
 render() {
   const { todos } = this.state
   return (
     <View style={styles.container}>
       <TodoList todos={todos} />
     </View>
   )
 }
```

```
}
// ...
```

Comme dans la version générée initialement, nous utilisons le composant `View` de React Native. Il s'agit du composant de base pour les applications mobiles, considérez-le simplement comme un conteneur pour d'autres composants, similaire à l'élément HTML `div`.

En utilisant la propriété `style` de ce composant, nous pouvons lui associer des règles de style afin d'en modifier la présentation. Notez cependant que malgré la similitude (et bien que ce soit techniquement possible), nous ne définissons pas de style en ligne directement dans l'attribut ; en React Native il est préférable que les styles soient définis à l'aide de `Stylesheet.create`. Ajoutons donc la déclaration de style à la fin du fichier :

```
// ...

const styles = StyleSheet.create({
 container: {
   flex: 1,
   marginTop: 25
 }
})

export default App
```

En réalité `Stylesheet.create` enregistre les styles de manière globale pour l'application, et ne fait que renvoyer des ID pour les « classes » créées. Par exemple, pour cette déclaration il se pourrait très bien que l'objet `styles` contienne comme valeur { `container: 1` }. En passant `styles.container` à l'attribut `style` de notre composant, nous indiquons à React Native d'utiliser le style déclaré ici.

Concernant le style proprement dit, il s'agit des attributs CSS dont vous avez l'habitude (en notation *camelCase*, comme c'est le cas dans l'attribut `style` des éléments du DOM dans le navigateur). Attention cependant : tous les éléments sont affichés en mode `flex` et il n'est pas possible de faire autrement ; pas de `block` ou `inline` notamment. Cela pourra vous demander de travailler vos connaissances sur FlexBox si vous n'en avez pas l'habitude, mais rassurez-vous c'est finalement relativement intuitif et extrêmement agréable à utiliser.

Je vous suggère l'article *A Complete Guide to Flexbox* (https://css-tricks.com/snippets/css/a-guide-to-flexbox/) qui fait très bien office d'aide-mémoire à ce sujet.

Le composant `TodoList` n'apporte pas vraiment de nouveauté. Nous affichons un composant `Todo` par tâche qui nous est passée en paramètre (sans oublier l'attribut `key`), et ici également nous définissons le style de notre composant, en l'occurrence pour nous assurer que les tâches prendront toute la largeur du composant (`alignItems: 'stretch'`).

```
// src/components/TodoList.js
import React from 'react'
import { View, StyleSheet } from 'react-native'
import Todo from './Todo'

const TodoList = ({ todos }) => (
 <View style={styles.todoList}>
   {todos.map(todo => (
     <Todo todo={todo} key={todo.id} />
   ))}
 </View>
)

const styles = StyleSheet.create({
 todoList: {
   flex: 1,
   alignItems: 'stretch'
 }
})

export default TodoList
```

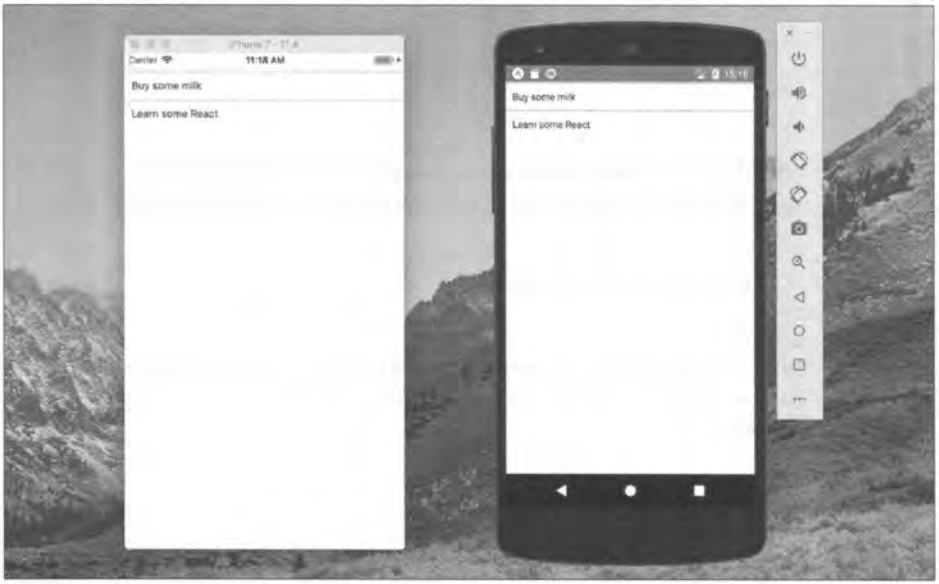
Enfin, le composant `Todo` génère l'affichage pour une tâche donnée :

```
// src/components/Todo.js
import React from 'react'
import { View, Text, StyleSheet } from 'react-native'

const Todo = ({ todo }) => (
 <View style={styles.todo}>
   <Text>{todo.label}</Text>
 </View>
)
```

```
const styles = StyleSheet.create({
 todo: {
   padding: 10,
   borderTopWidth: 1,
   borderStyle: 'solid',
   borderColor: 'lightgray'
 }
})

export default Todo
```

La seule nouveauté ici est l'utilisation du composant `Text`, qui permet, comme son nom vous l'aura fait deviner, d'afficher du texte. Il s'agit en quelque sorte de l'équivalent de la balise `span` du HTML, mais en React Native il est nécessaire de l'utiliser pour afficher du texte. Il n'est pas possible par exemple de mettre une chaîne de caractères dans un composant View : `<View>Hello</View>` génèrera une erreur.

Une fois ces trois composants écrits, vous devriez être en mesure de lancer l'application. Vous verrez ainsi s'afficher nos deux tâches de test.



Affichage des tâches dans iOS et Android

2.3 Gérer des entrées de l'utilisateur

Comme prochaine étape, ajoutons la possibilité de modifier une tâche existante. Cela nous montrera comment permettre à l'utilisateur de saisir du texte. Nous allons pour cela modifier notre composant `Todo`. Il disposera maintenant d'un state local, qui contiendra notamment un attribut booléen `editMode`, qui lorsqu'il sera à `true` indiquera que l'on souhaite afficher la tâche en mode modification.

```
// src/components/Todo.js
// ...
class Todo extends Component {
 state = {
   editMode: false
   // …
 }
 render() {
   const { editMode } = this.state
   return (
     <View style={styles.todo}>
       {editMode ? this.renderEditMode() : this.renderViewMode()}
     </View>
   )
 }
}
```

En mode visualisation, nous allons comme précédemment afficher le libellé de la tâche, mais également un bouton permettant de passer en mode édition :

```
renderViewMode = () => {
 const { todo } = this.props
 return (
   <Fragment>
     <Text style={styles.todoLabel}>{todo.label}</Text>
     <Button title="Edit" onPress={this.onEditPress} />
   </Fragment>
 )
}
```

Notez l'utilisation de `Fragment`, nouveauté de React 16 permettant de renvoyer plusieurs éléments sans les intégrer dans un élément englobant. Nous aurions pu utiliser une `View`, mais cela aurait eu quelques conséquences sur la mise en forme. Ici `Fragment` nous simplifie la vie.

Pour afficher un bouton, nous utilisons le composant `Button` de React Native. Son attribut `title` permet de définir le texte à afficher, tandis que l'attribut `onPress` permet de définir l'action à effectuer lorsque le bouton est pressé. Relativement intuitif non ? Il est intéressant de noter que par défaut, le visuel du bouton est adapté à la plateforme (iOS ou Android) et aux règles communes d'ergonomie sur chacune d'elles.

Sans grande surprise, lorsque le bouton est pressé, nous ne faisons que passer en mode édition :

```
onEditPress = () => {
 this.setState({
   editMode: true
 })
}
```

Les choses deviennent intéressantes lorsque l'on est en mode édition. Pour ce qui est du rendu, nous affichons cette fois un champ texte pour modifier le libellé de la tâche, ainsi que deux boutons : le premier pour enregistrer le nouveau libellé, et le second pour annuler les modifications. Nous souhaitons que les deux boutons fassent également repasser en mode visualisation.

```
state = {
 editMode: false,
 label: this.props.todo.label
}
renderEditMode = () => {
 const { label } = this.state
 return (
   <Fragment>
     <TextInput
       style={[styles.editInput, styles.todoLabel]}
       value={label}
       onChangeText={this.onChange}
       autoFocus
     />
     <Button title="Save" onPress={this.onSavePress} />
```

```
        <Button title="Cancel" onPress={this.onCancelPress} />
      </Fragment>
    )
  }
```

Pour le champ texte, le composant à utiliser est `TextInput`. Sans surprise, son attribut `value` définit la valeur du champ, tandis que l'attribut `onChangeText` définit la fonction à appeler lorsque la valeur est modifiée. Assez pratique, l'attribut `autoFocus` permet de donner le focus automatiquement au champ dès qu'il est affiché, en l'occurrence pour notre exemple dès que l'on passe en mode édition.

À quelques noms d'attributs près, peu de différence avec ce que l'on a fait au premier chapitre côté web. Petite différence tout de même : lorsque la fonction passée à `onChangeText` est appelée, c'est directement la nouvelle valeur qui est passée en paramètre, et non un objet `event` :

```
onChange = label => {
 this.setState({ label })
}
```

Les méthodes `onSavePress` et `onCancelPress`, appelées respectivement lorsque les boutons **Save** et **Cancel** sont pressés, appellent toutes deux `setState` afin de quitter le mode édition. Mais en plus, `onCancelPress` redéfinit l'attribut `label` du state afin qu'à la prochaine édition on ait bien le libellé original. Quant à `onSavePress`, elle appelle la fonction `updateTodoLabel` passée en propriété au composant. Nous continuons ainsi d'appliquer la bonne pratique consistant à ne pas mettre à jour l'état de notre application n'importe où. Ici, mieux vaut laisser cette responsabilité au composant stockant cet état, à savoir `App`.

```
onSavePress = () => {
 const { updateTodoLabel } = this.props
 const { label } = this.state
 updateTodoLabel(label)
 this.setState({ editMode: false })
}
onCancelPress = () => {
 this.setState({
   editMode: false,
   label: this.props.todo.label
 })
```

```
}
```

N'oublions pas également de déclarer le style `todoLabel` que nous utilisons dans cette nouvelle version du composant `Todo`:

```
const styles = StyleSheet.create({
 // ...
 todoLabel: {
   fontSize: 18,
   padding: 10,
   flex: 1
 }
})
```

Il ne reste qu'à mettre à jour les composants `TodoList` et `App` pour passer la bonne propriété `updateTodoLabel`. Pour ce qui est de la mise à jour proprement dite (dans `App`), j'ai réutilisé l'algorithme vu dans l'exemple du premier chapitre, afin de mettre à jour la tâche dont l'ID est donné en paramètre, mais bien en passant par `setState` et non en modifiant directement l'objet.

```
// src/components/TodoList.js
// ...
<Todo
 todo={todo}
 updateTodoLabel={label => updateTodoLabel(todo.id, label)}
 key={todo.id}
/>
// ...
```

```
// src/components/App.js
updateTodoLabel = (todoId, label) => {
 const { todos } = this.state
 const todoIndex = todos.findIndex(t => t.id === todoId)
 const todosBefore = todos.slice(0, todoIndex)
 const todosAfter = todos.slice(todoIndex + 1)
 const newtodo = { ...todos[todoIndex], label }
 this.setState({
   todos: [...todosBefore, newtodo, ...todosAfter]
 })
}
render() {
```

```
    // ...
    <TodoList todos={todos} updateTodoLabel={this.updateTodoLabel}
/>
    // ...
}
```

Pour terminer cette section, vous trouverez ci-dessous deux petits exercices afin que notre application soit complète. Ils ne présentent pas de nouveauté majeure par rapport à ce que nous avons vu, mais vous permettront de vérifier que vous avez bien compris les notions présentées.

L'intégralité du code de l'application, incluant ces deux nouvelles fonctionnalités, est bien entendu disponible dans les exemples téléchargeables accompagnant le livre.

Exercice 1 : permettre d'ajouter une tâche

Le but est d'ajouter sur l'application un bouton permettant l'ajout d'une nouvelle tâche. Vous pouvez vous référer à l'exemple du premier chapitre afin de voir comment gérer cette création pour ce qui est de l'ID à donner à la nouvelle tâche.

Exercice 2 : marquer une tâche comme effectuée

Lorsque l'utilisateur clique sur le libellé d'une tâche (s'il n'est pas en mode édition), la tâche doit être marquée comme effectuée (attribut `isDone`) et ainsi afficher une icône (une emoji fera l'affaire). Encore une fois, n'hésitez pas à retourner au second chapitre pour voir l'aspect algorithmique et vous concentrer ici sur la partie interface.

Afin d'intercepter le clic sur un élément de l'interface qui n'est pas un bouton, vous aurez besoin d'utiliser un composant que nous n'avons pas encore vu : `TouchableOpacity`. Il permet d'une part de signaler au système que le composant est cliquable (pour des raisons d'accessibilité), mais d'autre part d'afficher un retour visuel lors du clic. Il s'utilise ainsi :

```
<TouchableOpacity onPress={...}>
  <Text>Cliquez ici!</Text>
</TouchableOpacity>
```

Voici à quoi peut ressembler l'application une fois ces deux fonctionnalités ajoutées

Nous en avons terminé avec cette première application React Native. Voyons à présent deux fonctionnalités très utilisées sur mobile en général et donc en React Native : la navigation dans une application et l'affichage de listes.

2.3.1 Gestion de la navigation

Nous avons vu dans la première section de ce chapitre comment créer une application grâce à React Native et Expo. Nous avons créé des composants, avons modifié leur rendu grâce à ce qui ressemble à du CSS, et avons permis à l'utilisateur d'interagir avec eux à l'aide de boutons et autres champs de saisie.

Dans les deux prochaines sections, nous allons explorer deux patterns utilisés sur une très grande majorité des applications mobiles, que ce soit sur iOS ou Android, et cela bien avant l'apparition de React Native. La première est la navigation, c'est-à-dire le fait d'avoir plusieurs écrans et de permettre à l'utilisateur de naviguer entre eux.

Sur mobile certaines règles d'expérience utilisateur sont fréquemment observées à ce propos, comme le fait d'avoir un bouton retour permettant de revenir à l'écran précédent, ou encore que les animations respectent le sens de la navigation (aller vers la droite pour aller vers le détail, revenir vers la gauche pour revenir au plus général). Nous allons voir que certains outils nous permettent de mettre tout cela en œuvre de manière très simple.

La seconde fonctionnalité est d'afficher des listes. Cela peut paraître trivial ; en effet, en développement web, afficher une liste ne présente aucune difficulté, et se fait de la même manière que pour n'importe quel autre élément. D'ailleurs, dans le premier exemple, nous avons affiché une liste de tâches de manière naïve et cela fonctionnait très bien. Seulement, il se pourrait que l'on ait besoin d'afficher des listes de grande taille, au point non seulement que l'intégralité de la liste ne serait pas visible (on devrait pouvoir faire défiler la vue), mais également que l'on n'aurait pas envie d'effectuer le rendu de tous les éléments pour des raisons de performance. Ici également, React Native propose de s'en sortir aisément, nous verrons cela.

Au long de ces sections, nous allons construire une application de gestion de contacts. Elle sera constituée des écrans suivants :

- `contactsList` affichera la liste des contacts.
- `viewContact` affichera les détails d'un contact.
- `editContact` permettra de modifier les informations d'un contact.

L'application sera initialisée de la même manière que pour le premier exemple, c'est-à-dire avec Expo. De la même manière, j'ai supprimé le contenu du fichier App.js pour le remplacer simplement par :

```
// App.js
import App from './src/components/App'
export default App
```

La première version de notre application sera très simple puisqu'elle ne permettra d'afficher qu'un seul contact. Ainsi, nous nous concentrerons sur l'aspect navigation pour laisser l'affichage de liste à la section suivante.

React Native ne propose pas de système de gestion de navigation au sein d'une application. Il permet bien d'accéder aux API d'iOS et Android, mais celles-ci sont assez différentes dans leur fonctionnement, il n'y a donc pas de moyen de gérer simplement la navigation pour les deux plateformes à la fois. C'est pour répondre à ce problème qu'a été créé *React Navigation*. Il permet simplement de faire ce que l'on pourrait attendre de React Native, à savoir gérer la navigation avec un même code de manière native sous iOS et Android.

Techniquement, React Navigation réimplémente cette navigation en JavaScript (et non en utilisant les API natives), mais il est très peu probable que vous vous en rendiez compte, tant le résultat final est similaire (visuellement et en termes de réactivité) à ce qui serait fait sur une application native.

Avant de mettre en place une navigation, créons deux composants `ContactsList` et `ViewContact`, que l'on verra ensuite comment organiser au sein d'une navigation.

```
// src/components/ContactsList.js
import React, { Component } from 'react'
import { View, Button, Text } from 'react-native'
import PropTypes from 'prop-types'

class ContactsList extends Component {
 static propTypes = {
   goToContactDetails: PropTypes.func.isRequired
 }
 state = {
   contacts: [{ id: 1, name: 'John Smith' }]
 }
 render() {
   const { goToContactDetails } = this.props
   const [contact] = this.state.contacts
   return (
     <View>
       <Text>Name: {contact.name}</Text>
       <Button
         onPress={() => goToContactDetails(contact)}
         title="Contact details"
       />
     </View>
   )
 }
```

```
}

export default ContactsList
```

```
// src/component/ViewContact.js
import React, { Component } from 'react'
import { Text } from 'react-native'
import PropTypes from 'prop-types'

class ViewContact extends Component {
 static propTypes = {
   contact: PropTypes.object.isRequired
 }
 render() {
   const { contact } = this.props
   return <Text>Details for {contact.name}</Text>
 }
}

export default ViewContact
```

Comme vous le voyez, `PropTypes` a été utilisé pour indiquer les propriétés attendues par chacun de ces deux composants :

- `ContactsList` attend une fonction `goToContactDetails`, qui nous permettra d'ouvrir les détails d'un contact donné.
- `ViewContact` attend un objet contact contenant les informations sur le contact à afficher.

Pour afficher des composants au sein d'une navigation, React Navigation attend que certaines informations soient données au composant de manière statique. Pour cela il est pertinent de définir des composants (que l'on peut appeler des écrans, ou *screens*), dont le but est :

- de gérer leurs propriétés : leur titre, la couleur de l'en-tête, les boutons à afficher dans l'en-tête, etc. ;
- d'accéder aux paramètres qui leur sont envoyés : nous allons voir en quoi cela consiste ;
- de proposer le nécessaire pour naviguer entre ces écrans.

Notez que l'utilisation de composants dédiés n'est qu'une méthode pour extraire la gestion de la navigation de nos composants « métiers ». Cela n'est pas une obligation imposée par React Navigation.

Commençons par l'écran `ContactsListScreen`, qui gère la navigation pour l'écran affichant la liste des contacts :

```
// src/components/ContactsListScreen.js
import React, { Component } from 'react'
import ContactsList from './ContactsList'

class ContactsListScreen extends Component {
 static navigationOptions = {
   title: 'Home'
 }
 goToContactDetails = contact => {
   const { navigation } = this.props
   navigation.navigate('viewContact', { contact })
 }
 render() {
   return <ContactsList
     goToContactDetails={this.goToContactDetails} />
 }
}

export default ContactsListScreen
```

Des choses intéressantes sont à noter ici :

- Nous supposons qu'à notre composant sera passée une propriété `navigation`. Cette propriété sera en fait passée par React Navigation grâce à la manière dont nous allons déclarer la navigation dans quelques instants.
- Cette propriété `navigation` nous donne accès à une méthode `navigate`, permettant d'afficher un écran grâce à un ID (ici `viewContact`) et en lui envoyant des paramètres (ici un objet contenant le contact à afficher).
- Nous déclarons le membre statique `navigationOptions` permettant de définir les options de l'écran. Ici nous n'avons défini qu'un titre.

Comprenez donc que ce composant permet d'afficher `ContactsList` au sein de la navigation, tout en lui donnant le moyen, via la propriété `goToContactDetails`, d'afficher les détails d'un contact en utilisant la navigation.

Passons à l'écran `ViewContactScreen`, gérant la navigation pour l'écran permettant d'afficher un contact :

```
// src/components/ViewContactScreen.js
import React, { Component } from 'react'
import ViewContact from './ViewContact'

class ViewContactScreen extends Component {
 static navigationOptions = {
   title: 'Details'
 }
 render() {
   const { navigation } = this.props
   const contact = navigation.getParam('contact')
   return <ViewContact contact={contact} />
 }
}

export default ViewContactScreen
```

Ici également nous utilisons la propriété `navigation` pour récupérer le paramètre `contact` qui a été envoyé par `ContactsListScreen`. Ici, nous affichons donc le composant `ViewContact`, en lui fournissant par la propriété `contact` le contact sélectionné, envoyé en tant que paramètre de navigation.

Nos deux écrans sont prêts, il ne reste qu'à lier le tout. Pour cela, nous créons d'abord un objet *navigator* à l'aide de la fonction `createStackNavigator` de React Navigation, que nous encapsulons dans un *app container*, créé par `createAppContainer` :

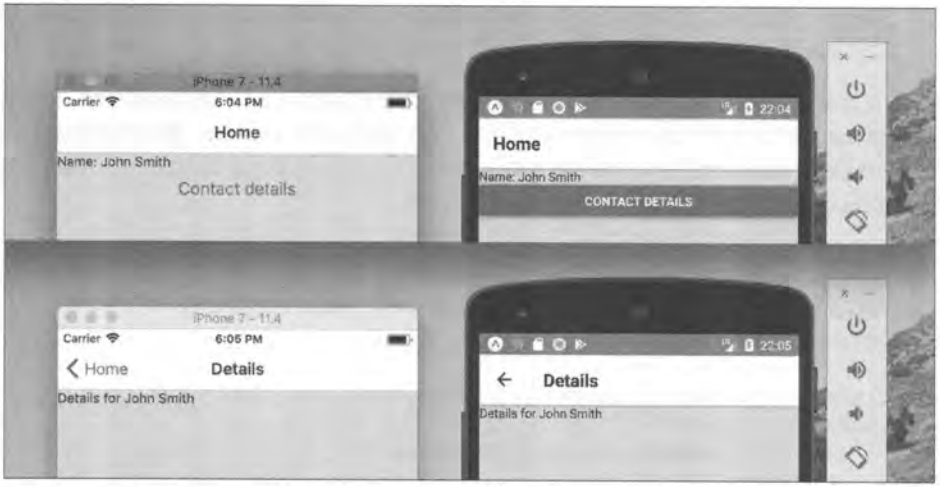
```
// src/components/App.js
import React, { Component } from 'react'
import {
 createAppContainer,
 createStackNavigator
} from 'react-navigation'
import ContactsListScreen from './ContactsListScreen'
import ViewContactScreen from './ViewContactScreen'

const Navigator = createStackNavigator({
 contactsList: ContactsListScreen,
 viewContact: ViewContactScreen
})
```

```
const AppContainer = createAppContainer(Navigator)

export default AppContainer
```

Par défaut, le premier écran déclaré est affiché au démarrage, ici `contactsList`. Vous pouvez remarquer que les écrans sont identifiés par des ID, d'où le `navigation.navigate('viewContact')` que l'on a vu dans `ContactsListScreen`.

Chose très intéressante que vous avez peut-être remarquée : à aucun moment nous n'avons géré de retour à la liste de contacts depuis l'affichage des détails d'un contact. Cela est en effet pris en charge par défaut par React Navigation (bien que paramétrable).

Si vous lancez l'application sur iOS et Android, vous pourrez observer que l'aspect visuel de l'en-tête respecte les bonnes pratiques d'expérience utilisateur de chaque plateforme. De plus le comportement diffère légèrement. Par exemple sur Android le bouton physique **Retour** permet également de revenir à la liste des contacts, et l'animation n'est pas la même.



Affichage de la navigation sur iOS et Android

3. Affichage de listes

À présent que nous avons su gérer la navigation au sein de notre application, passons à la deuxième notion à aborder : l'affichage de listes. Il est fréquent de vouloir afficher des listes dans une application, qu'il s'agisse d'une liste de contacts, d'articles ou de tâches. Tellement fréquent que React Native (et avant lui les plateformes iOS et Android) met à disposition le moyen de les gérer facilement, et surtout efficacement.

Efficacement dans le sens où pour l'affichage de listes comportant de nombreux éléments, il n'est pas nécessaire de générer le rendu pour tous les éléments, seuls ceux visibles importent. De plus, une fois que l'on fait défiler la liste vers le bas, les éléments du haut, devenus invisibles, peuvent probablement voir leur rendu libéré de la mémoire. Ce sont toutes ces optimisations que nous n'aurons pas à gérer nous-mêmes.

Pour créer une liste, deux composants s'offrent à nous :

- `FlatList` permet d'afficher une liste simple ;
- `SectionList` permet d'afficher une liste par section, comportant chacune un titre notamment.

Pour notre exemple, nous allons utiliser une `FlatList`. La première chose à faire est de définir un composant que l'on souhaite afficher pour chaque élément de la liste, en l'occurrence pour notre exemple, pour chaque contact. Appelons ce composant `ContactListItem` :

```
// src/components/ContactListItem.js
import React, { Component } from 'react'
import { View, Text, TouchableOpacity, StyleSheet }
 from 'react-native'

class ContactListItem extends Component {
 onPress = () => {
   const { openDetails } = this.props
   openDetails()
 }
 render() {
   const { contact } = this.props
   return (
     <TouchableOpacity onPress={this.onPress}>
```

```
        <View style={styles.contact}>
          <Text>{contact.name}</Text>
        </View>
      </TouchableOpacity>
    )
  }
}

const styles = StyleSheet.create({
 contact: {
   padding: 16,
   backgroundColor: 'white',
   borderBottomWidth: 1,
   borderBottomColor: 'lightgray'
 }
})

export default ContactListItem
```

Ce composant est extrêmement simple, nous ne faisons qu'afficher le nom du contact. Lorsque le composant est cliqué, nous appelons la fonction `openDetails` que l'on attend en propriété, et qui ouvrira l'écran permettant de visualiser les détails d'un contact.

À présent, mettons à jour le composant `ContactsList` afin d'afficher une vraie liste, en utilisant notre nouveau composant. `FlatList` attend qu'on lui passe au minimum trois propriétés :

- `data` : le tableau des éléments que l'on souhaite afficher ;
- `keyExtractor` : une fonction permettant d'associer à un élément de `data` une clé unique (cette propriété n'est en fait pas nécessaire si les éléments de `data` possèdent déjà un attribut `key`) ;
- `renderItem` : une fonction permettant de générer le rendu pour chaque élément de `data` (attention ce n'est pas directement l'élément qui est passé en paramètre, mais un objet dont l'attribut `item` contient l'élément).

```
// src/components/ContactsList.js
// ...
render() {
 const { goToContactDetails } = this.props
 const { contacts } = this.state
 return (
```

```
    <FlatList
      data={contacts}
      keyExtractor={contact => String(contact.id)}
      renderItem={({ item: contact }) => (
        <ContactListItem
          contact={contact}
          openDetails={() => goToContactDetails(contact)}
        />
      )}
    />
  )
}
// ...
```

Afin de tester convenablement notre liste pour l'affichage de nombreux éléments, nous allons ajouter un appel à une API qui permettra d'initialiser notre liste de contacts. Nous utiliserons l'API JSON Placeholder, que nous avions déjà aperçue au deuxième chapitre. Nous appellerons l'URL `/users` permettant de récupérer une liste de dix utilisateurs.

Pour cela ajoutons une méthode `loadContacts` dans notre composant `ContactsList` et appelons-la dans `componentDidMount` :

```
// ...
state = {
 loading: false,
 contacts: []
}
loadContacts = async () => {
 this.setState({ loading: true })
 const res =
   await fetch('https://jsonplaceholder.typicode.com/users')
 const contacts = await res.json()
 this.setState({ loading: false, contacts })
}
componentDidMount() {
 this.loadContacts()
}
```

Notez que j'ai modifié l'affichage des contacts dans `ContactListItem` afin d'agrandir le *padding* à 32, et ainsi que l'affichage des dix contacts nécessite de faire défiler la vue.

Maintenant que nous avons cette fonction permettant de charger les contacts, il est même possible de configurer `FlatView` pour activer le *pull-to-refresh*. C'est ce qui permet de rafraîchir une liste lorsque l'utilisateur "tire" la liste vers le bas, comme pour la faire défiler plus haut ; cette fonctionnalité est particulièrement appréciée des utilisateurs d'applications mobiles. Pour l'utiliser, il suffit de fournir deux propriétés supplémentaires à `FlatList` :

- `refreshing` : un booléen indiquant si le rafraîchissement est en cours.
- `onRefresh` : la fonction à appeler pour rafraîchir la liste.

```
const { loading, contacts } = this.state
return <FlatList
 data={contacts}
 keyExtractor={contact => String(contact.id)}
 refreshing={loading}
 onRefresh={this.loadContacts}
 renderItem={/* ... */}
/>
```

Cette liste convenablement gérée clôt ce chapitre d'introduction à React Native. Nous allons dans le prochain chapitre poursuivre avec cette application de gestion de contacts, en ajoutant la possibilité d'utiliser l'appareil photo du téléphone pour ajouter des photos de contacts. Nous en profiterons également pour rendre son architecture plus robuste, en y intégrant Redux. Ainsi l'ajout de nouvelles fonctionnalités comme l'édition de contacts sera plus aisé.

Chapitre 6
Fonctionnalités avancées avec React Native

1. Utiliser une fonctionnalité native : l'appareil photo

Dans l'exemple du chapitre précédent, nous avons créé une application permettant de gérer des contacts, en affichant des listes et en gérant une navigation pour les écrans. Nous allons dans ce chapitre améliorer cet exemple en raffinant la navigation et en y ajoutant Redux. Mais tout d'abord, explorons une possibilité offerte par React Native et Expo : l'utilisation de fonctionnalités natives du smartphone. Nous allons voir comment permettre à l'utilisateur de prendre des photos et de les afficher dans l'application. Cela nous permettra d'ajouter une fonctionnalité à l'application : celle d'associer des photos de profil à nos contacts.

Pour cette section, nous nous contenterons de réaliser un composant permettant de prendre une photo et de l'afficher ensuite. Dans les sections suivantes, nous intégrerons cette fonctionnalité à l'application.

Tout d'abord, nous aurons besoin d'installer deux dépendances à notre projet :

- `expo-camera` pour accéder à l'appareil photo du téléphone.
- `expo-permissions` pour demander l'autorisation au téléphone (et donc, à l'utilisateur) d'accéder à l'appareil photo.

Installer ces dépendances avec NPM ne sera pas suffisant, car il y a également des opérations à faire dans le code natif de l'application, c'est-à-dire le code spécifique de chaque plateforme, iOS et Android. Nous n'avons pas accès directement à ce code, car c'est Expo qui le gère pour nous. C'est pourquoi Expo propose une commande pour réaliser tout cela pour nous (y compris l'installation via NPM) :

```
$ expo install expo-camera
$ expo install expo-permissions
```

Créons ensuite un composant `TakePicture`. Il disposera d'un state local comportant deux éléments :

- `hasCameraPermission`, qui vaudra `true` si l'on peut accéder à l'appareil photo, et `false` sinon. En attendant d'avoir cette information (le temps que l'utilisateur réponde à la demande), il vaudra `null`.
- `photos`, tableau qui contiendra les photos qui seront prises.

Nous stockerons les photos au format renvoyé par Expo, c'est-à-dire un objet comportant divers attributs (chemin du fichier, taille...), dont nous ne nous soucions guère puisque nous pouvons les afficher directement grâce au composant `Image` de React Native.

```
export default class TakePicture extends Component {
 state = {
   hasCameraPermission: null,
   photos: [],
 }
```

Dans la méthode `componentDidMount`, nous demandons l'autorisation d'utiliser l'appareil photo. Bien heureusement, la demande ne sera faite à l'utilisateur qu'une seule fois pour l'application, et non à chaque démarrage.

