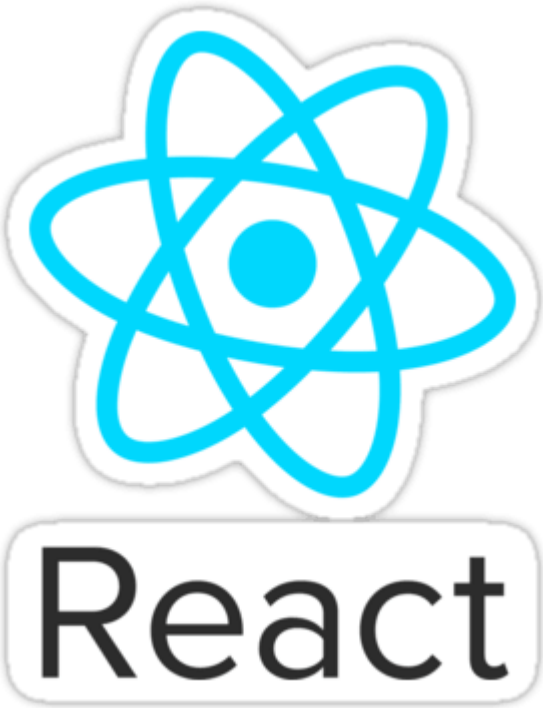
Maîtriser le framework JavaScript de Facebook



votre intervenant :

DESORBAIX Alexandre

Prenant à contrepied les modèles traditionnels, le Framework maintenu par Facebook favorise la simplicité et la performance des composants de RIA.

Vous apprendrez dans ce cours à développer des applications avec ReactJS, JSX et Flux/Redux et découvrirez le principe et les bénéfices du développement isomorphique.

Objectifs pédagogiques

- Développer avec ReactJS.
- Concevoir une SPA avec ReactJS et Flux.
- Comprendre le subset JavaScript JSX.
- Optimiser les performances des RIA.

Présentation

Participants. (Tour de table)

Développeurs JavaScript, architectes et chefs de projets web.

Prérequis.

Bonne connaissance de JavaScript, pratique du développement web.

Récapitulatif matinal.

Chaque journée commence par un exercice collectif de restitution des concepts abordés la veille. L'objectif étant de renforcer (répétition) les connaissances acquises et les utiliser comme socle pour la journée à venir.

Concertation personnelle.

Votre formateur passera vous assister individuellement aussi souvent que possible.
N'hésitez pas à le solliciter pendant la journée de formation ou pour revenir sur un point particulier en fin de journée.

Références à l'ouvrage et autres références

La formation est illustrée par la projection d'une version numérique (pdf) de votre support et l'utilisation d'autres ressources pertinentes (site internet, démonstration).

Autres références

- reactjs officiel
- codementor.io
- Playground
- Playground JSX
- Playground No JSX
- node.js
- npmjs
- Webpack

[TOC]

Introduction

ReactJS

React (appelé aussi React.js) est un moteur de rendu JavaScript qui se démarque par une architecture voulue **efficace et performante.**

Initialement créé par Facebook pour développer *le fil d'actualité de son réseau social*. **React est publié en open source en mai 2013,** sous Licence Apache 2.0.

La logique *best of breed*

React cherche à offrir la meilleure réponse technologique à une problématique précise. **Rendre le DOM rapidement**

React agit comme un **moteur de rendu** intermédiaire. Il s'agit d'**une librairie JavaScript** et non d'un framework. En termes de performance, **React optimise les opérations** sur le DOM en utilisant un **DOM virtuel.**

 react

Pour **exprimer la structure du Virtual DOM,** React utilise JSX. Un langage qui étend **JavaScript (subset)** avec une syntaxe déclarative permettant de définir le mode de rendu HTML du composant.

> - ReactJS permet de fabriquer des composants (Web).
> - Un composant ReactJS génère du code (HTML) à chaque changement d'état.
> - ReactJS ne gère que la partie interface de l'application Web (Vue).
> - ReactJS peut être utilisé avec une autres bibliothèque ou un framework (AngularJS).

> En tant que **Librairie JavaScript ReactJS** statisfait aux problématiques de développement en utilisant `l'écosystème industrialisé moderne`

 séparation

Avant de commencer,

Démistyfication des outils

&

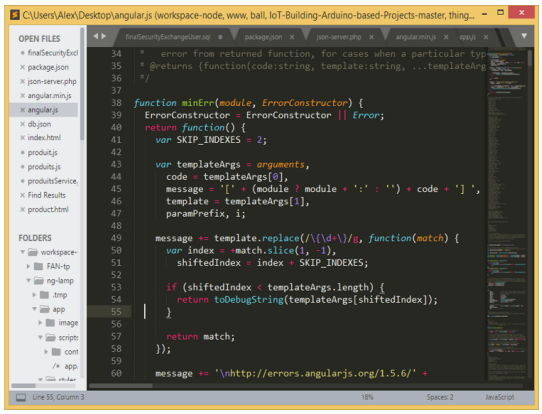
Configuration du poste

---

les IDEs

Les IDE ou interfaces de développements sont les outils d'édition de code. Plus ou moins avancées mais surtout du plus ou moins onéreux aussi. plusieurs outils sont heureusement gratuit et assez complet pour faciliter l'edition de code à déstination du web :

- sublime Text(shareware)
- eclipse
- atom *visual studio comunity
- visual studio code

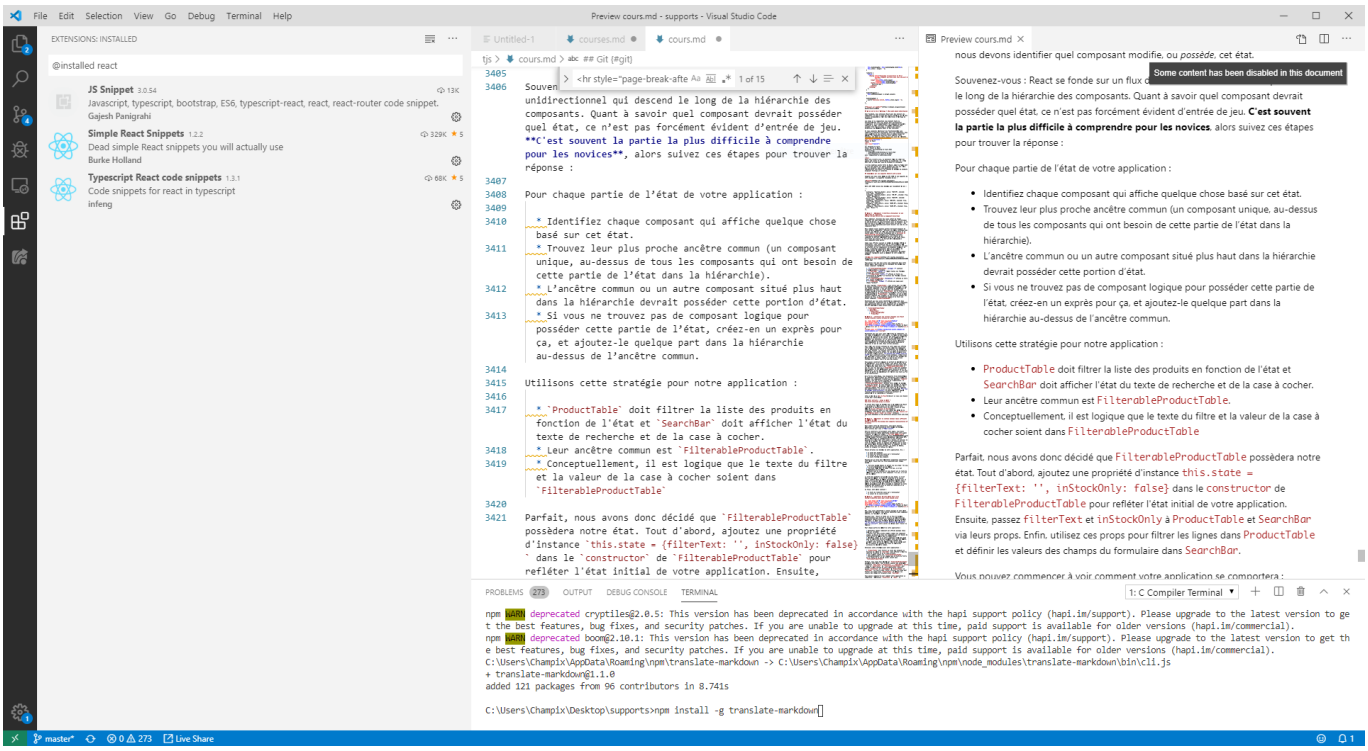


Penchons nous sur visual studio code et ses extensions & snipets gratuits pour un petit peu tous les besoins possibles.

vs-code {#vscode-js}

L'interface de développement (IDE) selon microsoft :

- Multiplateforme
- multilanguage
- support git natif
- plugins
- snippets
- terminal (cmd, shell, ...)



---

vs-code plugins {#vs-plugin}

Vs code est un bonne outils mais il n'en serait rien si les extension n'etaient pas aussi nombreuse. toutes ces extension, qui elles aussi sont multiplateforme, etendent vs code de snippet (petit morceaux de code avec accès rapide) mais aussi de prise en charge de language ou meme de debugger.

ces Extensions sont developper par des tiers donc il y a de trés bon projets et d'autres moins interessants.

Nous allons regarder de plus pret quelques extensions

- react
- react-native
- redux

Git & versionning {#git}

Le versionning est le faite marquer les differentes évolutions d'une application, au cours de son developpement, comme des changements détats (lignes de codes, ...) de la version précédente, qui elle même est deja un changement de la versione précedentes. Cette logique oeut s'incrire aussi dans une démarche agile. cette methode ou chaque étapes peut etre matériaiisées par une suite de changement permet aussi d'afficher des différentiels de ces changements.

Le systeme de versionning le plus en vogue du moment avec prise en chareg :

- pull, push, sync
- branches
- revert, approuve,
- pull request, merge
- ...

Trois grandes plateformes sont présentes sous différents formats. certain existent en serveurs privées, dautres en ligne et certaines proposent les deux formats. dans ces plateformes 3 reviennent souvent :

Concentrons nous plus particulierement sur l'une d'elle : bitbucket.org.

Bitbucket {#git-bitbucket}

cette plateforme propose des espces de stockage pour le versionnage de vos applis. cette plateforrme supporte plusieurs standards du versionning : git, mercurial. Elle offre une accès gratuit à un éspace pour creer un nombre de repository illimités limité seulement par leur tailles 1Go max (par fichiers et par repositories).

pour creer un compte : https://bitbucket.org/account/signup/

pour se logger : https://bitbucket.org/account/signin/

A vous creer votre compte

Création 1ere repository

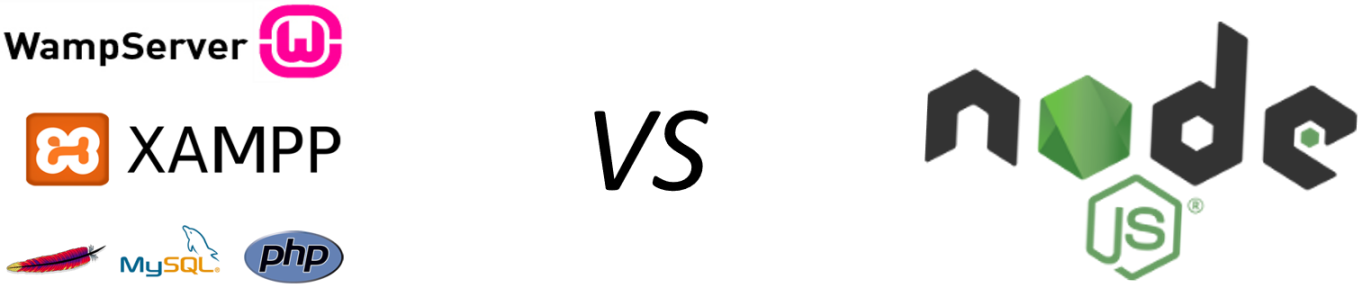
- Edit, commit, approuve du readme.md
- Clone local de la repository

Ouverture du répertoire sous Visual Studio Code

- Edition du readme.md sou vscode
- 1er commit & push

Les serveurs de dev

Il éxiste plein de manières de desservir des fichiers http sur un réseau. Beaucoup de technologies font plus ou moins la même chose... plus ou moins. L'exemple qui nous vient tous en tête chez les developpeurs web c'est le fameux WAMP, ou LAMP pour les linuxiens. Mais d'autres téchnologies marche bien et offre surtout une extensibilité quasie infinie. c'est le cas de Node Node.js avec ces "plugins". Car Node.js n'eest pas un serveur web par définition, nous regarderons donc de plus pres certains projets intéressant pour notre usage.



Wamp {#wamp}

Wamp est un rassemblement de plusieurs logiciels serveurs. il est composé de :

- apache

  pour repondre aux demandes faites pour HTTP. Exemple
  requette HTTP1.0 GET /index.php et si besoin est renvoyé le retour de la requette assemblée ou non par php

- php

  pour repondre aux demandes faites par apache pour dynamiser le contenu d'une requette la route vers
  un fichier php portant l'extension php directement .

- mysql

  C'est un SGBDR où S.ysteme de G.estion de B.ases de D.onnées R.eletionnel. il permet le stockage et l'interraction avec celles ci par le biais du langage SQL. Il est dit Relationnel car il permet de faire la relation entre plusieurs table pour limiter le stockage et la redondance d'information

Tout cette ensemble est bien pratique mais un peut lourd pour notre usage. pour faire une application js nous navons pas besoin de php et la base de donnees peut exister sous d'autre forme que mysql. pour ces raisons nous selectionnerons plutot node js qui en plus d'etre ecris pour le js nous propose une serie de logiciels serveur remplissant tous nos besoins.

Node js {#node-js}

&

Node.js est avant tout un moteur de JS qui opère en dehors de tout client de navigation. il permet donc l'interrprétation et l'execution de code js. Sont role se limitant à etre un moteur pour executer du js. il est donc dépendent de differents projets pour prendre chacun des rôles que l'on souhaite lui faire gérrer.

les dépendences c'est le role de npm qui permet de "tirer" des dépendences et les installer soit pour un projet, soit pour disponible pour un utilisateur et rend disponible et pret à executer les projets téléchargés.

Il existe des outils concurents pour gerrer ces dépencdences es : yarn. Ce gestionnaire est celui qui est utilisé par la dernière version de create-react-app lorsque il assemble un nouveau projet.

Les outils pour notre projets

ESLint / TsLint {#js-lint}

ESLint

se définit sur sont github https://github.com/eslint/eslint comme :

> A fully pluggable tool for identifying and reporting on patterns in JavaScript

soit un outil pluggable qui **identitfie** et **avertie** sur les pattern d'écriture de code utilisés dans le code JS

TsLint

se définit quant à lui sur son site https://palantir.github.io/tslint/ comme :

> TSLint is an extensible static analysis tool that checks TypeScript code for readability, maintainability, and functionality errors.

soit un outils d'analyse qui verifie du code TypeScript sur les points et erreurs suivantes :

- lisibilité
- maintenabilité
- fonctionabilité

le choix de l'un ou de l'autre rivens au choix de langage utilisée. pour votre projets.

use strict {#js5-strict}

Cette simple ligne permet de définir une zone ou un segment de code **es5** en indiquant qu'il doit répondre à des règles d'optimisation d'ecriture. Ces règles facilite l'interpretation & l'execution du code. il est souvent

donseiller de se placer dans des zones de code dite strict. Ce mode n'a pas pour but d'éxécuter du code non strict 😅

```
"use strict";
```

babel {#babel}

Babel est le transpileur autrement dit l'outil qui se charge de rendre compatible notre code. le jsx, tsx, ts, es 6 et autres technologies non pris en charge dans les navigateurs vont être convertis en code comprehensible c'est à dire transformés vers differents niveaux de languages js, et permettre son fonctionnement dans de "vieux" navigateurs.

