Table des matières

[Installation des outils requis 1](#_Toc84966575)

[1. NodeJS 1](#_Toc84966576)

[2. Un éditeur de texte 1](#_Toc84966577)

[3. Et ensuite ? 1](#_Toc84966578)

[Création du premier projet 2](#_Toc84966579)

[1. Explication du code 4](#_Toc84966580)

[2. Composants et propriétés 5](#_Toc84966581)

[Le langage JSX 7](#_Toc84966582)

[Un composant par fichier 11](#_Toc84966583)

[Ajouter du style 14](#_Toc84966584)

Installation des outils requis

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.

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.

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.

# 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](http://localhost:1234/), vous devriez voir le texte « *Hello !* ». Félicitations, c’est votre première application React !

## 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. 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.

# 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.

# 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 ?

# 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.