```
async componentDidMount() {
 const { status } = await Permissions.askAsync(
   Permissions.CAMERA
 )
 this.setState({ hasCameraPermission: status === 'granted' })
}
```

Dans le cas où nous n'avons pas (ou pas encore) la permission, nous afficherons un message à l'utilisateur :

```
render() {
 const { hasCameraPermission, photos } = this.state
 if (hasCameraPermission === null) {
   return <Text>Waiting for permission...</Text>
 }
 if (hasCameraPermission === false) {
   return <Text>We need permission to access camera.</Text>
 }
 // ...
```

L'affichage de ce que « voit » l'appareil photo se fait grâce au composant `Camera` fourni par Expo Camera. Pour ce qui est d'interagir avec, c'est-à-dire dans notre cas prendre la photo, nous aurons besoin d'un accès à ce composant. Pour cela, il nous faudra utiliser une possibilité offerte par React : les références, ou *refs*.

Le principe est de pouvoir accéder à un composant une fois qu'il est monté dans le DOM ou dans l'application mobile. D'abord, on crée un objet *ref* grâce à la fonction `createRef` de React. Dans notre cas, nous le stockerons comme membre de la classe :

```
class TakePictureTemp extends Component {
 state = { ... }
 cameraRef = createRef()
```

Lorsque nous utiliserons le composant `Camera`, nous fournirons le fournirons en propriété `ref`. Ainsi dès que le composant sera monté, il sera possible d'accéder à son instance en utilisant `cameraRef.current`. Ainsi nous pourrons accéder à l'API proposée par le composant.

La suite de notre méthode `render` ressemblerait donc à cela, en affichant l'objet `Camera`, puis un bouton permettant de prendre la photo, et enfin une galerie horizontale des photos qui ont été prises :

```
render() {
 // ...
 return (
   <View style={styles.container}>
     <Camera
       ref={this.camera}
```

```
        style={styles.camera}
        type={Camera.Constants.Type.back}
      />
      <Button title="Take picture" onPress={this.takePicture} />
      <View style={styles.picturesContainer}>
        <ScrollView horizontal>
          {photos.map((photo, index) => (
            <Image key={index}
                   style={styles.picture}
                   source={photo} />
          ))}
        </ScrollView>
      </View>
    </View>
  )
}

// ...
const styles = StyleSheet.create({
 container: {
   flex: 1,
 },
 camera: {
   flex: 1,
 },
 picturesContainer: {
   height: 100,
   flexDirection: 'row',
 },
 picture: {
   width: 100,
   height: 100,
 },
})
```

Il ne nous reste qu'à écrire la méthode `takePicture` qui va effectivement prendre la photo et l'ajouter au tableau des photos dans le state :

```
takePicture = async () => {
 const photo = await this.camera.current.takePictureAsync()
 this.setState({ photos: [photo, ...this.state.photos] })
}
```

Afin de tester notre composant, je vous suggère de modifier temporairement le fichier App.js à la racine du projet pour exporter `TakePicture` plutôt que le composant `App` contenant le navigateur (dans src/App).

Il va de soi que pour profiter pleinement de l'appareil photo il est préférable de lancer l'application sur un vrai appareil. Si vous ne l'avez jamais fait, Expo explique la procédure dans la console lorsque vous lancez l'application. Néanmoins, sous iOS par exemple, vous pourrez tout de même simuler la prise de photo, les photos retournées seront simplement noires avec une incrustation de la date de la photo.

Retournons à présent à notre application de gestion de contacts. Avant d'y intégrer cet écran de prise de photo, je vous propose d'ajouter Redux dans l'application. En effet, la gestion d'un état global pour l'application (sans utiliser Redux) peut être laborieuse du fait qu'avec la navigation on ne peut pas toujours passer des propriétés explicitement à un composant (notamment aux écrans).

2. Ajouter Redux à l'application

Ajouter Redux à une application React Native ne présente pas de différence par rapport à une application web. Il faudra commencer par installer les dépendances nécessaires :

```
$ yarn add redux react-redux redux-thunk
```

Il n'est pas nécessaire ici de séparer la gestion du store en plusieurs services, nous aurons donc un seul fichier store.js dans lequel nous placerons actions et reducer.

```
// src/store.js
import { createStore, applyMiddleware } from 'redux'
import thunk from 'redux-thunk'

const initialState = {
 loading: false,
 contacts: [],
 error: null
}
```

```
const actionTypes = {
 LOAD_CONTACTS: 'LOAD_CONTACTS',
 LOAD_CONTACTS_START: 'LOAD_CONTACTS_START',
 LOAD_CONTACTS_SUCCESS: 'LOAD_CONTACTS_SUCCESS',
 LOAD_CONTACTS_FAILURE: 'LOAD_CONTACTS_FAILURE'
}

export const actions = {
 loadContacts: () => async dispatch => {
   dispatch(actions.loadContactsStart())
   try {
     const res = await
       fetch('https://jsonplaceholder.typicode.com/users')
     const contacts = await res.json()
     dispatch(actions.loadContactsSuccess(contacts))
   } catch (err) {
     dispatch(actions.loadContactsFailure(err))
   }
 },
 loadContactsStart: () =>
   ({ type: actionTypes.LOAD_CONTACTS_START }),
 loadContactsSuccess: contacts => ({
   type: actionTypes.LOAD_CONTACTS_SUCCESS,
   contacts
 }),
 loadContactsFailure: error => ({
   type: actionTypes.LOAD_CONTACTS_FAILURE,
   error
 })
}

const reducer = (state = initialState, action) => {
 switch (action.type) {
   case actionTypes.LOAD_CONTACTS_START:
     return { ...state, error: null, loading: true }
   case actionTypes.LOAD_CONTACTS_SUCCESS:
     return { ...state, loading: false,
              contacts: action.contacts }
   case actionTypes.LOAD_CONTACTS_FAILURE:
     return { ...state, loading: false, error: action.error }
   default:
     return state
 }
}
```

```
export default createStore(reducer, applyMiddleware(thunk))
```

Notre composant `ContactsList` est alors modifié pour ne plus avoir à gérer de state local, mais plutôt utiliser celui de Redux :

```
//...
import { connect } from 'react-redux'
import { actions } from '../store'

class ContactsList extends Component {
 //...
 componentDidMount() {
   const { loadContacts } = this.props
   loadContacts()
 }
 render() {
   const {
     goToContactDetails,
     loadContacts,
     loading,
     contacts
   } = this.props
   return (
     <FlatList
       data={contacts}
       keyExtractor={contact => String(contact.id)}
       refreshing={loading}
       onRefresh={() => loadContacts()}
       renderItem={/* ... */}
     />
   )
 }
}

const mapStateToProps = state => ({
 loading: state.loading,
 contacts: state.contacts
})

const mapDispatchToProps = {
 loadContacts: actions.loadContacts
}
```

```
export default connect(
 mapStateToProps,
 mapDispatchToProps
)(ContactsList)
```

Il ne nous reste plus qu'à encapsuler notre application dans le composant `Provider` de React-Redux pour finalement relier tout cela :

```
// src/components/App.js
// ...
import { Provider } from 'react-redux'
import store from '../store'

const Navigator = createStackNavigator({
 /* ... */
})
const AppContainer = createAppContainer(Navigator)

export default () => (
 <Provider store={store}>
   <AppContainer />
 </Provider>
)
```

Comme vous pouvez le constater, il n'y a aucune difficulté à utiliser Redux avec React Native. Dès lors que l'on utilise React Navigation (ou n'importe quel autre système de navigation), il se montre d'autant plus utile même pour de petites applications.

3. Plus loin avec la navigation

Pour clôturer ce chapitre, allons un peu plus loin dans la navigation en ajoutant la possibilité de créer et de modifier des contacts. Cela nous donnera l'occasion d'une part de voir comment afficher des fenêtres modales (ou simplement modales) avec React Navigation, et d'autre part de réfléchir au meilleur moyen de faire travailler Redux et React Navigation ensemble.

Les modifications effectuées dans cette section sont importantes bien qu'une grande partie n'apporte pas de nouveautés aux notions que nous avons vues, c'est pourquoi je ne mettrai ici que quelques bouts de code ciblés.

Je vous encourage à vous référer aux exemples téléchargeables accompagnant le livre si vous souhaitez voir l'ensemble du code source final de l'application.

3.1 Une modale pour l'édition d'un contact

Pour commencer, voyons déjà ce que l'on entend par « modale ». Jusqu'ici, la navigation que nous avons vue n'était composée que d'écrans s'affichant les uns à la suite des autres (seulement deux en l'occurrence), avec la possibilité de revenir en arrière. Pour React Navigation, une modale est différente dans la mesure où il s'agit d'un écran pouvant être affiché à n'importe quelle étape du flow de navigation. Sur le plan expérience utilisateur sur mobile, une modale s'affiche généralement différemment : sur iOS par exemple l'écran surgit du bas du téléphone (et repart vers le bas lorsque la modale est fermée).

Dans React Navigation, la gestion des modales se fait lors de la création du navigateur. Par exemple, si l'on souhaitait afficher l'écran des détails d'un contact sous forme de modale, on pourrait écrire :

```
createStackNavigator(
  {
    contactsList: ContactsListScreen,
    viewContact: ViewContactScreen
  },
  { mode: 'modal' }
)
```

Mais ici ce n'est pas ce que l'on souhaite ; nous voulons conserver la navigation actuelle pour les détails, mais afficher une modale pour éditer ou créer un contact. Pour cela nous pouvons imbriquer les navigateurs les uns dans les autres. Nous aurons donc un navigateur pour la liste et les détails, et un pour la modale. Cela permet à chacun des navigateurs de gérer sa propre navigation (même si pour la création et l'édition nous n'aurons qu'un seul écran), et d'avoir l'en-tête standard avec titre et boutons.

Comme chacun des navigateurs a son en-tête, on peut en profiter pour supprimer celui du navigateur principal (sans quoi deux en-têtes seraient affichés) :

```
createStackNavigator(
 {
   mainNavigator: createStackNavigator({
     contactsList: ContactsListScreen,
     viewContact: ViewContactScreen
   }),
   editContactNavigator: createStackNavigator({
     editContact: EditContactScreen
   })
 },
 {
   mode: 'modal',
   headerMode: 'none'
 }
)
```

Pour afficher l'écran **EditContactScreen**, nous devons créer un bouton `Add` dans l'écran **ContactsListScreen**, comme bouton de droite du `header` :

```
static navigationOptions = ({ navigation }) => ({
 title: 'Home',
 headerRight: (
   <AddContactButton
     goToEditContact={() => navigation.push('editContact')}
   />
 )
})
```

Le comportement de ce bouton sera double. Il devra bien évidemment afficher l'écran d'édition de contact, mais également dispatcher une action permettant d'initialiser les informations du contact à éditer (ici un contact vierge, puisqu'on souhaite faire une création). Il devra donc avoir accès à Redux, ce qui n'est pas le cas ici dans `navigationOptions`, puisqu'il s'agit d'un membre statique de la classe, qui n'a donc pas accès aux propriétés que l'on pourrait définir grâce à React-Redux.

C'est pourquoi nous créerons un composant indépendant `AddContactButton` qui, lui, sera bien connecté à Redux :

```
// src/components/AddContactButton.js
// ...
class AddContactButton extends Component {
 onPress = () => {
   const { goToEditContact, setContactToEdit } = this.props
   setContactToEdit({ name: 'New contact' })
   goToEditContact()
 }
 render() {
   return <Button onPress={this.onPress} title="Add" />
 }
}

const mapDispatchToProps = {
 setContactToEdit: actions.setContactToEdit
}

export default connect(
 null,
 mapDispatchToProps
)(AddContactButton)
```

Pour ce qui est de l'édition d'un contact existant, nous utiliserons un composant similaire `EditContactButton`. Par contre, lui attendra une propriété `contactId` permettant de savoir quel contact éditer :

```
// src/components/EditContactButton.js
// ...
onPress = () => {
 const {
   contactId,
   contacts,
   goToEditContact,
   setContactToEdit
 } = this.props
 const contact = contacts.find(c => c.id === contactId)
 setContactToEdit(contact)
 goToEditContact(contactId)
}
// ...
```

```
const mapStateToProps = state => ({
 contacts: state.contacts
})

const mapDispatchToProps = {
 setContactToEdit: actions.setContactToEdit
}

export default connect(
 mapStateToProps,
 mapDispatchToProps
)(EditContactButton)
```

Sur l'écran d'édition d'un contact, nous ferons également en sorte d'afficher le nom du contact dans l'en-tête. Pour cela, la solution la plus simple serait de passer le contact (ou au moins son nom) en paramètre de navigation : `navigation.navigate('editContact', { contactName: ... })`. Cependant il est préférable de n'utiliser les paramètres de navigation que pour stocker les « vrais » paramètres de navigation (par exemple l'ID du contact en cours d'édition).

Pour afficher le nom, le plus propre est encore une fois de passer par un nouveau composant connecté à Redux :

```
// src/components/EditContactScreen.js
static navigationOptions = ({ navigation }) => {
 const contactId = navigation.getParam('contactId')
 return {
   headerTitle: (
     <ContactName contactId={contactId}
                  defaultTitle="Create contact" />
   ),
   // ...
 }
}
```

Vous aurez en effet fréquemment à vous poser la question suivante : dois-je stocker cette donnée dans le state de Redux ou comme paramètre de navigation ? Encore une fois, il n'y a pas de règle universelle, mais je recommanderais d'en stocker le moins possible dans les paramètres de navigation. Il est préférable de n'utiliser ces derniers que pour stocker les paramètres indispensables pour la navigation, et de passer par Redux pour le reste.

De la même manière, je pense préférable de ne pas utiliser Redux pour stocker les paramètres de navigation.

3.2 Intégration de la prise de photo

Finalement, il nous reste à intégrer notre fonctionnalité d'ajout de photos. Pour cela nous pouvons nous en sortir sans nouvelle action dans le store. Le composant `TakePicture` peut être simplifié pour ne plus afficher la galerie de photos en bas. Lorsqu'une photo sera prise, il dispatchera une action `updateContact` pour mettre à jour le contact en cours d'édition et lui ajouter la photo, puis naviguera vers l'écran précédent, c'est-à-dire l'écran d'édition de contact :

```
// src/components/TakePicture.js
takePicture = async () => {
 const { contact, updateContact, navigation } = this.props
 const photo = await this.camera.current.takePictureAsync()
 updateContact({ ...contact, photo })
 navigation.goBack()
}
```

Pour ce qui est de l'affichage de la photo, nous utiliserons globalement le même procédé que pour notre galerie de photos. Dans le composant `Edit-Contact` par exemple, affichons la photo de sorte que lorsqu'elle est cliquée on soit redirigé vers l'écran pour prendre une nouvelle photo :

```
// src/components/EditContact.js
<TouchableOpacity
 style={styles.pictureContainer}
 onPress={() => navigation.push('takePicture')}
>
 {contact.photo ? (
   <Image style={styles.picture} source={contact.photo} />
 ) : (
   <Text>Add photo</Text>
 )}
</TouchableOpacity>
```

Cela suppose naturellement que l'on ait créé un écran pour la prise de photo, et qu'on l'ait ajouté au navigateur de l'application :

```
// src/components/TakePicture.js
import React, { Component } from 'react'
import TakePicture from './TakePicture'

class TakePictureScreen extends Component {
 static navigationOptions = {
   headerTitle: 'Take picture',
 }
 render() {
   return <TakePicture />
 }
}

export default TakePictureScreen
```

```
// src/components/App.js
// ...
const Navigator = createStackNavigator(
 {
   // ...
   editContact: createStackNavigator({
     editContact: EditContactScreen,
     takePicture: TakePictureScreen,
   }),
 },
 // ...
```

Prenant exemple sur ces composants, vous pouvez également afficher la photo à d'autres endroits. Dans les exemples téléchargeables accompagnant le livre, vous pourrez voir que j'ai affiché la photo également sur la fiche contact `ViewContact`, ainsi que dans la liste des contacts (voir `ContactListItem`).

Je m'arrête ici pour la présentation de la création et l'édition de contacts. J'espère que cet exemple d'application de gestion de contacts vous aura plu et vous aura permis d'en apprendre plus sur React Native. Si vous vous êtes senti un peu perdu entre la navigation, le state de Redux et l'affichage des listes, pas de panique !

Ce sont des notions qui, vues ensemble, peuvent être déconcertantes. C'est pourquoi je vous encourage à examiner attentivement le code source complet de cet exemple, ou mieux encore essayer de créer par vous-même une petite application mêlant plusieurs écrans de navigation. De nombreuses subtilités viendront se mettre en travers de votre apprentissage, mais elles donneront naissance à de bonnes pratiques que vous vous imposerez, et avec de la pratique rien ne sera insurmontable !

4. Conclusion

Ces deux chapitres consacrés à React Native vous auront permis, je l'espère, de voir que développer des applications mobiles n'est pas beaucoup plus difficile que développer pour le Web. J'aurais aimé présenter beaucoup d'autres choses à ce sujet, mais le développement avec React Native pourrait se voir consacrer un livre entier ! Notamment je n'ai pas abordé la gestion de fichiers ressources (images par exemple), ou certaines possibilités offertes par Expo (affichage d'un *splashscreen* au démarrage de l'application), ou encore la possibilité de persister des données localement dans le téléphone (les photos que l'on a prises par exemple, pour qu'elles soient disponibles au prochain lancement de l'application).

Si vous souhaitez aller plus loin dans le développement avec React Native, je ne doute pas que vous trouverez les ressources nécessaires sur Internet. La documentation de React Native (https://facebook.github.io/react-native/) est bien sûr le premier site à mettre dans vos favoris ! Elle contient de nombreux tutoriels pratiques, des fonctions les plus simples aux plus avancées (problématiques de performances par exemple).

La documentation de React Navigation (https://reactnavigation.org/docs/en/getting-started.html) est également très complète et très orientée « guides pratiques ». Il en est de même enfin de la documentation d'Expo (https://docs.expo.io/versions/latest/), qui vous expliquera par exemple comment créer un *package* de votre application pour la distribuer sur les stores d'Apple et de Google.

Chapitre 7
Gestion de formulaires et du routage

1. Introduction

Ce chapitre, ainsi que le suivant, développent grâce à un exemple pratique complet quatre notions que l'on retrouve dans un grand nombre d'applications :

- Tout d'abord les formulaires, moyen classique de recueillir une entrée de l'utilisateur lorsque celle-ci comporte plusieurs données. Plusieurs champs sont affichés, lesquels sont ensuite validés, parfois même la saisie est contrainte (en nombre de caractères par exemple), puis l'utilisateur soumet le formulaire pour que sa saisie soit prise en compte.
- Ensuite, le routage qui permet d'organiser une application en routes, souvent elles-mêmes organisées hiérarchiquement. Généralement dans une application web, cela se traduit par la mise à jour de l'URL dans la barre d'adresse du navigateur.
- Puis l'authentification, procédé permettant à un utilisateur de s'inscrire et s'identifier dans l'application, qui lui donne par exemple accès à des ressources qui lui sont réservées.
- Enfin le stockage de données distantes, ce qui permet à l'utilisateur de créer des données auxquelles il pourra accéder plus tard, éventuellement depuis une autre machine.

Ces quatre fonctions ne sont pas propres à React ; néanmoins, elles sont tellement fréquentes qu'il existe des moyens de faciliter leur implémentation grâce à des bibliothèques utilisées conjointement à React.

Afin d'observer ces notions de manière concrète, au long de ces deux chapitres nous développerons une application complète (minimaliste) de gestion de dépenses :

- comportant des formulaires permettant l'inscription d'un utilisateur ou encore la saisie d'une nouvelle dépense ou la modification d'une dépense existante
- mettant à jour l'URL en fonction de la dépense en cours d'édition, mais aussi permettant à l'utilisateur d'accéder directement à une dépense en accédant à une URL spécifique à cette dépense
- permettant l'authentification des utilisateurs par le biais d'une inscription et d'un écran de connexion
- permettant à plusieurs utilisateurs de saisir des dépenses après s'être inscrits, tout en faisant en sorte que chacun ne voit que ses propres dépenses.

Il devrait vous être possible de réaliser l'application au fur et à mesure de la lecture du chapitre. Néanmoins si vous souhaitez vous y référer, le code source complet de l'application est bien évidemment disponible dans les exemples téléchargeables accompagnant le livre.

Notez que dans ces deux chapitres nous réaliserons une application web, mais la plupart des notions développées sont applicables aux applications mobiles. Notamment, les bibliothèques utilisées sont prévues pour fonctionner avec React Native. Je vous suggère de les mettre en œuvre sur une application web pour commencer, puis à vous référer aux sites respectifs des bibliothèques pour voir l'adaptation à React Native.

2. Création de formulaires avec Formik

Avoir besoin de formulaires est très fréquent sur les applications web ou mobiles, que ce soit pour donner l'occasion à un utilisateur de s'inscrire ou de s'identifier, ou pour créer une ressource, faire une recherche, ou contacter un support clientèle.

Lorsqu'on crée un formulaire avec React, se pose la question de la manière dont on va stocker les données saisies par l'utilisateur :

- Les stockera-t-on dans un state global, par exemple grâce à Redux ? Dans ce cas, toute modification de l'utilisateur entraînera la création d'une action et la mise à jour du state.
- Ou bien utilisera-t-on un state local au composant contenant le formulaire ?

Les deux solutions sont tout à fait acceptables, et chacune dispose des outils facilitant son implémentation. La première peut être mise en œuvre grâce à *Redux-Form* (https://redux-form.com) par exemple, qui facilite la création des actions nécessaires, et permet la gestion des données saisies dans le state.

Pour ma part, j'opte plutôt pour la deuxième solution, car je considère que les données saisies sur le formulaire, tant qu'elles ne sont pas validées (« soumises ») par l'utilisateur, n'ont pas besoin d'être partagées avec le reste de l'application. Un state local fera donc l'affaire, et pour cela j'utilise la bibliothèque *Formik* (https://jaredpalmer.com/formik).

Notez que les deux bibliothèques Redux-Form et Formik proposent à peu de choses près les mêmes fonctionnalités de base, mais qu'aucune n'est indispensable pour créer des formulaires avec React. Cela étant dit certaines tâches communes sont grandement facilitées :

- La mise à jour du state (local ou global) lors d'une entrée de l'utilisateur.
- La validation des entrées selon des règles définies, effectuée dès que le focus sur un champ est perdu, et l'affichage des messages d'erreurs éventuels.
- Le fait que la soumission ne soit possible qu'après validation du formulaire complet.
- etc.

Passons à l'action et créons une application comme nous en avons l'habitude (voir le premier chapitre), en installant cette fois-ci le paquet `formik` en plus :

```
$ yarn add formik
```

Pour l'instant, le composant principal de l'application `App` ne fera qu'utiliser le composant `SignUpForm` que nous allons créer ensuite, permettant à un utilisateur de s'inscrire à notre application.

```
// src/components/App.js
import React, { Component } from 'react'
import SignUpForm from './SignUpForm'

class App extends Component {
 render() {
   return <SignUpForm />
 }
}

export default App
```

2.1 Premier formulaire : inscription d'un utilisateur

Le formulaire d'inscription que nous allons créer comportera trois champs, un pour saisir un nom d'utilisateur et deux pour le mot de passe, dont la saisie doit être répétée. Commençons naïvement par créer notre formulaire de manière classique :

```
// src/components/SignUpForm.js
import React from 'react'

const SignUpForm = () => (
 <form>
   <label>
     User name:
     <input type="text" name="username" />
   </label>
   <label>
     Password:
     <input type="password" name="password" />
   </label>
   <label>
```

```
      Password (repeat):
      <input type="password" name="passwordRepeat" />
    </label>
    <button type="submit">Sign up</button>
  </form>
)

export default SignUpForm
```

Ce formulaire fonctionne très bien, à ceci près qu'il ne fait aucune validation sur les entrées de l'utilisateur, et envoie directement au back-end (à supposer qu'il y en ait un) les valeurs saisies.

Pour effectuer une validation sur les données, la première étape est d'intégrer notre formulaire au sein du composant `Formik` :

```
import { Formik } from 'formik'

const SignUpForm = () => (
  <Formik
    initialValues={{
      username: '',
      password: '',
      passwordRepeat: ''
    }}
    onSubmit={values => console.log(values)}
  >
    {({ values, handleChange, handleSubmit }) => (
      <form onSubmit={handleSubmit}>
        <label>
          User name:
          <input
            type="text"
            name="username"
            value={values.username}
            onChange={handleChange}
          />
        </label>
        {/* ... */}
        <button type="submit">Sign up</button>
      </form>
    )}
  </Formik>
)
```

La syntaxe peut vous étonner ; en effet l'unique enfant de `Formik` est une fonction. Il s'agit d'un pattern appelé *render props* que nous verrons plus en détail dans un prochain chapitre. Pour le moment, considérez juste que les paramètres de cette fonction (ici `values`, `handleChange` et `handleSubmit`) sont des données ou fonctions que Formik nous fournit pour que nous les utilisions dans notre formulaire.

Détaillons tout d'abord les paramètres que nous fournissons à `Formik` :

- `initialValues` contient les valeurs avec lesquelles le formulaire doit être initialisé. Il peut s'agir de valeurs vides pour le cas d'un formulaire de création, mais aussi de données d'un objet existant pour un formulaire de modification.
- `onSubmit` est la fonction appelée lorsque le formulaire est soumis. Elle reçoit en paramètres les valeurs saisies par l'utilisateur. Son intérêt est qu'elle n'est appelée que si les données sont valides, comme nous allons le voir dans un instant.

Afin que tout fonctionne bien, il est nécessaire de brancher Formik sur notre formulaire. Pour cela :

- l'attribut `onSubmit` doit appeler `handleSubmit` (afin de gérer la validation notamment) ;
- chaque champ (`input`) doit avoir son attribut `name` renseigné avec la clé correspondante dans `initialValues` et l'objet retourné à `onSubmit` ;
- l'attribut `onChange` de chaque champ doit appeler `handleChange`.

Avec tout cela nous obtenons un formulaire géré avec Formik aux fonctions minimales. D'ailleurs si vous lancez l'application vous ne verrez pas vraiment de changement avec la version précédente. Passons à ce qui nous intéresse : la validation.

La validation d'un formulaire avec Formik s'implémente grâce à une fonction à laquelle est passé un objet contenant les valeurs saisies, et renvoyant un objet contenant les erreurs de validations, selon la clé de la valeur erronée. Cette fonction est à passer en propriété du composant `Formik` :

```
<Formik
 initialValues={{ username: '' }}
 validate={values => {
```

```
    const errors = {}
    if (!values.username) {
      errors.username = 'Please enter a user name.'
    } else if (
        !values.username.match(/[a-z][a-z0-9_\-.]{2,15}/i)
      ) {
      errors.username = 'Please enter a valid user name.'
    }
    return errors
  }}
// ...
```

La validation sera alors appelée à la soumission du formulaire. Celle-ci sera stoppée si la validation échoue. Il est alors important de signaler les erreurs à l'utilisateur, celles-ci étant disponibles dans les paramètres de la fonction générant le formulaire :

```
{({ values, handleChange, handleSubmit, errors }) => (
 <form onSubmit={handleSubmit}>
   <label>
     User name:
     <input
       type="text"
       name="username"
       value={values.username}
       onChange={handleChange}
     />
     {errors.username && <span className="error">{errors.user-
name}</span>}
   </label>
```

À présent, si vous essayez de soumettre le formulaire sans avoir saisi de nom d'utilisateur, l'erreur vous est signalée.

Validation de champ

Pour améliorer légèrement l'expérience utilisateur, Formik permet de valider la saisie non pas uniquement à la soumission du formulaire, mais également lorsque le focus est perdu par le champ, puis à partir de ce moment, dès que la valeur est à nouveau modifiée.

On prévient ainsi l'utilisateur le plus tôt possible, sans toutefois afficher tous les messages d'erreurs à l'affichage initial du formulaire.

Pour cela, il suffit de définir l'attribut `onBlur` des champs avec la fonction `handleBlur` fournie par Formik. De plus, on affichera le message d'erreur que si le champ a été visité (`touched`) :

```
{({ values, handleChange, handleBlur, handleSubmit, touched,
errors }) => (
 <form onSubmit={handleSubmit}>
   <label>
     User name:
     <input
       // ...
       onBlur={handleBlur}
     />
     {errors.username &&
       touched.username && (
         <span className="error">{errors.username}</span>
       )}
   </label>
)}
```

Notre exemple est complet (pour ce qui est du premier champ), mais peut-être trouvez-vous que notre composant contient beaucoup de code (même pour un seul champ). Bonne nouvelle : Formik propose des composants permettant de simplifier considérablement le code. Ainsi, la fonction générant notre formulaire devient :

```
<Formik /* ... */>
 {({ handleSubmit }) => (
   <form onSubmit={handleSubmit}>
     <label>
       User name:
       <Field type="text" name="username" />
       <ErrorMessage
         name="username" className="error"
         component="span" />
```

```
        </label>
        {/* ... */}
        <button type="submit">Sign up</button>
      </form>
    )}
  </Formik>
```

Le composant `Field` crée notre champ input en définissant convenablement les propriétés `onChange`, `onBlur` et `value` ; le composant `ErrorMessage` affiche le message d'erreur, mais seulement s'il y a eu erreur. Au final, cette version raccourcie est quasiment équivalente à ce que nous avions avant. Notez que les composants `Field` et `ErrorMessage` peuvent accéder aux données et paramètres du formulaire grâce à la notion de contexte de React, que nous verrons en détail dans un prochain chapitre.

Je vous propose à titre d'exercice de compléter cette implémentation du formulaire en y ajoutant les deux champs de mots de passe avec leur validation respective :

- le premier mot de passe doit contenir au minimum huit caractères ;
- le second doit être identique au premier.

Validation du formulaire complet

À présent que notre formulaire est prêt, mettons-le de côté pour le moment, nous le réutiliserons lorsque nous verrons comment gérer l'authentification plus loin dans le chapitre.

Pour le moment, mettons en pratique ce que nous venons de voir en créant un deuxième formulaire légèrement plus complexe.

2.2 Deuxième formulaire : création/modification d'une dépense

Le deuxième formulaire que nous allons créer ne comportera pas beaucoup de nouveautés par rapport à celui que nous avons vu, mais il sera utilisé pour deux fonctions : la création d'une nouvelle dépense et la modification d'une dépense existante.

Commençons en ne prenant en compte que la création. Le formulaire s'appellera `ExpenseForm` et, pour faciliter la lisibilité, sera défini comme une classe (bien que nous n'utilisions pas de state local ni les méthodes de cycle de vie du composant).

Définissons les valeurs par défaut du formulaire, la validation ainsi que l'action à effectuer à la soumission directement comme membres de classe afin de simplifier la méthode `render` :

```
// src/components/ExpenseForm.js
import React, { Component } from 'react'
import { Formik, Field, ErrorMessage } from 'formik'

class ExpenseForm extends Component {
 defaultValues = {
   title: '',
   date: new Date()
     .toISOString()
     .split('T')
     .shift() // yyyy-mm-dd,
   amount: 0,
   notes: ''
 }
 // ...
```

Une dépense sera définie par un titre, une date au format « yyyy-mm-dd » (pour faciliter l'utilisation de l'`input` de type date), un montant, et un commentaire.

```
// ...
validate = values => {
 const errors = {}
 if (!values.title) {
   errors.title = 'Please enter a title.'
 }
 if (!values.date) {
   errors.date = 'Please enter a date.'
 }
 if (!values.amount) {
   errors.amount = 'Please enter a non-zero amount.'
 }
 return errors
}
// ...
```

La validation vérifiera que le titre, la date et le montant sont bien remplis. Nous ne faisons pas de vérification particulière sur le commentaire.

```
// ...
onSubmit = expense => {
 this.props.onSubmit(expense)
}
// ...
```

À la soumission, nous appellerons la fonction `onSubmit` passée en propriété du composant, en supposant que celle-ci attend comme paramètre la dépense à créer.

Pour rendre plus concis le code de `render`, j'ai également créé la méthode `renderError`, qui ne fait qu'appeler `ErrorMessage` avec les bons paramètres pour ne pas avoir à les répéter.