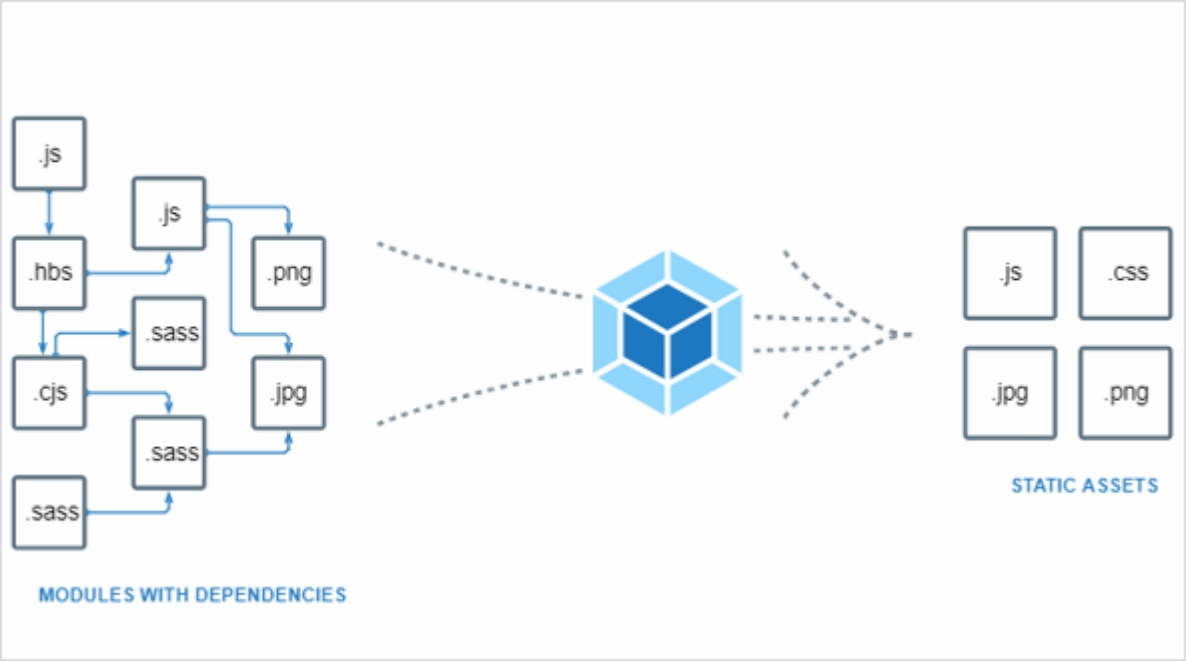
Il est un des projets node js aborder dans la partie précédente sur node js. Grace a la fonction create-react-app cette dépéndence serat automatiiquement instaler dans notre projet.

webpack

webpack vas permettre d'assembler le code pour les préprocesseur tels que lass ou less qui generre du css qui doit etre pré-génerrer pour être fonctionnel.

Tout comme Babel ce projet sera automatiquement installer et configuer pour notre projet par creat-react-app. ne détaillant pas ici le less, et le sass je vous invite à visiter pour affiner vos connaissances sur cette outil

https://webpack.js.org/


.js
.hbs
.cjs
.sass
.js
.png
.sass
.jpg
.sass
MODULES WITH DEPENDENCIES
.js
.css
.jpg
.png
STATIC ASSETS

Bien démarrer

Cette page est un aperçu de la documentation React et des ressources associées.

React est une bibliothèque JavaScript pour la construction d'interfaces utilisateur (UI). Pour découvrir à quoi sert React, allez sur notre page d'accueil ou dans le tutoriel.

Essayer React {#try-react}

React a été conçu dès le départ pour une adoption progressive, et **vous pouvez utiliser React *a minima* ou autant que nécessaire.** Que vous souhaitiez avoir un aperçu de React, ajouter de l'interactivité à une simple page HTML ou démarrer une application React complète, les liens de cette section vous aideront à bien démarrer.

Terrains de jeu en ligne {#online-playgrounds}

Si vous souhaitez tester React, vous pouvez utiliser un terrain de jeu en ligne. Essayez un modèle *Hello World* sur CodePen, CodeSandbox ou Glitch.

Si vous préférez utiliser votre propre éditeur de texte, vous pouvez aussi télécharger ce ficher HTML, l'éditer et l'ouvrir à partir du système de fichiers local dans votre navigateur. Il transforme le code à la volée lors de l'exécution, ce qui est particulièrement lent. Pour cette raison, nous vous recommandons de ne l'utiliser que pour des démos simples.

Ajouter React à un site Web {#add-react-to-a-website}

Vous pouvez ajouter React à une page HTML en une minute et ensuite étendre progressivement sa présence ou la limiter à quelques éléments d'interface dynamiques.

Créer une nouvelle application React {#create-a-new-react-app}

Lorsque vous démarrez un projet React, une simple page HTML avec des balises de script reste peut-être la meilleure option. Ça ne prend qu'une minute à mettre en place !

Au fur et à mesure que votre application grandit, vous voudrez peut-être envisager une configuration plus intégrée. Il y a plusieurs boîtes à outils JavaScript que nous conseillons pour des applications plus importantes. Chacune d'entre elles peut fonctionner avec peu ou pas de configuration et vous permet de tirer pleinement parti du riche écosystème de React.

Apprendre React {#learn-react}

Les gens qui découvrent React viennent d'horizons et de styles d'apprentissage variés. Que vous préfériez une approche plutôt théorique ou pratique, nous espérons que vous trouverez cette section utile.

- Si vous préférez **apprendre en faisant**, commencez par notre tutoriel.

- Si vous préférez **apprendre les concepts étape par étape**, commencez par notre guide des fondamentaux.

Comme toute technologie que vous ne connaissez pas encore, React a une certaine courbe d'apprentissage. Avec de la pratique et un peu de patience, vous *arriverez* à le maîtriser.

Premiers exemples {#first-examples}

La page d'accueil de React contient quelques petits exemples React que vous pouvez modifier directement en ligne. Même si vous ne connaissez rien encore à React, essayez de changer le code de ces exemples et voyez comment cela affecte le résultat.

React pour les débutant·e·s {#react-for-beginners}

Si vous trouvez que la documentation React va un peu trop vite pour vous, consultez cet aperçu de React par Tania Rascia. Il présente les concepts les plus importants de React de façon détaillée et conviviale pour les débutant·e·s. Une fois que vous avez terminé, essayez à nouveau la documentation !

React pour les designers {#react-for-designers}

Si vous avez plutôt une expérience de designer, ces ressources sont un excellent moyen de commencer.

Ressources JavaScript {#javascript-resources}

La documentation de React suppose une certaine habitude de la programmation en langage JavaScript. Pas besoin d'être un·e expert·e, mais il est plus difficile d'apprendre à la fois React et JavaScript.

Nous vous recommandons de parcourir cet aperçu JavaScript pour vérifier votre niveau de connaissances. Ça vous prendra entre 30 minutes et une heure, mais vous vous sentirez plus en confiance pour apprendre React.

> Astuce
>
> Chaque fois que quelque chose en JavaScript vous semble déroutant, MDN et javascript.info sont d'excellents sites pour vérifier. On trouve aussi des forums de soutien communautaire dans lesquels vous pouvez demander de l'aide.

Tutoriel pratique {#practical-tutorial}

Si vous préférez **apprendre par la pratique**, allez voir notre tutoriel pratique. Dans ce tutoriel, nous construisons un jeu de morpion en React. Vous pourriez être tenté·e de l'ignorer sous prétexte que vous ne construisez pas de jeux—mais donnez-lui sa chance. Les techniques que vous apprendrez dans ce tutoriel sont fondamentales pour la construction de n'importe quel type d'appli React, et les maîtriser vous apportera une compréhension profonde de React.

Guide étape par étape {#step-by-step-guide}

Si vous préférez **apprendre les concepts étape par étape** notre guide des fondamentaux est le meilleur endroit pour commencer. Chaque chapitre s'appuie sur les connaissances introduites dans les précédents, afin que vous ne manquiez de rien pour avancer.

Penser en React {#thinking-in-react}

De nombreux utilisateurs de React estiment que c'est en lisant Penser en React que React a enfin « cliqué » pour eux. C'est probablement le plus ancien guide pas-à-pas sur React, mais il est toujours aussi pertinent.

Cours recommandés {#recommended-courses}

Parfois les gens préféreront des livres et cours vidéo créés par des tiers à la documentation officielle. Nous maintenons une liste de ressources fréquemment recommandées, dont certaines sont gratuites.

Guides avancés {#advanced-concepts}

Une fois que vous serez à l'aise avec les fondamentaux et que vous aurez joué un peu avec React, vous pourriez être intéressé·e par des sujets plus avancés. Cette section présente les fonctionnalités puissantes, mais moins utilisées, de React, telles que le contexte et les refs.

Référence de l'API {#api-reference}

Cette section de la documentation est utile lorsque vous souhaitez en savoir plus sur une API React spécifique. Par exemple, la référence de l'API `React.Component` peut vous fournir des détails sur le fonctionnement de `setState()`, et sur les utilités respectives des différentes méthodes de cycle de vie.

Glossaire et FAQ {#glossary-and-faq}

Le glossaire contient un aperçu des termes les plus couramment employés dans la documentation de React. Il y a également une section FAQ dédiée aux questions et réponses courtes sur des sujets fréquents, tels que faire des requêtes AJAX, gérer l'état local des composants, et la structure de fichiers.

Se tenir au courant {#staying-informed}

Le blog React est la source officielle des mises à jour, par l'équipe de React. Tout ce qui est important, y compris les notes de publication ou les avis de dépréciation, y est publié en priorité.

Vous pouvez également suivre le compte @reactjs sur Twitter, mais rien d'essentiel ne vous échappera si vous ne lisez que le blog.

Toutes les versions de React ne méritent pas leur propre article de blog, mais vous pouvez trouver un *changelog* détaillé pour chaque version dans le fichier `CHANGELOG.md` du dépôt React, ainsi que sur la page des *Releases*.

Documentation versionnée {#versioned-documentation}

Cette documentation reflète toujours la dernière version stable de React. Depuis React 16, vous pouvez trouver les anciennes versions de la documentation sur une page séparée. Notez que la documentation des versions antérieures est figée au moment de la publication et n'est plus mise à jour par la suite.

Quelque chose vous manque ? {#something-missing}

Si quelque chose manque dans la documentation ou si vous en trouvez une partie déroutante, veuillez créer une *issue* sur le dépôt de la documentation avec vos suggestions d'amélioration, ou tweetez en mentionnant le compte @reactjs. Nous adorons avoir de vos nouvelles !

Créer une nouvelle appli React

Utilisez une boîte à outils intégrée pour la meilleure expérience utilisateur et développeur possible.

Cette page décrit quelques boîtes à outils populaires qui facilitent les tâches telles que :

- La montée à l'échelle avec de nombreux fichiers et composants.
- L'utilisation de bibliothèques tierces depuis npm.
- La détection précoce des erreurs courantes.
- L'édition à la volée du CSS et du JS en développement.
- L'optimisation pour la production.

Les boîtes à outils recommandées sur cette page **ne nécessitent aucune configuration pour démarrer**.

Vous n'avez peut-être pas besoin d'une boîte à outils {#you-might-not-need-a-toolchain}

Si vous ne rencontrez pas les problèmes décrits ci-dessus ou si vous n'êtes pas encore à l'aise avec l'utilisation d'outils JavaScript, envisagez d'ajouter React comme une simple balise `<script>` sur une page HTML, éventuellement avec du JSX.

C'est également **la façon la plus simple d'intégrer React au sein d'un site web existant**. Vous pourrez toujours étendre votre outillage si ça vous semble utile !

Boîtes à outils recommandées {#recommended-toolchains}

L'équipe React recommande en premier lieu ces solutions :

- Si vous **apprenez React** ou **créez une nouvelle application web monopage**, alors utilisez Create React App.
- Si vous construisez un **site web rendu côté serveur avec Node.js**, essayez Next.js.
- Si vous construisez un **site web statique orienté contenu**, essayez Gatsby.
- Si vous construisez une **bibliothèque de composants** ou une **intégration avec du code déjà existant**, essayez des boîtes à outils plus flexibles.

Create React App {#create-react-app}

Create React App est un environnement confortable pour **apprendre React**, et constitue la meilleure option pour démarrer **une nouvelle application web monopage** en React.

Il configure votre environnement de développement de façon à vous permettre d'utiliser les dernières fonctionnalités de JavaScript, propose une expérience développeur agréable et optimise votre application pour la production. Vous aurez besoin de Node >= 8.10 et de npm >= 5.6 sur votre machine. Pour créer un projet, exécutez :

```
npx create-react-app mon-app
cd mon-app
npm start
```

> Remaque :

> `npx` sur la première ligne n'est pas une faute de frappe -- c'est un exécuteur de paquets qui est inclus dans npm 5.2+.

Create React App ne prend pas en charge la logique côté serveur ni les bases de données ; il crée simplement une chaîne de construction pour la partie frontale, de sorte que vous pouvez utiliser le serveur de votre choix. Sous le capot, il utilise Babel et webpack, mais vous n'avez pas besoin de connaître ces outils.

Lorsque vous êtes prêt·e à déployer en production, exécutez `npm run build` pour créer une version optimisée de votre application dans le répertoire `build`. Vous pouvez en apprendre davantage sur Create React App dans son README et son guide utilisateur.

Next.js {#nextjs}

Next.js est un framework populaire et léger pour les **applications statiques rendues côté serveur** construites avec React. Il fournit des solutions prêtes à l'emploi pour **les styles et le routage**, et suppose que vous utilisez Node.js comme environnement serveur.

Apprenez Next.js grâce à son guide officiel.

Gatsby {#gatsby}

Gatsby est la meilleure option pour créer des **sites web statiques** avec React. Il vous permet d'utiliser des composants React, mais génère du HTML et du CSS pré-rendus afin de garantir le temps de chargement le plus rapide.

Apprenez Gatsby grâce à son guide officiel et à une collection de kits de démarrage.

Boîtes à outils plus flexibles {#more-flexible-toolchains}

Les boîtes à outils suivantes offrent plus de flexibilité et de choix. Nous les recommandons pour les utilisateurs expérimentés :

- **Neutrino** combine la puissance de webpack avec la simplicité des préréglages. Il inclut un préréglage pour les applications React et les composants React.
- **nwb** est particulièrement utile pour publier des composants React sur npm. Il peut également être utilisé pour créer des applications React.
- **Parcel** est un *bundler* d'applications web rapide et sans configuration qui fonctionne avec React.
- **Razzle** est un framework de rendu côté serveur qui ne requiert aucune configuration, mais offre plus de flexibilité que Next.js.

Créer une boîte à outils à partir de zéro {#creating-a-toolchain-from-scratch}

Une boîte à outils de construction en JavaScript comprend généralement :

- Un **gestionnaire de paquets**, tel que Yarn ou npm. Il vous permet de tirer parti d'un vaste écosystème de paquets tiers, et de les installer ou les mettre à jour facilement.
- Un ***bundler***, tel que webpack ou Parcel. Il vous permet d'écrire du code modulaire et de le regrouper en petits paquets permettant d'optimiser le temps de chargement.

- Un **compilateur** tel que Babel. Il vous permet d'écrire du JavaScript moderne qui fonctionnera quand même dans les navigateurs les plus anciens.

Si vous préférez configurer votre propre boîte à outils JavaScript à partir de zéro, jetez un œil à ce guide qui re-crée certaines des fonctionnalités de Create React App.

Pensez à vous assurer que votre outillage personnalisé est correctement configuré pour la production.

Démat monde

Le plus petit exemple de React ressemble à ceci :

```
ReactDOM.render(
  <h1>Bonjour, monde !</h1>,
  document.getElementById('root')
);
```

Il affiche un titre qui dit « Bonjour, monde ! » sur la page.

Essayer sur CodePen

Cliquez sur le lien ci-dessus pour ouvrir un éditeur en ligne. Vous êtes libres de faire quelques changements et de voir comment ils affectent l'affichage. La plupart des pages de ce guide auront des exemples modifiables comme celui-ci.

Comment lire ce guide {#how-to-read-this-guide}

Dans ce guide, nous examinerons les blocs qui constituent une application React : les éléments et les composants. Une fois que vous les aurez maîtrisés, vous pourrez créer des applications complexes à partir de petits blocs réutilisables.

> Astuce
>
> Ce guide est destiné aux personnes qui préfèrent **apprendre étape par étape**. Si vous préférez apprendre par la pratique, allez voir notre tutoriel. Vous trouverez peut-être que les deux sont complémentaires.

Ceci est le premier chapitre d'un guide étape par étape à propos des concepts principaux de React. Vous pouvez trouver une liste des chapitres dans la barre latérale de navigation. Si vous lisez ceci depuis un appareil mobile, vous pouvez accéder à la navigation en appuyant sur le bouton situé dans le coin en bas à droite de votre écran.

Chacun des chapitres de ce guide s'appuie sur les connaissances introduites dans les chapitres précédents. **Vous pouvez apprendre l'essentiel de React en lisant les chapitres du guide « Fondamentaux » dans l'ordre où ils apparaissent dans la barre latérale.** Par exemple, le prochain chapitre s'intitule « Introduction à JSX ».

Niveau de connaissances supposé {#knowledge-level-assumptions}

React est une bibliothèque JavaScript, donc nous supposerons que vous avez une compréhension décente du langage JavaScript. **Si vous ne vous sentez pas à l'aise, nous vous recommandons de passer par un tutoriel JavaScript pour vérifier votre niveau de connaissances** et vous permettre de suivre ce guide sans être perdu·e. Il vous prendra entre 30 minutes et une heure environ, mais au moins vous n'aurez pas le sentiment d'apprendre React et JavaScript en même temps.

> Remarque
>
> Ce guide utilise occasionnellement quelques nouvelles syntaxes de JavaScript dans les exemples. Si vous n'avez pas travaillé avec JavaScript ces dernières années, ces trois points devraient vous aider.

Commençons ! {#lets-get-started}

Continuez à défiler pour atteindre le lien vers le prochain chapitre de ce guide juste avant le pied de page.

Introduction à JSX

Observez cette déclaration de variable :

```
const element = <h1>Bonjour, monde !</h1>;
```

Cette drôle de syntaxe n'est ni une chaîne de caractères ni du HTML.

Ça s'appelle du JSX, et c'est une extension syntaxique de JavaScript. Nous recommandons de l'utiliser avec React afin de décrire à quoi devrait ressembler l'interface utilisateur (UI). JSX vous fait sûrement penser à un langage de balisage, mais il recèle toute la puissance de JavaScript.

JSX produit des « éléments » React. Nous verrons comment les retranscrire dans le DOM dans la prochaine section. Dans la suite de ce document, nous verrons les bases de JSX dont vous aurez besoin pour bien démarrer.

Pourquoi JSX ? {#why-jsx}

Le fonctionnement d'une UI conditionnera toujours les logiques de rendu, de la gestion des événements à la préparation des données pour l'affichage, en passant par l'évolution de l'état au fil du temps. React a choisi d'assumer pleinement cet état de fait.

Au lieu de séparer artificiellement les *technologies* en mettant le balisage et la logique dans des fichiers séparés, React sépare les *préoccupations* via des unités faiblement couplées appelées « composants », qui contiennent les deux. Nous reviendrons sur les composants dans une prochaine section, mais si l'idée d'injecter des balises dans du JS vous met mal à l'aise, cette présentation vous fera peut-être changer d'avis.

React ne vous oblige pas à utiliser JSX, mais la plupart des gens y trouvent une aide visuelle quand ils manipulent l'interface utilisateur dans le code JavaScript. Ça permet aussi à React de produire des messages d'erreurs et d'avertissements plus utiles.

Ceci étant posé, commençons !

Utiliser des expressions dans JSX {#embedding-expressions-in-jsx}

Dans l'exemple suivant, nous déclarons une variable appelée `name` et nous l'utilisons ensuite dans JSX en l'encadrant avec des accolades :

```
const name = 'Clarisse Agbegnenou';
const element = <h1>Bonjour, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Vous pouvez utiliser n'importe quelle expression JavaScript valide dans des accolades en JSX. Par exemple, `2 + 2`, `user.firstName`, ou `formatName(user)` sont toutes des expressions JavaScript valides.

Dans l'exemple suivant, on intègre le résultat de l'appel d'une fonction JavaScript, `formatName(user)`, dans un élément `<h1>`.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Kylian',
  lastName: 'Mbappé'
};

const element = (
  <h1>
    Bonjour, {formatName(user)} !
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Essayer sur CodePen

On découple le JSX en plusieurs lignes pour une meilleure lisibilité. Par la même occasion, nous recommandons également de le mettre entre parenthèses afin d'éviter les pièges d'insertion de point-virgule automatique, même si cette pratique n'est pas obligatoire.

JSX n'est rien d'autre qu'une expression {#jsx-is-an-expression-too}

Après la compilation, les expressions JSX deviennent de simples appels de fonctions JavaScript, dont l'évaluation renvoie des objets JavaScript.

Ça signifie que vous pouvez utiliser JSX à l'intérieur d'instructions `if` ou de boucles `for`, l'affecter à des variables, l'accepter en tant qu'argument, et le renvoyer depuis des fonctions :

```
function getGreeting(user) {
  if (user) {
    return <h1>Bonjour, {formatName(user)} !</h1>;
  }
  return <h1>Bonjour, Belle Inconnue.</h1>;
}
```

Spécifier des attributs en JSX {#specifying-attributes-with-jsx}

Vous pouvez utiliser des guillemets pour spécifier des littéraux chaînes de caractères dans les attributs :

```
const element = <div tabIndex="0"></div>;
```

Vous pouvez aussi utiliser des accolades pour utiliser une expression JavaScript dans un attribut :

```
const element = <img src={user.avatarUrl}></img>;
```

Ne mettez pas de guillemets autour des accolades quand vous utilisez une expression JavaScript dans un attribut. Vous pouvez utiliser soit des guillemets (pour des valeurs textuelles) soit des accolades (pour des expressions), mais pas les deux à la fois pour un même attribut.

> **Attention :**
>
> Dans la mesure où JSX est plus proche de JavaScript que de HTML, React DOM utilise la casse `camelCase` comme convention de nommage des propriétés, au lieu des noms d'attributs HTML.
>
> Par exemple, `class` devient `className` en JSX, et `tabindex` devient `tabIndex`.

Spécifier des éléments enfants en JSX {#specifying-children-with-jsx}

Si une balise est vide, vous pouvez la fermer immédiatement avec `/>`, comme en XML :

```
const element = <img src={user.avatarUrl} />;
```

Les balises JSX peuvent contenir des enfants :

```
const element = (
  <div>
    <h1>Bonjour !</h1>
    <h2>Content de te voir ici.</h2>
  </div>
);
```

JSX empêche les attaques d'injection {#jsx-prevents-injection-attacks}

Vous ne risquez rien en utilisant une saisie utilisateur dans JSX :

```
const title = response.potentiallyMaliciousInput;
// Ceci est sans risque :
const element = <h1>{title}</h1>;
```

Par défaut, React DOM échappe toutes les valeurs intégrées avec JSX avant d'en faire le rendu. Il garantit ainsi que vous ne risquez jamais d'injecter quoi que ce soit d'autre que ce vous avez explicitement écrit dans votre

application. Tout est converti en chaîne de caractères avant de produire le rendu. Ça aide à éviter les attaques XSS (cross-site-scripting).

JSX représente des objets {#jsx-represents-objects}

Babel compile JSX vers des appels à `React.createElement()`.

Ces deux exemples sont identiques :

```
const element = (
  <h1 className="greeting">
    Bonjour, monde !
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Bonjour, monde !'
);
```

`React.createElement()` effectue quelques vérifications pour vous aider à écrire un code sans bug, mais pour l'essentiel il crée un objet qui ressemble à ceci :

```
// Remarque : cette structure est simplifiée
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Bonjour, monde !'
  }
};
```

Ces objets sont appelés des « éléments React ». Vous pouvez les considérer comme des descriptions de ce que vous voulez voir sur l'écran. React lit ces objets et les utilise pour construire le DOM et le tenir à jour.

Nous explorerons la retranscription des éléments React dans le DOM dans la prochaine section.

> **Astuce :**
>
> Nous recommandons d'utiliser la définition de langage « Babel » dans votre éditeur préféré, afin que les codes ES6 et JSX soient correctement colorisés. Ce site utilise le thème de couleurs Oceanic Next, qui est compatible avec ce mode syntaxique.

JSX dans le détail

Fondamentalement, JSX fournit juste du sucre syntaxique pour la fonction `React.createElement(component, props, ...children)`.

Le code JSX :

```
<MyButton color="blue" shadowSize={2}>
  Cliquez ici
</MyButton>
```

est compilé en :

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Cliquez ici'
)
```

Il est aussi possible d'utiliser la balise auto-fermante si il n'y a pas d'enfants. Donc :

```
<div className="sidebar" />
```

est compilé en :

```
React.createElement(
  'div',
  {className: 'sidebar'},
  null
)
```

Si vous souhaitez voir comment certains éléments JSX spécifiques sont compilés en JavaScript, vous pouvez utiliser le compilateur Babel en ligne.

Spécifier le type d'un élément React {#specifying-the-react-element-type}

La première partie d'une balise JSX détermine le type de l'élément React en question.

Les types commençant par une lettre majuscule indiquent que la balise JSX fait référence à un composant React. Ces balises sont compilées en références directes à la variable nommée, donc si vous utilisez l'expression JSX `<Foo />`, l'identifiant `Foo` doit être présent dans la portée.

React doit être présent dans la portée {#react-must-be-in-scope}

Étant donné que JSX se compile en appels à `React.createElement`, la bibliothèque `React` doit aussi être présente dans la portée de votre code JSX.

Par exemple, les deux imports sont nécessaires dans le code ci-dessous même si `React` et `CustomButton` ne sont pas directement référencés depuis JavaScript :

```
import React from 'react';
import CustomButton from './CustomButton';

function WarningButton() {
  // return React.createElement(CustomButton, {color: 'red'}, null);
  return <CustomButton color="red" />;
}
```

Si vous n'utilisez pas un *bundler* JavaScript mais que vous chargez React à partir d'une balise `<script>`, il est déjà dans la portée en tant que variable globale `React`.

Utiliser la notation à points pour un type JSX {#using-dot-notation-for-jsx-type}

Vous pouvez également référencer un composant React en utilisant la notation à points dans JSX. C'est pratique si vous avez un seul module qui exporte de nombreux composants React. Par exemple si `MyComponents.DatePicker` est un composant, vous pouvez directement l'utiliser dans JSX comme ceci :

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imaginez un sélecteur de dates {props.color} ici.</div>;
  }
}

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

Les composants utilisateurs doivent commencer par une majuscule {#user-defined-components-must-be-capitalized}

Quand un élément commence par une lettre minuscule il fait référence à un composant natif tel que `<div>` ou `<span>`, ce qui donne une chaîne de caractères `'div'` ou `'span'` passée à `React.createElement`. Les types qui commencent par une lettre majuscule comme `<Foo />` sont compilés en `React.createElement(Foo)` et correspondent à un composant défini ou importé dans votre fichier JavaScript.

Nous recommandons de nommer vos composants avec une initiale majuscule. Si vous avez un composant qui démarre avec une lettre minuscule, affectez-le à une variable avec une initiale majuscule avant de l'utiliser dans votre JSX.

Par exemple, ce code ne s'exécutera pas comme prévu :

```
import React from 'react';

// Faux ! C’est un composant, il devrait commencer par une lettre majuscule
:
function hello(props) {
  // Correct ! Cette utilisation de <div> fonctionne car div est une balise
HTML valide :
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Faux ! React pense que <hello /> est une balise HTML car il commence
pas par une majuscule :
  return <hello toWhat="World" />;
}
```

Pour corriger ça, nous allons renommer `hello` en `Hello` et utiliser `<Hello />` lorsqu'on y fait référence :

```
import React from 'react';

// Correct ! C’est un composant, il doit avoir une initiale majuscule :
function Hello(props) {
  // Correct ! Cette utilisation de <div> fonctionne car div est une balise
HTML valide :
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Correct ! React sait que <Hello /> est un composant car il commence
par une majuscule.
  return <Hello toWhat="World" />;
}
```

Choix du type au moment de l'exécution {#choosing-the-type-at-runtime}

Vous ne pouvez pas utiliser une expression générale pour le type d'un élément React. Si vous voulez utiliser une expression pour définir le type d'un élément, affectez-la d'abord à une variable dont l'initiale est majuscule. Ça arrive en général lorsque vous voulez afficher un composant différent en fonction d’une prop :

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};
```

```
function Story(props) {
  // Faux ! Un type JSX ne peut pas être une expression.
  return <components[props.storyType] story={props.story} />;
}
```

Pour corriger ça, nous allons d'abord affecter le type à une variable dont l'identifiant commence par une majuscule :

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Correct ! Un type JSX peut être une variable commençant par une
majuscule.
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

Les props en JSX {#props-in-jsx}

Il y a différents moyens de définir les props en JSX.

Les expressions JavaScript comme props {#javascript-expressions-as-props}

Vous pouvez passer n'importe quelle expression JavaScript comme prop, en l'entourant avec des accolades `{}`. Par exemple, dans ce code JSX :

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

Pour `MyComponent`, la valeur de `props.foo` sera `10` parce que l'expression `1 + 2 + 3 + 4` est calculée.

Les instructions `if` et les boucles `for` ne sont pas des expressions en JavaScript, donc elle ne peuvent pas être directement utilisées en JSX. Au lieu de ça, vous pouvez les mettre dans le code environnant. Par exemple :

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>pair</strong>;
  } else {
```

```
    description = <i>impair</i>;
  }
  return <div>{props.number} est un nombre {description}</div>;
}
```

Vous pouvez en apprendre davantage sur les conditions et les boucles au sein des sections correspondantes de la documentation.

Les littéraux chaînes {#string-literals}

Vous pouvez passer un littéral chaîne comme prop. Les deux expressions JSX ci-dessous sont équivalentes :

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

Quand vous passez un littéral chaîne, sa valeur subit un échappement HTML inverse. Ces deux expressions JSX sont donc équivalentes :

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

Ce comportement n'est en général pas pertinent (au sens où vous n'avez pas à vous en soucier particulièrement), ce n'est mentionné ici que par souci d'exhaustivité.

Les props valent `true` par défaut {#props-default-to-true}

Si vous n'affectez aucune valeur à une prop, sa valeur par défaut sera `true`. Ces deux expressions JSX sont équivalentes :

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

En général, nous déconseillons cette syntaxe car ça peut être confondu avec la notation ES6 de propriétés concises `{foo}` qui est l'abréviation de `{foo: foo}` et non de `{foo: true}`. Ce comportement existe uniquement par souci de cohérence avec HTML.

Décomposition des props {#spread-attributes}

Si vous avez déjà un objet `props` et souhaitez l'utiliser en JSX, vous pouvez utiliser l'opérateur de décomposition *(spread operator, NdT)* `...` pour passer l'ensemble de l'objet props. Ces deux composants sont équivalents :

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

Vous pouvez également choisir certaines props que votre composant utilisera en passant toutes les autres props avec l'opérateur de *rest*.

```
const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "primary" ? "PrimaryButton" :
"SecondaryButton";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log("cliqué !")}>
        Bonjour monde !
      </Button>
    </div>
  );
};
```

Dans l'exemple ci-dessus, la prop `kind` est extraite pour le composant principal et *n'est pas* passée à l'élément `<button>` du DOM. Toutes les autres props sont passées via l'objet `...other`, ce qui rend ce composant très flexible. Vous pouvez voir qu'il passe les props `onClick` et `children`.

La décomposition des props peut être utile, mais elle permet aussi de passer trop facilement des props inutiles aux composants, ou de passer des attributs HTML invalides au DOM. Nous vous conseillons d'utiliser cette syntaxe avec parcimonie.