```
// ...
renderError(name) {
 return <ErrorMessage name={name}
   component="span" className="error" />
}
// ...
```

La méthode `render` devient alors relativement simple :

```
// ...
 render() {
   return (
     <Formik
       initialValues={this.defaultValues}
       validate={this.validate}
       onSubmit={this.onSubmit}
     >
       {({ handleSubmit, isSubmitting }) => (
         <form className="expense-form" onSubmit={handleSubmit}>
           <label>
             Title: <Field type="text" name="title" />
             {this.renderError('title')}
           </label>
           <label>
             Date: <Field type="date" name="date" />
             {this.renderError('date')}
           </label>
           <label>
             Amount (€): <Field type="number" name="amount" />
             {this.renderError('amount')}
           </label>
           <label>
             Notes: <Field component="textarea" name="notes" />
             {this.renderError('notes')}
           </label>
           <button type="submit">Create</button>
         </form>
       )}
     </Formik>
   )
 }
}

export default ExpenseForm
```

Pour utiliser notre composant `ExpenseForm`, mettons à jour le composant `App` afin de lui faire gérer un state local, comportant les dépenses créées, l'ID de la prochaine dépense créée, ainsi qu'un flag indiquant si le formulaire de création doit être affiché :

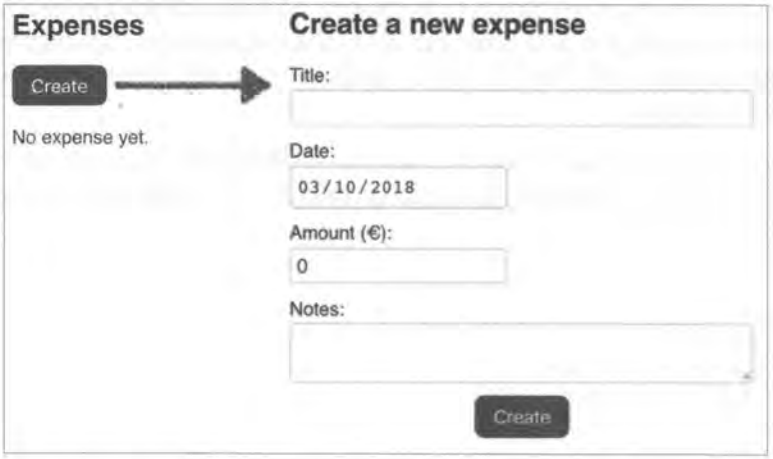
```
// src/components/App.js
import React, { Component } from 'react'
import ExpenseForm from './ExpenseForm'

class App extends Component {
 state = {
   expenses: [],
   nextExpenseId: 0,
   isCreatingExpense: false,
 }
// ...
```

Dans le composant, définissons ensuite les méthodes `showCreationForm` et `createExpense`, permettant respectivement d'afficher le formulaire et de créer une nouvelle dépense à l'aide des informations passées en paramètres :

```
// ...
showCreationForm = () => {
 this.setState({ isCreatingExpense: true })
}
createExpense = expenseInfos => {
 this.setState({
   expenses: [
     { id: this.state.nextExpenseId, ...expenseInfos },
     ...this.state.expenses
   ],
   nextExpenseId: this.state.nextExpenseId + 1,
   isCreatingExpense: false
 })
}
// ...
```

Enfin, faisons en sorte dans la méthode `render` que notre application affiche soit le formulaire de création d'une dépense, soit la liste des dépenses créées en fonction du flag `isCreatingExpense` :

```
// ...
 render() {
   if (this.state.isCreatingExpense) {
     return (
       <Fragment>
         <h2>Create a new expense</h2>
         <ExpenseForm onSubmit={this.createExpense} />
       </Fragment>
     )
   }
   return (
     <Fragment>
       <h2>Expenses</h2>
       <button onClick={this.showCreationForm}>Create</button>
       {this.state.expenses.length > 0 ? (
         <ul>
           {this.state.expenses.map(expense => (
             <li key={expense.id}>
               {expense.title}: {expense.amount} € spent on{' '}
               {new Date(expense.date).toDateString()}
             </li>
           ))}
         </ul>
       ) : (
         <p>No expense yet.</p>
       )}
     </Fragment>
   )
 }
}

export default App
```



Création d'une dépense

Notre application est complète pour ce qui est de la création d'une dépense ! Si vous la lancez, vous devriez être capable de créer une nouvelle dépense. Mais si vous ne saisissez pas certaines informations obligatoires, des messages d'erreurs doivent être alors affichés.

Passons à l'édition d'une dépense. Tout d'abord, mettons à jour le formulaire `ExpenseForm`. La seule modification que nous aurons besoin d'effectuer sera par rapport aux valeurs initiales du formulaire. En effet nous souhaitons que, si une dépense est passée en paramètre au composant, ce soit celle-ci qui serve pour l'initialisation. Pour cela, mettons à jour la propriété `initialValues` passée à Formik :

```
<Formik
 initialValues={this.props.expense || this.defaultValues}
 // ...
```

Pour bien signaler que c'est une édition et non une création, modifions également le libellé du bouton de soumission :

```
<button type="submit">
 {this.props.expense ? 'Update' : 'Create'}
</button>
```

C'est tout pour le formulaire ! Bien évidemment, nous pourrions ajouter des comportements spécifiques à la création ou à la modification : afficher un champ en lecture seule, afficher l'ID de la dépense, etc. Mais faisons au plus simple pour le moment.

Pour ce qui est du composant App, nous allons ajouter dans son state un attribut `currentlyEditedExpense` qui contiendra la dépense en cours d'édition s'il y en a une :

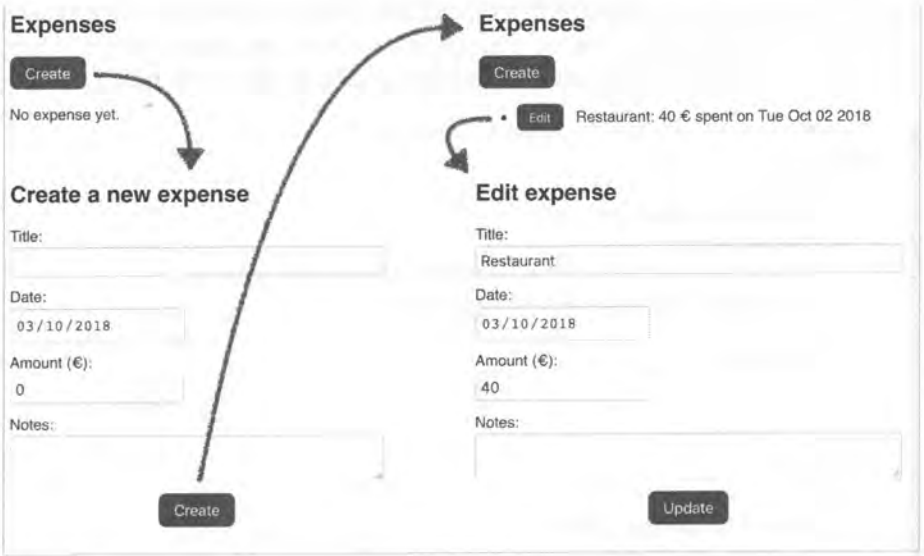
```
state = {
 expenses: [],
 nextExpenseId: 0,
 isCreatingExpense: false,
 currentlyEditedExpense: null
}
```

On définira ensuite deux méthodes permettant d'afficher le formulaire d'édition et de mettre à jour une dépense, de la même manière que ce qui est fait pour la création :

```
showEditionForm = expense => {
 this.setState({ currentlyEditedExpense: expense })
}
updateExpense = expenseInfos => {
 const { expenses } = this.state
 const expenseIndex = expenses.findIndex(e =>
   e.id === expenseInfos.id
 )
 const expensesBefore = expenses.slice(0, expenseIndex)
 const expensesAfter = expenses.slice(expenseIndex + 1)
 this.setState({
   expenses: [
     ...expensesBefore,
     expenseInfos,
     ...expensesAfter
   ],
   currentlyEditedExpense: null
 })
}
```

Pour finir, il nous reste à afficher le formulaire d'édition dans `render` en ajoutant un cas supplémentaire, et d'ajouter à la liste des dépenses un bouton d'édition pour chaque dépense, permettant d'afficher le formulaire :

```
if (this.state.currentlyEditedExpense) {
 return (
   <Fragment>
     <h2>Edit expense</h2>
     <ExpenseForm
       expense={this.state.currentlyEditedExpense}
       onSubmit={this.updateExpense}
     />
   </Fragment>
 )
}
return (
 <Fragment>
   {/* ... */}
   <li key={expense.id}>
     <button onClick={() => this.showEditionForm(expense)}>
       Edit
     </button>
     {expense.title}: {expense.amount} € spent on{' '}
   {/* ... */}
```



Création et modification

Ainsi, notre application permet à présent de créer une nouvelle dépense, et de modifier une dépense existante. À titre d'exercice, vous pouvez si vous le souhaitez ajouter au formulaire un bouton d'annulation, permettant de revenir à la liste des dépenses sans apporter de modification. Pour cela, le plus simple est d'ajouter une propriété `onCancel` au formulaire, appelée au moment du clic.

Dans la prochaine section, nous allons voir comment structurer cette petite application grâce au routage, qui permettra une meilleure navigation entre la liste des tâches et les formulaires.

3. Gestion du routage avec React Router

Notre application permet déjà une navigation cohérente grâce à la manière dont nous l'avons gérée, c'est-à-dire dans le state local du composant App. Nous avons deux attributs `isCreatingExpense` et `currentlyEditedExpense`, indiquant respectivement si on est sur le formulaire de création ou le formulaire d'édition. Dans ce cas, que va-t-on ajouter dans cette section ?

En utilisant une stratégie de routage, nous allons :

- mettre à jour l'URL de la page en fonction de l'emplacement de l'application dans lequel se trouve l'utilisateur (liste des dépenses, création ou modification) ;
- permettre à l'utilisateur de naviguer dans l'historique grâce aux boutons **Précédent** et **Suivant** de son navigateur (pour revenir à la liste des dépenses par exemple) ;
- offrir la possibilité de se rendre à un emplacement de l'application défini dans l'URL, par exemple en ajoutant l'URL aux favoris, en la partageant, ou en rafraîchissant la page.

Toutes ces fonctionnalités nous sont offertes par *React Router* (https://reacttraining.com/react-router/), bibliothèque la plus utilisée pour gérer le routage d'une application React, que ce soit pour le Web ou le mobile.

Afin d'implémenter le routage dans notre application, procédons par étape. Nous allons tout d'abord réorganiser (ou refactorer) quelque peu notre application afin de la préparer à l'arrivée de React Router. Dans un second temps nous définirons les routes que nous souhaitons avoir dans notre application, puis nous ajouterons enfin React Router.

3.1 Refactoring et définition des routes

Le refactoring préalable est en réalité très simple, mais il nous permettra d'y voir plus clair lorsque nous allons ajouter React Router à l'application. Il consiste à définir clairement ce que nous souhaitons afficher en fonction du state, et plus particulièrement de la partie concernant la navigation.

Pour mieux comprendre, passons sans plus tarder à la pratique : commençons à ajouter une méthode `renderCreateExpenseForm` à la classe `App`, contenant le JSX à retourner dans le cas où l'on souhaite afficher le formulaire de création :

```
renderCreateExpenseForm = () => {
 return (
   <Fragment>
     <h2>Create a new expense</h2>
     <ExpenseForm
       onSubmit={this.createExpense}
       onCancel={this.onCreateExpenseCancel}
     />
   </Fragment>
 )
}
```

De la même manière, une nouvelle méthode `renderEditExpenseForm` permettra d'afficher le formulaire d'édition d'une dépense :

```
renderEditExpenseForm = () => {
 const expense = this.state.currentlyEditedExpense
 return (
   <Fragment>
     <h2>Edit expense</h2>
     <ExpenseForm
       expense={expense}
       onSubmit={this.updateExpense}
       onCancel={this.onUpdateExpenseCancel}
     />
   </Fragment>
 )
}
```

Enfin, la méthode `renderExpensesList` affichera la liste des dépenses, avec les boutons de création et d'édition d'une dépense (je ne remets pas la totalité du code ici, mais il s'agit du même code que nous avions dans la version précédente de l'application).

```
renderExpensesList = () => {
 return (
   <Fragment>
     <h2>Expenses</h2>
     <button className="primary"
```

```
            onClick={this.showCreationForm}>
        Create
      </button>
      {/* ... */}
    </Fragment>
  )
}
```

Ainsi, notre méthode `render` est considérablement simplifiée, mais met bien en évidence la stratégie de routage que nous allons mettre en place pour notre navigation :

```
render() {
 if (this.state.isCreatingExpense) {
   return this.renderCreateExpenseForm()
 }
 if (this.state.currentlyEditedExpense) {
   return this.renderEditExpenseForm()
 }
 return this.renderExpensesList()
}
```

Notre application aura donc trois routes :

- La première permettra d'afficher les dépenses, nous utiliserons l'URL racine /
- La deuxième permettra d'accéder directement au formulaire de création, l'URL sera `/create`
- Enfin, la troisième affichera le formulaire de création d'une dépense spécifique, dont l'ID sera également dans l'URL : `/<id>/edit`

Notez que nous allons devoir gérer également les cas d'erreurs :

- L'URL n'est pas reconnue, c'est-à-dire ne correspond à aucun des trois cas ci-dessus.
- On accède à l'URL d'édition, mais avec un ID qui ne correspond à aucune dépense existante.

3.2 Ajout du routage avec React Router

Pour utiliser React Router, il est bien évidemment nécessaire de l'ajouter aux dépendances : `yarn add react-router-dom`. Puis dans notre fichier App.js, importons tout de suite le nécessaire, nous verrons au fur et à mesure à quoi tout cela sert :

```
import {
 BrowserRouter as Router,
 Route,
 Link,
 Switch
} from 'react-router-dom'
```

La définition du routage de l'application se fait grâce à l'utilisation de composants, nous allons donc le faire dans la méthode `render` :

```
render() {
 return (
   <Router>
     <Switch>
       <Route exact path="/"
         render={this.renderExpensesList} />
       <Route exact path="/create"
         render={this.renderCreateExpenseForm} />
       <Route exact path="/:id/edit"
         render={this.renderEditExpenseForm} />
       <Route render={this.renderNotFound} />
     </Switch>
   </Router>
 )
}
```

Il n'est pas difficile de comprendre à peu près ce que fait ce code dans les grandes lignes. Le composant `Router` permet de définir la portion de l'application concernée par le routage. Il est tout à fait possible d'inclure ce composant dans un élément précis de l'application (c'est-à-dire sans englober le tout), mais dans ce cas ce qui est à l'extérieur n'aura pas accès au contexte (c'est-à-dire l'endroit où se trouve l'utilisateur) et ne pourra pas non plus le mettre à jour (se rendre à un endroit spécifique). La plupart du temps, c'est toute l'application qui sera englobée dans un `Router`.

Nous définissons donc quatre routes, les trois premières correspondant à ce que nous avons établi précédemment (liste, création, modification), et la quatrième s'affichant si aucune des trois premières ne correspond à l'URL. Pour chacune d'entre elles nous utilisons l'attribut `render` qui permet comme son nom l'indique de définir (sous forme de fonction) quoi afficher lorsqu'il s'agit de la route courante.

Notez que la route associée à l'édition prend un paramètre `:id`. Grâce à cette syntaxe, toute URL respectant le modèle `/<id>/edit` correspondra ici, et la valeur `<id>` sera placée dans le paramètre `id` que nous allons ensuite récupérer (voir plus loin dans cette section).

Notez également que pour les trois premières routes nous spécifions l'attribut `exact`. Cela indique à React Router que nous souhaitons bien une correspondance exacte, c'est-à-dire par exemple que l'URL `/create/test`, ne doit pas correspondre à la route de la création. Nous verrons un peu plus loin dans le chapitre l'intérêt de ne pas procéder par correspondance exacte.

Le composant `Switch` permet de définir que, parmi les routes définies en son sein, seule une devra être affichée. Autrement dit dès qu'une route correspondra à l'URL, c'est uniquement celle-ci qui sera prise en compte. Sans le `Switch` il serait possible d'avoir plusieurs correspondances (ce qui est tout à fait correct dans beaucoup de cas, mais pas pour ce que nous souhaitons faire). En réalité ici, si nous n'avions que les trois premières routes, dans la mesure où celles-ci sont exclusives (c'est-à-dire qu'une URL ne peut pas correspondre à plus qu'une route) il ne serait pas nécessaire d'utiliser `Switch`. Mais la quatrième route correspondrait alors toujours puisqu'elle n'a pas de modèle d'URL défini.

À présent que nous avons indiqué à React Router la stratégie de routage que nous souhaitions implémenter, voyons les mises à jour à faire dans le comportement associé à chaque route.

Pour la liste des dépenses tout d'abord, peu de modifications à faire ici. Nous n'allons mettre à jour que les boutons de création et d'édition d'une dépense. Dans la mesure où la navigation n'est plus définie dans le state du composant, nous allons remplacer les deux boutons en utilisant le composant `Link` de React Router.

Dans son utilisation la plus simple, celui-ci prend en paramètre l'URL où l'on souhaite se rendre au moment du clic :

```
// Pour la création:
<Link to="/create">
 Create
</Link>

// Pour l'édition:
<Link to={`/${expense.id}/edit`}>
 Edit
</Link>
```

Ainsi, lorsque l'un des boutons sera cliqué, l'URL affichée dans le navigateur sera mise à jour en conséquence, et le formulaire de création ou d'édition sera affiché. Si l'utilisateur clique sur le bouton **Précédent** du navigateur, il reviendra alors à la liste des dépenses (sans que l'application soit rechargée). S'il clique ensuite sur **Suivant**, il sera à nouveau sur le formulaire.

Voyons ensuite ce qu'il en est de l'affichage du formulaire de création :

```
renderCreateExpenseForm = ({ history }) => {
 return (
   <Fragment>
     <h2>Create a new expense</h2>
     <ExpenseForm
       onSubmit={expenseInfos => {
         this.createExpense(expenseInfos)
         history.push('/')
       }}
       onCancel={() => {
         history.push('/')
       }}
     />
   </Fragment>
 )
}
```

Nous avons ici mis à jour les deux attributs `onSubmit` et `onCancel`. Dans les deux cas, l'idée est de revenir à la liste des dépenses, cela se fait au moyen de la méthode `history.push`, qui permet de se rendre à une route spécifique. L'objet `history` est passé automatiquement en paramètre à la fonction du fait que nous avons passé cette fonction en propriété `render` du composant `Route`.

Pour ce qui est de l'édition, la première étape consiste à récupérer la valeur du paramètre `id` extrait de l'URL et d'en déduire la dépense à modifier :

```
renderEditExpenseForm = ({ history, match }) => {
 const expenseId = Number(match.params.id)
 const expense = this.state.expenses.find(e => e.id ===
expenseId)
 // ...
```

Nous récupérons l'ID grâce à l'objet `match` également passé en paramètre, contenant des informations ayant servi à établir la correspondance (le *match*) avec la route sur laquelle nous arrivons, et notamment les paramètres, dans l'attribut `params`. Notez que les paramètres sont toujours des chaînes de caractères, d'où l'utilisation de `Number` pour convertir l'ID en entier.

Dans le cas où l'ID ne permet pas de trouver une dépense existante, nous affichons un message d'erreur ainsi qu'un lien pour revenir à la liste des dépenses :

```
// ...
if (!expense) {
 return (
   <Fragment>
     <p>No expense with this ID.</p>
     <Link to="/">Go back to expense list</Link>
   </Fragment>
 )
}
// ...
```

Et dans le cas où l'ID est correct, nous affichons comme prévu le formulaire, en utilisant comme pour la création `history.push` pour revenir à la liste des dépenses en cas d'annulation ou après la modification :

```
// ...
 return (
   <Fragment>
     <h2>Edit expense</h2>
     <ExpenseForm
       expense={expense}
       onSubmit={expenseInfos => {
         this.updateExpense(expenseInfos)
         history.push('/')
       }}
```

```
        onCancel={() => {
          history.push('/')
        }}
      />
    </Fragment>
  )
}
```

Il ne nous reste qu'à gérer le cas d'une route invalide. Ici aussi nous affichons une erreur et proposons de revenir à la liste des dépenses :

```
renderNotFound = () => {
  return (
    <Fragment>
      <p>Nothing here...</p>
      <Link to="/" className="button">
        Go back to expense list
      </Link>
    </Fragment>
  )
}
```

Afin que la nouvelle version de notre composant `App` soit débarrassée de tout le superflu, nous pouvons à présent supprimer ce qui nous servait à gérer la navigation avant l'ajout de React Router :

- Les références aux attributs `isCreatingExpense` et `currentlyEditedExpense`, que ce soit dans le state initial, ou dans les appels à `setState` au moment de la création et de la modification.
- Les méthodes d'annulation (si vous les aviez ajoutées en exercice à la fin de la section précédente).

Si vous lancez l'application, vous devriez pouvoir jouer avec les boutons de création, d'édition et d'annulation et constater les effets sur l'URL affichée. Testez aussi les boutons **Précédent** et **Suivant** du navigateur pour vérifier que tout se passe bien. De plus en accédant à l'URL `/create` vous devriez arriver directement sur l'écran de création d'une dépense.

Pour ce qui est de l'édition, il n'est en réalité pas possible d'accéder à une dépense créée grâce à l'URL, et pour cause nous n'avons persisté les données du state nulle part, les dépenses créées sont donc effacées dès que l'application est rechargée.

Pour remédier à cela et tester le routage dans son ensemble, je vous propose d'utiliser le *local storage* du navigateur pour persister le state du composant App. Voyez cela comme une astuce temporaire et non quelque chose à mettre en œuvre dans vos applications. Il existe des moyens plus propres de mettre en place une persistance de données.

3.3 Persister des données dans le navigateur

Pour stocker des données dans le *local storage* du navigateur et les récupérer créons deux méthodes dans la classe App :

```
saveStateToLocalStorage = () => {
 window.localStorage.setItem('state',
   JSON.stringify(this.state)
 )
}
loadStateFromLocalStorage = () => {
 const stateJSON = window.localStorage.getItem('state')
 if (stateJSON) {
   this.setState(JSON.parse(stateJSON))
 }
}
```

La première `saveStateToLocalStorage` permet de placer le state dans le *local storage*, et la seconde `loadStateFromLocalStorage` de le récupérer (les valeurs stockées doivent être des chaînes de caractères, d'où l'utilisation de `JSON.parse` et `JSON.stringify`). Pour sauvegarder le state à chaque mise à jour, nous pouvons passer notre méthode en second paramètre de chaque appel de `setState` :

```
createExpense = expenseInfos => {
 this.setState(
   { ... },
   this.saveStateToLocalStorage
 )
}
updateExpense = expenseInfos => {
 // ...
 this.setState(
   { ... },
   this.saveStateToLocalStorage
```

```
  )
}
```

Pour rappel, la mise à jour du state grâce à `setState` est asynchrone. En plaçant `saveStateToLocalStorage` en second paramètre, on est assuré qu'elle sera appelée une fois le state mis à jour, ce qui n'est pas le cas si on l'appelait simplement après avoir appelé `setState`.

Enfin, pour récupérer le state stocké dans le *local storage*, on appellera `loadStateFromLocalStorage` dans la méthode `componentDidMount` :

```
componentDidMount() {
 this.loadStateFromLocalStorage()
}
```

Cette nouvelle version de l'application est complète. Vous pouvez à présent naviguer dans la liste des dépenses, en créer de nouvelles, rafraîchir la page, accéder aux dépenses existantes depuis leurs URL, etc.

Pour conclure cette section consacrée au routage, nous allons effectuer un dernier changement dans l'application. Nous allons ajouter un écran permettant de visualiser les détails d'une dépense (sans formulaire, donc), ce qui va me donner l'occasion de vous présenter la possibilité d'imbriquer des routes.

3.4 Ajout d'un nouvel écran

Le routage de l'application va être légèrement modifié. En effet lorsque l'on cliquera sur une dépense, ce n'est plus vers l'édition que l'on souhaitera être dirigé, mais vers le nouvel écran de visualisation d'une dépense. Sur cet écran se trouvera un bouton d'édition affichant le formulaire, et le bouton d'annulation de celui-ci reviendra vers l'écran de visualisation. Un bouton retour sur la visualisation permettra de revenir vers la liste.

Les routes de notre application seront donc maintenant les suivantes :

- 1. `/` : liste des dépenses
- 2. `/create` : création d'une dépense
- 3. `/:id` : redirection vers `/:id/details`
- 4. `/:id/details` : affichage des détails d'une dépense

– 5. `/:id/edit` : modification d'une dépense

Les routes 1 et 2 restent donc inchangées. Pour les routes 3, 4 et 5, nous allons créer un nouveau composant `Expense`, auquel le composant `App` délèguera la gestion du routage pour les URL commençant par `:id`.

Commençons donc par créer le composant `Expense`. Notez que nous partons du principe qu'à ce composant seront passées trois propriétés :

– la dépense à visualiser ou éditer `expense`,
– une fonction permettant de mettre à jour la dépense `updateExpense`,
– l'objet `match` fourni par le router.

```
// src/components/Expense.js
import React, { Component, Fragment } from 'react'
import { Link, Route, Redirect, Switch } from 'react-router-dom'
import ExpenseForm from './ExpenseForm'

class Expense extends Component {
// ...
```

En mode affichage, en plus d'afficher les détails d'une dépense, nous afficherons deux boutons permettant de revenir à la liste des dépenses et d'éditer la dépense :

```
renderDetails = () => {
 const { expense } = this.props
 return (
   <Fragment>
     <h2>Expense details</h2>
     <Link to="/">Back</Link>
     <Link to={`/${expense.id}/edit`}>Edit</Link>
     <p>
       Title: <strong>{expense.title}</strong>
     </p>
     {/* ... */}
 )
}
```

Pour ce qui est de l'édition, la méthode est en réalité celle que nous avions dans App, mais nous n'avons plus besoin de vérifier ici qu'à un ID correspond bien une dépense :

```
renderEdit = ({ history, match }) => {
 const { expense, updateExpense } = this.props
 return (
   <Fragment>
     <h2>Edit expense</h2>
     <ExpenseForm
       expense={expense}
       onSubmit={expenseInfos => {
         updateExpense(expenseInfos)
         history.push(`/${expense.id}`)
       }}
       onCancel={() => {
         history.push(`/${expense.id}`)
       }}
     />
   </Fragment>
 )
}
```

Au cas où un utilisateur saisirait une URL non valide (mais commençant par `/:id` où l'ID est bien celui d'une dépense valide), nous afficherons là encore une erreur :

```
renderNotFound = ({ match }) => {
 const { expense } = this.props
 return (
   <Fragment>
     <p>Nothing here...</p>
     <Link to={`/${expense.id}`} className="button">
       Go back to expense details
     </Link>
   </Fragment>
 )
}
```

Enfin, la méthode `render` de notre composant va, comme dans `App`, décrire les routes que l'on souhaite avoir dans notre application, ou plus précisément dans cette sous-partie de l'application (les détails d'une dépense) :

```
render() {
 const { match } = this.props
 return (
   <Switch>
     <Redirect exact from={`${match.path}`}
               to={`${match.path}/details`} />
     <Route exact path={`${match.path}/details`}
            render={this.renderDetails} />
     <Route exact path={`${match.path}/edit`}
            render={this.renderEdit} />
     <Route render={this.renderNotFound} />
   </Switch>
 )
}
```

Afin de ne pas avoir besoin de répéter le début de l'URL dans les chemins des routes, nous utilisons `match.path`, qui ici correspond à `/:id`. Cela permet par exemple de greffer notre composant dans l'application à un autre endroit, sans qu'il ait besoin de connaître l'URL qui a permis son affichage.

Notez qu'il n'est pas nécessaire de définir un deuxième `Router`. Une application n'a besoin que d'un seul routeur, dans lequel on peut mettre autant de `Switch` et de `Route` que l'on souhaite.

Remarquez enfin comment le composant `Redirect` permet simplement de faire une redirection d'une route vers une autre.

Il ne nous reste qu'à mettre à jour le composant `App` pour utiliser notre nouveau composant. Tout d'abord, la définition des routes change quelque peu : la route d'édition disparaît au profit de la route `/:id` :

```
<Route exact path="/" render={this.renderExpensesList} />
<Route exact path="/create"
 render={this.renderCreateExpenseForm} />
<Route path="/:id" render={this.renderExpenseView} />
<Route render={this.renderNotFound} />
```

Pour cette route nous ne mettons plus l'attribut `exact`, car nous souhaitons qu'elle inclue toutes les URL commençant par `/:id`.

Dans la liste des dépenses (méthode `renderExpensesList`), on n'affichera plus un bouton d'édition, mais un lien vers la dépense :

```
this.state.expenses.map(expense => (
 <li key={expense.id}>
   <Link to={`/${expense.id}`}>{/* ... */}</Link>
 </li>
))
```

Enfin, la méthode `renderEditExpenseForm` est renommée en `renderExpenseView`, et dans le cas où l'ID de dépense est valide, on n'affiche plus le formulaire, mais notre nouveau composant `Expense` :

```
return (
 <Expense match={match} expense={expense}
   updateExpense={this.updateExpense} />
)
```

Ce sera tout pour notre exploration du routage avec React Router. N'hésitez pas à lancer l'application, à naviguer entre les formulaires, utiliser les boutons **Précédent** et **Suivant** du navigateur, et tester différentes URL pour accéder directement à différents écrans.

J'espère que cette section vous aura convaincu que mettre en place un routage dans une application React n'est non seulement pas compliqué (même si cela demande de la rigueur dès qu'il y a plus de routes dont certaines imbriquées), mais également vraiment intéressant du point de vue expérience utilisateur. En effet, cela permet d'utiliser les possibilités du navigateur pour revenir à des écrans précédents, ou encore de sauvegarder ou partager des URL.

4. Prochaines étapes

Dans ce chapitre, nous avons vu comment :

- créer des formulaires à l'aide de Formik, et gérer la validation des données saisies par l'utilisateur ;
- gérer la navigation au sein de l'application grâce au routage, en créant différentes URL pour les ressources et écrans disponibles, en gérant des liens permettant de passer de l'un à l'autre.

Dans le chapitre suivant, nous allons retrouver le formulaire créé dans la première section de ce chapitre, afin d'imposer à l'utilisateur de s'inscrire et de s'identifier pour pouvoir gérer ses dépenses. Cela nous donnera l'occasion de voir une nouvelle fonctionnalité du routage : donner accès à certaines routes uniquement selon certaines conditions, par exemple que l'utilisateur soit authentifié.

Nous verrons ensuite comment appeler une API externe pour stocker des données distantes.

Chapitre 8
Sécurité et persistance avec Firebase

1. Découverte de Firebase

À présent que nous savons créer des formulaires et gérer le routage dans une application, la prochaine étape est d'ajouter une couche de sécurité en permettant aux utilisateurs de s'inscrire et de s'identifier pour utiliser cette application, ou certaines ressources privées de celle-ci. En effet, nous allons dans ce chapitre réutiliser le formulaire d'inscription créé au début du livre, mais également adapter notre routage pour rendre certaines routes privées. En l'occurrence, un utilisateur devra être identifié pour créer et visualiser des dépenses, puis les visualiser.

Pour gérer cela, nous allons utiliser Firebase de Google. Il s'agit d'un ensemble d'outils mettant à disposition un back-end, avec notamment la gestion de l'authentification d'utilisateurs et la persistance de données dans le cloud. Le but de ce chapitre n'est pas de présenter Firebase en détail, mais plutôt de profiter de la simplicité avec laquelle il permet de mettre en œuvre un back-end sans avoir à le créer ou l'héberger soi-même. Les concepts vus ici pourront être appliqués à n'importe quel back-end.

Pour commencer à utiliser Firebase, il vous faudra créer et configurer l'application dans la console de Firebase. Pour cela, je vous renvoie à la section Création d'une application Firebase qui se trouve en annexe de ce livre et qui décrit la procédure.

Dans la suite, je supposerai que vous avez une application configurée (avec l'authentification pour cette section, et la base de données temps réel pour la suivante), avec le code JavaScript de configuration (`var config = ...`).

Pour nous connecter à Firebase dans notre application, commençons par installer la bibliothèque `firebase` :

```
$ yarn add firebase
```

Créons ensuite un fichier firebase.js qui contiendra la configuration nécessaire, et initialisera la connexion à Firebase :

```
// src/firebase.js
import firebase from 'firebase/app'
import 'firebase/auth'

const config = {
 apiKey: 'xxxxxxxx'
 // etc.
}

if (firebase.apps.length === 0) {
 firebase.initializeApp(config)
}
```

Il n'est pas nécessaire d'exporter quoi que ce soit ; considérez que l'objet Firebase est défini globalement et qu'il ne peut y en avoir qu'une seule instance. Dans l'ensemble de l'application, lorsque nous aurons besoin d'interagir avec Firebase, nous importerons `firebase` depuis `firebase/app`.

La vérification `if (firebase.apps.length === 0)` sert à nous assurer que Firebase n'a pas déjà été initialisé (ce qui provoquerait l'affichage d'un avertissement dans la console), cas qui n'arrive en réalité qu'en développement lorsque le *hot reload* est activé (avec Parcel notamment).

Il ne nous reste qu'à appeler cette configuration dans index.js :

```
// src/index.js
import 'babel-polyfill'
import React from 'react'
import { render } from 'react-dom'
import App from './components/App'
import './firebase'
// ...
```

À présent, commençons par mettre en place l'authentification de l'utilisateur dans notre application.

2. Gestion de l'authentification

2.1 Inscription d'un utilisateur

Nous allons donc mettre à jour le formulaire `SignUpForm` créé dans la première section du chapitre Gestion de formulaires et du routage, afin d'appeler Firebase lorsque le formulaire est soumis (c'est-à-dire notamment lorsque les informations saisies ont été validées). Tout d'abord, l'authentification de Firebase fonctionne avec un couple adresse e-mail/mot de passe, et non avec un nom d'utilisateur. Commençons donc par remplacer dans le fichier les instances de `username` par `email`, et mettre à jour la validation du formulaire. Pour cela, j'ai pour ma part utilisé la bibliothèque `email-validator` :

```
import emailValidator from 'email-validator'
// ...
// Dans la validation:
if (!values.email) {
 errors.email = 'Please enter your e-mail address.'
} else if (!emailValidator.validate(values.email)) {
 errors.email = 'Please enter a valid e-mail address.'
}
```

Dans un deuxième temps, nous allons prévoir le cas où l'inscription ne se passe pas comme prévu, c'est-à-dire le cas où une erreur est renvoyée par Firebase. Pour afficher un message à l'utilisateur, nous allons devoir stocker un flag `hasError` dans le state local du composant, et donc convertir notre composant en classe :