Les éléments enfants en JSX {#children-in-jsx}

Dans les expressions JSX qui comportent une balise ouvrante et une balise fermante, le contenu entre ces deux balises est passé comme une prop spéciale : `props.children`. Il existe plusieurs moyens pour passer ces enfants :

Littéraux chaînes {#string-literals-1}

Vous pouvez mettre une chaîne de caractères entre une balise ouvrante et une fermante et `props.children` sera juste cette chaîne de caractères. C'est utile pour la plupart des éléments HTML natifs. Par exemple :

```
<MyComponent>Bonjour monde !</MyComponent>
```

C'est du JSX valide, et `props.children` dans `MyComponent` sera simplement la chaîne de caractères `"Bonjour monde !"`. Le HTML subit un échappement inverse, donc vous pouvez généralement écrire du JSX de la même façon que vous écrivez du HTML, c'est-à-dire :

```
<div>Ce contenu est valide en HTML &amp;amp; en JSX.</div>
```

JSX supprime les espaces en début et en fin de ligne. Il supprime également les lignes vides. Les sauts de lignes adjacents aux balises sont retirés ; les sauts de lignes apparaissant au sein de littéraux chaînes sont ramenés à une seule espace. Du coup, tous les codes ci-dessous donnent le même résultat :

```
<div>Bonjour monde</div>

<div>
  Bonjour monde
</div>

<div>
  Bonjour
  monde
</div>

<div>

  Bonjour monde
</div>
```

Éléments JSX enfants {#jsx-children}

Vous pouvez fournir des éléments JSX supplémentaires en tant qu'enfants. C'est utile pour afficher des composants imbriqués :

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

Vous pouvez mélanger différents types d'enfants, comme par exemple des littéraux chaînes et des éléments JSX. Là encore, JSX est similaire à HTML, de sorte que le code suivant est valide tant en HTML qu'en JSX :

```
<div>
  Voici une liste :
```

```
  <ul>
    <li>Élément 1</li>
    <li>Élément 2</li>
  </ul>
</div>
```

Un composant React peut aussi renvoyer un tableau d'éléments :

```
render() {
  // Pas besoin d'enrober les éléments de la liste dans un élément
supplémentaire !
  return [
    // N'oubliez pas les "keys" :)
    <li key="A">Premier élément</li>,
    <li key="B">Deuxième élément</li>,
    <li key="C">Troisième élément</li>,
  ];
}
```

Les expressions JavaScript comme enfants {#javascript-expressions-as-children}

Vous pouvez passer n'importe quelle expression JavaScript en tant qu'enfant, en l'enrobant avec des accolades `{}`. Ainsi, ces expressions sont équivalentes :

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

C'est souvent utile pour afficher une liste d'expressions JSX de longueur quelconque. Par exemple, ce code affiche une liste HTML :

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['finir la doc', 'envoyer la pr', 'tanner Chris pour une
revue'];
  return (
    <ul>
      {todos.map((message) => <Item key={message} message={message} />)}
    </ul>
  );
}
```

Les expressions JavaScript peuvent être mélangées avec d'autres types d'enfants. C'est souvent utile en remplacement de gabarits textuels :

```
function Hello(props) {
  return <div>Bonjour {props.addressee} !</div>;
}
```

Les fonctions comme enfants {#functions-as-children}

En temps normal, les expressions Javascript insérées dans JSX produiront une chaîne, un élément React ou une liste de ces types. Cependant, `props.children` fonctionne exactement comme n'importe quelle prop dans le sens où elle peut passer n'importe quel genre de données, pas seulement celles que React sait afficher.

Par exemple, si vous avez un composant personnalisé, vous pouvez lui faire accepter une fonction de rappel dans `props.children` :

```
// Appelle la fonction de rappel children à raison de numTimes fois
// afin de produire une répétition du composant
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  return <div>{items}</div>;
}

function ListOfTenThings() {
  return (
    <Repeat numTimes={10}>
      {(index) => <div key={index}>Ceci est l'élément {index} de la
liste</div>}
    </Repeat>
  );
}
```

Les enfants passés à un composant personnalisé peuvent être n'importe quoi, du moment que ce composant les transforme en quelque chose que React peut comprendre avant le rendu. Cette utilisation n'est pas courante, mais elle fonctionne si vous voulez étendre ce dont JSX est capable.

Les booléens ainsi que `null` et `undefined` sont ignorés {#booleans-null-and-undefined-are-ignored}

`false`, `null`, `undefined`, et `true` sont des enfants valides. Ils ne sont simplement pas exploités. Ces expressions JSX produiront toutes la même chose :

```
<div />

<div></div>

<div>{false}</div>

<div>{null}</div>

<div>{undefined}</div>

<div>{true}</div>
```

Ça peut être utile pour afficher des éléments React de façon conditionnelle. Ce JSX produit un composant `<Header />` uniquement si `showHeader` est à `true` :

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

Une mise en garde s'impose : certaines valeurs *falsy*, comme le nombre `0`, sont tout de même affichées par React. Par exemple, ce code ne se comportera pas comme vous l'espérez car il affichera `0` lorsque `props.messages` est un tableau vide :

```
<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

Pour corriger ça, assurez-vous que l'expression avant `&&` est toujours un booléen :

```
<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>
```

Réciproquement, si vous voulez qu'une valeur comme `false`, `true`, `null`, ou `undefined` soit bien affichée, vous devez d'abord la convertir en chaîne :

```
<div>
  Ma variable Javascript est {String(myVariable)}.
</div>
```

Le rendu des éléments

Les éléments sont les blocs élémentaires d'une application React.

Un élément décrit ce que vous voulez voir à l'écran :

```
const element = <h1>Bonjour, monde</h1>;
```

Contrairement aux éléments DOM d'un navigateur, les éléments React sont de simples objets peu coûteux à créer. React DOM se charge de mettre à jour le DOM afin qu'il corresponde aux éléments React.

> Remarque
>
> On pourrait confondre les éléments avec le concept plus répandu de « composants ». Nous présenterons les composants dans la prochaine section. Les éléments représentent la base des composants, aussi nous vous conseillons de bien lire cette section avant d'aller plus loin.

Afficher un élément dans le DOM {#rendering-an-element-into-the-dom}

Supposons qu'il y ait une balise `<div>` quelque part dans votre fichier HTML :

```
<div id="root"></div>
```

Nous parlons de nœud DOM « racine » car tout ce qu'il contient sera géré par React DOM.

Les applications développées uniquement avec React ont généralement un seul nœud DOM racine. Si vous intégrez React dans une application existante, vous pouvez avoir autant de nœuds DOM racines isolés que vous le souhaitez.

Pour faire le rendu d'un élément React dans un nœud DOM racine, passez les deux à la méthode `ReactDOM.render()` :

`embed:rendering-elements/render-an-element.js`

Essayer sur CodePen

Cet exemple de code affichera « Bonjour, monde » sur la page.

Mettre à jour un élément affiché {#updating-the-rendered-element}

Les éléments React sont immuables. Une fois votre élément créé, vous ne pouvez plus modifier ses enfants ou ses attributs. Un élément est comme une image d'un film à un instant T : il représente l'interface utilisateur à un point précis dans le temps.

Avec nos connaissances actuelles, la seule façon de mettre à jour l'interface utilisateur est de créer un nouvel élément et de le passer à `ReactDOM.render()`.

Prenons l'exemple de cette horloge :

`embed:rendering-elements/update-rendered-element.js`

Essayer dans CodePen

À chaque seconde, nous appellons `ReactDOM.render()` depuis une fonction de rappel passée à `setInterval()`.

> Remarque
>
> En pratique, la plupart des applications React n'appellent `ReactDOM.render()` qu'une seule fois. Dans les prochaines sections, nous apprendrons comment encapsuler un tel code dans des composants à état.
>
> Nous vous conseillons de lire les sujets abordés dans l'ordre car ils s'appuient l'un sur l'autre.

React met à jour le strict nécessaire {#react-only-updates-whats-necessary}

React DOM compare l'élément et ses enfants avec la version précédente, et applique uniquement les mises à jour DOM nécessaires pour refléter l'état voulu.

Vous pouvez vérifier ce comportement en inspectant le dernier exemple avec les outils de développement du navigateur :

 L'inspecteur montrant des mises à jour atomiques

Même si nous créons à chaque seconde un élément décrivant l'arborescence complète de l'interface utilisateur, seul le nœud texte dont le contenu a été modifié est mis à jour par React DOM.

L'expérience nous montre que réfléchir à quoi devrait ressembler une interface utilisateur à un moment donné plutôt que de réfléchir à comment elle devrait évoluer permet d'éliminer toute une catégorie de bugs.

Composants et props

Les composants vous permettent de découper l'interface utilisateur en éléments indépendants et réutilisables, vous permettant ainsi de considérer chaque élément de manière isolée. Cette page fournit une introduction au concept de composants. Vous trouverez une [référence détaillée de l'API des composants ici](/docs/react-component.html).

Conceptuellement, les composants sont comme des fonctions JavaScript. Ils acceptent des entrées quelconques (appelées « props ») et renvoient des éléments React décrivant ce qui doit apparaître à l'écran.

Fonctions composants et composants à base de classes {#function-and-class-components}

Le moyen le plus simple de définir un composant consiste à écrire une fonction JavaScript :

```
function Welcome(props) {
  return <h1>Bonjour, {props.name}</h1>;
}
```

Cette fonction est un composant React valide car elle accepte un seul argument « props » (qui signifie « propriétés ») contenant des données, et renvoie un élément React. Nous appelons de tels composants des « fonctions composants », car ce sont littéralement des fonctions JavaScript.

Vous pouvez également utiliser une classe ES6 pour définir un composant :

```
class Welcome extends React.Component {
  render() {
    return <h1>Bonjour, {this.props.name}</h1>;
  }
}
```

Les deux composants ci-dessus sont équivalents du point de vue de React.

Les classes possèdent quelques fonctionnalités supplémentaires dont nous discuterons dans les prochaines sections. En attendant, nous utiliserons les fonctions composants pour leur concision.

Produire le rendu d'un composant {#rendering-a-component}

Jusqu'ici, nous n'avons rencontré que des éléments React représentant des balises DOM :

```
const element = <div />;
```

Mais ces éléments peuvent également représenter des composants définis par l'utilisateur :

```
const element = <Welcome name="Sara" />;
```

Lorsque React rencontre un élément représentant un composant défini par l'utilisateur, il transmet les attributs JSX à ce composant sous la forme d'un objet unique. Nous appelons cet objet « props ».

Par exemple, ce code affiche « Bonjour, Sara » sur la page :

```
function Welcome(props) {
  return <h1>Bonjour, {props.name}</h1>;
}

const element = <Welcome name="Sara" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Essayer sur CodePen

Récapitulons ce qui se passe dans cet exemple :

1. On appelle `ReactDOM.render()` avec l'élément `<Welcome name="Sara" />`.
2. React appelle le composant `Welcome` avec comme props `{name: 'Sara'}`.
3. Notre composant `Welcome` retourne un élément `<h1>Bonjour, Sara</h1>` pour résultat.
4. React DOM met à jour efficacement le DOM pour correspondre à `<h1>Bonjour, Sara</h1>`.

> Remarque
>
> React considère les composants commençant par des lettres minuscules comme des balises DOM. Par exemple, `<div />` représente une balise HTML div, mais `<Welcome />` représente un composant, et exige que l'identifiant `Welcome` existe dans la portée courante.
>
> Pour en apprendre davantage sur le raisonnement qui sous-tend cette convention, lisez donc JSX en profondeur.

Composition de composants {#composing-components}

Les composants peuvent faire référence à d'autres composants dans leur sortie. Ça nous permet d'utiliser la même abstraction de composants pour n'importe quel niveau de détail. Un bouton, un formulaire, une boîte de dialogue, un écran : dans React, ils sont généralement tous exprimés par des composants.

Par exemple, nous pouvons créer un composant `App` qui utilise plusieurs fois `Welcome` :

```
function Welcome(props) {
  return <h1>Bonjour, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
```

```
      <Welcome name="Edite" />
    </div>
  );
}

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

Essayer sur CodePen

En règle générale, les nouvelles applications React ont un seul composant `App` à la racine. En revanche, si vous intégrez React à une application existante, vous pouvez commencer tout en bas par un petit composant comme `Button` et remonter progressivement la hiérarchie des vues.

Extraire des composants {#extracting-components}

N'ayez pas peur de scinder des composants en composants plus petits.

Prenons par exemple ce composant `Comment` :

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Essayer sur CodePen

Il accepte comme props `author` (un objet), `text` (une chaîne de caractères) et `date` (une date), et décrit un commentaire sur un réseau social en ligne.

Les nombreuses imbrications au sein du composant le rendent difficile à maintenir, et nous empêchent d'en réutiliser des parties individuelles. Essayons donc d'en extraire quelques composants.

Pour commencer, nous allons extraire `Avatar` :

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

Le composant `Avatar` n'a pas besoin de savoir qu'il figure dans un composant `Comment`. C'est pourquoi nous avons donné à sa prop un nom plus générique : `user` plutôt que `author`.

Nous vous recommandons de nommer les props du point de vue du composant plutôt que de celui du contexte dans lequel il est utilisé.

On peut maintenant simplifier un poil`Comment` :

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Ensuite, nous allons extraire un composant `UserInfo` qui affiche un `Avatar` à côté du nom de l'utilisateur :

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
```

```
        {props.user.name}
      </div>
    </div>
  );
}
```

Ce qui nous permet de simplifier encore davantage `Comment` :

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Essayer sur CodePen

Au début, extraire des composants peut vous sembler fastidieux, mais disposer d'une palette de composants réutilisables s'avère rentable sur des applications de plus grande taille. En règle générale, si une partie de votre interface utilisateur est utilisée plusieurs fois (`Button`, `Panel`, `Avatar`) ou si elle est suffisamment complexe en elle-même (`App`, `FeedStory`, `Comment`), c'est un bon candidat pour un composant réutilisable.

Les props sont en lecture seule {#props-are-read-only}

Que vous déclariez un composant sous forme de fonction ou de classe, il ne doit jamais modifier ses propres props. Considérons cette fonction `sum` :

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

Ces fonctions sont dites « pures » parce qu'elles ne tentent pas de modifier leurs entrées et retournent toujours le même résultat pour les mêmes entrées.

En revanche, cette fonction est impure car elle modifie sa propre entrée :

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

React est plutôt flexible mais applique une règle stricte :

Tout composant React doit agir comme une fonction pure vis-à-vis de ses props.

Bien entendu, les interfaces utilisateurs des applications sont dynamiques et évoluent dans le temps. Dans la prochaine section, nous présenterons un nouveau concept « d'état local ». L'état local permet aux composants React de modifier leur sortie au fil du temps en fonction des actions de l'utilisateur, des réponses réseau et de n'importe quoi d'autre, mais sans enfreindre cette règle.

État et cycle de vie

Cette page présente les concepts d'état local et de cycle de vie dans un composant React. Vous pouvez trouver [la référence d'API des composants ici](/docs/react-component.html).

Prenons l'exemple de l'horloge dans une des sections précédentes. Dans Le rendu des éléments, nous avons appris une seule façon de mettre à jour l'interface utilisateur (UI). On appelle `ReactDOM.render()` pour changer la sortie rendue :

```
function tick() {
  const element = (
    <div>
      <h1>Bonjour, monde !</h1>
      <h2>Il est {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Essayer sur CodePen

Dans cette section, nous allons apprendre à faire un composant `Clock` vraiment réutilisable et isolé. Il mettra en place son propre minuteur et se mettra à jour tout seul à chaque seconde.

Nous commençons par isoler l'apparence de l'horloge :

```
function Clock(props) {
  return (
    <div>
      <h1>Bonjour, monde !</h1>
      <h2>Il est {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Essayer sur CodePen

Cependant, il manque une contrainte cruciale : le fait que la `Clock` mette en place le minuteur et mette à jour son interface utilisateur devrait être un détail d'implémentation de la `Clock`.

Idéalement, on veut écrire ceci une seule fois et voir la `Clock` se mettre à jour elle-même :

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

Pour implémenter ça, on a besoin d'ajouter un « état local » au composant `Horloge`.

L'état local est similaire aux props, mais il est privé et complètement contrôlé par le composant.

Convertir une fonction en classe {#converting-a-function-to-a-class}

Vous pouvez convertir un composant fonctionnel comme `Clock` en une classe en cinq étapes :

1. Créez une classe ES6, avec le même nom, qui étend `React.Component`.
2. Ajoutez-y une méthode vide appelée `render()`.
3. Déplacez le corps de la fonction dans la méthode `render()`.
4. Remplacez `props` par `this.props` dans le corps de la méthode `render()`.
5. Supprimez la déclaration désormais vide de la fonction.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Essayer sur CodePen

Le composant `Clock` est maintenant défini comme une classe au lieu d'une fonction.

La méthode `render` sera appelée à chaque fois qu'une mise à jour aura lieu, mais tant que l'on exploite le rendu de `<Clock />` dans le même nœud DOM, une seule instance de la classe `clock` sera utilisée. Cela nous permet d'utiliser des fonctionnalités supplémentaires telles que l'état local et les méthodes de cycle de vie.

Ajouter un état local à une classe {#adding-local-state-to-a-class}

Nous allons déplacer la `date` des props vers l'état en trois étapes :

1. Remplacez `this.props.date` avec `this.state.date` dans la méthode `render()` :

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

2. Ajoutez un constructeur de classe qui initialise `this.state`:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Notez que l'on passe `props` au constructeur de base :

```
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
```

Les composants à base de classe devraient toujours appeler le constructeur de base avec `props`.

3. Supprimez la prop `date` de l'élément `<Clock />` :

```
ReactDOM.render(
  <Clock />,
```

```
  document.getElementById('root')
);
```

Nous rajouterons plus tard le code du minuteur dans le composant lui-même.

Le résultat ressemble à ceci :

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

Essayer sur CodePen

Ensuite, nous allons faire en sorte que le composant `Clock` mette en place son propre minuteur et se mette à jour toutes les secondes.

Ajouter des méthodes de cycle de vie à une classe {#adding-lifecycle-methods-to-a-class}

Dans des applications avec de nombreux composants, il est très important de libérer les ressources utilisées par les composants quand ils sont détruits.

Nous voulons mettre en place un minuteur quand une `Horloge` apparaît dans le DOM pour la première fois. Le terme React « montage » désigne cette phase.

Nous voulons également nettoyer le minuteur quand le DOM produit par l'`Horloge` est supprimé. En React, on parle de « démontage ».

Nous pouvons déclarer des méthodes spéciales sur un composant à base de classe pour exécuter du code quand un composant est monté et démonté :

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {

  }

  componentWillUnmount() {

  }

  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

On les appelle des « méthodes de cycle de vie ».

La méthode `componentDidMount()` est exécutée après que la sortie du composant a été injectée dans le DOM. C'est un bon endroit pour mettre en place le minuteur :

```
  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }
```

Notez qu'on a enregistré l'ID du minuteur directement sur `this` (`this.timeID`).

Alors que `this.props` est mis en place par React lui-même et que `this.state` a un sens bien spécial, vous pouvez très bien ajouter manuellement d'autres champs sur la classe si vous avez besoin de stocker quelque chose qui ne participe pas au flux de données (comme un ID de minuteur).

Nous allons détruire le minuteur dans la méthode de cycle de vie `componentWillUnmount()` :

```
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
```

Enfin, nous allons implémenter une méthode appelée `tick()` que le composant `Clock` va exécuter toutes les secondes.

Elle utilisera `this.setState()` pour planifier une mise à jour de l'état local du composant :

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

Essayer sur CodePen

Maintenant l'horloge se met à jour toutes les secondes.

Récapitulons ce qui se passe et l'ordre dans lequel les méthodes sont invoquées :

1. Quand `<Clock />` est passé à `ReactDOM.render()`, React appelle le constructeur du composant `Clock`. Puisque `Clock` a besoin d'afficher l'heure actuelle, il initialise `this.state` avec un objet

contenant l'heure actuelle. Nous mettrons cet état à jour par la suite.

2. React appelle ensuite la méthode `render()` du composant `Clock`. C'est comme cela que React découvre ce qu'il faut afficher à l'écran. React met ensuite à jour le DOM pour correspondre à la sortie de la méthode `render()` du composant `Clock`.

3. Quand la sortie de la `Clock` est insérée dans le DOM, React appelle la méthode de cycle de vie `componentDidMount()`. À l'intérieur, le composant `Clock` demande au navigateur de mettre en place un minuteur pour appeler la méthode `tick()` du composant une fois par seconde.

4. Chaque seconde, le navigateur appelle la méthode `tick()`. À l'intérieur, le composant `Clock` planifie une mise à jour de l'interface utilisateur en appelant `setState()` avec un objet contenant l'heure actuelle. Grâce à l'appel à `setState()`, React sait que l'état a changé, et invoque à nouveau la méthode `render()` pour savoir ce qui devrait être affiché à l'écran. Cette fois, la valeur de `this.state.date` dans la méthode `render()` est différente, la sortie devrait donc inclure l'heure mise à jour. React met à jour le DOM en accord avec cela.

5. Si le composant `Clock` finit par être retiré du DOM, React appellera la méthode de cycle de vie `componentWillUnmount()` pour que le minuteur soit arrêté.

Utiliser l'état local correctement {#using-state-correctly}

Il y'a trois choses que vous devriez savoir à propos de `setState()`.

Ne modifiez pas l'état directement {#do-not-modify-state-directly}

Par exemple, ceci ne déclenchera pas un rafraîchissement du composant :

```
// Erroné
this.state.comment = 'Bonjour';
```

À la place, utilisez `setState()` :

```
// Correct
this.setState({comment: 'Bonjour'});
```

Le seul endroit où vous pouvez affecter `this.state`, c'est le constructeur.

Les mises à jour de l'état peuvent être asynchrones {#state-updates-may-be-asynchronous}

React peut grouper plusieurs appels à `setState()` en une seule mise à jour pour des raisons de performance.

Comme `this.props` et `this.state` peuvent être mises à jour de façon asynchrone, vous ne devez pas vous baser sur leurs valeurs pour calculer le prochain état.

Par exemple, ce code peut échouer pour mettre à jour un compteur :

```
// Erroné
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

Pour remédier à ce problème, utilisez la seconde forme de `setState()` qui accepte une fonction à la place d'un objet. Cette fonction recevra l'état précédent comme premier argument et les props au moment de la mise à jour comme second argument :

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Nous avons utilisé une fonction fléchée ci-dessus, mais une fonction normale marche aussi :

```
// Correct
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

Les mises à jour de l'état sont fusionnées {#state-updates-are-merged}

Quand vous invoquez `setState()`, React fusionne les objets que vous donnez avec l'état actuel.

Par exemple, votre état peut contenir plusieurs variables indépendantes :

```
  constructor(props) {
    super(props);
    this.state = {
      posts: [],
      comments: []
    };
  }
```

Ensuite, vous pouvez les mettre à jour indépendamment avec des appels séparés à `setState()` :

```
  componentDidMount() {
    fetchPosts().then(response => {
      this.setState({
        posts: response.posts
      });
```

```
    });

    fetchComments().then(response => {
      this.setState({
        comments: response.comments
      });
    });
  }
```

La fusion n'est pas profonde, donc `this.setState({comments})` laisse `this.state.posts` intacte, mais remplace complètement `this.state.comments`.

Les données descendent {#the-data-flows-down}

Ni parent ni enfant ne peuvent savoir si un certain composant est à état ou non, et ne devraient pas se soucier de savoir s'il est défini par une fonction ou une classe.

C'est pourquoi on dit souvent que l'état est local ou encapsulé. Il est impossible d'y accéder depuis un autre composant.

Un composant peut choisir de passer son état à ses enfants via des props :

```
<h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
```

Cela marche également avec des composants définis par l'utilisateur :

```
<FormattedDate date={this.state.date} />
```

Le composant `FormattedDate` reçoit la `date` dans ses props et ne sait pas si elle vient de l'état de la `Clock`, des props de la `Clock`, ou a été tapée à la main :

```
function FormattedDate(props) {
  return <h2>Il est {props.date.toLocaleTimeString()}.</h2>;
}
```

Essayer sur CodePen

On appelle souvent cela un flux de données « du haut vers le bas » ou « unidirectionnel ». Un état local est toujours possédé par un composant spécifique, et toute donnée ou interface utilisateur dérivée de cet état ne peut affecter que les composants « en-dessous » de celui-ci dans l'arbre de composants.

Si vous imaginez un arbre de composants comme une cascade de props, chaque état de composant est une source d'eau supplémentaire qui rejoint la cascade à un point quelconque, mais qui coule également vers le bas.

Pour démontrer que tous les composants sont réellement isolés, nous pouvons créer un composant `App` qui affiche trois `<Clock>`s :

```
function Application() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}

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

Essayer sur CodePen

Chaque `Clock` met en place son propre minuteur et se met à jour indépendamment.

Dans une application React, le fait qu'un composant soit à état ou non est considéré comme un détail d'implémentation du composant qui peut varier avec le temps. Vous pouvez utiliser des composants sans état à l'intérieur de composants à état, et vice-versa.

Gérer les événements

La gestion des événements pour les éléments React est très similaire à celle des éléments du DOM. Il y a tout de même quelques différences de syntaxe :

- Les événements de React sont nommés en `camelCase` plutôt qu'en minuscules.
- En JSX on passe une fonction comme gestionnaire d'événements plutôt qu'une chaîne de caractères.

Par exemple, le HTML suivant :

```
<button onclick="activateLasers()">
  Activer les lasers
</button>
```

est légèrement différent avec React:

```
<button onClick={activateLasers}>
  Activer les lasers
</button>
```

Autre différence importante : en React, on ne peut pas renvoyer `false` pour empêcher le comportement par défaut. Vous devez appeler explicitement `preventDefault`. Par exemple, en HTML, pour annuler le comportement par défaut des liens qui consiste à ouvrir une nouvelle page, vous pourriez écrire :

```
<a href="#" onclick="console.log('Le lien a été cliqué.'); return false">
  Clique ici
</a>
```

En React, ça pourrait être :

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('Le lien a été cliqué.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Clique ici
    </a>
  );
}
```

Ici, `e` est un événement synthétique. React le définit en suivant les spécifications W3C, afin que vous n'ayez pas à vous préoccuper de la compatibilité entre les navigateurs. Pour en apprendre davantage, consultez le guide de référence de `SyntheticEvent`.

Lorsque vous utilisez React, vous n'avez généralement pas besoin d'appeler la méthode `addEventListener` pour ajouter des écouteurs d'événements *(event listeners, NdT)* à un élément du DOM après que celui-ci est créé. À la place, on fournit l'écouteur lors du rendu initial de l'élément.

Lorsque vous définissez un composant en utilisant les classes ES6, il est d'usage que le gestionnaire d'événements soit une méthode de la classe. Par exemple, ce composant `Toggle` affiche un bouton qui permet à l'utilisateur de basculer l'état de "ON" à "OFF".

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // Cette liaison est nécéssaire afin de permettre
    // l'utilisation de `this` dans la fonction de rappel.
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

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

Essayer dans CodePen

En JSX, vous devez être prudent·e avec l'utilisation de `this` dans les fonctions de rappel. En JavaScript, les méthodes de classes ne sont pas liées par défaut. Si vous oubliez de lier `this.handleClick` et l'utilisez dans `onClick`, `this` sera `undefined` quand la fonction sera appelée.

Ce n'est pas un comportement spécifique à React, ça fait partie du fonctionnement normal des fonctions en JavaScript. En général, si vous faites référence à une méthode sans l'appeler avec `()`, comme dans `onClick={this.handleClick}`, vous devriez lier cette méthode.

Si vous ne souhaitez pas utiliser `bind`, vous avez deux alternatives possibles. Si vous avez l'habitude d'utiliser la syntaxe des champs de classes, qui est encore expérimentale, vous pourriez l'utiliser pour lier les fonctions

de rappel :

```
class LoggingButton extends React.Component {
  // Cette syntaxe nous assure que `this` est bien lié dans la méthode
handleClick.
  // Attention : cette syntaxe est encore *expérimentale*.
  handleClick = () => {
    console.log('this vaut :', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Clique ici
      </button>
    );
  }
}
```

Cette syntaxe est activée par défaut dans Create React App.

Si vous n'utilisez pas la syntaxe des champs de classe, vous pouvez utiliser les fonctions fléchées pour les fonctions de rappel.

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this vaut :', this);
  }

  render() {
    // Cette syntaxe nous assure que `this` est bien lié dans la méthode
handleClick
    return (
      <button onClick={(e) => this.handleClick(e)}>
        Clique ici
      </button>
    );
  }
}
```

Cette syntaxe n'est toutefois pas sans défauts, car elle crée une nouvelle fonction de rappel à chaque affichage de `LoggingButton`. Dans la plupart des cas ce n'est pas gênant. Néanmoins, si cette fonction était passée comme prop à des composants plus bas dans l'arbre, ces composants risqueraient de faire des ré-affichages superflus. Nous recommandons donc, en règle générale, de lier ces méthodes dans le constructeur ou d'utiliser un champ de classe afin d'éviter ce genre de problèmes de performances.

Passer des arguments à un gestionnaire d'événements {#passing-arguments-to-event-handlers}

Au sein d'une boucle, il est courant de vouloir passer un argument supplémentaire à un gestionnaire d'événements. Par exemple, si `id` représente la ligne sélectionnée, on peut faire au choix :

```
<button onClick={(e) => this.deleteRow(id, e)}>Supprimer la ligne</button>
<button onClick={this.deleteRow.bind(this, id)}>Supprimer la ligne</button>
```

Les lignes précédentes sont équivalentes et utilisent respectivement les fonctions fléchées et `Function.prototype.bind`.

Dans les deux cas, l'argument `e` represente l'événement React qui sera passé en second argument après l'ID. Avec une fonction fléchée, nous devons passer l'argument explicitement, alors qu'avec `bind` tous les arguments sont automatiquement transmis.

Démat monde

En React, vous pouvez concevoir des composants distincts qui encapsulent le comportement voulu. Vous pouvez alors n'afficher que certains d'entre eux, suivant l'état de votre application.

L'affichage conditionnel en React fonctionne de la même façon que les conditions en Javascript. On utilise l'instruction Javascript `if` ou l'opérateur ternaire pour créer des éléments représentant l'état courant, et on laisse React mettre à jour l'interface utilisateur (UI) pour qu'elle corresponde.

Considérons ces deux composants :

```
function UserGreeting(props) {
  return <h1>Bienvenue !</h1>;
}

function GuestGreeting(props) {
  return <h1>Veuillez vous inscrire.</h1>;
}
```

Nous allons créer un composant `Greeting` qui affiche un de ces deux composants, selon qu'un utilisateur est connecté ou non :

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  // Essayez de changer ça vers isLoggedIn={true} :
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

Essayer sur CodePen

Cet exemple affiche un message différent selon la valeur de la prop `isLoggedIn`.

Variables d'éléments {#element-variables}

Vous pouvez stocker les éléments dans des variables. Ça vous aide à afficher de façon conditionnelle une partie du composant tandis que le reste ne change pas.

Prenons ces deux nouveaux composants, qui représentent les boutons de Déconnexion et de Connexion :

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
```

```
      Connexion
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Déconnexion
    </button>
  );
}
```

Dans l'exemple ci-dessous, nous allons créer un composant à état appelé `LoginControl`.

Il affichera soit `<LoginButton />`, soit `<LogoutButton />`, selon son état courant. Il affichera aussi un `<Greeting />` de l'exemple précédent :

```
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
```

```
}

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

Essayer sur CodePen

Même si déclarer une variable et utiliser une instruction `if` reste une bonne façon d'afficher conditionnellement un composant, parfois vous voudrez peut-être utiliser une syntaxe plus concise. Nous allons voir, ci-dessous, plusieurs façons d'utiliser des conditions à la volée en JSX.

Condition à la volée avec l'opérateur logique && {#inline-if-with-logical--operator}

Vous pouvez utiliser n'importe quelle expression dans du JSX en l'enveloppant dans des accolades. Ça vaut aussi pour l'opérateur logique Javascript `&&`. Il peut être pratique pour inclure conditionnellement un élément :

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Bonjour !</h1>
      {unreadMessages.length > 0 &&
        <h2>
          Vous avez {unreadMessages.length} message(s) non-lu(s).
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

Essayer sur CodePen

Ça fonctionne parce qu'en JavaScript, `true && expression` est toujours évalué à `expression`, et `false && expression` est toujours évalué à `false`.

Du coup, si la condition vaut `true`, l'élément juste après `&&` sera affiché. Si elle vaut `false`, React va l'ignorer et le sauter.

Alternative à la volée avec opérateur ternaire {#inline-if-else-with-conditional-operator}

Une autre méthode pour l'affichage conditionnel à la volée d'éléments consiste à utiliser l'opérateur ternaire Javascript `condition ? trueValue : falseValue`.

Dans l'exemple ci-dessous, on l'utilise pour afficher conditionnellement un bloc de texte.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      L'utilisateur <b>{isLoggedIn ? 'est actuellement' : 'n'est pas'}</b>
connecté.
    </div>
  );
}
```

On peut aussi l'utiliser pour des expressions plus longues, même si c'est moins clair :

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn ? (
        <LogoutButton onClick={this.handleLogoutClick} />
      ) : (
        <LoginButton onClick={this.handleLoginClick} />
      )}
    </div>
  );
}
```

Tout comme en Javascript, c'est à vous de choisir un style approprié selon les préférences de lisibilité en vigueur pour vous et votre équipe. Souvenez-vous aussi que chaque fois que des conditions deviennent trop complexes, c'est peut-être le signe qu'il serait judicieux d'en extraire un composant.

Empêcher l'affichage d'un composant {#preventing-component-from-rendering}

Dans de rares cas, vous voudrez peut-être qu'un composant se masque alors même qu'il figure dans le rendu d'un autre composant. Pour ce faire, il suffit de renvoyer `null` au lieu de son affichage habituel.

Dans l'exemple ci-dessous, `<WarningBanner />` s'affichera en fonction de la valeur de la prop `warn`. Si la valeur est `false`, le composant ne s'affiche pas :

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
```

```
      <div className="warning">
        Attention !
      </div>
    );
  }

  class Page extends React.Component {
    constructor(props) {
      super(props);
      this.state = {showWarning: true};
      this.handleToggleClick = this.handleToggleClick.bind(this);
    }

    handleToggleClick() {
      this.setState(state => ({
        showWarning: !state.showWarning
      }));
    }

    render() {
      return (
        <div>
          <WarningBanner warn={this.state.showWarning} />
          <button onClick={this.handleToggleClick}>
            {this.state.showWarning ? 'Masquer' : 'Afficher'}
          </button>
        </div>
      );
    }
  }

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

Essayer sur on CodePen

Renvoyer `null` depuis la méthode `render` d'un composant n'affecte pas l'appel aux méthodes du cycle de vie du composant. Par exemple, `componentDidUpdate` sera quand même appelée.