```
class SignUpForm extends Component {
 state = {
   hasError: false
 }
 render() {
   const { hasError } = this.state
   return (
     <Fragment>
```

```
        <h2>Sign up</h2>
        <Formik /*...*/>
          {({ handleSubmit, isSubmitting }) => (
            <form>
              {/* ... */}
              <footer>
                <button type="submit" disabled={isSubmitting}>
                  {isSubmitting ? 'Signing up…' : 'Sign up'}
                </button>
              </footer>
              {hasError && (
                <p>
                  <span className="error">
                    Something wrong happened.
                  </span>
                </p>
              )}
            </form>
          )}
        </Formik>
      </Fragment>
    )
  }
}
```

Notez également que pour éviter que le formulaire ne soit soumis à nouveau (et donc les données envoyées à l'API plusieurs fois) avant que l'on ait reçu la première réponse (en double ou triple cliquant sur le bouton), nous désactivons le bouton lorsque le flag `isSubmitting` fourni par Formik est à `true`.

Bien, il nous reste le plus important : appeler Firebase pour procéder à l'inscription de l'utilisateur. Cela se passera dans le paramètre `onSubmit` donné à Formik :

```
import firebase from 'firebase/app'
// ...

onSubmit={async (values, { setSubmitting }) => {
 const { email, password } = values
 this.setState({ hasError: false })
 try {
   await firebase
     .auth()
     .createUserWithEmailAndPassword(email, password)
```

```
    console.log('Signed up successfully!')
  } catch (err) {
    console.error(err)
    this.setState({ hasError: true })
    setSubmitting(false)
  }
}}
```

Le code devrait vous paraître relativement simple à comprendre. L'appel à `firebase.auth().createUserWithEmailAndPassword` permet d'inscrire l'utilisateur à l'aide de son adresse e-mail et de son mot de passe. Cette fonction étant asynchrone, elle renvoie une promesse, d'où l'utilisation d'`async/await`. Si une erreur s'est produite, on définit le flag `hasError` à `true` pour afficher le message, et on appelle `setSubmitting` pour réactiver le bouton et permettre de soumettre à nouveau le formulaire.

Si l'inscription se passe bien, l'utilisateur sera également connecté automatiquement (sans passer par l'écran de connexion donc). Nous allons un peu plus loin voir qu'il n'est pas nécessaire de prévenir explicitement le composant `App` si celui-ci souhaite effectuer un traitement comme rediriger vers une autre page.

Pour tester votre formulaire et l'inscription, je vous suggère de modifier temporairement le fichier index.js pour afficher le composant `SignUpForm` au lieu d'`App`. Une fois un utilisateur créé, vous pourrez vous rendre dans la console Firebase de l'application pour constater que l'utilisateur a bien été ajouté à la liste des utilisateurs de l'application.

Dans la console Firebase, liste des utilisateurs inscrits

Suite logique, afin qu'un utilisateur inscrit préalablement puisse s'identifier, passons au formulaire de connexion.

2.2 Connexion d'un utilisateur

Le formulaire de connexion `SignInForm` sera extrêmement ressemblant au formulaire d'inscription, je vous suggère donc de copier le fichier, renommer la classe, et de supprimer le code spécifique à l'inscription, c'est-à-dire la gestion de la deuxième saisie du mot de passe (`passwordRepeat`). Changez ensuite le libellé du bouton **Sign up** par **Sign in**, et vous aurez un beau formulaire de connexion à peu de frais.

Remarque

En raison de la ressemblance entre les deux formulaires, il peut être tentant de ne faire qu'un seul composant pour mutualiser le code commun. C'est tout à fait possible, mais d'expérience j'ai tendance à préférer faire deux composants distincts, dans la mesure où rapidement le formulaire d'inscription devient de plus en plus complexe, en demandant d'autres informations : nom, site web, comptes de réseaux sociaux, validation de conditions d'utilisation, etc., alors que le formulaire de connexion reste toujours très simple. Mieux vaut un peu de duplication qu'une mauvaise abstraction (https://www.sandimetz.com/blog/2016/1/20/the-wrong-abstraction).

La partie intéressante de notre formulaire de connexion reste bien évidemment ce qui se passe à la soumission du formulaire. C'est encore une fois très similaire à ce qui se fait à l'inscription :

```
onSubmit={async (values, { setSubmitting }) => {
 const { email, password } = values
 this.setState({ hasError: false })
 try {
   await firebase.auth().signInWithEmailAndPassword(
     email, password
   )
   setSubmitting(false)
   console.log('Signed in successfully!')
 } catch (err) {
   console.error(err)
   this.setState({ hasError: true })
   setSubmitting(false)
```

```
  }
}
```

Est-il nécessaire d'expliquer le code ?

De la même manière vous pouvez tester ce formulaire en faisant en sorte que dans index.js on affiche temporairement `<SignInForm />`.

Maintenant que nos utilisateurs peuvent s'inscrire et s'authentifier, il nous reste à faire quelque chose de cette authentification, en adaptant l'affichage de l'application et les possibilités offertes en conséquence.

2.3 Gestion de l'authentification dans l'application

L'idée est d'inclure la gestion de l'authentification dans notre application de gestion de dépenses. Mais dans un premier temps, je vous propose de faire cela dans un composant temporaire (appelons-le `App2`), afin de mieux comprendre comment fonctionnera notamment le routage couplé à l'authentification.

Le but de cette application temporaire sera d'afficher deux pages de contenu :

- L'une d'elles sera la page d'accueil, accessible à l'URL / par tout le monde, connecté ou non.
- L'autre sera une page privée, accessible à l'URL `/private` mais uniquement par les utilisateurs connectés. Si un utilisateur non connecté tente d'y accéder, il sera redirigé vers la page de connexion.

En plus de cela nous aurons donc deux pages avec nos formulaires de connexion et d'inscription, accessibles respectivement aux URL `/signin` et `/signup`. Celles-ci ne seront accessibles qu'aux utilisateurs non connectés. En plus de cela, nous aurons une page `/signout` permettant de se déconnecter.

Mais avant de gérer le routage, commençons par voir comment vérifier qu'un utilisateur est connecté. Cela se fait de manière très simple avec Firebase, il suffit de souscrire à un évènement déclenché dès que l'état de l'authentification change.

Cet évènement est donc déclenché :

- au démarrage de l'application, pour recevoir l'état initial (utilisateur connecté ou non) ;
- lorsque l'utilisateur vient de se connecter (et de s'inscrire donc) ou de se déconnecter.

Notez que notre application devra donc gérer trois états pour l'authentification : l'état connecté, l'état non connecté, mais également l'état où l'on ne sait pas encore si l'utilisateur est connecté ou non.

Souscrivons à l'évènement dès le démarrage de l'application, pour mettre à jour son state local en conséquence :

```
// src/components/App2.js
class App extends Component {
 state = {
   isLoadingUser: true,
   user: null
 }
 componentDidMount() {
   this.unsubscribeAuth = firebase.auth()
     .onAuthStateChanged(user => {
       this.setState({ user, isLoadingUser: false })
     })
 }
 componentWillUnmount() {
   if (this.unsubscribeAuth) {
     this.unsubscribeAuth()
   }
 }
 // ...
}
```

Nous utilisons donc `firebase.auth().onAuthStateChanged`, qui permet d'appeler la fonction passée en paramètre dès que l'authentification évolue. Cela nous renvoie une nouvelle fonction, qui permettra d'annuler cette souscription, et ainsi éviter des erreurs de React si nous tentons d'appeler `setState` alors que le composant n'est pas monté.

À présent que nous avons les informations sur l'authentification, commençons par créer un en-tête pour notre application. Celui-ci affichera l'e-mail de l'utilisateur connecté s'il y en a un, et les liens vers les pages d'inscription, de connexion et de déconnexion.

```
renderHeader = () => {
 const { user, isLoadingUser } = this.state
 if (isLoadingUser) {
   return <header>Loading user info...</header>
 }
 if (user) {
   return (
     <header>
       <span>
         Signed in as <strong>{user.email}</strong>.
       </span>
       <span>
         <NavLink to="/signout">Sign out</NavLink>
       </span>
     </header>
   )
 }
 return (
   <header>
     <span>Not logged in.</span>
     <span>
       <NavLink to="/signin">Sign in</NavLink>{' '}
       <NavLink to="/signup">Sign up</NavLink>
     </span>
   </header>
 )
}
```

Notez que nous utilisons le composant `NavLink` pour les liens, et non `Link` comme nous l'avons fait jusque-là. Cela revient exactement au même, à ceci près que `NavLink` ajoute une classe « active » aux liens correspondant à la route courante, ce qui permet de les mettre en évidence, et rend donc ce composant particulièrement adapté pour les liens de navigation.

La page d'accueil et la page privée seront générées par les deux méthodes `renderHome` et `renderPrivate` :

```
renderHome = () => <p>Welcome!</p>
renderPrivate = () => <p>Welcome to this private section!</p>
```

Trois méthodes nous permettront d'afficher les pages d'inscription, de connexion et de déconnexion :

```
renderSignout = () => <Signout />
renderSignup = () => <SignUpForm />
renderSignin = () => <SignInForm />
```

Le composant `SignOut` est extrêmement simple, puisqu'il se contente d'appeler `firebase.auth().signOut()`. Nous n'avons pas besoin d'y gérer la redirection après la déconnexion, cela sera pris en charge par le composant `App2` et le routage, comme nous allons le voir.

```
// src/components/Signout.js
class Signout extends Component {
 componentDidMount() {
   firebase.auth().signOut()
 }
 render() {
   return <p>Signing out...</p>
 }
}
```

Enfin, il ne nous reste plus que la page à afficher dans le cas où la route entrée est invalide :

```
renderNotFound = () => (
 <Fragment>
   <p>Nothing here...</p>
   <Link to="/">Go to home</Link>
 </Fragment>
)
```

Bien, nous avons toutes les pages nécessaires, il nous reste à créer le routage pour les afficher correctement.

Voici une première implémentation simple :

```
render() {
 return (
   <Router>
     <Fragment>
       {this.renderHeader()}
       <Switch>
         <Route exact path="/signout"
           render={this.renderSignout} />
         <Route exact path="/signup"
           render={this.renderSignup} />
         <Route exact path="/signin"
           render={this.renderSignin} />
         <Route exact path="/private"
           render={this.renderPrivate} />
         <Route exact path="/"
           render={this.renderHome} />
         <Route render={this.renderNotFound} />
       </Switch>
     </Fragment>
   </Router>
 )
}
```

Cette implémentation fonctionne très bien, mais elle ne prend pas en compte le fait que l'utilisateur soit connecté ou non. En effet un utilisateur non connecté peut se rendre à l'adresse `/private` et voir la page privée, ce que nous ne souhaitons pas.

Nous pourrions gérer cela dans la méthode `renderPrivate`, afin de rediriger l'utilisateur vers l'écran de connexion s'il n'est pas connecté. Mais dans la mesure où il s'agit d'un traitement que nous aurons à refaire pour plusieurs routes dans la version finale de l'application (d'ailleurs, nous allons également le faire pour la route `/signout` ici), faisons les choses plus proprement en créant une fonction dédiée. Celle-ci attend en paramètre une fonction (de type « `renderXXX` » comme celles que nous avons ici) et fait en sorte qu'elle ne soit appelée que si l'utilisateur est connecté.

Dans le cas contraire, il est redirigé vers l'écran de connexion :

```
signedInOnly = render => props => {
 const { user, isLoadingUser } = this.state
 if (isLoadingUser) {
   return <p>Loading…</p>
 }
 if (user) {
   return render(props)
 }
 return <Redirect to="/signin" />
}
```

Nous pouvons mettre à jour le routage de manière à utiliser cette fonction dans les routes `/private` et `/signout` :

```
<Route
 exact
 path="/signout"
 render={this.signedInOnly(this.renderSignout)}
/>
<Route
 exact
 path="/private"
 render={this.signedInOnly(this.renderPrivate)}
/>
```

Le principe est exactement le même pour les routes qui ne doivent être accessibles qu'aux utilisateurs non connectés. Créons la méthode `notSignedInOnly` faisant plus ou moins l'inverse de `signedInOnly`:

```
notSignedInOnly = render => props => {
 const { user, isLoadingUser } = this.state
 if (isLoadingUser) {
   return <p>Loading…</p>
 }
 if (!user) {
   return render(props)
 }
 return <Redirect to="/" />
}
```

Puis utilisons-la pour les routes `/signin` et `/signup` :

```
<Route
 exact
 path="/signup"
 render={this.notSignedInOnly(this.renderSignup)}
/>
<Route
 exact
 path="/signin"
 render={this.notSignedInOnly(this.renderSignin)}
/>
```

Notre routage est complet, mais il reste une fonctionnalité non essentielle, mais très pratique que nous pouvons mettre en place. Supposons que vous ne soyez pas connecté à l'application, mais que vous essayiez de vous rendre à l'adresse `/private`. Vous êtes alors redirigé vers l'écran de connexion, mais après vous être connecté, vous êtes sur la page d'accueil. Ne serait-ce pas appréciable d'arriver sur la page privée, puisque c'est bien la page que vous aviez demandée initialement ?

Pour mettre en place cela, il nous faut d'abord modifier légèrement la méthode `signedInOnly` :

```
signedInOnly = (render, withRedirect = true) => props => {
 const { user, isLoadingUser } = this.state
 if (isLoadingUser) {
   return <p>Loading...</p>
 }
 if (user) {
   return render(props)
 }
 return (
   <Redirect
     to={{
       pathname: '/signin',
       state: withRedirect ? { from: props.location } : null
     }}
   />
 )
}
```

Par défaut, la méthode gère maintenant la redirection vers la route demandée après la connexion. Nous laissons néanmoins la possibilité de ne pas faire cette redirection ; en effet, si nous l'activons pour la route `/signout`, cela provoque un comportement gênant : l'utilisateur est immédiatement déconnecté après s'être connecté.

Pour gérer la redirection, tout ce que nous faisons ici consiste à mettre l'URL de la page demandée (`props.location`) dans l'attribut state de l'objet passé à `Redirect`. Ce state sera ensuite accessible dans la suite du routage, c'est-à-dire, pour nous, dans le composant `SignInForm`.

Pour y avoir accès dans `SignInForm`, nous devons faire en sorte que celui-ci reçoive en paramètre les informations sur le routage. Pour cela, il suffit d'appeler `withRouter` au moment d'exporter le composant en bas du fichier :

```
// src/components/SignInForm.js
// ...
import { withRouter } from 'react-router'

// ...
export default withRouter(SignInForm)
```

Ainsi, le composant `SignInForm` recevra toujours en propriété les informations sur le routage, et notamment les objets `location` et `history` qui nous permettront d'effectuer la redirection si l'authentification a réussi, c'est-à-dire dans la fonction passée à `onSubmit` :

```
try {
 await firebase.auth()
   .signInWithEmailAndPassword(email, password)
 setSubmitting(false)

 const { location, history } = this.props
 if (location.state) {
   const { from } = location.state
   if (from) {
     history.push(from)
   }
 }
} catch (err) {
// ...
```

Le routage est réellement complet cette fois-ci, du moins pour notre application temporaire. Il ne reste qu'à transférer ce que nous avons fait ici vers le composant App, ce qui ne présente pas de difficulté. Il suffit :

- de fusionner les deux state et les méthodes `componentDidMount` ;
- de transférer toutes les méthodes de App2 à l'exception de `renderHome`, `renderPrivate` et `renderNotFound` ;
- d'adapter le routage de App pour y inclure les nouvelles routes `/signin`, `/signup` et `/signout`, et d'utiliser `signInOnly` sur les routes de gestion de dépenses (ce qui inclut la route /).

Je vous propose de suivre ces étapes pour finaliser cette version de l'application à titre d'exercice. La version finale est disponible avec les exemples téléchargeables accompagnant le livre.

C'en est terminé pour la gestion de l'authentification et de ce que cela implique sur le routage. Dans la prochaine et dernière section de ce chapitre, nous verrons comment utiliser Firebase pour stocker des données de l'application. En l'occurrence, les dépenses saisies par l'utilisateur ne seront plus simplement stockées dans le navigateur, mais bien dans Firebase, et surtout associées à son compte, ce qui lui permettra de les retrouver de n'importe où.

3. Persistance de données avec Firebase

Dans cette dernière section du chapitre, nous allons mettre en place la dernière brique de notre application pour la rendre complète : le stockage des dépenses dans Firebase, plutôt que dans le stockage local du navigateur. Ainsi l'utilisateur pourra retrouver ses dépenses depuis n'importe quel navigateur. De plus, il ne verra que ses propres dépenses, et non celles des autres utilisateurs.

Pour cela, nous utiliserons la fonction *Database* de Firebase. Il s'agit d'une base de données accessible en passant par la bibliothèque que nous avons déjà utilisée dans la section précédente. Nous ne nous attacherons pas à comprendre en détail comment fonctionne cette base de données ni à décrire toutes ses fonctionnalités. Néanmoins il peut être utile de savoir certaines choses à son sujet.

Tout d'abord, au sein d'une application Firebase, il n'y a qu'une seule base de données. Les données peuvent y être hétérogènes, et semblables à un objet JSON. Cela ressemble à ce que vous pouvez retrouver dans MongoDB, à la différence qu'il n'y a pas de notion de collection. Ou bien imaginez qu'il n'y a qu'une seule collection.

Les données sont organisées de manière hiérarchique. Pour notre application, voici un exemple de que nous pourrons retrouver :

```
{
 "users": {
   "a8G0FIKc1KgW1HkH6TjH8W9Tjjg2": {
     "expenses": {
       "-LOuERjsbDYssMPnRaTe": {
         "amount": 30,
         "date": "2018-10-16",
         "notes": "Ceci est la première dépense.",
         "title": "Dépense 1"
       }
     }
   }
 }
}
```

Dans cette base de données, nous avons un utilisateur dont l'ID est `a 8G0...` (Firebase utilise la terminologie *uid* pour *user ID*), et celui-ci dispose d'une dépense, dont l'ID est `-LOuERjsbDYssMPnRaTe`. Notez que les dépenses sont ici stockées par utilisateur, car comme nous le verrons plus loin cela nous permettra de restreindre l'accès aux dépenses de l'utilisateur qui les a créées.

Avant de commencer l'intégration de Firebase dans notre application, commençons par en configurer les accès. Dans Firebase, cela se fait en définissant des règles (*rules*) d'accès aux données. Pour cela, rendez-vous dans la console de votre application Firebase (voir la section Création d'une application Firebase en annexe du livre), et dans la section **Développer - Database**. Ouvrez ensuite l'onglet **Règles**. Par défaut, en fonction de ce que vous avez configuré à la création de la base de données, l'ensemble des données est protégé en lecture comme en écriture.

Règles d'accès à la base de données

Pour chacune des ressources, Firebase nous permet de définir sous quelles conditions il est possible de lire ou écrire des données. Pour notre cas, nos règles seront simples, puisque nous nous contenterons de limiter l'accès aux ressources `users/USER_ID/*` à l'utilisateur d'ID « `USER_ID` » :

```
{
 "rules": {
   "users": {
     "$user_id": {
       ".read": "auth.uid === $user_id",
       ".write": true
     }
   }
 }
}
```

Encore une fois, l'idée ici n'est pas de comprendre le détail de ces règles (vous trouverez beaucoup d'explications dans la documentation de Firebase). Enregistrez ces règles (bouton **Publier**), puis passons à ce qui nous intéresse : l'utilisation de cette base de données dans notre application.

Comme vous allez le voir, les modifications ne seront pas bien complexes. Commençons par activer l'accès à la base de données dans l'initialisation de Firebase (fichier firebase.js) :

```
// src/firebase.js
import firebase from 'firebase/app'
import 'firebase/auth'
import 'firebase/database'
// ...
```

Les accès à Firebase se feront ensuite dans le composant App. Dès que l'utilisateur se connecte (ou au chargement de l'application s'il est déjà connecté), nous allons souscrire aux mises à jour de ses dépenses. Cela signifie que nous allons définir un traitement à effectuer dès qu'une dépense est créée ou mise à jour.

Pour cela, commençons par créer une méthode `subscribeToExpenses`:

```
subscribeToExpenses() {
 const uid = this.state.user.uid
 this.expensesRef = firebase.database()
   .ref(`users/${uid}/expenses`)
 this.expensesRef.on('value', snapshot => {
   const expensesById = snapshot.val() || {}
   const expenses = Object.entries(expensesById)
     .map(([id, expense]) => ({
       id,
       ...expense
     }))
   this.setState({ expenses })
 })
}
```

Pour souscrire à des évènements dans Firebase, il est nécessaire tout d'abord de définir ce que Firebase appelle une *ref*. Il s'agit d'une référence vers une partie de la base de données, en l'occurrence ici `users/${id}/expenses`, où `${id}` est l'ID de l'utilisateur connecté.

Lorsque la valeur de cette référence est modifiée, nous recevons alors un évènement auquel nous nous abonnons via `this.expensesRef.on('value', ...`. Cela se produira ici dès qu'une dépense sera créée, modifiée, supprimée, réordonnée, etc.

Bien évidemment si notre application devait gérer beaucoup de dépenses et que nous souhaitions l'optimiser, il existe des évènements plus précis auxquels nous abonner (un pour l'ajout, un autre pour la création, etc.).

Dans l'évènement, nous recevons un *snapshot*, nous permettant, via sa méthode `val`, d'obtenir sa valeur. Pour nous ce sera donc l'ensemble des dépenses. Comme Firebase stocke les ensembles d'éléments comme des objets, où les clés sont les ID, nous passons par une petite étape de transformation pour créer le tableau `expenses`, correspondant aux valeurs de `expensesById` où chaque dépense contient également un attribut `id` avec son ID.

Une fois ce tableau créé, nous le plaçons dans l'attribut `expenses` du state. Et c'est dorénavant le seul endroit où nous allons avoir besoin de mettre à jour cet élément. En effet, lorsque nous créerons ou modifierons une dépense, celle-ci sera créée ou modifiée dans Firebase, puis l'évènement `value` sera reçu, mettant ainsi à jour le state. Cela nous permet de simplifier considérablement nos méthodes `createExpense` et `updateExpense` :

```
createExpense = async expenseInfos => {
 const uid = this.state.user.uid
 const expenseRef = firebase
   .database()
   .ref(`users/${uid}/expenses`)
   .push()
 await expenseRef.set(expenseInfos)
}

updateExpense = async expenseInfos => {
 const uid = this.state.user.uid
 const { id, ...expense } = expenseInfos
 const expenseRef = firebase.database()
   .ref(`users/${uid}/expenses/${id}`)
 await expenseRef.set(expense)
}
```

Notez tout d'abord dans `createExpense` l'appel à `.push` sur une référence, permettant d'ajouter un nouvel élément à une collection (`users/${uid}/expenses`), et renvoyant une référence vers le nouvel élément. Dans `updateExpense`, on récupère directement la référence : `users/${uid}/expenses/${id}`. Dans les deux cas, nous faisons appel à `.set` pour mettre à jour la référence de la dépense.

Il ne nous reste qu'à appeler notre méthode `subscribeToExpenses` lorsque l'utilisateur se connecte, et à annuler la souscription lorsqu'il se déconnecte ainsi que lorsque le composant est détruit :

```
componentDidMount() {
 this.unsubscribeAuth = firebase.auth().onAuthStateChanged(
   user => {
     this.setState({ user, isLoadingUser: false })
     if (user) {
       this.subscribeToExpenses()
     } else {
       this.setState({ expenses: {} })
       this.unsubscribeToExpenses()
     }
   }
 )
}

componentWillUnmount() {
 if (this.unsubscribeAuth) {
   this.unsubscribeAuth()
 }
 this.unsubscribeToExpenses()
}

unsubscribeToExpenses() {
 if (this.expensesRef) {
   this.expensesRef.off()
   this.expensesRef = null
 }
}
```

Notre application est bel et bien complète cette fois-ci. Vous pouvez tester la création de dépenses avec plusieurs utilisateurs et visualiser ces dépenses dans la console de Firebase. Mieux encore, vous pouvez ouvrir deux navigateurs avec le même utilisateur, et constater que la création ou la modification d'une dépense d'un côté apparaît immédiatement de l'autre ! C'est l'un des aspects intéressants de Firebase : la mise à jour des données se fait en temps réel (d'où l'appellation *realtime database*).

Dans cette section, nous avons eu un premier aperçu de ce que permet Firebase en termes de stockage de données. Encore une fois, l'objectif n'est pas de vous convaincre que Firebase est une solution à tous les problèmes. On lui reproche notamment son côté fermé, en ceci que le système de règles n'est pas très souple, et surtout qu'une application écrite pour utiliser Firebase ne peut pas facilement s'adapter pour utiliser un autre back-end.

L'idée était plutôt d'utiliser Firebase comme prétexte pour vous montrer comment une application peut faire appel à un service distant, pour y stocker des données, pour s'authentifier, etc. J'ai choisi Firebase car il est simple à mettre en œuvre et ne nécessite pas de récupérer et héberger un autre projet back-end.

Incontestablement, Firebase est un outil intéressant pour permettre à une application web ou mobile de stocker des données distantes, sans avoir à mettre en place une API complexe. Pensez par exemple à une application n'ayant besoin de stocker que les préférences de l'utilisateur ; est-il nécessaire de monter un serveur avec une API REST, en Node.js par exemple, tout cela pour accéder à une base de données contenant une seule collection ?

4. En conclusion

Faisons un petit résumé de ce que nous avons appris dans ce chapitre :

- Nous avons vu comment permettre à l'utilisateur de s'inscrire et s'identifier grâce à Firebase, et ce que cela impliquait sur le routage de l'application.
- Enfin, nous avons utilisé Firebase pour y stocker des données distantes, accessibles dès l'utilisateur connecté, quel que soit l'emplacement où il se trouve.

Les notions développées dans ces deux derniers chapitres vous rapprochent encore du stade où vous aurez tous les éléments en main pour créer des applications web ou mobiles autonomes. En fait, je pense même que vous avez déjà atteint ce stade ou, du moins, vous en être très proches. Pensez au nombre d'applications que vous pouvez commencer à développer grâce aux connaissances que vous avez acquises depuis le début de ce livre.

Vous savez créer des composants React et les ordonnancer au sein d'une application complexe, aidé de Redux ou non. Vous savez même le faire pour le web et le mobile. Et maintenant, vous savez permettre à un utilisateur de s'identifier, et de saisir des données, qui seront sauvegardées sur un service externe.

Autant dire que vous en savez beaucoup ! Si votre objectif est de pratiquer React dans le monde professionnel, sans aucun doute vous avez appris le nécessaire pour passer à l'action : répondre à une offre d'emploi, ou au moins créer une application mettant en œuvre ce que vous avez vu pour la valoriser lors d'une candidature.

Si nous avons utilisé Firebase comme API dans ce chapitre, il a un concurrent très intéressant qui est de plus en plus populaire : GraphQL. Il ne s'agit pas d'un service à proprement parler, mais d'un standard pour envoyer des requêtes à un serveur pour lire ou écrire des données. Et surtout : il se marie très bien à React. Voyons cela dans le prochain chapitre.

Chapitre 9
Connecter React à une API GraphQL

1. Présentation de GraphQL et premières requêtes

Il y a quelques années, pour appeler une API distante, la référence en termes de moyen technique pour y parvenir était SOAP, avec lequel on encapsulait requêtes et résultats dans de verbeux flux XML. Puis est arrivé REST qui redonnait sa place aux basiques de HTTP et profitait du JSON pour décrire les échanges. Beaucoup plus simple à appréhender que SOAP, les développeurs web notamment n'ont pas mis longtemps à l'adopter.

Aujourd'hui, un nouveau moyen de requêter une API fait parler de lui : GraphQL (http://www.graphql.com/). Explorons ses avantages dans ce chapitre en développant une application React faisant appel à une API GraphQL pour persister des données.

1.1 Qu'est-ce que GraphQL ?

Pour faire simple, GraphQL est une convention pour échanger des données entre un client (un front-end React par exemple) et un serveur (Node.js, PHP...).

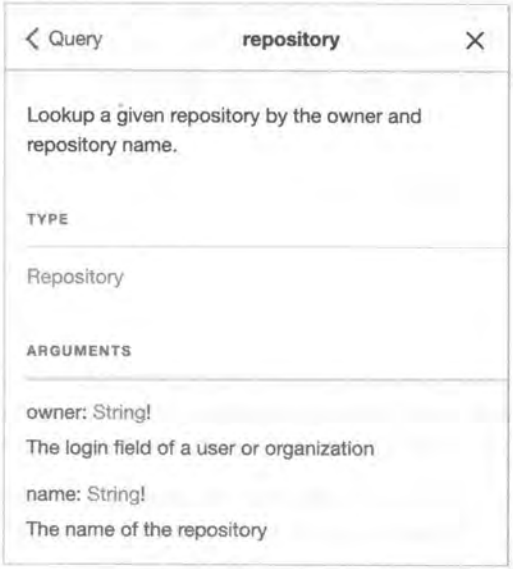
À la base d'une API GraphQL, on trouve donc :

- la description des objets pouvant être retournés par une requête : par exemple, un utilisateur comporterait un ID numérique, un nom, une date de naissance, tandis qu'un article aurait comme champs un ID, un titre, un contenu textuel, et une référence vers l'utilisateur l'ayant créé ;
- la description des méthodes qui peuvent être appelées sur l'API, avec les paramètres d'entrée de ces méthodes, et la description de ce qu'elles renvoient. Elles-mêmes séparées en deux types :
  - les *queries* qui permettent de lire des données ;
  - les *mutations* pour créer, mettre à jour ou supprimer des données.

Afin de mieux comprendre ces notions, allons dès maintenant jouer avec une API GraphQL : celle de GitHub.

1.2 Premières requêtes avec l'API de GitHub

GitHub fournit un outil en ligne pour requêter directement son API GraphQL : son GraphQL API Explorer (https://developer.github.com/v4/explorer/). Sur celui-ci, en haut à droite, cliquez sur le bouton **Docs** pour ouvrir **Documentation Explorer**. Pour notre première requête, nous souhaitons récupérer des informations sur le *repository* GitHub de React. Il s'agit donc d'une requête en lecture (une *query*), cliquons donc sur **Query** pour obtenir la liste des *queries* disponibles, puis descendons jusqu'à la ***query repository*** et cliquons dessus pour avoir les détails.



Documentation de la query « repository »

On apprend ici :

- que la *query* attend deux paramètres : un *owner* et un *name*, tous deux de type chaîne de caractères (`String`), et tous deux requis (ceci est indiqué par le point d'exclamation après le type : `!`) ;
- qu'elle renvoie un objet de type `Repository`. En cliquant sur ce type, vous pourrez découvrir les nombreux champs qu'il contient.

Notez que cette documentation n'est pas juste un bout de page HTML statique, elle est en réalité générée automatiquement par l'API GraphQL. Elle est donc nécessairement toujours à jour avec l'implémentation de l'API. En quelque sorte, on peut dire que la documentation fait partie intégrante de l'API elle-même.

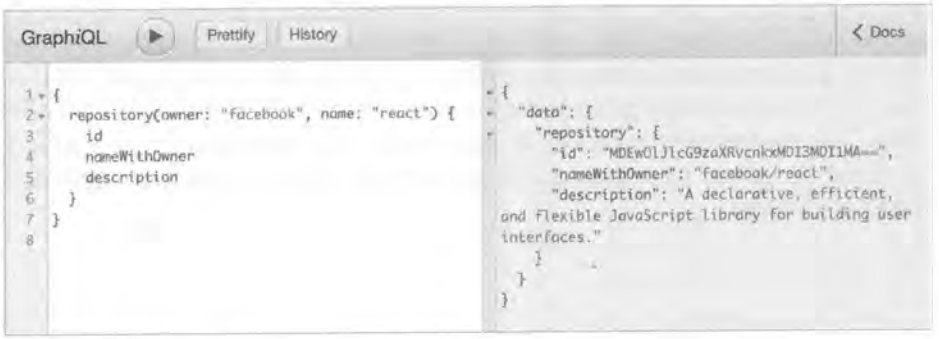
Nous souhaitons obtenir les informations sur le repository de React. Écrivons donc la requête (je vous suggère de ne pas la copier-coller, mais plutôt de l'écrire directement dans l'éditeur afin de contempler l'autocomplétion rendue disponible par GraphQL) :

```
{
 repository(owner: "facebook", name: "react") {
   id
   nameWithOwner
   description
 }
}
```

Ici, nous appelons la *query* `repository`, en lui passant les paramètres `owner` et `name`. Du résultat de cette requête, nous souhaitons obtenir les champs `id`, `nameWithOwner` et `description`. Que remarque-t-on de cette requête ?

- D'abord, la syntaxe est propre à GraphQL, même si on note des ressemblances avec JSON ou JavaScript. Elle est relativement simple à prendre en main, nous aurons l'occasion de voir plusieurs exemples par la suite.
- Ensuite, nous fournissons les données que nous souhaitons recevoir de cette requête. En effet, il n'est pas possible de dire simplement que l'on souhaite tous les champs disponibles. Cela peut sembler être une contrainte au premier abord, mais cela garantit que vous aurez les champs attendus, ni plus ni moins, même si l'API est mise à jour.

En exécutant cette requête, on obtient un résultat au format JSON :



Résultat de la requête

Vous verrez que les résultats sont toujours au format suivant : un objet comportant un attribut `data`, étant lui-même un objet dont les clés sont les noms des *queries* appelées (on peut en effet appeler plusieurs *queries* dans un même appel).

1.3 Ajout de données liées à la requête

Ajoutons quelques données à notre requête ; nous souhaitons ajouter aux informations du repository la liste de ses *issues*. Pour cela l'API de GraphQL propose le champ `issu` sur le type `Repository`. Mais petite particularité de l'API GitHub : il ne s'agit pas d'un tableau, mais d'un objet contenant notamment un attribut `nodes`, lui de type tableau.