---

listes & clefs

Tout d'abord, voyons comment transformer des listes en JavaScript.

Dans le code suivant, on utilise la méthode `map()` pour prendre un tableau de nombres et doubler leurs valeurs. On affecte le nouveau tableau retourné par `map()` à une variable `doubled` et on l'affiche dans la console :

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

Ce code affiche `[2, 4, 6, 8, 10]` dans la console.

Avec React, transformer un tableau en une liste d'éléments est presque identique.

Afficher plusieurs composants {#rendering-multiple-components}

On peut construire des collections d'éléments et les inclure dans du JSX en utilisant les accolades `{}`.

Ci-dessous, on itère sur le tableau de nombres en utilisant la méthode JavaScript `map()`. On retourne un élément `<li>` pour chaque entrée du tableau. Enfin, on affecte le tableau d'éléments résultant à `listItems` :

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

On inclut tout le tableau `listItems` dans un élément `<ul>`, et on l'affiche dans le DOM :

```
ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```

Essayer sur CodePen

Ce code affiche une liste à puces de nombres entre 1 et 5.

Composant basique de liste {#basic-list-component}

Généralement, on souhaite afficher une liste au sein d'un composant.

On peut transformer l'exemple précédent pour en faire un composant qui accepte un tableau de nombres et produit une liste d'éléments.

```
function NumberList(props) {
  const numbers = props.numbers;
```

```
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

En exécutant ce code, vous obtiendrez un avertissement disant qu'une clé devrait être fournie pour les éléments d'une liste. Une « clé » *(key, NdT)*, est un attribut spécial que vous devez inclure quand vous créez une liste d'éléments. Nous verrons pourquoi c'est important dans la prochaine section.

Assignons une `key` aux éléments de notre liste dans `numbers.map()` afin de corriger le problème de clés manquantes.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Essayer sur CodePen

Les clés {#keys}

Les clés aident React à identifier quels éléments d'une liste ont changé, ont été ajoutés ou supprimés. Vous devez donner une clé à chaque élément dans un tableau afin d'apporter aux éléments une identité stable :

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
```

```
    {number}
  </li>
);
```

Le meilleur moyen de choisir une clé est d'utiliser quelque chose qui identifie de façon unique un élément d'une liste parmi ses voisins. Le plus souvent on utilise l'ID de notre donnée comme clé :

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

Quand vous n'avez pas d'ID stable pour les éléments affichés, vous pouvez utiliser l'index de l'élément en dernier recours :

```
const todoItems = todos.map((todo, index) =>
  // Ne faites ceci que si les éléments n'ont pas d'ID stable
  <li key={index}>
    {todo.text}
  </li>
);
```

Nous vous recommandons de ne pas utiliser l'index comme clé si l'ordre des éléments est susceptible de changer. Ça peut avoir un effet négatif sur les performances, et causer des problèmes avec l'état du composant. Vous pouvez lire l'article de Robin Pokorny pour une explication en profondeur de l'impact négatif de l'utilisation de l'index comme clé (en anglais). Si vous choisissez de ne pas donner explicitement de clé aux éléments d'une liste, React utilisera l'index par défaut.

Si vous voulez en apprendre davantage, consultez cette explication en profondeur de la raison pour laquelle les clés sont nécessaires.

Extraire des composants avec des clés {#extracting-components-with-keys}

Les clés n'ont une signification que dans le contexte du tableau qui les entoure.

Par exemple, si on extrait un composant `ListItem`, on doit garder la clé sur l'élément `<ListItem />` dans le tableau, et non sur l'élément `<li>` dans le composant `ListItem` lui-même.

Exemple : utilisation erronée des clés

```
function ListItem(props) {
  const value = props.value;
  return (
    // Erroné ! Pas la peine de spécifier la clé ici :
    <li key={value.toString()}>
```

```
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Erroné : la clé doit être spécifiée ici :
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Exemple : utilisation correcte des clés

```
function ListItem(props) {
  // Correct ! Pas la peine de spécifier la clé ici :
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Correct ! La clé doit être spécifiée dans le tableau.
    <ListItem key={number.toString()}
              value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Essayer sur CodePen

Gardez en tête cette règle simple : chaque élément à l'intérieur d'un appel à `map()` a besoin d'une clé.

Les clés n'ont besoin d'être uniques qu'au sein de la liste {#keys-must-only-be-unique-among-siblings}

Les clés utilisées dans un tableau doivent être uniques parmi leurs voisins. Cependant, elles n'ont pas besoin d'être globalement uniques. On peut utiliser les mêmes clés dans des tableaux différents :

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      <hr />
      {content}
    </div>
  );
}

const posts = [
  {id: 1, title: 'Bonjour, monde', content: 'Bienvenue sur la doc de React
!'},
  {id: 2, title: 'Installation', content: 'Vous pouvez installer React
depuis npm.'}
];
ReactDOM.render(
  <Blog posts={posts} />,
  document.getElementById('root')
);
```

Essayer sur CodePen

Les clés servent d'indicateur à React mais ne sont pas passées à vos composants. Si vous avez besoin de la même valeur dans votre composant, passez-la dans une prop avec un nom différent :

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

Dans l'exemple ci-dessus, le composant `Post` peut accéder à `props.id`, mais pas à `props.key`.

Intégrer `map()` dans du JSX {#embedding-map-in-jsx}

Dans les exemples précédents, nous déclarions séparément la variable `listItems` pour ensuite l'inclure dans le JSX :

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()}
              value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

JSX permet d'intégrer des expressions quelconques entre accolades. Nous pouvons donc utiliser `map()` directement dans notre code JSX :

```
function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()}
                  value={number} />
      )}
    </ul>
  );
}
```

Essayer sur CodePen

Ça rend parfois le code plus lisible, mais il faut éviter d'en abuser. Comme avec JavaScript, c'est vous qui décidez quand ça vaut le coup d'extraire l'expression dans une variable pour plus de lisibilité. Gardez en tête

que si le corps de `map()` est trop profond ou trop riche, c'est sans doute le signe qu'il faudrait extraire un composant.

Formulaires

Les formulaires HTML fonctionnent un peu différemment des autres éléments du DOM en React car ils possèdent naturellement un état interne. Par exemple, ce formulaire en HTML qui accepte juste un nom :

```
<form>
  <label>
    Nom :
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Envoyer" />
</form>
```

Ce formulaire a le comportement classique d'un formulaire HTML et redirige sur une nouvelle page quand l'utilisateur le soumet. Si vous souhaitez ce comportement en React, vous n'avez rien à faire. Cependant, dans la plupart des cas, vous voudrez pouvoir gérer la soumission avec une fonction JavaScript, qui accède aux données saisies par l'utilisateur. La manière classique de faire ça consiste à utiliser les « composants contrôlés ».

Composants contrôlés {#controlled-components}

En HTML, les éléments de formulaire tels que `<input>`, `<textarea>`, et `<select>` maintiennent généralement leur propre état et se mettent à jour par rapport aux saisies de l'utilisateur. En React, l'état modifiable est généralement stocké dans la propriété `state` des composants et mis à jour uniquement avec `setState()`.

On peut combiner ces deux concepts en utilisant l'état local React comme « source unique de vérité ». Ainsi le composant React qui affiche le formulaire contrôle aussi son comportement par rapport aux saisies de l'utilisateur. Un champ de formulaire dont l'état est contrôlé de cette façon par React est appelé un « composant contrôlé ».

Par exemple, en reprenant le code ci-dessus pour afficher le nom lors de la soumission, on peut écrire le formulaire sous forme de composant contrôlé :

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Le nom a été soumis : ' + this.state.value);
    event.preventDefault();
  }
```

```
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Nom :
          <input type="text" value={this.state.value} onChange=
{this.handleChange} />
        </label>
        <input type="submit" value="Envoyer" />
      </form>
    );
  }
}
```

Essayer sur CodePen

À présent que l'attribut `value` est défini sur notre élément de formulaire, la valeur affichée sera toujours `this.state.value`, faisant ainsi de l'état local React la source de vérité. Puisque `handleChange` est déclenché à chaque frappe pour mettre à jour l'état local React, la valeur affichée restera mise à jour au fil de la saisie.

Dans un composant contrôlé, chaque changement de l'état aura une fonction gestionnaire associée. Ça permet de modifier ou valider facilement, à la volée, les saisies de l'utilisateur. Par exemple, si nous voulions forcer les noms en majuscules, on pourrait écrire `handleChange` de la manière suivante :

```
handleChange(event) {
  this.setState({value: event.target.value.toUpperCase()});
}
```

La balise `textarea` {#the-textarea-tag}

En HTML, une balise `<textarea>` définit son texte via ses enfants :

```
<textarea>
  Bonjour, voici du texte dans une zone de texte
</textarea>
```

En React, un `<textarea>` utilise à la place l'attribut `value`. Du coup, un formulaire utilisant un `<textarea>` peut être écrit d'une manière très similaire à un formulaire avec un élément `<input>` mono-ligne.

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Écrivez un essai à propos de votre élément du DOM préféré'
```

```
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Un essai a été envoyé : ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Essay:
          <textarea value={this.state.value} onChange={this.handleChange}
/>
        </label>
        <input type="submit" value="Envoyer" />
      </form>
    );
  }
}
```

Remarquez que `this.state.value` est initialisé dans le constructeur, permettant que le textarea démarre avec du texte à l'intérieur.

La balise `select` {#the-select-tag}

En HTML, `<select>` crée une liste déroulante. Par exemple, ce HTML crée une liste déroulante de parfums.

```
<select>
  <option value="grapefruit">Pamplemousse</option>
  <option value="lime">Citron vert</option>
  <option selected value="coconut">Noix de coco</option>
  <option value="mango">Mangue</option>
</select>
```

Notez que l'option Noix de coco est sélectionnée au départ, grâce à l'attribut `selected`. React, au lieu d'utiliser l'attribut `selected`, utilise un attribut `value` sur la balise racine `select`. C'est plus pratique dans un composant contrôlé car vous n'avez qu'un seul endroit à mettre à jour. Par exemple :

```
class FlavorForm extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Votre parfum favori est : ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Choisissez votre parfum favori :
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Pamplemousse</option>
            <option value="lime">Citron vert</option>
            <option value="coconut">Noix de coco</option>
            <option value="mango">Mangue</option>
          </select>
        </label>
        <input type="submit" value="Envoyer" />
      </form>
    );
  }
}
```

Essayer sur CodePen

Au final, ça permet aux balises `<input type="text">`, `<textarea>`, et `<select>` de fonctionner de manière très similaire—elles acceptent toutes un attribut `value` que vous pouvez utiliser pour implémenter un composant contrôlé.

> Note
>
> Vous pouvez passer un tableau pour l'attribut `value`, permettant de sélectionner plusieurs valeurs dans un élément `select` :
>
> ```
> <select multiple={true} value={['B', 'C']}>
> ```

La balise `input type="file"` {#the-file-input-tag}

En HTML, un `<input type="file">` permet à l'utilisateur de sélectionner un ou plusieurs fichiers depuis son appareil et de les téléverser vers un serveur ou de les manipuler en JavaScript grâce à l'API File.

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

Sa valeur étant en lecture seule, c'est un composant **non-contrôlé** en React. Ce cas de figure et le sujet des composants non-contrôlés en général sera détaillé plus tard dans la documentation.

Gérer plusieurs saisies {#handling-multiple-inputs}

Quand vous souhaitez gérer plusieurs champs contrôlés, vous pouvez ajouter un attribut `name` à chaque champ et laisser la fonction gestionnaire choisir quoi faire en fonction de la valeur de `event.target.name`.

Par exemple :

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked :
target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Participe :
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
```

```
          Nombre d'invités :
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Essayer sur CodePen

Notez l'utilisation de la syntaxe des propriétés calculés pour mettre à jour la valeur de l'état correspondant au nom du champ.

```
this.setState({
  [name]: value
});
```

C'est équivalent à ce code ES5 :

```
var partialState = {};
partialState[name] = value;
this.setState(partialState);
```

Qui plus est, comme `setState()` fusionne automatiquement un état partiel dans l'état local courant, il nous a suffi de l'appeler avec les parties modifiées.

Valeur nulle des champs contrôlés {#controlled-input-null-value}

Définir la prop `value` sur un composant contrôlé empêche l'utilisateur de changer la saisie sauf si vous le permettez. Si vous spécifiez une `value` mais que le champ reste modifiable, alors `value` doit s'être accidentellement retrouvée à `undefined` ou `null`.

Le code suivant illustre ce cas de figure. (Le champ est verrouillé au démarrage mais devient modifiable après un court délai.)

```
ReactDOM.render(<input value="Salut" />, mountNode);

setTimeout(function() {
  ReactDOM.render(<input value={null} />, mountNode);
}, 1000);
```

Alternatives aux composants contrôlés {#alternatives-to-controlled-components}

Il est parfois fastidieux d'utiliser les composants contrôlés, car il vous faut écrire un gestionnaire d'événement pour chaque possibilité de changement des données, et gérer toute modification des saisies via un composant React. Ça peut devenir particulièrement irritant lors de la conversion d'un projet en React, ou l'intégration d'une application React avec une bibliothèque non-React. Dans ces situations, il est intéressant de connaître les composants non-contrôlés, une technique alternative pour implémenter les formulaires de saisie.

Solutions clé en main {#fully-fledged-solutions}

Si vous cherchez une solution complète gérant la validation, l'historique des champs visités, et la gestion de soumission de formulaire, Formik fait partie des choix populaires. Ceci dit, il repose sur les mêmes principes de composants contrôlés et de gestion d'état local—alors ne faites pas l'impasse dessus.

Faire remonter l'état

Plusieurs composants ont souvent besoin de refléter les mêmes données dynamiques. Nous conseillons de faire remonter l'état partagé dans leur ancêtre commun le plus proche. Voyons comment ça marche.

Dans cette section, nous allons créer un calculateur de température qui détermine si l'eau bout à une température donnée.

Commençons par un composant appelé `BoilingVerdict`. Il accepte une prop `celsius` pour la température, et il affiche si elle est suffisante pour faire bouillir l'eau :

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>L'eau bout.</p>;
  }
  return <p>L'eau ne bout pas.</p>;
}
```

Ensuite, nous allons créer un composant appelé `Calculator`. Il affiche un `<input>` qui permet de saisir une température et de conserver sa valeur dans `this.state.temperature`.

Par ailleurs, il affiche le `BoilingVerdict` pour la température saisie.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Saisissez la température en Celsius :</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />
        <BoilingVerdict
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

Essayer sur CodePen

Ajouter un deuxième champ {#adding-a-second-input}

Il nous faut à présent proposer, en sus d'une saisie en Celsius, une saisie en Fahrenheit, les deux devant rester synchronisées.

On peut commencer par extraire un composant `TemperatureInput` du code de `Calculator`. Ajoutons-y une prop `scale` qui pourra être soit `"c"`, soit `"f"` :

```
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Saisissez la température en {scaleNames[scale]} :</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Nous pouvons désormais modifier le composant `Calculator` pour afficher deux saisies de température :

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

Essayer sur CodePen

Nous avons maintenant deux champs de saisie, mais lorsque vous saisissez la température dans un des deux, l'autre ne se met pas à jour. Nous avons besoin de les garder synchronisés.

Qui plus est, nous ne pouvons pas afficher le `BoilingVerdict` depuis `Calculator`. Le composant `Calculator` n'a pas accès à la température saisie, car elle est cachée dans le `TemperatureInput`.

Écrire des fonctions de conversion {#writing-conversion-functions}

D'abord, écrivons deux fonctions pour convertir de Celsius à Fahrenheit et réciproquement :

```
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}
```

Ces deux fonctions convertissent des nombres. Écrivons une autre fonction qui prend en arguments une chaîne de caractères `temperature` et une fonction de conversion, et qui renvoie une chaîne. Nous utiliserons cette nouvelle fonction pour calculer la valeur d'un champ en fonction de l'autre.

Elle renvoie une chaîne vide pour une `temperature` incorrecte, et arrondit la valeur de retour à trois décimales :

```
function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}
```

Par exemple, `tryConvert('abc', toCelsius)` renvoie une chaîne vide, et `tryConvert('10.22', toFahrenheit)` renvoie `'50.396'`.

Faire remonter l'état {#lifting-state-up}

Pour l'instant, les deux éléments `TemperatureInput` conservent leur propre état local indépendamment l'un de l'autre :

```
class TemperatureInput extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    // ...
```

Cependant, nous voulons que les deux champs soient synchronisés. Lorsqu'on modifie le champ en Celsius, celui en Fahrenheit doit refléter la température convertie, et réciproquement.

Avec React, partager l'état est possible en le déplaçant dans le plus proche ancêtre commun. On appelle ça « faire remonter l'état ». Nous allons supprimer l'état local de `TemperatureInput` et le déplacer dans le composant `Calculator`.

Si le composant `Calculator` est responsable de l'état partagé, il devient la « source de vérité » pour la température des deux champs. Il peut leur fournir des valeurs qui soient cohérentes l'une avec l'autre. Comme les props des deux composants `TemperatureInput` viennent du même composant parent `Calculator`, les deux champs seront toujours synchronisés.

Voyons comment ça marche étape par étape.

D'abord, remplaçons `this.state.temperature` par `this.props.temperature` dans le composant `TemperatureInput`. Pour le moment, faisons comme si `this.props.temperature` existait déjà, même si nous allons devoir la passer depuis `Calculator` plus tard :

```
  render() {
    // Avant : const temperature = this.state.temperature;
    const temperature = this.props.temperature;
    // ...
```

On sait que les props sont en lecture seule. Quand la `temperature` était dans l'état local, le composant `TemperatureInput` pouvait simplement appeler `this.setState()` pour la changer. Cependant, maintenant que `temperature` vient du parent par une prop, le composant `TemperatureInput` n'a pas le contrôle dessus.

Avec React, on gère généralement ça en rendant le composant « contrôlé ». Tout comme un élément DOM `<input>` accepte des props `value` et `onChange`, notre `TemperatureInput` peut accepter des props `temperature` et `onTemperatureChange` fournies par son parent `Calculator`.

Maintenant, quand le composant `TemperatureInput` veut mettre à jour la température, il appelle `this.props.onTemperatureChange` :

```
  handleChange(e) {
    // Avant : this.setState({temperature: e.target.value});
    this.props.onTemperatureChange(e.target.value);
    // ...
```

> Remarque
>
> Les noms de props `temperature` et `onTemperatureChange` n'ont pas de sens particulier. On aurait pu les appeler n'importe comment, par exemple `value` et `onChange`, qui constituent une convention de nommage répandue.

La prop `onTemperatureChange` sera fournie par le composant parent `Calculator`, tout comme la prop `temperature`. Elle s'occupera du changement en modifiant son propre état local, entraînant un nouvel affichage des deux champs avec leurs nouvelles valeurs. Nous allons nous pencher très bientôt sur l'implémentation du nouveau composant `Calculator`.

Avant de modifier le composant `Calculator`, récapitulons les modifications apportées au composant `TemperatureInput`. Nous en avons retiré l'état local, et nous lisons désormais `this.props.temperature` au lieu de `this.state.temperature`. Plutôt que d'appeler `this.setState()` quand on veut faire un changement, on appelle désormais `this.props.onTemperatureChange()`, qui est fournie par le `Calculator` :

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    const temperature = this.props.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Saisissez la température en {scaleNames[scale]} :</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Intéressons-nous maintenant au composant `Calculator`.

Nous allons stocker la valeur courante de `temperature` et de `scale` dans son état local. C'est l'état que nous avons « remonté » depuis les champs, et il servira de « source de vérité » pour eux deux. C'est la représentation minimale des données dont nous avons besoin afin d'afficher les deux champs.

Par exemple, si on saisit 37 dans le champ en Celsius, l'état local du composant `Calculator` sera :

```
{
  temperature: '37',
  scale: 'c'
}
```

Si plus tard on change le champ Fahrenheit à 212, l'état local du composant `Calculator` sera :

```
{
  temperature: '212',
  scale: 'f'
}
```

On pourrait avoir stocké les valeurs des deux champs, mais en fait ce n'est pas nécessaire. Stocker uniquement la valeur la plus récente et son unité s'avère suffisant. On peut déduire la valeur de l'autre champ rien qu'à partir des valeurs de `temperature` et de `scale` stockées.

Les champs restent synchronisés car leurs valeurs sont calculées depuis le même état :

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }

  handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature,
toFahrenheit) : temperature;
```

```
    return (
      <div>
        <TemperatureInput
          scale="c"
          temperature={celsius}
          onTemperatureChange={this.handleCelsiusChange} />
        <TemperatureInput
          scale="f"
          temperature={fahrenheit}
          onTemperatureChange={this.handleFahrenheitChange} />
        <BoilingVerdict
          celsius={parseFloat(celsius)} />
      </div>
    );
  }
}
```

Essayer sur CodePen

Désormais, quel que soit le champ que vous modifiez, `this.state.temperature` et `this.state.scale` seront mis à jour au sein du composant `Calculator`. L'un des deux champ recevra la valeur telle quelle, et l'autre valeur de champ sera toujours recalculée à partir de la valeur modifiée.

Récapitulons ce qui se passe quand on change la valeur d'un champ :

- React appelle la fonction spécifiée dans l'attribut `onChange` de l'élément DOM `<input>`. Dans notre cas, c'est la méthode `handleChange` du composant `TemperatureInput`.
- La méthode `handleChange` du composant `TemperatureInput` appelle `this.props.onTemperatureChange()` avec la nouvelle valeur. Ses props, notamment `onTemperatureChange`, ont été fournies par son composant parent, `Calculator`.
- Au dernier affichage en date, le composant `Calculator` a passé la méthode `handleCelsiusChange` de `Calculator` comme prop `onTemperatureChange` du `TemperatureInput` en Celsius, et la méthode `handleFahrenheitChange` de `Calculator` comme prop `onTemperatureChange` du `TemperatureInput` en Fahrenheit. L'une ou l'autre de ces méthodes de `Calculator` sera ainsi appelée en fonction du champ modifié.
- Dans ces méthodes, le composant `Calculator` demande à React de le rafraîchir en appelant `this.setState()` avec la nouvelle valeur du champ et l'unité du champ modifié.
- React appelle la méthode `render` du composant `Calculator` afin de savoir à quoi devrait ressembler son UI. Les valeurs des deux champs sont recalculées en fonction de la température actuelle et de l'unité active. La conversion de température est faite ici.
- React appelle les méthodes `render` des deux composants `TemperatureInput` avec leurs nouvelles props, spécifiées par le `Calculator`. React sait alors à quoi devraient ressembler leurs UI.
- React appelle la méthode `render` du composant `BoilingVerdict`, en lui passant la température en Celsius dans les props.
- React DOM met à jour le DOM avec le verdict d'ébullition, et retranscrit les valeurs de champs souhaitées. Le champ que nous venons de modifier reçoit sa valeur actuelle, et l'autre champ est mis à jour avec la température convertie.

Chaque mise à jour suit ces mêmes étapes, ainsi les champs restent synchronisés.

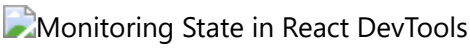
Ce qu'il faut retenir {#lessons-learned}

Il ne doit y avoir qu'une seule « source de vérité » pour toute donnée qui change dans une application React. En général, l'état est d'abord ajouté au composant qui en a besoin pour s'afficher. Ensuite, si d'autres composants en ont également besoin, vous pouvez faire remonter l'état dans l'ancêtre commun le plus proche. Au lieu d'essayer de synchroniser les états de différents composants, vous devriez vous baser sur des données qui se propagent du haut vers le bas.

Faire remonter l'état implique d'écrire plus de code générique *(boilerplate code, NdT)* qu'avec une liaison de données bidirectionnelle, mais le jeu en vaut la chandelle, car ça demande moins de travail pour trouver et isoler les bugs. Puisque tout état « vit » dans un composant et que seul ce composant peut le changer, la surface d'impact des bugs est grandement réduite. Qui plus est, vous pouvez implémenter n'importe quelle logique personnalisée pour rejeter ou transformer les saisies des utilisateurs.

Si quelque chose peut être dérivé des props ou de l'état, cette chose ne devrait probablement pas figurer dans l'état. Par exemple, plutôt que de stocker à la fois `celsiusValue` et `fahrenheitValue`, on stocke uniquement la dernière `temperature` modifiée et son unité `scale`. La valeur de l'autre champ peut toujours être calculée dans la méthode `render()` à partir de la valeur de ces données. Ça nous permet de vider ou d'arrondir la valeur de l'autre champ sans perdre la valeur saisie par l'utilisateur.

Quand vous voyez quelque chose qui ne va pas dans l'UI, vous pouvez utiliser les outils de développement React pour examiner les props et remonter dans l'arborescence des composants jusqu'à trouver le composant responsable de la mise à jour de l'état. Ça vous permet de remonter à la source des bugs :


Monitoring State in React DevTools

Composition ou héritage ?

React fournit un puissant modèle de composition, aussi nous recommandons d'utiliser la composition plutôt que l'héritage pour réutiliser du code entre les composants.

Dans cette section, nous examinerons quelques situations pour lesquelles les débutants en React ont tendance à opter pour l'héritage, et montrerons comment les résoudre à l'aide de la composition.

Délégation de contenu {#containment}

Certains composants ne connaissent pas leurs enfants à l'avance. C'est particulièrement courant pour des composants comme `Sidebar` ou `Dialog`, qui représentent des blocs génériques.

Pour de tels composants, nous vous conseillons d'utiliser la prop spéciale `children`, pour passer directement les éléments enfants dans votre sortie :

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

Ça permet aux autres composants de leur passer des enfants quelconques en imbriquant le JSX :

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Bienvenue
      </h1>
      <p className="Dialog-message">
        Merci de visiter notre vaisseau spatial !
      </p>
    </FancyBorder>
  );
}
```

Essayer sur CodePen

Tout ce qui se trouve dans la balise JSX `<FancyBorder>` est passé comme prop `children` au composant `FancyBorder`. Puisque `FancyBorder` utilise `{props.children}` dans une balise `<div>`, les éléments passés apparaissent dans la sortie finale.

Bien que cela soit moins courant, vous aurez parfois besoin de plusieurs « trous » dans un composant. Dans ces cas-là, vous pouvez créer votre propre convention au lieu d'utiliser `children` :

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}
```

Essayer sur CodePen

Des éléments React tels que `<Contacts />` et `<Chat />` sont simplement des objets, vous pouvez les passer comme props au même titre que n'importe quelle autre donnée. Cette approche peut vous rappeler la notion de “slots” présente dans d'autres bibliothèques, mais il n'y a aucune limitation à ce que vous pouvez passer en props avec React.

Spécialisation {#specialization}

Parfois, nous voyons nos composants comme des « cas particuliers » d'autres composants. Par exemple, nous pourrions dire que `WelcomeDialog` est un cas particulier de `Dialog`.

Avec React, on réalise aussi ça avec la composition ; un composant plus « spécialisé » utilise un composant plus « générique » et le configure grâce aux props :

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
```

```
    );
  }

  function WelcomeDialog() {
    return (
      <Dialog
        title="Bienvenue"
        message="Merci de visiter notre vaisseau spatial !" />
    );
  }
```

Essayer sur CodePen

La composition fonctionne tout aussi bien pour les composants à base de classe :

```
  function Dialog(props) {
    return (
      <FancyBorder color="blue">
        <h1 className="Dialog-title">
          {props.title}
        </h1>
        <p className="Dialog-message">
          {props.message}
        </p>
        {props.children}
      </FancyBorder>
    );
  }

  class SignUpDialog extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.handleSignUp = this.handleSignUp.bind(this);
      this.state = {login: ''};
    }

    render() {
      return (
        <Dialog title="Programme d'exploration de Mars"
                message="Comment devrions-nous nous adresser à vous ?">
          <input value={this.state.login}
                 onChange={this.handleChange} />
          <button onClick={this.handleSignUp}>
            Inscrivez-moi !
          </button>
        </Dialog>
      );
    }

    handleChange(e) {
      this.setState({login: e.target.value});
```

```
    }

    handleSignUp() {
      alert(`Bienvenue à bord, ${this.state.login} !`);
    }
  }
```

Essayer sur CodePen

Qu'en est-il de l'héritage ? {#so-what-about-inheritance}

Chez Facebook, nous utilisons React pour des milliers de composants, et nous n'avons pas encore trouvé de cas où nous aurions recommandé de créer des hiérarchies d'héritage de composants.

Les props et la composition vous donnent toute la flexibilité dont vous avez besoin pour personnaliser l'apparence et le comportement d'un composant de manière explicite et sûre. Souvenez-vous qu'un composant peut accepter tout type de props, y compris des valeurs primitives, des éléments React et des fonctions.

Si vous souhaitez réutiliser des fonctionnalités sans rapport à l'interface utilisateur entre les composants, nous vous suggérons de les extraire dans un module Javascript séparé. Les composants pourront alors importer cette fonction, cet objet ou cette classe sans avoir à l'étendre.

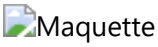
Penser en React

React est, à notre avis, la meilleure façon de créer des applis web vastes et performantes en JavaScript. Il a très bien tenu le coup pour nous, à Facebook et Instagram.

L'un des nombreux points forts de React, c'est la façon dont il vous fait penser aux applis pendant que vous les créez. Dans ce document, nous vous guiderons à travers l'élaboration avec React d'un tableau de données de produits proposant filtrage et recherche.

Commençons par une maquette {#start-with-a-mock}

Imaginez que nous avons déjà une API JSON et une maquette de notre designer. La maquette ressemble à ceci :

 Maquette

Notre API JSON renvoie des données qui ressemblent à ceci :

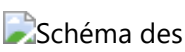
```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name:
"Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name:
"Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name:
"Basketball"},
  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod
Touch"},
  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone
5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus
7"}
];
```

Étape 1 : décomposer l'interface utilisateur en une hiérarchie de composants {#step-1-break-the-ui-into-a-component-hierarchy}

Pour commencer, dessinez des cases autour de chaque composant (et sous-composant) sur la maquette, et attribuez un nom à chacune. Si vous travaillez avec un designer, il se peut qu'elle l'ait déjà fait, alors allez lui parler ! Ses noms de calques Photoshop pourraient devenir les noms de vos composants React !

Mais comment savoir quelles parties devraient disposer de leurs propres composants ? Utilisez les mêmes techniques que lorsque vous décidez de créer une nouvelle fonction ou un nouvel objet. L'une de ces techniques est le principe de responsabilité unique, qui stipule qu'un composant ne devrait idéalement faire qu'une seule chose. S'il finit par grossir, il devrait être décomposé en sous-composants plus petits.

Comme vous affichez souvent un modèle de données JSON à un utilisateur, vous constaterez que si votre modèle a été correctement construit, votre interface utilisateur (et donc la structure de vos composants) correspondra aisément. En effet, l'interface utilisateur (UI) et les modèles de données tendent à adhérer à la même *architecture d'information*. Séparez votre UI en composants, où chaque composant représente juste un élément de votre modèle de données.

 Schéma des composants

Vous pouvez voir que nous avons cinq composants dans notre petite appli. Nous avons mis en italiques les données que chaque composant représente.

1. `FilterableProductTable` **(orange) :** contient l'intégralité de l'exemple
2. `SearchBar` **(bleu) :** reçoit toutes les *données saisies par l'utilisateur*
3. `ProductTable` **(vert) :** affiche et filtre la *collection de données* en fonction des *données saisies par l'utilisateur*
4. `ProductCategoryRow` **(turquoise) :** affiche un titre pour chaque *catégorie*
5. `ProductRow` **(rouge) :** affiche une ligne pour chaque *produit*

Si vous regardez `ProductTable`, vous verrez que l'en-tête du tableau (contenant les titres “Name” et “Price”) n'a pas son propre composant. C'est une question de préférence, et honnêtement les deux se valent. Dans cet exemple, nous l'avons laissé au sein de `ProductTable` car il fait partie de l’affichage de la *collection de données*, qui est de la responsabilité de `ProductTable`. Cependant, si cet en-tête devenait complexe (par exemple, si nous devions ajouter des options de tri), il deviendrait logique d'en faire son propre composant `ProductTableHeader`.

Maintenant que nous avons identifié les composants dans notre maquette, organisons-les en hiérarchie. Les composants qui apparaissent dans un autre composant sur la maquette doivent apparaître comme enfants dans cette hiérarchie :

- `FilterableProductTable`
  - `SearchBar`
  - `ProductTable`
    - `ProductCategoryRow`
    - `ProductRow`

Étape 2 : construire une version statique avec React {#step-2-build-a-static-version-in-react}

Voir le Pen Penser en React : Étape 2 sur CodePen.