```
{
 repository(owner: "facebook", name: "react") {
   id
   nameWithOwner
   description
   issues(last: 10) {
     nodes {
       number
       title
     }
   }
 }
}
```

Notez aussi que l'on peut passer des paramètres lorsqu'on récupère des données liées à un objet, comme les *issues*. En réalité, l'API de GitHub nous impose de spécifier le dernier élément à récupérer (paramètre `last`), sans quoi une erreur est retournée.

Requête avec les issues

Grâce à cette requête, nous obtenons les informations demandées sur le repository, avec les dix premières *issues*, et pour chacune, les données souhaitées uniquement : son numéro et son titre. Vous devriez commencer à voir les avantages du langage de requêtes GraphQL, même si cela peut paraître un brin magique...

1.4 Écrire des données

À présent que nous savons lire des données depuis l'API, intéressons-nous à la modification des données existantes au moyen des *mutations*. Dans la mesure où les données modifiées sont les « vraies » données de GitHub (ce n'est pas juste un bac à sable), soyons prudents et faisons une opération simple. Nous allons ajouter une étoile (star) au projet React.

Pour écrire une *mutation*, la syntaxe est similaire à une *query*, mais on placera le mot-clé `mutation` avant la première accolade (pour une *query* on peut mettre le mot-clé `query` mais ce n'est pas obligatoire).

```
mutation {
 addStar(input: {
   starrableId: "MDEwOlJlcG9zaXRvcnkxMDI3MDI1MA=="
 }) {
   clientMutationId
```

```
    starrable {
      id
    }
  }
}
```

Dans notre cas la *mutation* appelée est `addStar`, et la documentation nous apprend qu'elle prend en paramètre un objet `input` de type `AddStarInput`, lui-même comportant un attribut obligatoire `starrableId`, correspondant à l'ID de la ressource (ici un repository) sur laquelle on souhaite ajouter une étoile.

Comme pour une *query*, une *mutation* renvoie elle-même des données. Il est donc nécessaire de spécifier quelles données nous souhaitons récupérer, ici `clientMutationId` et l'attribut `id` de `starrable`.

Après avoir exécuté cette *mutation*, vous pourrez constater que vous venez d'ajouter une étoile au projet React. Pour valider à nouveau, vous pourrez remplacer `addStar` par `removeStar` dans la requête.

1.5 Prochaines étapes

Maintenant que nous avons vu comment faire des *queries* et des *mutations* sur une API GraphQL, vous devriez commencer à sentir la puissance de ce langage de requêtes. Mais d'un autre côté, il est possible que vous vous demandiez, parmi les fonctionnalités que nous venons de voir, lesquelles sont assurées par GraphQL, et lesquelles le sont par l'API GitHub spécifiquement.

En effet, en utilisant l'API de GitHub, on pourrait penser que GraphQL prend à sa charge des fonctionnalités comme le filtrage, la pagination, ou encore les jointures entre différents objets. En réalité, il n'en est rien, GraphQL n'est ni plus ni moins que le langage permettant d'effectuer les requêtes, et de décrire ce qu'elles prennent en paramètres et renvoient en résultat. Tout le reste est bien de la responsabilité du serveur qui fournit l'API.

Afin de mieux comprendre cela, nous allons dans la prochaine section voir comment créer notre propre API GraphQL pour stocker des données sur un serveur, le tout sans écrire la moindre ligne de code serveur.

Cela nous permettra également d'aller un peu plus loin dans les possibilités offertes par les requêtes et les *mutations*.

2. Création d'une API avec Graphcool

Graphcool (https://www.graph.cool/) est un service proposant de créer une API GraphQL sans écrire la moindre ligne de code. En effet comme nous l'avons vu dans la première section de ce chapitre, GraphQL n'est qu'un langage, la responsabilité de récupérer les données, de les filtrer, de faire les jointures, etc. reste celle du serveur (et donc du développeur).

Graphcool permet de réaliser tout cela de manière automatique, ce qui est idéal pour débuter avec GraphQL, surtout quand le but est comme ici d'aboutir le plus rapidement possible à un front-end travaillant avec l'API.

2.1 Installation et création du projet Graphcool

Pour le développement, Graphcool se présente comme un outil en ligne de commande que vous pouvez installer avec NPM : `npm install -g graphcool`. Il propose un moyen d'initialiser rapidement un projet, mais pour faire encore plus simple, créons les fichiers nécessaires à la main avec le strict minimum.

Tout d'abord le fichier *package.json* :

```
{
 "name": "hello-graphcool",
 "version": "0.1.0"
}
```

Puis le fichier graphcool.yml, manifeste qui indique comment Graphcool va devoir gérer notre application :

```
# Chemin vers le fichier de définition des types
# disponibles dans notre API
types: ./types.graphql
# Permissions: par défaut tout est accessible à tout le monde
permissions:
 - operation: '*'
```

Et enfin le plus intéressant, le fichier *types.graphql*, qui permet de définir les types qui seront disponibles dans notre API:

```
type User @model {
 id: ID! @isUnique
 name: String
}
```

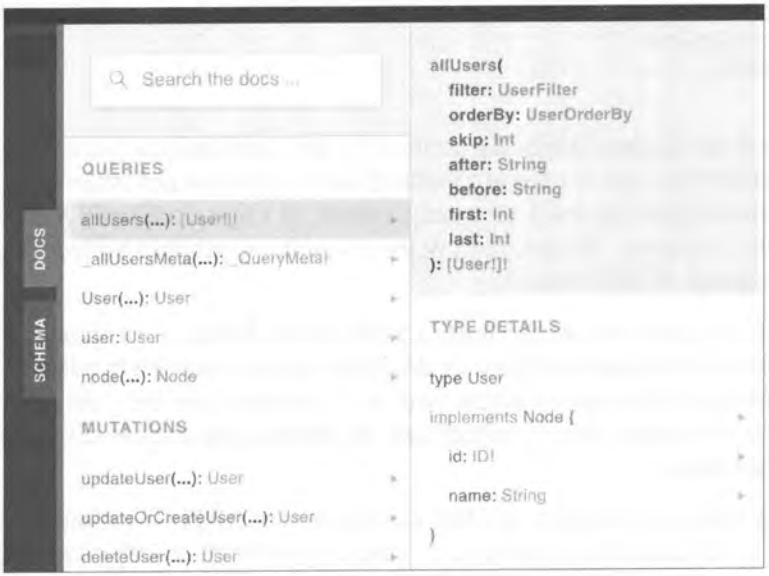
Il s'agit bien d'un fichier GraphQL, contenant des annotations permettant d'ajouter des informations sur des attributs (`@isUnique`) ou des types (`@model`). Une surcouche à GraphQL en quelque sorte : il s'agit du *GraphQL Schema Definition Language* (https://www.prisma.io/blog/graphql-sdl-schema-definition-language-6755bcb9ce51).

La magie de Graphcool est de parvenir, à partir de ce fichier, à proposer une API permettant d'interroger une base de données reproduisant les modèles décrits. Ici, il est probable que quelque part sera générée une base disposant d'une table ou collection `User` permettant de stocker des entités disposant d'un ID et d'un nom.

Voyons dès à présent comment générer et utiliser cette API. Une fois les fichiers créés, la commande `graphcool deploy` permet de déployer chez Graphcool une instance de notre API. Quelques questions vous seront posées quant à l'emplacement où déployer l'API, les choix par défaut devraient très bien convenir.

Une fois la commande exécutée, Graphcool vous indique les URL de l'API, mais pour pouvoir jouer avec, le plus simple est de lancer la commande `graphcool playground`, qui ouvrira une interface permettant de faire des requêtes, similaire à celle que nous avons vue pour l'API GitHub dans la première section.

Dans le playground, commençons par ouvrir la documentation (onglet **Docs** dans le panneau latéral à droite de la page).



Documentation de notre API

Vous pouvez constater que Graphcool a généré énormément de choses à partir de notre simple modèle `User` et de ses deux attributs `id` et `name` :

- Des *queries* : `allUsers` pour récupérer plusieurs utilisateurs, `user` pour en récupérer un seul...
- Des *mutations* : `createUser`, `updateUser`, `deleteUser`...
- Des *subscriptions*, pour être averti des mises à jour des utilisateurs, etc.

Commençons par créer un utilisateur grâce à la *mutation* `createUser` :

```
mutation {
 createUser(name: "Sébastien") {
   id
   name
 }
}
```

Après cela, vérifions qu'il nous est bien renvoyé si nous appelons la *query* `allUsers` :

```
{
 allUsers {
   id
   name
 }
}
```

Félicitations, vous venez de créer votre première API GraphQL ! Voyons maintenant comment ajouter d'autres modèles, ainsi que des relations entre eux.

2.2 Ajout de nouveaux modèles et relations

Notre but sera ici de créer une API permettant de stocker des données pour un service similaire à Instagram, en beaucoup plus minimaliste. Les utilisateurs peuvent poster des photos (en fournissant directement l'URL), et d'autres utilisateurs peuvent y ajouter des commentaires. Nous réutiliserons notre API dans la prochaine section, lorsque nous créerons le front-end de notre application avec React.

Notre application devra donc manipuler trois types d'entités : les utilisateurs (que nous avons déjà vus), les posts, et les commentaires. Commençons par créer le type `Post` dans notre fichier types.graphql :

```
type Post @model {
 id: ID! @isUnique
 createdAt: DateTime!

 author: User! @relation(name: "UserPosts")
 imageUrl: String!
}
```

Par rapport au type `User` que nous avions vu, vous pouvez remarquer :

- que nous avons ici un champ `createdAt`, qui est en lecture seule et est géré par Graphcool (nous ne pouvons que l'utiliser dans les *queries*, pas le modifier) ;
- qu'un attribut `author` permet de faire une liaison avec le modèle `User`.

La relation avec l'utilisateur est de type « 1:n », c'est-à-dire qu'à un post correspond un utilisateur, mais qu'un utilisateur peut avoir plusieurs posts. Pour réaliser la liaison dans l'autre sens, ajoutons un attribut `posts` au modèle `User` :

```
type User @model {
 # ...
 posts: [Post!]! @relation(name: "UserPosts")
}
```

Le nom de la relation `UserPosts` permet d'indiquer à Graphcool la correspondance entre les deux attributs dans `User` et `Post`. En effet rien ne nous empêcherait d'avoir plusieurs relations entre ces deux modèles.

De la même manière, créons un nouveau type `Comment` pour définir les commentaires :

```
type Comment @model {
 id: ID! @isUnique
 createdAt: DateTime!
 content: String!

 post: Post! @relation(name: "PostComments")
 author: User! @relation(name: "UserComments")
}
```

Nous avons ici deux relations, l'une permettant de définir le post associé au commentaire, et l'autre permettant d'y associer un auteur. À titre d'exercice, vous pouvez ajouter les attributs nécessaires pour ces deux relations dans `Post` et `User`. Voici l'intégralité du fichier types.graphql en version définitive :

```
type User @model {
 id: ID! @isUnique
 name: String

 posts: [Post!]! @relation(name: "UserPosts")
 comments: [Comment!]! @relation(name: "UserComments")
}

type Post @model {
 id: ID! @isUnique
 createdAt: DateTime!
```

```
  author: User! @relation(name: "UserPosts")
  imageUrl: String!

  comments: [Comment!]! @relation(name: "PostComments")
}

type Comment @model {
  id: ID! @isUnique
  createdAt: DateTime!
  content: String!

  post: Post! @relation(name: "PostComments")
  author: User! @relation(name: "UserComments")
}
```

Après avoir redéployé notre API sur Graphcool (`graphcool deploy`), nous pouvons créer des utilisateurs, des posts et des commentaires.

Par exemple pour créer un nouveau post avec la *mutation* `createPost` (pensez à remplacer l'ID de l'utilisateur par celui d'un utilisateur que vous avez créé) :

```
mutation {
  createPost(
    authorId: "cjur4185307sq0106c5uy3wyh"
    imageUrl:
      "https://images.unsplash.com/photo-1555706195-38f133d1419f"
  ) {
    id
    author {
      id
      name
    }
    imageUrl
  }
}
```

Pour créer un commentaire, même principe avec la *mutation* `createComment` :

```
mutation {
 createComment(
   postId: "cjur4y17m0asx01234iwn30rv"
   authorId: "cjur4wgyl0b94014663afkelp"
   content: "Great picture!"
 ) {
   id
   createdAt
   content
 }
}
```

Après avoir créé quelques utilisateurs, posts et commentaires, nous pouvons par exemple écrire une requête permettant de récupérer :

- les posts avec les URL des images, par ordre décroissant de date de création,
- avec pour chaque post le nom de son auteur,
- ainsi que les commentaires associés, par ordre croissant de date de création, et avec le nom de leur auteur également.

Avec une API REST, cette requête nécessiterait généralement plusieurs appels réseau, à moins de fournir une route permettant d'exécuter spécifiquement cette requête. Avec GraphQL, c'est le client (le front-end) qui peut choisir exactement quelles données récupérer.

Voici notre requête finale, très proche de celle que nous utiliserons dans la suite du chapitre pour afficher dans notre application React les images postées :

```
{
 allPosts(orderBy: createdAt_DESC) {
   id
   createdAt
   imageUrl
   author {
     id
     name
   }
   comments(orderBy: createdAt_ASC) {
```

```
      id
      createdAt
      author {
        id
        name
      }
      content
    }
  }
}
```

Remarquez comme il est facile de comprendre ce que renvoie cette requête sans être expert du GraphQL. Avec un peu d'habitude, les requêtes GraphQL ne sont pas beaucoup plus difficiles à écrire qu'à lire.

Mais maintenant que nous avons créé notre API GraphQL et que nous savons dialoguer avec elle par les *queries* et *mutations*, nous n'avons pourtant vu qu'une partie de la magie qu'apporte GraphQL. Il nous reste à appeler cette API dans une application React, et encore une fois des outils vont rendre ceci extrêmement simple.

3. Appel d'une API avec React et Apollo Client

À présent que notre API est prête, voyons comment l'appeler, et notamment depuis une application React. Une API GraphQL repose sur une API REST classique, donc il serait bien évidemment possible de l'appeler grâce aux fonctions classiques de requêtes HTTP, par exemple la fonction `fetch`. Ce serait cependant passer à côté de fonctionnalités très pratiques offertes par GraphQL.

Il existe plusieurs bibliothèques tierces permettant d'appeler une API GraphQL ; la plus connue et la plus réputée dans l'écosystème est probablement Apollo (https://apollographql.com), c'est donc celle-ci que nous allons découvrir ensemble. Apollo permet de réaliser l'intégralité d'une application client-serveur échangeant grâce à GraphQL, mais nous n'allons utiliser que la partie cliente, sobrement appelée Apollo Client (que j'appellerai simplement Apollo dans la suite du chapitre).

Voilà à quoi notre application finale ressemblera

Créons dès maintenant notre projet React, de la même manière que les autres projets, en y ajoutant les dépendances d'Apollo :

```
yarn init -y
yarn add -D parcel-bundler
yarn add react react-dom apollo-boost @apollo/react-hooks graphql
```

3.1 Lire des données en envoyant des queries

Pour le moment laissons de côté React pour nous concentrer sur l'appel de notre API, dans le fichier src/index.js. Commençons par créer un objet client qui nous permettra d'effectuer des *queries* et des *mutations*. La création de celui-ci nécessite bien entendu l'URI de l'API. Lorsque vous déployez votre API Graphcool via la commande `graphcool deploy`, l'URL vous est donnée ; il s'agit de l'URL désignée par « Simple API ».

```
import ApolloClient, { gql } from 'apollo-boost'

const client = new ApolloClient({
 uri: 'https://api.graph.cool/simple/v1/...',
})
```

Pour effectuer une requête, créons ensuite un objet `query` en appelant la fonction `gql`. Notez la syntaxe particulière, nous utilisons une *template string* pour appeler la fonction : `const query = gql`...`` . Récupérons la requête de la section précédente afin de récupérer tous les posts, avec leur auteur et leurs commentaires :

```
const query = gql`
 {
   allPosts(orderBy: createdAt_DESC) {
     id
     createdAt
     imageUrl
     author {
       id
       name
     }
     comments(orderBy: createdAt_ASC) {
       id
       createdAt
       author {
         id
         name
       }
       content
     }
   }
 }
`
```

Il ne nous reste plus qu'à utiliser le client pour exécuter la requête, et cela se fait de la manière la plus intuitive qui soit : en appelant sa méthode `query`, ce qui renvoie sous forme d'une promesse le résultat de la requête :

```
client.query({ query }).then(result => console.log(result))
```

Vous devriez voir dans votre console un résultat similaire à celui-ci :


```
{
 "data": {
   "allPosts": [
     {
       "id": "ck0l1dlqk04ta0186q6utxerf",
       "createdAt": "2019-09-15T13:49:45.000Z",
       "imageUrl": "https://images.unsplash.com/photo-
1555706195-38f133d1419f",
       "author": {
         "id": "ck0l1cxab04rj0186wh5l8sq4",
         "name": "Sébastien",
         "__typename": "User"
       },
       "comments": [
         {
           "id": "ck0l1e3wd04r90138jduombg9",
           "createdAt": "2019-09-15T13:50:09.000Z",
           "author": {
             "id": "ck0l1cxab04rj0186wh5l8sq4",
             "name": "Sébastien",
             "__typename": "User"
           },
           "content": "Great picture!",
           "__typename": "Comment"
         }
       ],
       "__typename": "Post"
     }
   ]
 },
 "loading": false,
 "networkStatus": 7,
 "stale": false
}
```

Vous observerez que c'est très similaire à ce que nous obtenions dans le playground, ce qui n'est bien évidemment pas un hasard. Nous pourrions développer notre application React en appelant les requêtes ainsi, mais Apollo vient avec des outils qui permettent de faire encore beaucoup mieux, en associant une ou plusieurs requêtes spécifiques à un composant chargé d'en afficher le résultat. Voyons cela tout de suite.

Créons un composant `PostsList`, qui sera responsable comme son nom l'indique de l'affichage d'une liste de posts. Apollo met à notre disposition un *hook* `useQuery` nous permettant d'exécuter une requête GraphQL. La requête sera exécutée au premier rendu du composant, et le composant sera à nouveau rendu lorsque l'état de la requête changera. Le hook `useQuery` renvoie plusieurs valeurs qui nous intéressent :

- `loading`, un booléen à `true` si la requête est en cours d'exécution.
- `error`, qui contient l'erreur éventuelle renvoyée par la requête.
- `data`, qui contient le résultat de la requête s'il n'y a pas eu d'erreur.

Nous avons donc la possibilité dans notre composant de gérer trois situations : le cas où la requête est en cours d'exécution, le cas d'erreur, et le cas où le résultat est prêt à être affiché.

Cela nous donne le composant suivant :

```
import React from 'react'
import { useQuery } from '@apollo/react-hooks'

// ...

const PostsList = () => {
 const { loading, error, data } = useQuery(postsQuery)

 if (loading) {
   return <p>Loading...</p>
 }

 if (error) {
   return <p>Something bad happened.</p>
 }

 return (
```

```
    <div className="posts-list">
      {data.allPosts.map(post => (
        ...
      ))}
    </div>
  )
}
```

La requête `postsQuery` doit être déclarée préalablement, par exemple au-dessus du composant. Pour notre liste de posts, nous n'allons pas avoir besoin de récupérer l'ensemble des commentaires de chaque post, leur nombre suffira :

```
// ...
import gql from 'graphql-tag'

const postsQuery = gql`
 {
   allPosts(orderBy: createdAt_DESC) {
     id
     createdAt
     imageUrl
     author {
       id
       name
     }
     comments {
       id
     }
   }
 }
`
```

Nous disposons à présent de l'ensemble des éléments nécessaires pour afficher notre liste, avec les images associées à chaque post :

```
return (
 <div className="posts-list">
   {data.allPosts.map(post => (
     <div className="post" key={post.id}>
       <img className="post-image" src={post.imageUrl} />
       <div className="post-footer">
         <div>
           <span>?{post.author.name}</span>
```

```
            <span>?{new Date(post.createdAt)
              .toLocaleDateString()}</span>
          </div>
          <span>?{post.comments.length}</span>
        </div>
      </div>
    ))}
  </div>
)
```

Notez qu'Apollo nous fournit également une fonction très utile permettant de ré-exécuter une requête à intervalles réguliers. Cela nous permettra d'afficher automatiquement les nouveaux posts au fur et à mesure de leur création.

```
const { loading, error, data, startPolling }
 = useQuery(postsQuery)

useEffect(() => {
 startPolling(5000)
})
```

Remarque

Il existe également un système en GraphQL permettant de s'abonner à certains évènements en temps réel : les Subscriptions. Leur mise en œuvre côté client se révèle cependant plus complexe, car ils nécessitent la configuration de bibliothèques de gestion de websockets. La documentation d'Apollo stipulant elle-même de ne les utiliser que si cela est nécessaire étant donné les besoins en termes de temps réel, nous nous contenterons de vérifier toutes les 5 secondes la présence de nouveaux posts.

Maintenant que nous sommes capables de lire des données depuis l'API GraphQL grâce aux *queries*, il nous reste à voir comment en écrire grâce aux *mutations*. Mais avant d'envisager d'écrire des données, il est nécessaire d'aborder un point : l'authentification. En effet nulle API digne de ce nom ne laisserait un utilisateur non autorisé écrire des données.

3.2 Gestion de l'authentification

GraphQL ne propose pas de mécanisme d'authentification à proprement parler. Dans la mesure où il s'agit toujours d'une API REST en premier lieu, les mêmes mécanismes peuvent être utilisés. Graphcool non plus n'offre pas de surcouche pour l'authentification aux API GraphQL qu'il permet de générer. Il nous faut donc trouver un autre moyen.

La mise en place d'un système d'authentification côté serveur sort du cadre de ce livre, c'est pourquoi je ne m'attarderai pas sur ce point. Dans la version finale de notre application, que vous trouverez dans les exemples téléchargeables accompagnant le livre, j'ai utilisé le service Auth0 (http://auth0.com) qui permet de créer en quelques clics une authentification reposant sur le protocole standard OpenID Connect (OIDC). Une bibliothèque est également proposée pour JavaScript côté client que nous n'avons qu'à intégrer dans notre application.

Il reste cependant à connecter Graphcool à Auth0 afin de permettre de s'authentifier auprès de l'API Graphcool après s'être authentifié auprès de Auth0. Autrement dit, le workflow d'authentification de notre application ressemblera à cela :

- À la première ouverture, puisqu'il n'est pas authentifié, l'utilisateur est redirigé vers la page de connexion d'Auth0 (qui lui permet également de s'inscrire).
- Après authentification, Auth0 redirige vers notre application, en fournissant un token en paramètre.
- Nous effectuons une nouvelle requête à Auth0 avec le token pour vérifier sa validité (pour vérifier que l'URL n'a pas simplement été appelée avec un faux token). Auth0 nous renvoie alors l'ID de l'utilisateur.
- Nous effectuons une requête auprès de Graphcool en envoyant l'ID Auth0 de l'utilisateur. S'il n'existe pas chez Graphcool, il est créé. Nous obtenons alors l'ID de l'utilisateur, ainsi qu'un nouveau token qu'il faut inclure en entête de chaque requête à Graphcool.

Graphcool permet également de configurer des permissions sur chaque type d'objet manipulé. Dans notre cas, nous souhaitons que seuls des utilisateurs authentifiés puissent créer de nouveaux posts et commentaires.

Cela se configure dans le fichier graphcool.yml de l'API :

```
permissions:  # Objets
 - operation: 'User.read'
   authenticated: true
 - operation: 'Post.read'
   authenticated: true
 - operation: 'Post.create'
   authenticated: true
 - operation: 'Comment.read'
   authenticated: true
 - operation: 'Comment.create'
   authenticated: true
 # Relations
 - operation: 'UserPosts.connect'
   authenticated: true
 - operation: 'PostComments.connect'
   authenticated: true
 - operation: 'UserComments.connect'
   authenticated: true
```

Il est également possible d'aller plus loin dans les permissions, notamment en faisant en sorte qu'un utilisateur ne puisse pas créer de post ou de commentaire au nom d'un autre utilisateur. Dans la mesure où cette configuration n'est pas propre à GraphQL et change selon l'API que l'on utilise, pour notre exemple le front-end React assurera cette responsabilité. Bien évidemment, dans une application en production, l'API doit elle-même faire ces vérifications.

La gestion de l'authentification pour notre application (workflow défini plus haut) sera gérée dans un composant `Authenticate`, que vous pourrez retrouver dans les exemples accompagnant le livre (il ne présente que peu d'intérêt pour comprendre les mécanismes associés à l'appel d'une API GraphQL). L'important à savoir est que lorsque l'utilisateur est authentifié, le token à passer à chaque requête GraphQL est inscrit dans le *session storage* du navigateur. À la création de notre client Apollo, il est donc possible de récupérer cette valeur et de l'ajouter aux en-têtes de chaque requête si elle est présente :

```
const client = new ApolloClient({
 uri: 'https://api.graph.cool/simple/v1/…',
 request: operation => {
   const token = sessionStorage.getItem('auth_token')
```

```
    if (token) {
      operation.setContext({
        headers: { Authorization: `Bearer ${token}` }
      })
    }
  },
})
```

3.3 Écrire des données à l'aide de mutations

Notre utilisateur est maintenant authentifié, nous pouvons donc lui permettre de créer de nouveaux posts et commentaires. Pour cela créons un composant `CreatePost` qui contient dans un premier temps un simple formulaire.

```
const CreatePost = ({ userId }) => {
 const [imageUrl, setImageUrl] = useState('')
 const onImageUrlChange = useCallback(event => {
   setImageUrl(event.target.value)
 })
 const onSubmit = useCallback(event => {
   event.preventDefault()
   // ...
   setImageUrl('')
 })

 return (
   <div className="create-post">
     <form onSubmit={onSubmit}>
       <label htmlFor="imageUrl">Image URL:</label>
       <input
         disabled={loading}
         required
         placeholder="http://..."
         id="imageUrl"
         type="text"
         onChange={onImageUrlChange}
         value={imageUrl}
       />
       <button disabled={loading} type="submit">
         Create
       </button>
```

```
      </form>
    </div>
  )
}
```

De la même manière que nous avions utilisé `useQuery` pour récupérer la liste des posts, Apollo nous permet d'utiliser `useMutation` pour écrire des données en appelant une *mutation*. Créons donc cette *mutation* :

```
const createPostMutation = gql`
 mutation createPost($authorId: ID!, $imageUrl: String!) {
   createPost(authorId: $authorId, imageUrl: $imageUrl) {
     id
   }
 }
`

const CreatePost = ({ userId }) => {
 const [createPost, { loading, error }] =
   useMutation(createPostMutation)
 // ...
```

Encore une fois, nous récupérons également des attributs `loading` et `error` qui nous permettent d'adapter l'affichage du composant pendant l'appel à la *mutation* ou s'il y a eu une erreur. Il ne nous reste qu'à appeler cette *mutation* lors de la soumission du formulaire :

```
const onSubmit = useCallback(event => {
 event.preventDefault()
 createPost({
   variables: { authorId: userId, imageUrl },
 })
 setImageUrl('')
})
```

Vous remarquerez que lorsque l'on crée la *mutation* ou lorsque nous l'appelons, nous n'insérons pas les valeurs pour les variables directement dans la requête (par une concaténation de chaînes de caractères par exemple). Les valeurs sont fournies dans l'objet passé en paramètre à la *mutation* `createPost`, dans l'attribut `variables`. En plus d'être plus simple à utiliser, ce mécanisme assure également les protections en termes d'injection de code, par exemple.

Affichons également un message de chargement ou, le cas échéant, d'erreur, en bas du formulaire :

```
        //...
      </form>
      {loading && <div>Creating...</div>}
      {error && <div>Something bad happened.</div>}
    </div>
  )
}
```

Notre composant de création de post est complet. Il ne reste qu'à l'ajouter à notre application. Un endroit approprié peut être dans le composant `App`.

Si vous examinez attentivement le composant `CreatePost`, vous remarquerez que celui-ci prend en paramètre l'ID l'utilisateur connecté. Cet ID est renvoyé par le composant `Authenticate` évoqué plus haut et est passé en cascade en propriété aux composants qui en ont besoin. Bien évidemment, rien ne nous empêcherait de stocker cet ID dans un store Redux par exemple.

Notre application permet à présent de visualiser la liste des posts, et d'en créer de nouveaux. Il reste à mettre en place les mêmes fonctionnalités pour les commentaires, ce que je vous laisse faire à titre d'exercice. L'ensemble du code final est bien entendu disponible dans les exemples accompagnant le livre.

La seule petite différence entre la récupération des posts et celle des commentaires et que la seconde nécessite l'envoi d'un paramètre, en l'occurrence l'ID du post dont on souhaite récupérer les commentaires. Pour cela, la *query* prendra un paramètre, que nous fournirons en second paramètre du hook `useQuery`.

```
const commentsQuery = gql`
 query comments($postId: ID!) {
   Post(id: $postId) {
     comments(orderBy: createdAt_DESC) {
       id
       author {
         email
         id
       }
       createdAt
       content
     }
```

```
    }
  }


const CommentsList = ({ userId, postId }) => {
  const { loading, error, data, refetch } =
    useQuery(commentsQuery, { variables: { postId } })
  // ...
```

Cela conclut cet exemple, et le chapitre consacré à GraphQL. N'hésitez pas à récupérer l'application complète (API et partie cliente en React) dans les exemples téléchargeables accompagnant le livre, et à la lancer localement. Vous aurez besoin de créer votre propre application Auth0. Cette étape est indiquée dans le fichier README accompagnant le code.

Sachez que nous n'avons qu'effleuré la surface des possibilités offertes par Apollo. Il propose également des mécanismes avancés pour gérer un cache local des résultats de requêtes. Poussé à l'extrême, ce cache peut même se substituer à Redux. En effet, une quantité importante de données que l'on stocke généralement dans le store de Redux correspond en réalité à des données que nous mettons en cache.

Comme vous l'avez vu, créer une application React appelant une API GraphQL est relativement simple. Il n'y a pas à se soucier du format de la requête à envoyer ni du format des résultats renvoyés. Si un jour vous avez l'occasion de réaliser une telle application avec un système de types (TypeScript ou Flow par exemple), Apollo propose même une commande permettant de se connecter au serveur GraphQL, récupérer les schémas proposés par l'API, et générer les définitions de type correspondantes, ce qui facilite encore le développement.

Si ces derniers chapitres se sont concentrés sur des notions pratiques, le suivant et dernier de ce livre reviendra sur des concepts plus théoriques. Vous y apprendrez comment créer des composants robustes en vue de les réutiliser, dans vos applications ou pour les diffuser à la communauté.

Chapitre 10
Écrire des composants réutilisables

1. Introduction

Les chapitres précédents de ce livre vous ont, je l'espère, permis de découvrir React d'une façon aussi pratique que possible. Le but était de ne pas surcharger votre apprentissage de plein de notions théoriques avant que vous ne soyez capables de créer une application par vous-même.

À présent que les bases de React sont acquises (si vous êtes arrivés jusqu'ici cela ne fait aucun doute), nous allons revenir sur la première notion que nous avons vue : les composants. Depuis que nous les avons découverts au premier chapitre, nous avons trouvé toute sorte de moyens de les faire communiquer entre eux et de les organiser au sein de l'application. À présent, voyons plus en détail comment les rendre plus robustes et réutilisables.

Nous commencerons par voir brièvement les grands principes d'écriture de composants, puis nous verrons ce que React et les années de pratique de la communauté nous offrent pour arriver à ces fins.

Ce chapitre sera un brin plus théorique que les précédents : pas de cas pratique construit au fil du chapitre, mais plutôt un ensemble de petits exemples associés à chaque notion développée. Ceci a pour conséquence que les sous-sections de ce chapitre peuvent être lues dans n'importe quel ordre en fonction de votre besoin.

2. Principes pour des composants réutilisables

Qu'entend-on par composants réutilisables ? Tout d'abord, il est important de préciser que le but n'est pas forcément de diffuser vos composants à la communauté, ni même de forcément les réutiliser dans un autre projet. Mais développer vos composants comme s'ils allaient être utilisés dans une autre application et par une autre personne les rendra forcément plus faciles à utiliser et à maintenir par vous-même. Ce n'est d'ailleurs pas propre aux composants React, cette pratique est courante pour écrire du code de qualité en général.

Quels sont donc les principes à respecter au maximum pour écrire des composants de qualité ? Ou plutôt : quelles sont les bonnes pratiques de développement qui sont particulièrement applicables aux composants ?

2.1 Des composants aussi simples que possible

Développer une application complexe pourra vous inciter à écrire des composants complexes, mais c'est pourtant dans ce cas qu'il sera plus pertinent d'écrire de petits composants, même si ce n'est pas pour les réutiliser.

Par exemple, un composant d'affichage de fiche contact pourrait se présenter ainsi dans une première version :

```
const ContactView = ({ contact, editGeneralInfos, editAddress })
=> (
 <div>
   <section>
     <h2>General info</h2>
     <p>First name: {contact.firstName}</p>
     <p>Last name: {contact.lastName}</p>
     {/* ... */}
     <button onClick={() => editGeneralInfos()}>Edit</button>
   </section>
   <section>
     <h2>Address</h2>
     <p>
       {contact.address.addressLineOne}
```

```
        <br />
        {contact.address.addressLineTwo}
        <br />
        {/* ... */}
      </p>
      <button onClick={() => editAddress()}>Edit</button>
    </section>
  </div>
)
```

On peut voir qu'au fur et à mesure que la fiche contact va s'agrandir (pour y afficher de nouvelles informations) le composant va devenir de plus en plus complexe. Et pourtant on ne fait qu'y afficher des données.

Une première étape peut consister à le diviser en plusieurs composants, pour chaque section à afficher :

```
const ContactGeneralInfos = ({ contact, editGeneralInfos }) => (
  <section>
    <h2>General info</h2>
    <p>First name: {contact.firstName}</p>
    <p>Last name: {contact.lastName}</p>
    {/* ... */}
    <button onClick={() => editGeneralInfos()}>Edit</button>
  </section>
)

const ContactAddress = ({ contact, editAddress }) => (
  <section>
    <h2>Address</h2>
    <p>
      {contact.address.addressLineOne}
      <br />
      {contact.address.addressLineTwo}
      <br />
      {/* ... */}
    </p>
    <button onClick={() => editAddress()}>Edit</button>
  </section>
)
const ContactView = ({ contact, editGeneralInfos, editAddress }
=> (
  <div>
    <ContactGeneralInfos
```

```
      contact={contact}
      editGeneralInfos={editGeneralInfos}
    />
    <ContactAddress contact={contact}
      editAddress={editAddress} />
  </div>
)
```

2.2 Des composants pour une interface homogène

Écrire du code avec des composants réutilisables permet non seulement de le rendre plus facile à maintenir, mais également de rendre l'interface utilisateur aussi homogène que possible, que ce soit au niveau de l'affichage (CSS...) que celui du comportement (boutons...).

Dans l'exemple précédent, on peut facilement aller un peu plus loin en remarquant que les deux sections (général et adresse) partagent du code en commun : les deux sont affichées au sein d'une balise `<section>`, et comportent un titre `<h2>`. Ce peut donc être une bonne idée de créer un composant pour ces sections :

```
const ContactSection = ({ title, children }) => (
  <section>
    <h2>{title}</h2>
    {children}
  </section>
)

const ContactGeneralInfos = ({ contact, editGeneralInfos }) => (
  <ContactSection title="General info">
    <p>First name: {contact.firstName}</p>
    {/* ... */}
    <button onClick={() => editGeneralInfos()}>Edit</button>
  </ContactSection>
)

const ContactGeneralInfos = ({ contact, editAddress }) => (
  <ContactSection title="Address">
    <p>
      {contact.address.addressLineOne}
      {/* ... */}
```

```
    </p>
    <button onClick={() => editAddress()}>Edit</button>
  </ContactSection>
)
```

Ainsi, dans le cas où plus tard on ne souhaiterait plus utiliser une balise `<h2>` mais une balise `<h3>` par exemple, il suffira de modifier le composant `ContactSection`. Notez que l'on pourrait aller encore plus loin en intégrant par exemple le bouton d'édition au composant `ContactSection`. Bien évidemment, cela dépendra du besoin. Toutes les sections auront-elles forcément un et un seul bouton ?

2.3 Sortir la logique des composants

À l'origine, le but même d'un composant est d'afficher des données (cela est vrai ailleurs que dans React). Mais au fur et à mesure qu'une application se complexifie, il est courant que les composants soient aussi responsables de traitements de données en vue de leur affichage ou du déclenchement d'une action suite à un évènement (clic sur un bouton...).

Par exemple, supposons qu'un composant soit responsable d'afficher une arborescence de fichiers. On peut imaginer que les fichiers sont stockés dans un état (Redux ou autre composant) sous forme de liste plate, et qu'ils sont passés en propriété à notre composant :

```
const FileTree = ({ files }) => {
  const tree = {}
  files.forEach(file => {
    // Ajout du fichier à `tree`
  })

  // Retour de JSX pour l'affichage de `tree`
}

// Utilisation:
<FileTree files={[
  '/Documents/Comptes.ods',
  '/Documents/Présentation.odp',
  '/Images/Photos/Paysage01.jpg',
  // ...
]}>
```

Dans ce composant, la logique de création de l'arbre a-t-elle vraiment sa place ? L'idéal serait de la sortir de composant ; quelles sont les possibilités qui s'offrent à nous avec ce que nous avons déjà vu ?

- Extraire la création de l'arbre dans un autre fichier (un helper) et l'importer ici.
- Déléguer la création de l'arbre à l'entité qui a appelé `FileTree` : autre composant ou action Redux, par exemple.

Ces deux méthodes sont parfaitement valables ; nous verrons au long de ce chapitre d'autres possibilités qui s'appliqueraient très bien ici.

2.4 Limiter le state local et les effets de bord

Dans le chapitre Découverte de React nous avons vu comment un composant peut gérer un state local et déclencher des effets de bord (requêtes HTTP...) grâce aux méthodes du cycle de vie d'un composant (`componentWillMount`...). Et maintenant on apprend que ce n'est pas une bonne pratique ?

En réalité, ce n'est pas aussi simple, et il est parfaitement acceptable qu'un composant utilise ces possibilités. Cependant, toujours en supposant que l'application grossisse et se complexifie, il sera extrêmement bénéfique de limiter le nombre de composants qui utilisent un state local et déclenchent des effets de bord.

En effet, un composant sans state local (*stateless*) et sans effet de bord est beaucoup plus facile à utiliser, à maintenir, à tester, et surtout à réutiliser.

Reprenons un exemple de fiche contact, et supposons que les données du contact soient récupérées depuis une API :

```
class Contact extends React.Component {
 state = {
   contact: null
 }
 componentWillMount() {
   fetch('/api/contacts/' + this.props.contactId)
     .then(res => res.json())
     .then(contact => this.setState({ contact }))
 }
```

```
  editGeneralInfos = () => {
    // ...
  }
  editAddress = () => {
    // ...
  }
  render() {
    if (this.state.contact) {
      return (
        <div>
          <ContactGeneralInfos
            contact={this.state.contact}
            editGeneralInfos={this.editGeneralInfos}
          />
          <ContactAddress
            contact={this.state.contact}
            editAddress={this.editAddress}
          />
        </div>
      )
    } else {
      return <span>Loading...</span>
    }
  }
}
```

Ce composant a en réalité deux responsabilités bien distinctes :

- Récupérer les informations du contact.
- Afficher les informations du contact, ou un *placeholder* le temps de les récupérer.

De plus, je n'ai pas présenté dans l'exemple comment implémenter l'édition d'informations, mais il est probable que cela fasse surgir une nouvelle responsabilité. Par ailleurs, que se passe-t-il si je souhaite afficher les informations d'un contact déjà récupéré, ou bien d'un contact créé à l'aide d'un formulaire ? Je ne souhaite pas ici faire de nouvel appel à l'API...

Ici, la première chose à faire pour simplifier ce composant est d'extraire l'affichage des informations dans un autre composant (cela tombe bien, c'est justement le composant du premier exemple) :

```
class Contact extends React.Component {
 // ...
 render() {
   if (this.state.contact) {
     return (
       <ContactView
         contact={this.state.contact}
         editGeneralInfos={this.editGeneralInfos}
         editAddress={this.editAddress}
       />
     )
   } else {
     return <span>Loading...</span>
   }
 }
}
```

Ce composant comporte toujours un state local et fait toujours une requête HTTP, mais au moins nous avons extrait une partie de ses responsabilités (l'affichage des informations) dans un composant sans état ni effets de bord (`ContactView`). Et `ContactView` peut être utilisé pour afficher des informations d'un contact indépendamment de la manière dont celui-ci a été récupéré.

2.5 En conclusion

Les principes d'écriture de « beaux » composants sont nombreux, et un livre entier serait sûrement nécessaire pour les énumérer. L'important à retenir est que la plupart des bonnes pratiques d'écriture de code s'appliquent aux composants. Souvent il suffit de bien réfléchir à la manière de définir et diviser ses composants. Mais parfois on peut faire encore mieux.

Voyons quelques patterns permis par React pour cela.

3. Les high-order components

Les composants de haut niveau, plus couramment appelés *high-order components* ou HOC, sont un moyen particulièrement élégant de créer des composants réutilisables. Le principe est simple : un HOC est une fonction qui prend en paramètre une définition de composant (classe ou fonction), et renvoie une nouvelle définition de composant, qui ajoute du comportement à la première. Il s'agit en fait du pattern *Décorateur* appliqué aux composants React.

Vous en avez déjà utilisé sans même le savoir. La fonction `connect` de React-Redux est un HOC. En effet, en appelant `connect(mapStateToProps)`, vous obtenez une fonction prenant un composant en paramètre (celui que vous avez défini à côté), et renvoyant un composant, connecté à Redux. Autres exemples : les fonctions `withRouter` de React Router (chapitre Gestion de formulaires et du routage), ou encore `withNavigation` de React Navigation (cf. chapitre Développer pour le mobile avec React Native).

Voici un exemple très simple de HOC :

```
const addBorder = borderWidth => Component => props => (
 <div style={{
   borderColor: 'black', borderStyle: 'solid', borderWidth
 }}>
   <Component {...props} />
 </div>
)

const MyText = <p>Hello!</p>

const MyTextWithBorder = addBorder(5)(MyText)
```

Vous obtenez un composant `MyTextWithBorder` qui affiche le texte « Hello » avec une bordure de 5 pixels. Ici, `addBorder` est donc bien ce que l'on appelle un *high-order component*.

Quel est l'intérêt d'un HOC ? Un pattern très utile est d'extraire un comportement partagé par plusieurs composants dans des fonctions réutilisables. Voyons un exemple plus fourni.

3.1 Exemple : champ de saisie d'un numéro de téléphone

Pour appréhender les HOC, nous allons dans cette section utiliser le concept pour créer un champ de saisie de numéro de téléphone, qui :

- n'acceptera que les chiffres, parenthèses, tirets et espaces en entrée (à la frappe) ;
- mettra en forme le numéro de téléphone lorsque le focus sera perdu par le champ (évènement `blur`). (Seuls les numéros de téléphone nord-américains seront pris en compte : « (514) 555-0199 ».)

Saisie des chiffres dans le champ, et formatage du numéro à la perte de focus

Notez que l'on supposera que notre champ sera contrôlé, c'est-à-dire que nous utiliserons les propriétés `value` et `onChange` pour savoir quel texte afficher et comment le mettre à jour. Nous souhaitons également que la valeur ne contienne que les chiffres du numéro de téléphone (« 5145550199 »), sans se soucier de la mise en forme, et donc que le `onChange` soit appelé avec les chiffres uniquement (dans `event.target.value`).

Pour rendre notre HOC plus facile à écrire et maintenir, nous utiliserons la bibliothèque *Recompose* (https://github.com/acdlite/recompose), qui propose un grand nombre de fonctions utilitaires pour écrire des HOC. Nous en verrons quelques-unes dans cette section.

Pour développer notre composant, nous créerons deux HOC réutilisables, un pour chacun des points ci-dessus. Cela signifie que nous souhaitons que notre composant final soit défini ainsi :

```
const PhoneNumberInput = formatPhoneNumber(
 forbidNonPhoneNumberCharacters(props => <input {...props} />)
)
```

C'est le bon moment pour introduire la première fonction de Recompose que nous utiliserons : `compose`. Elle effectue la composition de plusieurs HOC pour les fusionner en un seul, de sorte que nous pouvons écrire plus simplement :

```
const PhoneNumberInput = compose(
 formatPhoneNumber,
 forbidNonPhoneNumberCharacters
)(props => <input {...props} />)
```

Et parce que nous souhaitons rendre nos HOC aussi réutilisables que possible (pour mettre en forme autre chose que des numéros de téléphone par exemple), rendons-les plus génériques :

```
// Ne garde que les chiffres, espaces, tirets et parenthèses
const forbiddenCharactersInPhoneNumber = /[^\d\s\-()]/g

// '5145551234' => '(514) 555-1234'
const formatPhoneNumber = value =>
 value.replace(/^(\d{3})(\d{3})(\d{4})$/, '($1) $2-$3')

// '(514) 555-1234' => '5145551234'
const parsePhoneNumber = formattedPhoneNumber =>
 formattedPhoneNumber.replace(/[^\d]/g, '').slice(0, 10)

const PhoneNumberInput = compose(
 formatInputValue({
   formatValue: formatPhoneNumber,
   parseValue: parsePhoneNumber
 }),
 forbidCharacters(forbiddenCharactersInPhoneNumber)
)(props => <input {...props} />)
```

Ne trouvez-vous pas cela déjà élégant si l'on peut réutiliser uniquement nos deux HOC pour mettre en forme à la fois des montants, des numéros de sécurité sociale, tout et n'importe quoi, juste en utilisant les bons paramètres ?

Le point réellement intéressant est qu'ici on utilise le composant `<input>` de base, mais nous pourrions utiliser n'importe quel composant, tant qu'il utilise les propriétés `value`, `onChange` et `onBlur`. Donc on peut imaginer utiliser notre champ de saisie de numéros de téléphone avec React Native par exemple, avec assez peu d'adaptation.

Passons maintenant au plus important : écrire nos deux HOC en utilisant les fonctions que Recompose met à notre disposition.

3.1.1 Premier HOC : n'accepter que certains caractères

L'idée ici est que lorsque la valeur de l'input est changée (évènement onChange), on intercepte cet évènement pour supprimer tout caractère interdit de la valeur, puis on appelle la propriété onChange parente avec la valeur propre.

Nous utiliserons ici la fonction `withHandlers` pour ajouter des nouveaux *handlers* d'évènements comme propriétés du composant encapsulé. Le bon point est que nous avons accès aux propriétés de notre composant (ici nous utiliserons onChange) pour créer notre nouveau *handler* :

```
const forbidCharacters = forbiddenCharsRegexp =>
 withHandlers({
   onChange: props => event => {
     // N'oublions pas que `onChange` n'est pas une
     // propriété requise (même si rien ne se produira
     // si elle est absente).
     if (props.onChange) {
       const value = event.target.value
       const cleanValue = value.replace(
         forbiddenCharsRegexp, '')
       // On ne modifie pas l'évènement original,
       // mais on le clone  en y redéfinissant
       // event.target.value avec la valeur propre.
       const newEvent = {
         ...event,
         target: { ...event.target, value: cleanValue }
       }
       // On émet à nouveau notre évènement au `onChange` parent.
       props.onChange(newEvent)
     }
   }
 })
```

Souvenez-vous qu'autant que possible le composant que nous créons à partir d'un autre doit respecter l'interface de ce dernier. Il doit donc accepter les mêmes propriétés avec le même type.

À présent, si nous souhaitons par exemple créer un champ n'acceptant que les chiffres, nous pouvons écrire :

```
const NumericField = forbidCharacters(/[^\d]/g)(
 props => <input {...props} />
)
```

Nous avons maintenant notre premier HOC pour interdire certains caractères ; écrivons à présent le deuxième, légèrement plus complexe, pour mettre en forme la valeur entrée par l'utilisateur.

3.1.2 Deuxième HOC : mettre en forme la valeur entrée

Pour notre deuxième HOC, nous devrons avoir dans notre composant un état local pour stocker la valeur entrée dans le champ sans la passer au composant parent. N'oubliez pas que nous souhaitons mettre en forme la valeur uniquement lorsque le focus sort du champ (évènement `blur`).

Recompose définit une fonction très simple pour ajouter un état local à un composant : `withState`. Elle prend en paramètre le nom de l'attribut dans l'état (qui sera donné comme propriété au composant enfant), le nom de la propriété contenant la fonction pour mettre à jour cet état (également donnée comme propriété), et la valeur initiale (valeur statique, ou bien fonction prenant en paramètre les propriétés et retournant la valeur initiale).

Pour ajouter notre état local, nous écrirons :

```
withState(
 'inputValue',
 'setInputValue',
 // `formatValue` est l'un des paramètres de notre HOC
 props => formatValue(props.value)
)
```

Maintenant que l'on a notre état, nous devons le mettre à jour lorsque la valeur de l'input est modifiée, donc nous définirons un *handler* `onChange` personnalisé :

```
withHandlers({
 onChange: props => event => {
   props.setInputValue(event.target.value)
 }
 // ...
```

Et à l'évènement `blur`, nous mettrons en forme la valeur, appellerons les `onChange` et `onBlur` parents, puis mettrons en forme également la valeur affichée :

```
  // ...
 onBlur: props => event => {
   // parseValue est l'autre paramètre de notre HOC
   const parsedValue = parseValue(props.inputValue)
   const formattedValue = formatValue(parsedValue)
   props.setInputValue(formattedValue)
   // On ne modifie pas l'évènement original, mais on le clone
   // en y redéfinissant event.target.value avec la valeur propre.
   const newEvent = {
     ...event,
     target: { ...event.target, value: parsedValue }
   }
   if (props.onChange) {
     props.onChange(newEvent)
   }
   if (props.onBlur) {
     props.onBlur(newEvent)
   }
 }
)
```

La dernière étape pour notre HOC consiste à nous assurer que seules les propriétés acceptées par `<input>` lui seront passées. Pour cela on utilisera la fonction `mapProps` de Recompose pour créer un nouvel objet de propriétés à partir des propriétés existantes, ainsi que la déstructuration de JavaScript pour exclure les `props` dont nous ne voulons pas :

```
mapProps(({ inputValue, setInputValue, ...otherProps }) => ({
 // otherProps est identique aux propriétés passées,
 // mais ne contient pas inputValue ni setInputValue.
 ...otherProps,
 value: inputValue
}))
```

En assemblant le tout avec `compose`, on obtient :

```
const formatInputValue = ({ formatValue, parseValue }) =>
 compose(
   withState(
     'inputValue', 'setInputValue',
```

```
    props => formatValue(props.value)
  ),
  withHandlers({
    onChange: props => event => {
      props.setInputValue(event.target.value)
    },
    onBlur: props => event => {
      const parsedValue = parseValue(props.inputValue)
      const formattedValue = formatValue(parsedValue)
      props.setInputValue(formattedValue)
      const newEvent = {
        ...event,
        target: { ...event.target, value: parsedValue }
      }
      if (props.onChange) {
        props.onChange(newEvent)
      }
      if (props.onBlur) {
        props.onBlur(newEvent)
      }
    }
  }),
  mapProps(({ inputValue, setInputValue, ...otherProps }) => ({
    ...otherProps,
    value: inputValue
  }))
)
```

Nous avons à présent deux *high-order components*, que l'on peut utiliser pour créer notre champ de saisie de numéro de téléphone. Vous trouverez l'ensemble du code source de cet exemple dans les exemples téléchargeables accompagnant le livre.

3.2 En conclusion

Les composants de haut niveau sont utilisés par de nombreuses bibliothèques de composants, à commencer par la fameuse React Redux. Le fonctionnement est élégant, car on retrouve les avantages de la programmation fonctionnelle classique, par exemple le fait de pouvoir composer des fonctions (avec `compose`).

Mais comme souvent il existe plusieurs solutions pour répondre à un même problème. Et les HOC ont un concurrent féroce très en vogue pour créer des composants réutilisables : les *render props*.

4. Les render props

Dans la section précédente, nous avons vu comment les high-order components pouvaient répondre au besoin de mutualiser de la logique commune à plusieurs composants. Dans certains cas d'utilisation, une manière alternative de répondre à ce besoin est d'utiliser des *render props* (terme qui ne trouve pas de traduction française satisfaisante).

Comme pour les HOC, le principe de base est très simple : donner à un composant une ou plusieurs propriétés qui sont des fonctions permettant de générer le rendu d'un composant. Voyons plus concrètement avec un exemple à quoi cela ressemble :

```
const ContactView = props => {
 const { contact, renderPhone } = props
 return (
   <div>
     <span>{contact.name}</span>
     <span>{renderPhone(contact.phone)}</span>
   </div>
 )
}
ContactView.propTypes = {
 contact: PropTypes.shape({
   name: PropTypes.string,
   phone: PropTypes.string
 }),
 renderPhone: PropTypes.func
```

```
}
ContactView.defaultProps = {
 renderPhone: phone => phone
}
```

Ici, le composant `ContactView` ne fait qu'afficher le nom d'une personne et son numéro de téléphone, mais vous remarquerez qu'on a la possibilité de lui passer en propriété une fonction `renderPhone` permettant de personnaliser l'affichage du numéro de téléphone. Grâce aux `defaultProps` du composant, le comportement par défaut sera simplement d'afficher le numéro, mais nous pouvons également utiliser notre composant ainsi :

```
<ContactView
 contact={{ name: 'Peter', phone: '514-555-1234' }}
 renderPhone={phone => <a href={`tel:${phone}`}>{phone}</a>}
/>
```

Ainsi, dans notre fiche contact, nous aurons un lien cliquable permettant de lancer un appel téléphonique.

Afin d'aller plus loin avec les *render props*, reprenons l'exemple de la section précédente. Pour rappel, nous avions créé un composant `PhoneNumberInput`, l'étape finale de réalisation ressemblant à ceci :

```
const PhoneNumberInput = compose(
 formatInputValue(/* ... */),
 forbidCharacters(/* ... */)
)(props => <input {...props} />)
```

Il est ainsi très facile d'utiliser notre composant comme un champ `input` classique :

```
<PhoneNumberInput
 value={this.state.phone}
 onChange={event => this.setState({
   phone: event.target.value
 })}
/>
```

L'inconvénient ici est que nous avons « figé » le champ de saisi à un champ `input` ; mais qu'en est-il si l'on souhaite utiliser le composant avec React Native ? Ou une bibliothèque telle que Material-UI définissant un champ `Input` enrichi ?

Grâce à une *render prop* nous pouvons laisser la possibilité d'utiliser n'importe quel composant de saisie à la place du classique `input`. Nous utiliserons donc le composant ainsi :

```
<PhoneNumberInput
 value={this.state.phone}
 onChange={event => this.setState({
   phone: event.target.value
 })}
 renderInput={props => <input {...props} />}
/>
```

Pour ce qui est des changements des `PhoneNumberInput`, une solution possible pour garder l'utilisation de notre HOC est de procéder ainsi :

```
const makePhoneNumberInput = compose(
 formatInputValue(/* ... */),
 forbidCharacters(/* ... */)
)

const PhoneNumberInput = makePhoneNumberInput(
 ({ renderInput, ...otherProps }) => renderInput(otherProps)
)
```

Les *render props* sont beaucoup utilisées par des bibliothèques tierces pour personnaliser l'affichage d'un composant. Par exemple, React Table (https://react-table.js.org), puissante bibliothèque d'affichage de tableaux, permet de gérer le comportement du tableau (colonnes, pagination, tri, évènements, etc.), mais laisse la possibilité de définir comment une cellule du tableau doit être affichée, ou bien une ligne, ou encore le texte à afficher lorsqu'il n'y a pas de données.

Un dernier cas d'utilisation très répandu des *render props* est une bizarrerie de React : le fait que le « contenu » d'un composant (ce qu'on lui passe dans sa balise) est en fait une propriété comme une autre, la propriété `children`. Par exemple, les deux lignes suivantes sont équivalentes :

```
<Composant x={1000}><span>Bonjour!</span></Composant>
<Composant x={1000} children={<span>Bonjour!</span>} />
```

Ainsi, il est donc possible de passer comme enfant d'un composant non seulement du texte ou d'autres composants, mais également une fonction !

```
<Composant x={1000} children={x => x + 1234} />
// Ou encore:
<Composant x={1000}>
 {x => x + 1234}
</Composant>
```

Bien entendu, cela suppose que le composant appelé s'attende bien à recevoir une fonction comme propriété `children` et la traite comme telle. Par exemple :

```
const Composant = props => {
 const makeBigger = props.children
 return <span>{makeBigger(props.x)}</span>
}
```

Vu comme cela, difficile de voir l'intérêt de cette possibilité, tout sauf intuitive ! D'ailleurs, il serait fort peu recommandable d'utiliser la propriété `children` ainsi dans ce genre d'exemple. Mais il existe certains cas d'utilisation où cela est beaucoup plus élégant : lorsque la *render prop* ne permet pas de personnaliser l'affichage du composant, mais qu'elle est en fait le cœur de la fonctionnalité apportée par le composant.

Prenons l'exemple d'un composant qui afficherait la taille de la fenêtre. Une version simple pourrait ressembler à ceci :

```
class App extends Component {
 constructor(props) {
   super(props)
   this.state = { width: 0, height: 0 }
 }
 onResize = () => {
   this.setState({
     width: window.innerWidth,
     height: window.innerHeight
   })
 }
 componentDidMount() {
   this.onResize()
   window.addEventListener('resize', this.onResize)
 }
 componentWillUnmount() {
   window.removeEventListener('resize', this.onResize)
 }
 render() {
```

```
    return (
      <span>
        {this.state.width}x{this.state.height}
      </span>
    )
  }
}
```

Cela fonctionne très bien, mais supposons maintenant qu'un autre composant ait besoin d'accéder à la taille actuelle de la fenêtre. Ou simplement que l'on préfère séparer la partie affichage du code qui réagit au redimensionnement de la fenêtre. Nous pouvons alors écrire un composant `WindowSize` dont l'unique but serait de gérer les informations sur la taille de la fenêtre, et enverrait la taille à une *render prop* qui lui serait passée :

```
class WindowSize extends Component {
 constructor(props) {
   super(props)
   this.state = { width: 0, height: 0 }
 }
 onResize = () => {
   this.setState({
     width: window.innerWidth,
     height: window.innerHeight
   })
 }
 componentDidMount() {
   this.onResize()
   window.addEventListener('resize', this.onResize)
 }
 componentWillUnmount() {
   window.removeEventListener('resize', this.onResize)
 }
 render() {
   const { width, height } = this.state
   return this.props.children(width, height)
 }
}

const App = () => (
 <WindowSize>
   {(width, height) => (
     <span>
```

```
        {width}x{height}
      </span>
    )}
  </WindowSize>
)
```

Ainsi, notre composant `App` ne conserve à sa charge que la partie affichage, tandis que la gestion de la taille courante de la fenêtre est faite dans `WindowSize`, composant totalement générique et réutilisable.

Notez que rien ne nous obligeait à utiliser la propriété `children` comme *render prop* ; nous aurions pu par exemple faire en sorte d'utiliser `WindowSize` ainsi :

```
<WindowSize
 renderWithSize={(width, height) => (
   <span>
     {width}x{height}
   </span>
 )}
/>
```

Mais, dans ce cas, nous perdons une information pertinente pour la lisibilité du code : fournir la taille de la fenêtre est ici la fonction principale de notre composant `WindowSize`. Il ne s'agit plus ici de personnaliser son comportement, contrairement aux exemples vus plus haut.

Ce dernier exemple a mis en évidence une bonne pratique qu'il est recommandé d'appliquer en React et même en développement web en général : séparer ce qui concerne l'affichage de ce qui concerne le comportement de l'application et des données à afficher. Les *render props* n'étaient ici qu'un moyen d'arriver à cette fin, mais une fonctionnalité de React vient généraliser ce principe encore plus loin, il s'agit des *hooks*.

5. Les hooks

Les *high-order components* et les *render props* permettent chacun à leur manière de répondre à un même besoin : faire en sorte qu'un composant soit le plus réutilisable possible, notamment en séparant ce qui peut être mutualisé (comportement, affichage de base...) de ce qui doit laisser un certain degré de liberté à l'utilisateur du composant (affichage personnalisé...).

Mais nous avons également vu dans le deuxième chapitre de ce livre que React propose une fonctionnalité qui pourrait également répondre à ce besoin, je parle naturellement des hooks. Nous avons vu comment utiliser certains des hooks standards de React (`useState`, `useCallback`, `useEffect`...), ou encore des hooks apportés par des bibliothèques tierces (`useSelector` de React Redux, `useQuery` de Apollo Client), mais bonne nouvelle : on peut également écrire nos propres hooks !

Reprenons par exemple ce que nous avions fait dans la section sur les *render props* : avoir un composant capable d'afficher la taille actuelle de la fenêtre. Il s'agissait de s'abonner à l'évènement `resize` de la fenêtre et de conserver cette taille dans un état interne au composant.

En utilisant les hooks, nous nous abonnerons à l'évènement dans `useEffect`, et nous conserverons la taille dans un état déclaré par `useState` :

```
const App = () => {
 const [width, setWidth] = useState(0)
 const [height, setHeight] = useState(0)

 const onResize = useCallback(event => {
   const { innerWidth, innerHeight } = event.target
   setWidth(innerWidth)
   setHeight(innerHeight)
 }, [width, setWidth, height, setHeight])

 useEffect(() => {
   setWidth(window.innerWidth)
   setHeight(window.innerHeight)
   window.addEventListener('resize', onResize)
   return () => {
     window.removeEventListener('resize', onResize)
   }
 }, [setWidth, setHeight, onResize])
```

```
  return (
    <span>
      {width}x{height}
    </span>
  )
}
```

Notez que cette fois-ci dans la fonction passée à `useEffect` nous retournons nous-mêmes une fonction. Ainsi nous indiquons à `useEffect` ce qu'il doit faire lorsque le composant est « démonté », en l'occurrence annuler l'abonnement à l'évènement *resize*.

Notre composant est relativement élégant mais, après tout, on est revenu en arrière par rapport à la section précédente, puisqu'on gère l'affichage au même endroit que la récupération de la taille de la fenêtre. Est-il possible de sortir la logique de notre composant ? Bien évidemment, car il est possible de créer nos propres hooks.

Pour notre besoin, nous allons créer un hook `useWindowSize` qui nous permettra d'obtenir la taille actuelle de la fenêtre (en lecture seulement).

```
const useWindowSize = () => {
  const [width, setWidth] = useState(0)
  const [height, setHeight] = useState(0)

  const onResize = useCallback(/* ... */)

  useEffect(/* ... */)

  return [width, height]
}
```

Pour utiliser notre hook, cela se passe comme pour n'importe quel hook que nous avons utilisé jusqu'ici :

```
const App = () => {
  const [width, height] = useWindowSize()
  return (
    <span>
      {width}x{height}
    </span>
  )
}
```

Mission accomplie : la logique est à présent dans le hook personnalisé `useWindowSize`, et notre composant `App` ne contient plus que l'affichage de la taille de la fenêtre.

6. Les contextes et le pattern Provider/Consumer

Dans la dernière section de ce chapitre nous allons voir un pattern que vous avez déjà utilisé sans forcément le savoir au fil de ce livre, il s'agit du pattern *Provider/Consumer*. Le but est de partager des informations entre les composants sans que celles-ci ne soient passées par les propriétés. Cela vous rappelle quelque chose ? C'est ce qui avait motivé l'utilisation de Redux dans le chapitre Concevoir une application avec Redux.

Aujourd'hui, React 16 fournit un outil permettant de mettre en place une gestion d'informations partagées sans avoir besoin de passer par Redux, ce qui peut être très pratique pour de petites applications, ne nécessitant pas tous les apports de Redux, comme des actions asynchrones. Cette fonctionnalité, ce sont les contextes.

Commençons par imaginer la situation suivante : notre application doit afficher divers composants graphiques avec une charte de couleurs précises (un arrière-plan et une couleur pour le texte). Nous souhaitons que les composants aient accès à ces couleurs sans passer un objet contenant ces couleurs en propriété dans toute la hiérarchie des composants.

On peut également imaginer que nous souhaitons que cette gestion de thèmes puisse être réutilisée dans plusieurs applications, qui n'utiliseraient pas forcément Redux, d'où l'intérêt de passer par une fonctionnalité standard de React.

À l'usage, cela pourrait donner ceci pour un composant `Box` :

```
// Box.js
import React from 'react'
import { injectTheme } from './theme'

const Box = ({ backgroundColor, textColor, text }) => {
 const style = {
   backgroundColor: backgroundColor,
   color: textColor
 }
```

```
  return <div style={style}>{text}</div>
}

export default injectTheme(Box)

// Utilisation dans un autre fichier:
<Box text="Hello!"/>
```

Lorsqu'on utilise le composant `Box`, on ne lui passe donc pas les propriétés `backgroundColor` et `textColor`, mais le composant a bien accès à ces valeurs, grâce à l'appel à `injectTheme` (qui est un high-order component). Cela vous rappelle Redux ? C'est bien normal, Redux est un parfait exemple d'application du pattern *Provider/Consumer*, qui utilise les contextes lui aussi.

Il nous reste donc à écrire `injectTheme` et faire en sorte qu'elle ait accès aux informations du thème. Pour cela, la première étape est de définir un contexte, grâce à `createContext` de React :

```
// theme.js
import React, { createContext } from 'react'

const ThemeContext = createContext()
```

L'objet `ThemeContext` contient alors deux propriétés propres à tout contexte : `Provider` et `Consumer`. Il serait possible d'utiliser ces deux objets directement, mais l'intérêt de définir un contexte personnalisé est justement de pouvoir les « décorer » pour rendre plus intuitive leur utilisation.

Le `Provider` est un composant qui devra contenir toute la hiérarchie de composants qui doivent avoir accès au contexte. De plus, on lui fournira la valeur à stocker dans le contexte, qui dans notre cas sera un objet contenant nos deux couleurs. Créons donc un composant `ThemeProvider` utilisant le `Provider` de notre contexte :

```
const ThemeProvider = ({
 backgroundColor,
 textColor,
 children
}) => {
 return (
   <ThemeContext.Provider value={{
     backgroundColor,
     textColor
```

```
    }}>
      {children}
    </ThemeContext.Provider>
  )
}
```

Avant de passer à `injectTheme`, voyons comment utiliser `ThemeProvider` dans l'application :

```
// App.js
import React from 'react'
import Box from './Box'
import { ThemeProvider } from './theme'

const App = () => (
 <ThemeProvider backgroundColor="red" textColor="blue">
   <Box text="Hello!" />
 </ThemeProvider>
)

export default App
```

Rappelez-vous lorsque nous avions utilisé Redux, l'ensemble de l'application était englobée de la même manière dans le `Provider` de React-Redux : `<Provider store={store}>`

Nous avons vu la partie *Provider* du contexte, qu'en est-il de la partie *Consumer* ? Notre fonction `injectTheme`, en bon high-order component, est une fonction prenant en paramètre un composant, et retournant un autre composant. Dans notre cas, ce nouveau composant devra avoir les propriétés `backgroundColor` et `textColor` en plus de celles qui sont passées au composant de base :

```
const injectTheme = Comp => props => (
 <ThemeContext.Consumer>
   {({ backgroundColor, textColor }) => (
     <Comp
       {...props}
       backgroundColor={backgroundColor}
       textColor={textColor}
     />
   )}
 </ThemeContext.Consumer>
)
```

Comme vous le voyez, le composant `Consumer` issu du contexte suit le pattern d'accepter une fonction comme enfant (propriété `children`, comme nous avons vu que c'était possible un peu plus haut dans le chapitre dans la section consacrée aux render props).
À cette fonction est passée la valeur du contexte, c'est-à-dire nos couleurs, que nous repassons comme propriétés au composant retourné.

Notez que nous avons fait le choix de proposer d'utiliser le `Consumer` grâce à un high-order component, mais nous pouvons tout aussi bien proposer un composant acceptant une fonction comme enfant également :

```
const ThemeConsumer = ({ children }) => (
 <ThemeContext.Consumer>
   {({ backgroundColor, textColor }) =>
     children(backgroundColor, textColor)
   }
 </ThemeContext.Consumer>
)
```

Notre composant `Box` peut alors s'écrire ainsi :

```
// box2.js
import React from 'react'
import { ThemeConsumer } from './theme'

const Box = ({ text }) => (
 <ThemeConsumer>
   {(backgroundColor, textColor) => (
     <div style={{
       backgroundColor: backgroundColor,
       color: textColor
     }}>
       {text}
     </div>
   )}
 </ThemeConsumer>
)

export default Box
```

Nous avons vu comment utiliser un contexte avec un HOC, avec une *render prop*... Et si vous avez aimé découvrir les hooks personnalisés dans la section précédente, bonne nouvelle : il existe un hook permettant d'utiliser un contexte : `useContext`.

On peut donc utiliser directement ce hook dans le composant `Box`, ou mieux : créer un hook personnalisé :

```
const useTheme = () => {
 return useContext(ThemeContext)
}
```

Le composant `Box` s'écrit alors :

```
const Box = ({ text }) => {
 const { backgroundColor, textColor } = useTheme()
 return <div style={...}>/* ... */</div>
}
```

Pour aller plus loin, remarquez qu'ici notre contexte ne sert qu'à stocker des données statiques, mais rien n'empêche de permettre la modification de ces données, grâce à ce que l'on fournit aux composants utilisant le contexte. Par exemple, le hook `useState` que nous avons vu dans la section précédente permet d'écrire un contexte permettant de lire et modifier une valeur de manière très élégante :

```
// counter.js
import React, { createContext, useState } from 'react'

const CounterContext = createContext()

const CounterProvider = ({ initialValue, children }) => {
 const [counter, setCounter] = useState(initialValue)
 return (
   <CounterContext.Provider value={{ counter, setCounter }}>
     {children}
   </CounterContext.Provider>
 )
}

const injectCounter = Comp => props => (
 <CounterContext.Consumer>
   {({ counter, setCounter }) => (
     <Comp {...props} counter={counter}
         setCounter={setCounter} />
   )}
 </CounterContext.Consumer>
)
```

```
export { CounterProvider, injectCounter }
```

On peut alors avoir deux composants utilisant ce contexte, par exemple l'un affichant la valeur du compteur, l'autre permettant de l'incrémenter :

```
// CounterDisplay.js
import React from 'react'
import { injectCounter } from './counter'

const CounterDisplay = ({ counter }) => (
 <span>
   Counter: <strong>{counter}</strong>
 </span>
)

export default injectCounter(CounterDisplay)

// IncrementCounterButton.js
import React from 'react'
import { injectCounter } from './counter'

const IncrementCounterButton = ({ counter, setCounter }) => (
 <button onClick={
   () => setCounter(counter + 1)
 }>Increment</button>
)

export default injectCounter(IncrementCounterButton)
```

Pour que le tout fonctionne, il suffit à présent que ces deux composants soient utilisés au sein du même `CounterProvider` :

```
<CounterProvider initialValue={1}>
 <CounterDisplay />
 <IncrementCounterButton />
</CounterProvider>
```

Vous trouverez dans les exemples téléchargeables accompagnant le livre la version complète de cette petite application.

Les contextes sont utilisés depuis longtemps par les bibliothèques disponibles pour React (notamment Redux), mais leur utilisation a grandement été simplifiée grâce à `createContext` en React 16. Leur plus grand avantage est de permettre de diviser la gestion d'un état global en plusieurs contextes. Ici, nous avons vu un contexte pour le thème graphique, un autre pour stocker une donnée à afficher ; on pourrait avoir un autre contexte pour gérer la langue d'une application et les traductions, par exemple. Et on pourrait même réutiliser ces contextes dans d'autres applications.

7. Conclusion

Créer des composants au maximum réutilisables et donc génériques est un besoin auquel est confronté tout développeur, que ce soit en React ou non. Il est fréquent que dans un projet on ait d'un côté les composants représentant les fonctionnalités d'une application (les pages ou écrans), et d'un autre, une bibliothèque de composants partagés, qui peuvent être utilisés par plusieurs fonctionnalités. Pour de gros projets, il est même courant que des composants soient partagés entre plusieurs applications, par exemple pour conserver une homogénéité graphique ou comportementale.

Dans ce chapitre, après avoir énoncé quelques principes bons à garder en tête au moment de développer de nouveaux composants, nous avons vu trois moyens de mettre cela en œuvre. Les *high-order components* et les *render props* ont largement fait leurs preuves, et si généralement ils répondent au même besoin, l'utilisation de l'un ou de l'autre est souvent une préférence du développeur. Certaines bibliothèques ou applications sont organisées autour de HOC, d'autres autour de render props. Quant aux hooks, ils semblent bien faire l'unanimité malgré leur jeunesse, c'est pourquoi la plupart des bibliothèques associées à React migrent progressivement leur API vers l'utilisation de hooks.

Enfin, nous avons vu comment les contextes permettent de sortir encore un peu de logique des composants, en déléguant à d'autres entités la gestion d'un état global et même sa mise à jour, à l'image de ce que fait Redux. Cela permet de simplifier les composants toujours dans le but que leur principale responsabilité soit l'affichage de données.

À la vitesse où évoluent React et son écosystème, il est fort probable que dans les prochains mois ou années de nouveaux patterns auront émergé, répondant à des besoins actuels ou nouveaux. Et c'est bien ce qui fait la force de React !

Chapitre 11
Tester une application React

1. Que tester dans une application React ?

En guise de dernier chapitre, il semble pertinent d'aborder le sujet des tests, et notamment des tests unitaires. Sur une application front-end, qu'elle soit en React ou non, la question se pose régulièrement de savoir quoi tester unitairement. En effet, si l'on prend par exemple un composant qui n'est responsable que de présentation en générant du HTML, il serait laborieux de tester précisément le rendu du composant dans le navigateur. De plus, il est possible que le moindre changement dans le CSS associé au composant change le rendu au point de mettre le test en échec, ce qui n'est généralement pas le but.

Par certains aspects, il est tout de même intéressant de tester quelques composants :

- pour tester les informations affichées ;
- pour tester le comportement du composant en réponse à des actions de l'utilisateur.

Par ailleurs, si votre application utilise Redux, il est possible de tester totalement le store en fonction des actions dispatchées, et ainsi tester la logique métier de l'application.

Dans ce chapitre, nous écrirons tout d'abord des tests sur les composants, en utilisant la bibliothèque Enzyme. Puis des tests sur la logique d'un store Redux couplé à Redux-Saga, à l'aide de Redux Saga Test Plan.

2. Test unitaire de composants avec Enzyme

2.1 Initialisation de l'application à tester

Afin de découvrir les tests unitaires de composants React, commençons par créer l'application que nous souhaiterons tester. Celle-ci sera initialisée de la même manière que la plupart des applications créées dans ce livre, avec Parcel :

```
$ yarn init -y
$ yarn add -D parcel-bundler
$ yarn add react react-dom
```

Le fichier public/index.html est habituel également :

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

Le composant principal de notre application, celui que nous allons tester, sera un formulaire de contact. Il sera réduit au strict minimum, en effet il ne contiendra qu'un seul champ permettant de saisir un nom.

Ses spécifications sont les suivantes :

- Le champ **nom** sera initialisé avec la propriété `contact.name`.
- Il sera éditable, autrement dit si l'on saisit une valeur celle-ci devra bien être affichée dans le champ.
- S'il est vide, un message d'erreur devra s'afficher.
- Lorsque le formulaire est soumis, la fonction fournie en propriété `updateContact` sera appelée, avec en paramètre le contact mis à jour avec le nom saisi, et ce uniquement si le champ n'est pas vide.

Écrire un tel composant ne présente aucune difficulté. Voici, par exemple, une implémentation utilisant des hooks :

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

export const ContactForm = ({ contact, updateContact }) => {
 const [name, setName] = useState(contact.name)

 const onChangeName = useCallback(
   event => {
     setName(event.target.value)
   },
   [setName],
 )
 const onSubmit = useCallback(event => {
   event.preventDefault()
   if (name !== '') {
     updateContact({ name })
   }
 })

 return (
   <form onSubmit={onSubmit}>
     <label>
       Name:
       <input name="name" value={name}
              onChange={onChangeName} />
       <br />
       {name === '' &&
         <span className="error">Please enter a name.</span>}
     </label>
     <br />
     <button type="submit">OK</button>
   </form>
 )
}
```

Le nom en cours d'édition est stocké dans un état local grâce au hook `useState`, initialisé avec le contact donné en propriété, et la soumission du formulaire, après vérification que le nom saisi n'est pas vide, appelle la fonction `updateContact`.

Pour lancer l'application avec ce composant et le tester manuellement, faisons en sorte qu'il soit affiché au chargement de l'application (dans src/index.js) :

```
import React from 'react'
import ReactDOM from 'react-dom'
import { ContactForm } from './ContactForm'

const App = () => (
 <ContactForm contact={{ name: 'John Doe' }}
              updateContact={console.log} />
)
ReactDOM.render(<App />, document.getElementById('app'))
```

Il ne reste qu'à ajouter la commande `"start": "parcel public/index.html"` au *package.json* pour lancer l'application avec `yarn start`. Sans surprise, le formulaire devrait répondre aux spécifications données. Nous pouvons passer à ce qui nous intéresse : tester unitairement ce composant.

2.2 Jest et Enzyme

Pour lancer les tests, nous utiliserons Jest (https://jestjs.io), qui s'est imposé comme un standard pour tester des applications React, bien qu'il soit possible d'utiliser n'importe quel outil de test JavaScript. Une question de préférence personnelle, donc.

Jest nous fournira plusieurs choses :

- Le moyen de lancer nos tests (via la commande `jest`) et de les organiser au sein de blocs `describe` et du cas de test `it`.
- Les fonctions pour effectuer les assertions, par exemple pour vérifier qu'une valeur obtenue est bien conforme à celle attendue (`expect(...).toEqual(...)`).
- Et enfin le moyen de créer facilement des *mocks* de fonctions, et de vérifier qu'ils ont été appelés et avec quels paramètres.

Si ces notions ne vous disent rien, ne vous en faites pas nous allons les voir en pratique dans un instant.

Associé à Jest, nous aurons également besoin d'un outil permettant de faire le rendu d'un composant React (celui que nous souhaitons tester). Il existe plusieurs solutions pour cela, et plusieurs outils. Citons par exemple ReactTestUtils (https://reactjs.org/docs/test-utils.html), intégré à React mais désormais déprécié au profit de deux autres librairies : React Testing Library (https://testing-library.com/docs/react-testing-library) et Enzyme d'AirBnB (https://airbnb.io/enzyme/). La première est relativement récente et minimaliste, là où Enzyme a fait ses preuves depuis plusieurs années. C'est pourquoi j'ai choisi de présenter Enzyme, bien que React Testing Library semble monter rapidement en popularité.

Enzyme a été créée par AirBnb, et présente la particularité de proposer deux modes de fonctionnement :

- Le *shallow rendering* qui permet de monter un composant avec un seul niveau de hiérarchie. C'est-à-dire que s'il fait appel à des composants enfants (excluant les éléments du DOM) ceux-ci ne seront pas rendus. Imaginez que ces composants seront accessibles tels des nœuds du DOM, vides, mais permettant de vérifier leurs propriétés (nous reviendrons sur un exemple concret en fin de section).
- Le *full DOM rendering*, effectuant le rendu du composant dans un DOM virtuel de la même manière que dans un navigateur, c'est-à-dire en affichant toute l'arborescence du composant et en permettant l'accès aux API du DOM.

Aucun mode n'est meilleur que l'autre dans l'absolu. Pour notre usage, c'est-à-dire tester unitairement un composant, en isolation totale des autres composants, le premier mode nous conviendra davantage.

Commençons par installer Jest, Enzyme, ainsi que quelques dépendances nécessaires :

```
$ yarn add -D jest enzyme enzyme-adapter-react-16 \
  @babel/core @babel/preset-env @babel/preset-react
```

L'inclusion des dépendances de Babel est nécessaire, car si elles sont automatiquement gérées par Parcel au lancement de l'application pour la transpilation du code ES2015 et JSX, ce n'est pas le cas lors du lancement des tests avec Jest.

C'est pourquoi nous aurons également besoin d'un fichier de configuration babel.config.js :

```
module.exports = api => {
 api.cache(true)
 return {
   presets: ['@babel/env', '@babel/react'],
 }
}
```

Pour ce qui est de Jest, il est également nécessaire d'ajouter une configuration propre à Enzyme à son lancement. Pour cela, créons le fichier jest.config.js :

```
module.exports = {
 setupFilesAfterEnv: ['<rootDir>src/setupTests.js'],
}
```

Enfin le fichier src/setupTests.js :

```
import { configure } from 'enzyme'
import Adapter from 'enzyme-adapter-react-16'

configure({ adapter: new Adapter() })
```

Nous sommes prêts à commencer l'écriture de nos tests pour le composant `ContactForm`, en utilisant Jest et Enzyme. Par convention, les tests se trouvent dans des fichiers dont le nom se termine par « .test.js ». Ainsi pour tester notre composant `ContactForm`, créons un fichier ContactForm.test.js situé à côté.

Pour organiser nos tests, Jest nous permet d'utiliser deux types de blocs :

- `it` (ou `test`) permet de définir un cas de test.
- `describe` permet de grouper plusieurs cas de tests (`it` ou `test`), ainsi que de définir des actions à effectuer avant chacun d'eux, ou avant le premier uniquement, ou encore à la fin de tous les tests englobés.

Un fichier de test aura donc par exemple la structure suivante :

```
describe('maFonction', () => {
 it('renvoie 0 si aucun paramètre n'est passé', () => {
   // ...
 })
 it('renvoie 1 si un paramètre est passé', () => {
```

```
    // ...
  })
})
```

Au sein d'un bloc `describe`, il est aussi possible d'avoir des blocs `beforeAll`, `beforeEach`, `afterAll` et `afterEach`, permettant comme leur nom l'indique d'effectuer respectivement des actions avant tous les tests, avant chaque test, après tous les tests et après chaque test.

2.3 Écriture des tests du composant

Initialisons à présent notre fichier de test avec un premier cas de test que nous allons remplir au fur et à mesure :

```
describe('ContactForm', () => {
  it('displays the name in the input', () => {
    // ...
  })
})
```

La première étape de notre test va être d'effectuer le rendu (en mode *shallow rendering*) de notre composant. Pour cela, nous utiliserons la fonction `shallow` fournie par Enzyme.

```
const wrapper = shallow(
  <ContactForm contact={{ name: 'John Doe' }}
               updateContact={() => {}} />,
)
```

L'objet renvoyé, appelé *wrapper* dans la terminologie d'Enzyme, propose plusieurs méthodes pour accéder au contenu du rendu, c'est-à-dire les composants générés. Pour notre test, nous souhaitons récupérer le champ de saisie (`input`) et vérifier que sa valeur est bien le nom du contact que l'on a passé en propriété :

```
expect(
  wrapper.find('input').get(0).props.value
).toEqual('John Doe')
```

La méthode `wrapper.find` prend en paramètre un sélecteur de type CSS semblable à ceux utilisés par `document.querySelector` par exemple (`'tag'`, `'.classe'`, `'#id'`, etc.).

Elle peut aussi prendre directement une définition de composant comme nous le verrons plus loin.

Elle renvoie potentiellement plusieurs éléments, d'où l'utilisation de `.get(0)` pour récupérer dans notre cas l'unique champ `input` affiché.

Voici donc notre premier fichier de test dans son ensemble :

```
import React from 'react'
import { shallow } from 'enzyme'
import { ContactForm } from './ContactForm'

describe('ContactForm', () => {
 it('displays the name in the input', () => {
   const wrapper = shallow(
     <ContactForm contact={{ name: 'John Doe' }}
                  updateContact={jest.fn()} />,
   )
   expect(
     wrapper.find('input').get(0).props.value
   ).toEqual('John Doe')
 })
})
```

Pour exécuter ce test, le plus simple est de définir la commande `test` dans le *package.json* : `"test": "jest"`, puis de lancer la commande `yarn test`. La sortie devrait ressembler à cela :

```
 PASS  src/ContactForm.test.js
 ContactForm
   ✓ displays the name in the input (13ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.623s
Ran all test suites.
```

Nous avons donc vérifié que la première des spécifications était vérifiée, à savoir que le contenu du champ de saisie était initialisé avec le nom du contact donné en propriété.

La prochaine étape est de vérifier que lorsque l'on tape un nouveau nom dans le champ, celui-ci est bien affiché (en réalité, cela devrait être une évidence, mais si l'on avait fait une erreur dans la définition de la propriété `onChange` par exemple, il se pourrait qu'il soit impossible de modifier la valeur).

Pour simuler une action de l'utilisateur, le `wrapper` renvoyé par `shallow` propose la méthode `simulate`, qui prend deux paramètres : le nom de l'évènement à déclencher (« `change` » dans notre cas) et l'évènement lui-même.

```
it('updates the name when updating input value', () => {
 const wrapper = shallow(
   <ContactForm contact={{ name: 'John Doe' }}
                updateContact={() => {}} />,
 )
 wrapper.find('input').simulate('change',
   { target: { value: 'Jane Doe' } }
 )
 expect(
   wrapper.find('input').get(0).props.value
 ).toEqual('Jane Doe')
})
```

On commence à remarquer que ces deux premiers tests sont assez difficiles à lire et ont beaucoup de code en commun. Essayons déjà de mutualiser certaines parties. Par exemple, le wrapper sera vraisemblablement toujours généré de la même manière. On peut donc le définir avant chaque test, dans un bloc `beforeEach` :

```
let wrapper

beforeEach(() => {
 wrapper = shallow(
   <ContactForm contact={{ name: 'John Doe' }}
                updateContact={() => {}} />,
 )
})
```

De même, le champ `input` sera toujours récupéré de la même manière, on peut donc créer une fonction pour simplifier cela :

```
const getInput = () => wrapper.find('input')
```

Le code du second cas de test par exemple devient ainsi plus léger :

```
it('updates the name when updating input value', () => {
 getInput().simulate('change',
   { target: { value: 'Jane Doe' } }
 )
 expect(getInput().get(0).props.value).toEqual('Jane Doe')
})
```

Comme troisième cas de test, vérifions que l'erreur est affichée si et seulement si le champ de saisie est vide. L'erreur étant affichée dans un élément de classe « `error` », créons déjà une fonction pour récupérer cet élément. On peut ensuite vérifier que l'erreur n'est pas affichée au chargement (puisque le nom est initialisé avec celui du contact), puis est affichée lorsqu'on vide le champ :

```
const getError = () => wrapper.find('.error')

it('displays the error message when input is empty', () => {
 expect(getError().length).toEqual(0)
 getInput().simulate('change', { target: { value: '' } })
 expect(getError().length).toEqual(1)
})
```

Afin de vérifier ensuite que la soumission du formulaire entraîne l'appel de la fonction `updateContact`, il nous faut définir cette fonction. Pour cela, Jest propose grâce à `jest.fn()` de créer des fonctions qui peuvent être *espionnées*, c'est-à-dire qu'il est possible de savoir si elles ont été appelées, et avec quels paramètres.

Définissons donc la fonction et modifions la création du composant dans le `beforeEach` pour lui injecter cette fonction en propriété `updateContact` :

```
let wrapper
let updateContact

beforeEach(() => {
 updateContact = jest.fn()
 wrapper = shallow(
   <ContactForm
     contact={{ name: 'John Doe' }}
     updateContact={updateContact}
   />,
 )
})
```

Nous pouvons ainsi écrire notre quatrième cas de test :

```
const getForm = () => wrapper.find('form')

it('calls updateContact on form submit', () => {
 getInput().simulate('change',
   { target: { value: 'Jane Doe' } }
 )
 getForm().simulate('submit', { preventDefault() {} })
 expect(updateContact).toHaveBeenCalledWith(
   { name: 'Jane Doe' }
 )
})
```

Deux choses sont importantes à noter ici. Premièrement, nous ne simulons pas un clic sur le bouton, mais directement la soumission du formulaire lui-même. En effet, le *shallow rendering* d'Enzyme ne propage pas les évènements comme ils le seraient dans le DOM. Un clic sur le bouton n'aurait donc dans notre cas aucun effet, puisque nous n'avons pas défini de propriété `onClick` dessus.

De plus, on fournit un objet évènement contenant une méthode `preventDefault` qui ne fait rien. Comme dans notre composant nous appelons `event.preventDefault()` à la soumission, une erreur serait déclenchée si nous ne fournissions pas cette méthode.

Remarquez la syntaxe très intuitive proposée par Jest pour vérifier qu'une fonction a été appelée avec les paramètres attendus, avec `expect(...).toHaveBeenCalledWith(...)`.

Le dernier cas de test à implémenter est relativement similaire, puisqu'il s'agit de vérifier que la fonction `updateContact` n'est pas appelée si le champ de saisie est vide :

```
it('doesn't call updateContact if input is empty', () => {
 getInput().simulate('change', { target: { value: '' } })
 getForm().simulate('submit', { preventDefault() {} })
 expect(updateContact).not.toHaveBeenCalled()
})
```

Notre test est complet, et vérifie l'ensemble des spécifications imposées. Constatez à quel point les descriptions des cas de tests (fournies à `it`) reflètent les spécifications.

De plus, les tests eux-mêmes sont relativement simples à comprendre même pour quelqu'un ne connaissant pas Enzyme, à condition de définir les bonnes fonctions utilitaires. Pour aller plus loin, on aurait pu par exemple définir des fonctions `getInputValue` et `setInputValue` :

```
const getInputValue = () => getInput().get(0).props.value
const setInputValue = value =>
 getInput().simulate('change', { target: { value } })
```

Bien évidemment, cela dépendra de la complexité des tests, il ne s'agit pas non plus de redéfinir son propre framework de test. Pour un développeur expérimenté qui aurait à lire les tests, voir un appel à `simulate('change', ...)` aura probablement plus de sens qu'un appel à une fonction `setInputValue` dont il devrait chercher la définition pour définir ce qu'elle fait exactement.

2.4 Spécificité du shallow rendering d'Enzyme

Dans la présentation d'Enzyme, nous avons vu qu'avec le *shallow rendering* le composant n'était rendu qu'avec un seul niveau de hiérarchie. Afin de montrer concrètement ce que cela signifie et implique, modifions notre composant `ContactForm`, afin que le message d'erreur ne soit plus affiché directement par le formulaire, mais par un composant `Error` :

```
// Error.js
import React from 'react'
export const Error = ({ message }) =>
 <span className="error">{message}</span>

// ContactForm.js
{name === '' && <Error message="Please enter a name." />}
```

Vous pouvez remarquer que le rendu sera exactement le même, ainsi que les éléments du DOM générés. Par contre, nos tests ne passent plus, en l'occurrence, celui chargé de vérifier qu'une erreur est affichée lorsque le champ de saisie est vide.

En effet, la hiérarchie du rendu de notre composant est à présent :

```
- form
 - label
   - (text)
   - input
   - br
   - Error
     - span
 - br
 - button
```

En mode *shallow rendering*, lorsqu'Enzyme rencontre notre composant `Error`, il ne va pas plus loin en profondeur et il est donc impossible de vérifier la présence de l'élément `span` à l'intérieur, pas plus que de récupérer son contenu.

En revanche, pour que nos tests passent à nouveau, il est bien possible de vérifier qu'un élément `Error` a été rendu :

```
const getError = () => wrapper.find(Error)
```

Si l'on souhaite récupérer le message d'erreur affiché, on peut ainsi vérifier la propriété `message` : `getError().get(0).props.message`.

Au premier abord, on peut voir cette limitation du *shallow rendering* comme une contrainte. Mais en réalité, non seulement ce n'est pas tant un problème, mais c'est même le principal avantage de ce mode de test de composants. En effet, nous ne souhaitons tester que le composant `ContactForm`, pas le composant `Error`. Ce dernier devra disposer de ces propres tests unitaires.

Voilà qui conclut cette présentation des tests de composants avec Enzyme. Sachez que nous n'avons qu'effleuré la surface des possibilités offertes par Jest et Enzyme. Il s'agit de deux bibliothèques puissantes, dont les documentations respectives sont très complètes, vous n'aurez donc pas de mal à trouver les informations nécessaires pour aller plus loin (Enzyme : https://airbnb.io/enzyme/, Jest : https://jestjs.io/docs/en/getting-started).

Dans la section suivante, nous allons toujours utiliser Jest, pour tester la partie Redux d'une application cette fois-ci, et encore une fois nous allons le coupler à une autre bibliothèque : Redux Saga Test Plan.

3. Tester le store Redux et les sagas avec Redux Saga Test Plan

Tester le store Redux d'une application peut très bien se faire sans bibliothèque externe. Après tout, un reducer n'est qu'une simple fonction n'ayant aucun effet de bord, les créateurs d'action également, ainsi que les sélecteurs. Seules les sagas pourraient poser un peu plus de difficultés, mais en sachant qu'elles ne sont que des fonctions génératrices, on pourrait tout de même s'en sortir. Néanmoins, ces tests ne seraient pas réellement pertinents.

Dans cette section, nous allons voir comment grâce à Redux Saga Test Plan, on peut écrire des tests sur un store entier (ou sur un morceau de store ou service, comme nous en avons écrit dans les chapitres Concevoir une application avec Redhux et Gérer les effets de bord avec Redux-Saga), en le considérant comme une boîte noire. Autrement dit, nous vérifierons que lorsque l'on dispatche une action :

- Les bons effets de bord sont déclenchés (dans notre cas, un appel à une API).
- Les bonnes actions résultantes sont dispatchées.
- Le state a bien été mis à jour (et nous utiliserons pour cela des sélecteurs).

Comme exemple accompagnant cette section, nous allons repartir de l'exemple précédent avec son `ContactForm`, en y ajoutant un appel à l'API de JSON Placeholder, que nous avons vue à plusieurs reprises, pour initialiser le nom du contact.

Notre store Redux sera extrêmement semblable à ce que nous avons vu dans les chapitres sur Redux et Redux Saga (dans un seul fichier pour aller à l'essentiel).

Il gère trois actions, une pour demander la récupération du contact, et deux pour gérer les cas de succès et d'erreur :

```
export const actionTypes = {
 GET_CONTACT: 'GET_CONTACT',
 GET_CONTACT_SUCCESS: 'GET_CONTACT_SUCCESS',
 GET_CONTACT_FAILURE: 'GET_CONTACT_FAILURE',
}

export const actions = {
```

```
  getContact: () => ({ type: actionTypes.GET_CONTACT }),
  getContactSuccess: contact => ({
    type: actionTypes.GET_CONTACT_SUCCESS,
    payload: contact,
  }),
  getContactFailure: error => ({
    type: actionTypes.GET_CONTACT_FAILURE,
    payload: error,
  }),
}
```

Le state contient l'ID du contact courant dont on souhaite récupérer les informations, les informations récupérées le cas échéant, ainsi que deux flags `loading` et `error`. Nous fournissons quatre sélecteurs permettant d'accéder à ces informations :

```
const initialState = {
  contactId: 2,
  loading: false,
  error: false,
  contact: null,
}

export const reducer = (state = initialState, action) => {
  switch (action.type) {
    case actionTypes.GET_CONTACT:
      return { ...state, loading: true, error: false }
    case actionTypes.GET_CONTACT_SUCCESS:
      return { ...state, loading: false,
               contact: action.payload }
    case actionTypes.GET_CONTACT_FAILURE:
      return { ...state, loading: false, error: true }
    default:
      return state
  }
}

export const selectors = {
  selectContactId: state => state.contactId,
  selectLoading: state => state.loading,
  selectError: state => state.error,
  selectContact: state => state.contact,
}
```

Nous n'aurons qu'une seule saga, déclenchée lorsque l'action `getContacts` est dispatchée. Dans ce cas, on appellera la fonction `api.getContact`, déclarée dans le fichier api.js, puis on dispatchera l'action de succès ou d'erreur selon le résultat :

```
function* getContactSaga() {
 try {
   const contactId = yield select(selectors.selectContactId)
   const contact = yield call(api.getContact, contactId)
   yield put(actions.getContactSuccess(contact))
 } catch (err) {
   yield put(actions.getContactFailure(err))
 }
}

export function* rootSaga() {
 yield takeEvery(actionTypes.GET_CONTACT, getContactSaga)
}
```

Il ne reste qu'à créer et exporter le store en lui ajoutant le `middleware` de Redux Saga :

```
const sagaMiddleware = createSagaMiddleware()
export const store = createStore(
 reducer,
 applyMiddleware(sagaMiddleware)
)
sagaMiddleware.run(rootSaga)
```

Les ajouts et modifications sur les composants React ne présentent que peu d'intérêt pour ce qui est des tests du store, je vous laisse donc le soin de les effectuer par vous-même si vous le souhaitez, ou de consulter les exemples téléchargeables associés au livre pour voir la version complète de l'application.

Passons à ce qui nous intéresse : les tests du store. Pour commencer, installons les dépendances nécessaires :

```
$ yarn add redux redux-saga @babel/polyfill
$ yarn add -D redux-saga-test-plan
```

Pour rappel, avec Parcel comme avec Jest, `@babel/polyfill` est nécessaire pour utiliser des fonctions génératrices. Ajoutons-le aux fichiers src/index.js et src/setupTests.js :

```
import '@babel/polyfill'
```

Notre test de store aura deux cas de tests, l'un pour gérer le succès de la récupération du contact, l'autre pour gérer le cas d'erreur :

```
// store.test.js
describe('Store', () => {
 it('dispatches getContactSuccess if success', async () => {
   // ...
 })

 it('dispatches getContactFailure on error', async () => {
   // ...
 })
})
```

Commençons par tester le cas de succès, ou *happy path*. Pour démarrer le test avec Redux Saga Test Plan, on utilise la fonction `expectSaga`, en lui passant la saga que l'on souhaite tester en paramètre. On pourrait ici passer la saga `getContactSaga`, mais le plus judicieux est de passer plutôt la saga racine `rootSaga`, afin de tester le comportement lorsqu'on dispatche une action `getContact` seulement.

Afin de tester le store dans son ensemble, il est également nécessaire de fournir le reducer que l'on souhaite utiliser à l'aide de `withReducer`. Il sera également testé, puisqu'il sera appelé dès qu'une action est dispatchée, comme s'il s'agissait d'un store normal et non un store de test.

Pour lancer la saga, et donc le test, on appelle ensuite la méthode `run` sur l'objet renvoyé.

```
expectSaga(rootSaga)
 .withReducer(reducer)
 .run()
```

Pour le moment, notre test ne « teste », en vérité, rien, si ce n'est qu'il n'y a pas d'erreur déclenchée. Mais comme nous n'avons pas encore dispatché d'action, ce serait peu probable.

Pour dispatcher une action au démarrage du test, on utilisera la méthode `dispatch` juste avant d'appeler `.run()` :

```
expectSaga(rootSaga)
  .withReducer(reducer)
  .dispatch(actions.getContact())
  .run()
```

Nous avons maintenant le nécessaire pour lancer le test avec les bonnes « entrées » (en l'occurrence la configuration du store et le fait d'avoir dispatché l'action souhaitée), nous pouvons passer à la vérification des « sorties ». Ces sorties sont dans notre cas de plusieurs types :

- Les effets déclenchés par la saga : `select`, `call`, `put`.
- Les modifications du state induites par les actions dispatchées dans le reducer.

Pour vérifier les effets déclenchés par la saga, il s'agit de fonctions à appeler dans la chaîne des appels de `expectSaga`, chaque effet étant testé par une fonction du même nom. Par exemple, si l'on souhaite vérifier que l'effet `select` est appelé avec le sélecteur `mySelector`, on écrira `.select(mySelector)`.

Dans notre cas, nous souhaitons un effet `select`, un effet `call` et un effet `put` :

```
.select(selectors.selectContactId)
.call(api.getContact, 2) // ID du contact dans l'initialState
.put(actions.getContactSuccess({ name: 'Jane Doe' }))
```

Nous avons cependant un souci : comment nous assurer de ce qui sera passé en paramètre de l'action `getContactSuccess` ? Dans la mesure où ce paramètre est issu d'un appel à une API, difficile de prévoir ce qui sera renvoyé. D'ailleurs, dans un contexte de test, on souhaite la plupart du temps nous affranchir de tout appel à une API. Pour cela, créez un *mock* de la fonction `api.getContact` grâce à Jest, pour renvoyer une valeur bien définie. Mais grâce à Redux Saga Test Plan c'est même beaucoup plus simple. En effet, il permet une syntaxe pour simuler le retour de n'importe quel effet (en créant un *mock* pour nous), et notamment l'effet `call`.

Pour cela, on fournit un ensemble de *providers* à l'exécution, chacun définissant un effet intercepté, et la valeur que l'on doit lui renvoyer. Ici, nous souhaitons intercepter l'effet `call` ayant pour paramètres `api.getContact` et 2, et renvoyer (de manière asynchrone) un faux contact :

```
.provide([
 [
   call(api.getContact, 2),
   Promise.resolve({ name: 'Jane Doe' })
 ],
])
```

Bien que ce ne soit pas nécessaire ici, nous pourrions tout aussi bien simuler le retour de l'effet `select` afin de simuler un autre ID de contact que celui présent dans le state :

```
.provide([
 [select(selectors.selectContactId), 42],
 // ...
```

Mis bout à bout, voici tous les éléments constituant notre appel à `expectSaga` :

```
expectSaga(rootSaga)
 .withReducer(reducer)
 .provide([
   [
     call(api.getContact, 2),
     Promise.resolve({ name: 'Jane Doe' })
   ],
 ])
 .select(selectors.selectContactId)
 .call(api.getContact, 2)
 .put(actions.getContactSuccess({ name: 'Jane Doe' }))
 .dispatch(actions.getContact())
 .run()
```

Remarquez que l'ordre dans lequel appeler les fonctions n'est pas intuitif. Tout d'abord, on fournit les providers, puis les effets attendus (`select`, `call`, `put`), et enfin on effectue les actions nécessaires pour lancer le test (`dispatch` et `run`).

Il nous reste une chose à vérifier pour que notre test soit complet : que le state a bien été mis à jour. Pour le moment nous n'avons finalement testé que la saga, mais nous pouvons aussi tester le reducer et les sélecteurs. Pour cela, le retour de l'appel à `expectSaga(...)...run()` est une promesse renvoyant notamment un attribut `storeState`, contenant comme son nom l'indique le contenu du state après l'exécution. Nous pouvons donc appeler nos sélecteurs sur ce state pour vérifier son contenu :

```
const { storeState } = await expectSaga(rootSaga)
 // ...
 .run()
expect(selectors.selectLoading(storeState)).toEqual(false)
expect(selectors.selectError(storeState)).toEqual(false)
expect(selectors.selectContact(storeState)).toEqual(
 { name: 'Jane Doe' }
)
```

Notre test est bien complet à présent. Le second test vérifiant le cas d'erreur est relativement similaire :

```
const { storeState } = await expectSaga(rootSaga)
 .withReducer(reducer)
 .provide([
   [call(api.getContact, 2), Promise.reject('some error')]
 ])
 .select(selectors.selectContactId)
 .call(api.getContact, 2)
 .put(actions.getContactFailure('some error'))
 .dispatch(actions.getContact())
 .run()
expect(selectors.selectLoading(storeState)).toEqual(false)
expect(selectors.selectError(storeState)).toEqual(true)
expect(selectors.selectContact(storeState)).toEqual(null)
```

Grâce à Redux Saga Test Plan, nous avons pu tester l'intégralité des éléments de notre store : les actions, les sélecteurs, le reducer et la saga. Le plus intéressant est que nous n'avons pas eu à nous soucier de certains détails d'implémentation, comme l'emplacement des informations dans le state. Si l'implémentation change l'attribut `contact` pour l'appeler `result` par exemple, du moment que le sélecteur `selectContacts` est mis à jour en conséquence, le test ne sera pas à modifier.

Autrement dit, on teste le store comme une boîte noire, en testant que pour des entrées données (actions dispatchées), on obtient la sortie escomptée (appel à une API, autre action dispatchée en retour, modification du state).

À titre personnel, tester ainsi un store Redux et ses sagas m'a permis de me rendre compte de l'intérêt et de la puissance de Redux Saga. La notion d'effet présentée au chapitre Gérer les effets de bord avec Redux-Saga n'est pas des plus faciles à appréhender, mais lorsqu'on l'utilise au quotidien et que l'on teste le store Redux et les sagas avec Redux Saga Test Plan, on a du mal à revenir en arrière.

Comme c'était le cas avec Enzyme dans la section précédente, nous n'avons, encore une fois, vu qu'une toute petite partie des possibilités offertes par Redux Saga Test Plan. Par exemple, il est possible de fournir des providers correspondant à un ensemble d'effets donnés. Le meilleur exemple étant un effet `call` sur une fonction donnée, mais sans spécifier les paramètres supplémentaires. Ou encore, il est possible de vérifier dynamiquement des effets déclenchés, de la même manière que nous avons vérifié le contenu du state après exécution. La documentation de Redux Saga Test Plan (http://redux-saga-test-plan.jeremyfairbank.com/) vous sera bien évidemment d'une aide précieuse pour aller plus loin.

4. En conclusion

Ainsi se clôt notre exploration du test d'une application React. Nous aurions pu aborder beaucoup d'autres sujets et manières de tester une application. Parmi les autres outils en vogue, citons par exemple Cypress (https://www.cypress.io) particulièrement adapté pour des tests *end-to-end*, qui vont jusqu'à lancer un navigateur pour ouvrir l'application testée, effectuer une série d'actions, pour vérifier que les éléments attendus sont présents sur la page.

Le sujet des tests front-end n'est en rien spécifique à React. D'ailleurs, toute la section consacrée à Redux Saga Test Plan pourrait s'appliquer à n'importe quelle bibliothèque de rendu, du moment que l'on utilise Redux et Redux Saga (ce qu'il est possible de faire avec Angular ou Vue par exemple). Mais il est tout de même intéressant de voir les différentes possibilités qui s'offrent à nous lorsqu'on souhaite tester une application dans un cadre professionnel, ou pour une application que l'on souhaite stable au fil de ses versions.

Certains projets ont pour philosophie de mettre le moins de logique possible dans les composants React, au point de réduire leur fonction au simple affichage de données. Cela est généralement une bonne pratique et justifie parfois de se passer de tests de composants. Néanmoins, si l'on place de la logique dans un hook personnalisé par exemple, ou un high-order component, ou encore que l'on souhaite tout de même avoir une certaine logique métier dans le composant, alors avoir des tests unitaires associés peut se révéler d'une grande aide pour la maintenabilité.

Conclusion

1. Aller plus loin

En guise de conclusion à cet ouvrage, quelques pistes vous seront proposées pour approfondir votre expérience avec React. Il sera évoqué, par exemple, comment il est possible de faire du React sans faire de JavaScript, comment utiliser React pour générer un site statique, ou encore quelques bibliothèques de composants seront abordées afin de simplifier la création d'interfaces utilisateur riches.

2. Reason : un autre langage pour faire du React

Lorsqu'on entend parler de React, il est bien entendu naturel de l'associer à JavaScript, puisque c'est le langage pour lequel il est prévu, bien qu'il soit possible de l'utiliser avec TypeScript par exemple. Pour cette raison, il paraît étonnant d'imaginer qu'un nouveau langage permette non seulement d'utiliser React, mais aille même jusqu'à intégrer JSX au cœur de sa syntaxe.

Reason (https://reasonml.github.io) est un langage créé au sein de Facebook, et même par les créateurs de React, c'est dire à quel point il a été conçu pour être utilisé conjointement à React. Dit simplement, Reason est une surcouche au langage OCaml, qui grâce à l'outil BuckleScript, peut être transpilé en JavaScript.

Finalement, on peut donc voir Reason comme une alternative à TypeScript, puisque l'un comme l'autre sont destinés à devenir du JavaScript. Ce qui pousse à utiliser TypeScript est d'ailleurs souvent la même raison que pour Reason: pouvoir typer ses variables, paramètres, objets, etc. pour intercepter plus vites des erreurs, à la compilation et non à l'exécution.

Mais là où TypeScript apporte des fonctionnalités à JavaScript, Reason constitue lui un langage totalement différent, bien que les efforts soient faits pour que la syntaxe soit similaire à JavaScript au maximum. Et notamment, Reason se veut un langage 100 % fonctionnel, au même titre qu'OCaml ou Haskell.

Et en tant que langage fonctionnel, Reason intègre en son cœur les notions phares de typage fort (et d'inférence de type), d'immutabilité, etc. Pour un développeur n'ayant jamais appréhendé de tels langages, cela peut paraître troublant et même une contrainte dans le développement. Mais toute personne ayant un jour essayé la programmation fonctionnelle sait en reconnaître les avantages (et les apprécier ou non), et l'appliquer au développement front-end avec React est maintenant possible avec Reason.

Pour plus d'informations sur Reason et React, vous pouvez consulter:

- le site web de Reason (https://reasonml.github.io/), qui vous permettra de vous familiariser avec le langage (ce que je recommande d'ailleurs avant de penser à son utilisation avec React) ;
- le site web de Reason-React : (https://reasonml.github.io/reason-react/).

Parcel, que nous avons utilisé pour les projets React dans ce livre, gère très bien Reason (https://parceljs.org/reasonML.html), ce serait dommage de s'en priver!

3. Générer des sites statiques avec Gatsby

L'utilisation principale de React est le développement d'applications web ou mobiles, donc parler de l'utiliser pour des sites statiques peut paraître étonnant. Tout d'abord, précisons que site statique ne signifie pas seulement site « vitrine », c'est-à-dire composé uniquement de pages statiques n'évoluant pas ou peu.

Par site statique, on entend ici tout site qui ne nécessite pas de génération de page côté serveur (en PHP, Node.js, Ruby, etc.).

On peut donc réaliser sous forme de site statique un blog, un site de e-commerce, ou encore évidemment les sites vitrines de sociétés (ce que l'on entend de manière classique par « site statique »). Pour les blogs par exemple, l'affichage d'articles est bien statique, il n'y aurait que la gestion des commentaires qui nécessiterait un traitement serveur, mais cette partie peut être déléguée à un service externe comme Disqus (https://disqus.com) par exemple.

L'avantage des sites statiques est qu'ils sont extrêmement légers puisque n'affichant que des pages statiques dont la récupération peut être optimisée par le cache du navigateur. L'idéal est même de faire en sorte que JavaScript ne soit pas nécessaire pour leur affichage, ce qui les rend alors légers à l'affichage également.

Mais alors, que peut apporter React pour ces sites statiques ? Et en quoi React peut-il être utile si JavaScript n'est même pas activé dans le navigateur ? Eh bien, grâce à un outil comme Gatsby (https://www.gatsbyjs.org), à partir d'un ensemble de composants React, plusieurs fichiers HTML seront générés, un pour chaque page du site.

Prenons un exemple simple : vous souhaitez mettre en œuvre un site contenant deux pages, la méthode classique consisterait à écrire deux pages HTML. Avec Gatsby, vous pouvez écrire ces pages avec React, c'est-à-dire en mutualisant les parties communes dans des composants partagés, en externalisant le *layout* du site dans un autre composant, etc.

Mais l'intérêt principal de Gatsby est le nombre de plugins dont il dispose. Par exemple, si vous créez un blog, vous pouvez écrire vos articles en Markdown, et demander à Gatsby de générer le HTML correspondant. Vous pourrez également gérer la pagination d'articles, des formulaires, lier les pages entre elles... le tout en React.

Une fois que votre site est prêt, une commande Gatsby suffit à construire une version du site prête à être déployée, constituée des fichiers HTML (un par page donc), des CSS, des images, etc. Le mieux, c'est que si JavaScript est tout de même activé dans le navigateur, alors les transitions entre pages se feront sans rechargement.

Gatsby est un outil extrêmement puissant, et, il faut le reconnaître, pas des plus simples à maîtriser au début. Si vous souhaitez vous y initier, un bon point de départ est le tutoriel du site officiel (https://www.gatsbyjs.org/tutorial/), qui vous expliquera comment créer un premier site, de manière très détaillée.

4. Des bibliothèques de composants utiles

La philosophie du développement en composants serait bien moins intéressante s'il n'était pas possible de trouver des bibliothèques de composants pour répondre à des besoins courants. Voyons-en ici quelques-unes, ainsi que d'autres bibliothèques qui vous seront utiles.

4.1 Material-UI et Paper

Material-UI (https://material-ui.com/) est une gigantesque bibliothèque de composants implémentant les recommandations graphiques du Material Design (https://material.io/design/). Cela donne des composants relativement beaux, uniformes, et donnant un look moderne à votre application sans trop d'effort. Du simple texte à des composants complexes comme des onglets ou des accordéons, en passant par les boutons et les cases à cocher... difficile de ne pas trouver son bonheur.

Il est possible de personnaliser la charte graphique des composants grâce à un système de thèmes, et l'ensemble est responsive, ce qui rend la bibliothèque pratique pour réaliser des applications qui s'affichent aussi bien sur mobile.

Pour ce qui est du développement mobile, une bibliothèque similaire existe: Paper (https://callstack.github.io/react-native-paper/). Les composants sont relativement les mêmes et il est également possible d'utiliser des thèmes.

L'idéal serait d'avoir une bibliothèque fonctionnant à la fois sur Web et sur mobile mais, en attendant, le site de Paper décrit une procédure pour utiliser les composants avec React Native Web, un projet permettant de développer en React Native pour le Web, et ainsi faire sauter la dernière barrière entre les développements web et mobile.

C'est encore un peu laborieux, mais possible et très encourageant pour l'avenir de React, notamment grâce aux dernières versions d'Expo qui permettent, grâce à React Native Web, de générer à partir du même code une application compatible iOS, Android, et pour le Web.

4.2 React-Select et React-Table

À présent, voici deux bibliothèques qui, contrairement aux deux précédentes, ne proposent pas un grand nombre de composants, mais seulement un pour chacune.

React-Select (https://react-select.com) propose une alternative à la liste de sélection classique `<select>` en ajoutant par exemple la possibilité de modifier le formatage des options, de sélectionner plusieurs options, de les organiser de manière hiérarchique ou encore de les charger dynamiquement en appelant une API au fur et à mesure que l'utilisateur tape des caractères. Le composant est vraiment puissant et propose un très grand nombre de possibilités de personnalisation.

Si vous démarrez une application comportant plusieurs listes de sélection, envisagez d'utiliser React-Select dès le début, cela vous facilitera la vie par la suite.

Dans le même ordre d'idée, React-Table (https://github.com/tannerlinsley/react-table) permet d'afficher des tableaux riches, qui peuvent être triés, filtrés, paginés, chargés dynamiquement... bref, encore un composant personnalisable à l'extrême. Si vous souhaitez afficher des tableaux un peu plus complexes que ceux proposés nativement en HTML, ne vous lancez pas dans un composant *from scratch*, utilisez React-Table.

4.3 React-Intl pour internationaliser votre application

L'internationalisation d'une application est un besoin très courant, au point qu'il existe un grand nombre de moyens de le faire. Parmi les bibliothèques disponibles, React-Intl (https://github.com/yahoo/react-intl/) est simple à mettre en place, et permet de gérer les traductions et le formatage de données (nombres, dates...) en fonction de la langue de l'utilisateur.

Son utilisation repose sur le pattern *Provider/Consumer* vu au chapitre Écrire des composants réutilisables, ce qui fait qu'après initialisation au démarrage de l'application, les possibilités sont données à chaque composant d'utiliser les traductions et autres fonctions.

Encore une fois, si vous devez gérer plusieurs langues dans votre application, inutile de réinventer la roue, React-Intl (ou une autre bibliothèque) vous aidera pour cela.

5. Le mot de la fin

Vous voilà arrivé à la fin de ce livre, et je vous en félicite! J'espère que vous aurez eu autant de plaisir à le lire que j'en ai eu à l'écrire, et que vous vous sentez suffisamment à l'aise avec React pour continuer votre apprentissage par vous-même et par la pratique.

Lorsque je me suis lancé dans ce projet d'écriture, j'ai longtemps réfléchi aux sujets que j'aborderai, et j'ai fréquemment changé d'avis en cours de route. Il m'a fallu trouver un compromis entre un livre qui ne ferait qu'effleurer la surface de React, et un livre présentant une multitude de possibilités et de bibliothèques, mais qui serait dépassé dans quelques mois du fait de l'évolution démesurément rapide de l'écosystème.

J'ai cherché à présenter des bases solides qui ne reposeraient pas sur l'état de l'art à un instant T, mais qui étaient pertinentes au début de React, le sont encore aujourd'hui, et le seront généralement encore dans le futur.

Cela n'a pas été facile ; par exemple, l'annonce de l'arrivée des hooks dans React 16.8 a quelque peu perturbé ce que j'avais prévu et déjà écrit. Et même s'il ne faut pas céder à la tendance (on parle de *hype-driven development*), les hooks promettent tellement de changer les pratiques que je ne pouvais pas ne pas les intégrer à ce livre.

Il me reste à présent à vous souhaiter de prendre beaucoup de plaisir avec React. Restez informé de ce qui s'y passe, découvrez de nouvelles approches, et surtout, pratiquez ! D'ailleurs, n'hésitez pas à retourner à la liste de ressources disponibles sur le Web proposée dans l'avant-propos de ce livre pour aller plus loin avec React.

Annexes

1. Utiliser les React Dev Tools

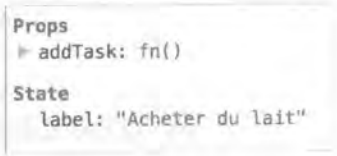
Les React Dev Tools (https://github.com/facebook/react-devtools) sont une extension pour les navigateurs Firefox et Chrome (et d'autres sous forme d'une application autonome) facilitant grandement l'écriture et la recherche de bugs sur une application React.

Une fois votre application lancée dans le navigateur, il suffit d'ouvrir les outils de développement classiques; un onglet **React** a normalement été ajouté par l'extension. Dans cet onglet, vous pouvez naviguer dans l'arbre des composants React de l'application, et voir les éléments générés dans le DOM :

Arbre des composants dans React Dev Tools

Mais la fonctionnalité la plus intéressante apparaît lorsque l'on sélectionne l'un de nos composants.

On peut alors voir dans la partie droite les valeurs de propriétés des composants, ainsi que le state courant :



Visualiser les propriétés et le state des composants

Lorsque votre application ne contient pas ce que vous attendez, le premier réflexe à avoir devra souvent être de vérifier que vos composants ont bien les propriétés ou state attendus. Et pour cela les React Dev Tools vous seront bien utiles.

2. Construire et déployer une application React

Afin de déployer une application React, il est auparavant nécessaire de construire un ensemble de fichiers qui peuvent être interprétés par un navigateur. Pour une application simple générée avec Parcel, comme vu dans le chapitre Découverte de React, ces fichiers seront :

- un fichier index.html ;
- un fichier CSS contenant tout le CSS importé depuis d'autres fichiers ;
- un fichier JavaScript contenant tout le JavaScript de notre application (JavaScript standard, c'est-à-dire notamment sans JSX) mais également celui des bibliothèques utilisées (notamment React).

Bien évidemment, Parcel est capable de nous générer tous ces fichiers. Nous avions dans le premier chapitre défini un script NPM `start: parcel public/index.html`. Il s'agit de la commande qui réalise cette génération, mais également lance un serveur de développement, et se met en attente pour regénérer les fichiers dès que les sources sont modifiées.

Pour simplement générer les fichiers, il est possible de lancer la commande `parcel --build public/index.html`. Les fichiers sont alors générés dans le répertoire dist, comme en mode développement. Cela peut mener à des erreurs (si l'on récupère les fichiers pour les mettre en production) ; c'est pourquoi je suggère d'utiliser un autre dossier, vidé avant la génération. Pour cela, on peut créer un nouveau script NPM :

```
"scripts": {
 "start": "parcel public/index.html",
 "build": "rm -rf build && parcel build public/index.html -d
build"
},
```

Il suffit alors de lancer la commande `yarn build` ou `npm run build` pour générer les fichiers. Il ne reste alors plus qu'à déployer ces fichiers sur un serveur pour rendre accessible votre application. Pour cela, comme une application React ne nécessite pas de traitements côté serveur pour être exécutée, tout hébergement de sites statiques peut convenir.

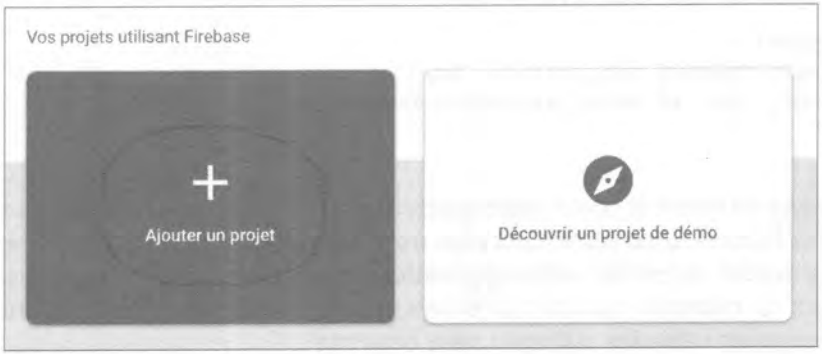
Par exemple, si vous êtes habitué à GitHub, vous pouvez utiliser GitHub Pages (https://pages.github.com/), ou encore héberger votre application sur Amazon S3. L'avantage est qu'un hébergement statique ne coûte généralement quasiment rien. Puis rien ne vous empêche ensuite de faire pointer un nom de domaine dessus.

3. Création d'une application Firebase

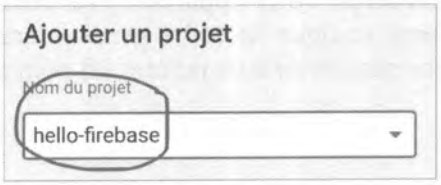
La procédure pour créer une application Firebase peut évoluer régulièrement, et il est tout à fait possible que cette procédure ait changé entre la dernière mise à jour de cette annexe et le moment où vous créez votre application. À ce jour, le site de Firebase est plutôt bien fait et intuitif, je n'ai pas de doute sur le fait que cette situation dure.

La première étape sera de vous connecter avec un compte Google (d'en créer un si vous n'en avez pas), puis de vous rendre à l'URL permettant d'accéder à la console d'administration de Firebase : console.firebase.google.com.

Suivez ensuite les étapes ci-dessous pour créer votre application, récupérer les informations nécessaires pour vous y connecter (un morceau de code JavaScript), et configurer l'authentification et la base de données temps réel (les deux fonctionnalités présentées au chapitre Gestion de formulaires et du routage).



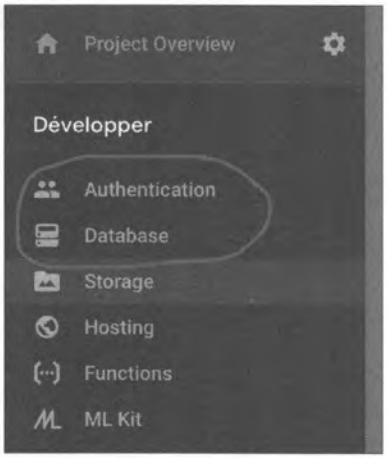
- Créez un nouveau projet Firebase.



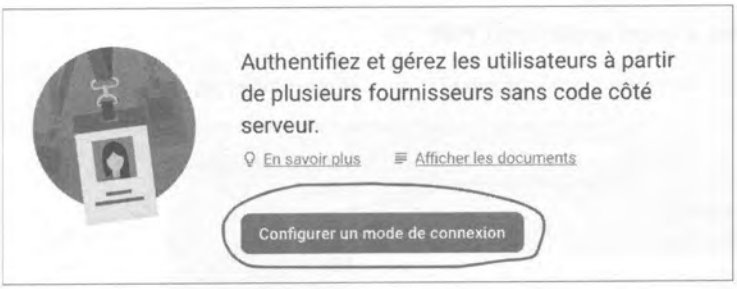
- Entrez le nom du projet.

- Sur le tableau de bord, cliquez sur l'icône **</>** pour obtenir les informations nécessaires pour connecter votre application.

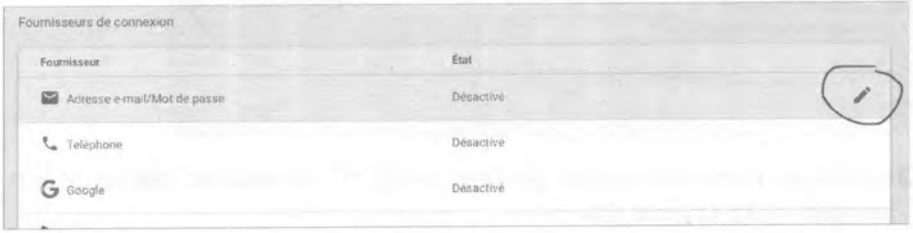
- Copiez le code commençant par "var config =". Vous aurez besoin de le coller dans votre application.



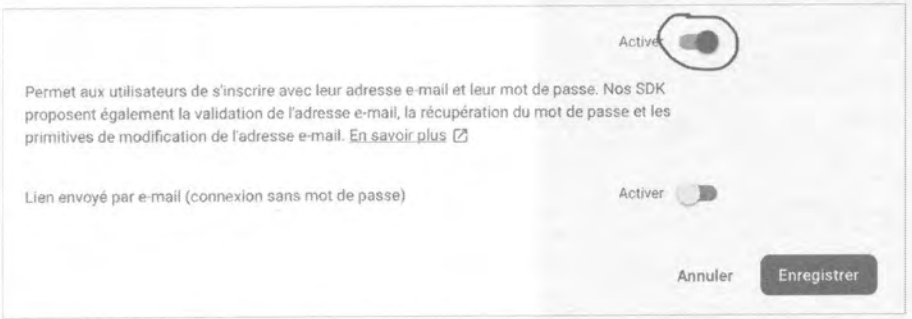
- La barre latérale permet de configurer l'authentification et la base de données à utiliser.



- Pour l'authentification, ajoutez un nouveau mode de connexion.

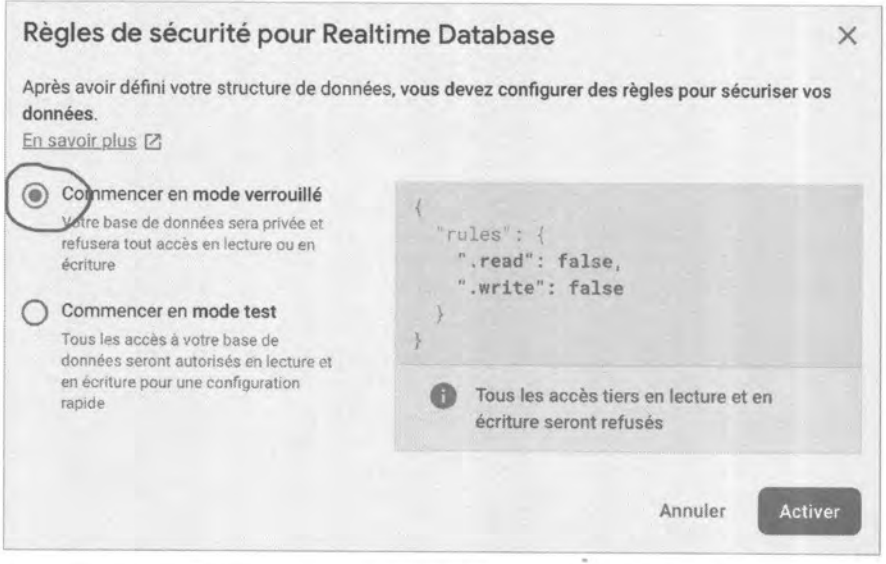


- Dans la liste des options proposées, éditez l'option **Adresse e-mail/mot de passe**.



- Puis activez cette option, et enregistrez.

- Pour la base de données, dans la section **Realtime Database**, créez une base de données.



- Choisissez de commencer en « mode verrouillé », puis validez en cliquant sur **Activer**.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

INFORMATIQUE
TECHNIQUE

React

Développez le Front End de vos applications web et mobiles avec JavaScript

Ce livre s'adresse aux **développeurs** qui souhaitent lever la complexité apparente du **framework Front End React** pour réaliser des **applications web et mobiles** bien architecturées et aisées à maintenir. Pour bien appréhender la lecture de ce livre, un minimum de connaissances sur le langage JavaScript, et en particulier sur les nouveautés apportées par ES2015, est un plus.

L'auteur commence par présenter les fonctionnalités natives de **React** avant d'expliquer comment la bibliothèque **Redux** permet de structurer et développer des applications plus complexes, notamment grâce aux apports de **Redux Saga**. Puis le lecteur étudie le développement mobile avec **React Native**, en détaillant notamment la mise en place de listes ou de la navigation.

Dans la suite du livre, l'auteur poursuit avec des notions plus avancées du développement avec React telles que le **routage**, la gestion de **formulaires**, les problématiques de **sécurité** ou l'utilisation de **Firebase** pour l'**authentification** ou le **stockage** de données distantes. L'auteur présente également **GraphQL** comme alternative à Firebase pour permettre l'appel à une API.

Dans les derniers chapitres, le lecteur trouvera les informations nécessaires pour développer des composants plus **faciles à maintenir grâce aux hooks**, ainsi que des pistes pour **apprendre à tester une application** développée avec React et Redux.

Tout au long du livre, les notions présentées sont accompagnées d'**exemples concrets** que le lecteur pourra mettre en pratique au fil de sa lecture. Des éléments complémentaires sont en téléchargement sur le site www.editions-eni.fr.

Ingénieur et développeur depuis près de dix ans, **Sébastien CASTIEL** est spécialisé dans le développement web et le développement Front End, notamment avec le langage JavaScript. Son envie de transmettre ses connaissances l'a tout naturellement conduit à se consacrer à l'écriture de ce livre pour partager au plus grand nombre son expertise sur le framework React. Il propose ainsi un livre 100% opérationnel pour permettre au lecteur de développer des applications web et mobiles performantes.

Sur **www.editions-eni.fr** :
➜ Le code source des exemples du livre.

Pour plus d'informations :

39 €

ISBN : 978-2-409-02272-2

9 782409 022722

www.editions-eni.fr