Maintenant que vous avez votre hiérarchie de composants, il est temps d'implémenter votre appli. La façon la plus simple consiste à construire une version qui prend votre modèle de données et affiche une UI inerte. Il est préférable de découpler ces processus, car la construction d'une version statique nécessite beaucoup de code et aucune réflexion, alors qu’ajouter de l'interactivité demande beaucoup de réflexion et peu de code. Nous verrons pourquoi.

Pour créer une version statique de votre appli qui affiche votre modèle de données, vous devrez créer des composants qui en réutilisent d'autres et transmettent les données au moyen des *props*. Les *props* sont un moyen de transmettre des données de parent à enfant. Si vous êtes à l’aise avec le concept d'*état local*, **n'utilisez pas d'état local du tout** pour construire cette version statique. L'état local est réservé à l'interactivité, c'est-à-dire aux données qui évoluent dans le temps. Comme il s'agit d'une version statique de l'appli, vous n'en avez pas besoin.

Vous pouvez construire l’appli en partant de l’extérieur ou de l’intérieur. En d'autres termes, vous pouvez aussi bien commencer par construire les composants les plus hauts dans la hiérarchie (dans notre cas, `FilterableProductTable`), que par ceux les plus bas (`ProductRow`). Dans des exemples plus simples, il

est généralement plus facile de partir de l'extérieur, et sur des projets plus importants, il est plus facile de partir de l'intérieur et d'écrire les tests au fil de la construction.

À la fin de cette étape, vous disposerez d'une bibliothèque de composants réutilisables qui afficheront votre modèle de données. Les composants n'auront que des méthodes `render()` puisque c'est une version statique de l'application. Le composant au sommet de la hiérarchie (`FilterableProductTable`) prendra votre modèle de données en tant que *prop*. Si vous modifiez les données et appelez `ReactDOM.render()` à nouveau, l'UI sera mise à jour. On comprend comment votre UI est mise à jour et où y apporter des modifications, car il n'y a rien de compliqué. **Le flux de données unidirectionnel** de React (également appelé *liaison unidirectionnelle*) permet de maintenir la modularité et la rapidité de l'ensemble.

Jetez un œil à la doc de React si vous avez besoin d'aide pour cette étape.

Petit entracte : props ou état ? {#a-brief-interlude-props-vs-state}

Il existe deux types de données dans le « modèle » de React : les props et l'état local. Il est important de bien comprendre la distinction entre les deux ; jetez un coup d'œil à la doc officielle de React si vous n'êtes pas sûr·e de la différence. Vous pouvez aussi consulter la FAQ : Quelle est la différence entre `state` et `props` ?

Étape 3 : déterminer le contenu minimal (mais suffisant) de l'état de l'UI {#step-3-identify-the-minimal-but-complete-representation-of-ui-state}

Pour rendre votre UI interactive, vous devez pouvoir déclencher des modifications à votre modèle de données. React utilise pour cela l'**état local**.

Afin de construire correctement votre appli, vous devez d'abord penser à l'état modifiable minimal dont votre appli a besoin. La règle est simple : *ne vous répétez pas (Don't Repeat Yourself, aussi désigné par l'acronyme DRY, NdT)*. Déterminez la représentation la plus minimale possible de l'état dont votre appli a besoin, et calculez le reste à la demande. Par exemple, si vous construisez une liste de tâches, gardez un tableau des tâches sous la main ; pas besoin d'une variable d'état pour le compteur. Au lieu de ça, quand vous voulez afficher le nombre de tâches, prenez la longueur du tableau de tâches.

Pensez à toutes les données de notre application. On a :

- La liste des produits
- Le texte de recherche saisi par l'utilisateur
- La valeur de la case à cocher
- La liste filtrée des produits

Passons-les en revue pour déterminer lesquelles constituent notre état. Posez-vous ces trois questions pour chaque donnée :

1. Est-elle passée depuis un parent via les props ? Si oui, ce n'est probablement pas de l'état.
2. Est-elle figée dans le temps ? Si oui, ce n'est probablement pas de l'état.
3. Pouvez-vous la calculer en vous basant sur le reste de l'état ou les props de votre composant ? Si oui, ce n'est pas de l'état.

La liste des produits est passée via les props, ce n'est donc pas de l'état. Le texte de recherche et la case à cocher semblent être de l'état puisqu'ils changent avec le temps et ne peuvent être calculés à partir d'autre

chose. Enfin, la liste filtrée des produits ne constitue pas de l'état puisqu'elle peut être calculée en combinant la liste originale des produits avec le texte de recherche et la valeur de la case à cocher.

Au final, notre état contient :

- Le texte de recherche saisi par l'utilisateur
- La valeur de la case à cocher

Étape 4 : identifier où votre état doit vivre {#step-4-identify-where-your-state-should-live}

Voir le Pen Penser en React : Étape 4 sur CodePen.

Bon, nous avons identifié le contenu minimal de notre état applicatif. À présent, nous devons identifier quel composant modifie, ou *possède*, cet état.

Souvenez-vous : React se fonde sur un flux de données unidirectionnel qui descend le long de la hiérarchie des composants. Quant à savoir quel composant devrait posséder quel état, ce n'est pas forcément évident d'entrée de jeu. **C'est souvent la partie la plus difficile à comprendre pour les novices**, alors suivez ces étapes pour trouver la réponse :

Pour chaque partie de l'état de votre application :

- Identifiez chaque composant qui affiche quelque chose basé sur cet état.
- Trouvez leur plus proche ancêtre commun (un composant unique, au-dessus de tous les composants qui ont besoin de cette partie de l'état dans la hiérarchie).
- L'ancêtre commun ou un autre composant situé plus haut dans la hiérarchie devrait posséder cette portion d'état.
- Si vous ne trouvez pas de composant logique pour posséder cette partie de l'état, créez-en un exprès pour ça, et ajoutez-le quelque part dans la hiérarchie au-dessus de l'ancêtre commun.

Utilisons cette stratégie pour notre application :

- `ProductTable` doit filtrer la liste des produits en fonction de l'état et `SearchBar` doit afficher l'état du texte de recherche et de la case à cocher.
- Leur ancêtre commun est `FilterableProductTable`.
- Conceptuellement, il est logique que le texte du filtre et la valeur de la case à cocher soient dans `FilterableProductTable`

Parfait, nous avons donc décidé que `FilterableProductTable` possèdera notre état. Tout d'abord, ajoutez une propriété d'instance `this.state = {filterText: '', inStockOnly: false}` dans le `constructor` de `FilterableProductTable` pour refléter l'état initial de votre application. Ensuite, passez `filterText` et `inStockOnly` à `ProductTable` et `SearchBar` via leurs props. Enfin, utilisez ces props pour filtrer les lignes dans `ProductTable` et définir les valeurs des champs du formulaire dans `SearchBar`.

Vous pouvez commencer à voir comment votre application se comportera : définissez `filterText` à `"ball"` et rafraîchissez votre appli. Vous verrez que le tableau de données est correctement mis à jour.

Étape 5 : ajouter le flux de données inverse {#step-5-add-inverse-data-flow}

Voir le Pen Penser en React : Étape 5 sur CodePen.

Pour le moment, nous avons construit une appli qui s'affiche correctement en fonction des props et de l'état qui descendent le long de la hiérarchie. À présent, il est temps de permettre la circulation des données dans l'autre sens : les composants de formulaire situés plus bas dans la hiérarchie ont besoin de mettre à jour l'état dans `FilterableProductTable`.

React rend ce flux de données explicite pour vous aider à comprendre le fonctionnement de votre programme, mais cela demande un peu plus de code qu'une liaison de données bidirectionnelle classique.

Si vous essayez de saisir du texte ou de cocher la case dans la version actuelle de l'exemple, vous verrez que React ne tient pas compte de vos saisies. C'est volontaire, car nous avons spécifié l'attribut `value` de l'élément `input` pour qu'il soit toujours égal à l'état passé depuis `FilterableProductTable`.

Réfléchissons à ce que nous voulons qu'il se passe. Nous voulons garantir que chaque fois que l'utilisateur met à jour le formulaire, nous mettons à jour l'état pour refléter la saisie de l'utilisateur. Puisque les composants ne peuvent mettre à jour que leur propre état, `FilterableProductTable` passera une fonction de rappel à `SearchBar`, qui devra être déclenchée chaque fois que l'état doit être mis à jour. Nous pouvons utiliser l'événement `onChange` des champs pour cela. Les fonctions de rappel passées par `FilterableProductTable` appelleront `setState()`, et l'application sera mise à jour.

Et c'est tout {#and-thats-it}

Avec un peu de chance, vous avez maintenant une idée de la façon de penser la construction de vos composants et applis en React. Bien que ça demande peut-être un peu plus de code que vous n'en avez l'habitude, souvenez-vous que le code est lu beaucoup plus souvent qu'il n'est écrit, et que ce type de code, modulaire et explicite, est moins difficile à lire. Plus vous écrirez de composants, plus vous apprécierez cette clarté et cette modularité, et avec la réutilisation du code, le nombre de vos lignes de code commencera à diminuer. 😀

React Flux & Redux

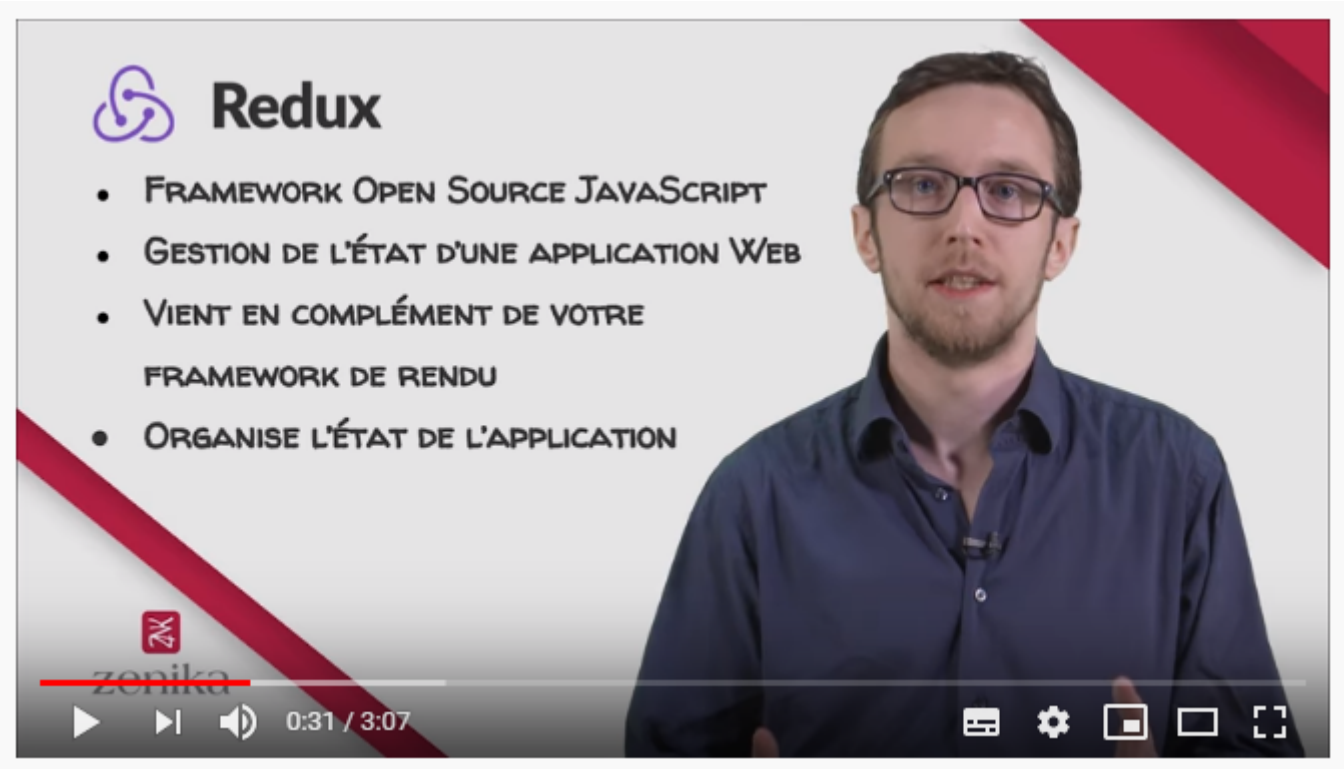
React FLux & Redux

flux

[...]

Redux

Une video explicative de Redux vous le présente ici.



https://www.youtube.com/watch?v=QEjf1W-rRIo

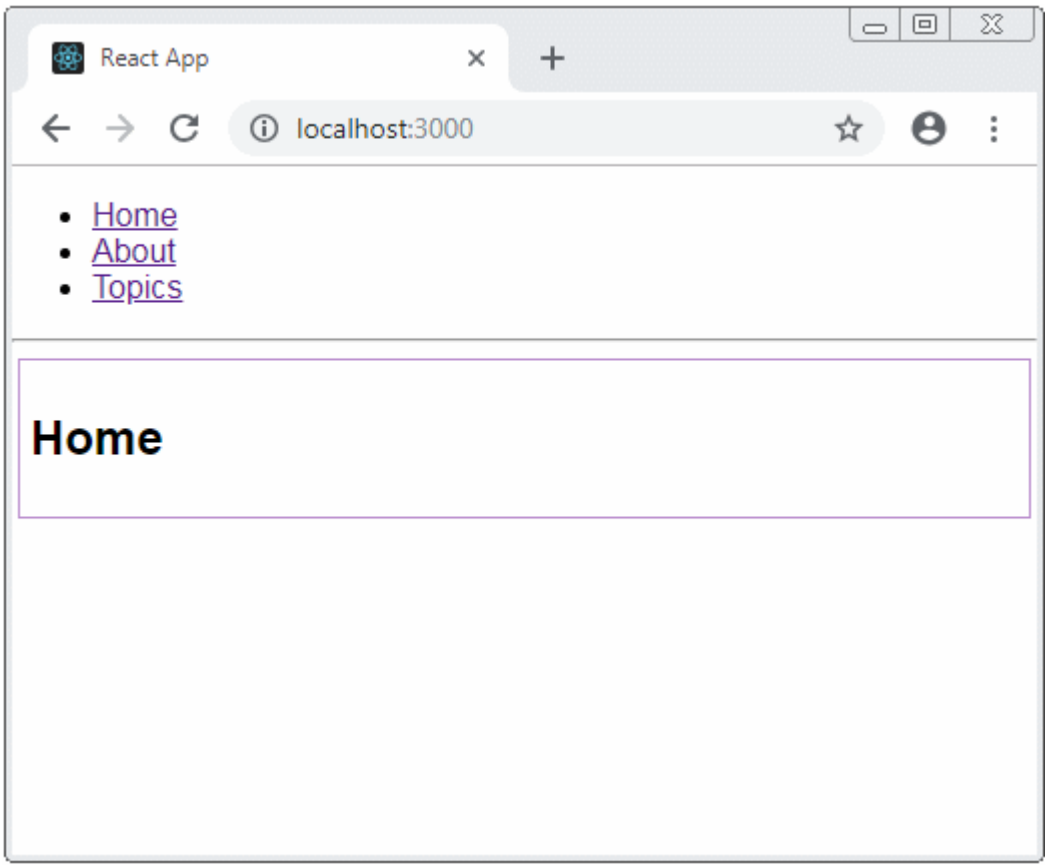
React Native

React Native [...]

react router

React router est une librairie standard de react. Cette lib assure la gestion et le controle des url pour l'acces a des vues spécifique ou composées par le biais d'une adresse spécifique

Ce concept est fort agréable lorsque l'on travaille sur une SPA ou S.ingle P.age A.pplication car dans notre application react, nous allons mettre en place plein de composants et qu'une seule page html. ce systeme permettra d'orchestrer les URL en fonction des besoins et de l'avancement de l'utilisateur sur les liens de l'app.



Deux catégories d'URL existent :

- les hash ou <HashRouter>

http://example.com/#/about

Url Caractérisée par la présence d'un # pour specifier au navigateur que la partie se trouvant apres le # est une partie d'adresse appartenant au client de navigation et donc à gérrer par js.

- url html5 ou <BrowserRouter>

http://example.com/about

Url ressemblant fortement aux url serveur habituelle, il est impossible de voir la part serveur de la part navigateur dans l'adresse hormis l'adresse principale du serveur.

Installons react-router dans une app

pour limiter les interraction avec notre projet nous allons onfigurer un nouveau projet graçe a creat-react-app à laquelle nous ajouterons react-router puis nous le configurons.

Etape 1 installer react-router-dom

Pour cette éttape nous assumons partir d'un projet fraichement créer par creat-react-app et nous nous placerons dans le répertoire du projet pour exécture les commandes suivantes.

```
npm install react-router-dom
```

Etape 2 configuration de notre appli

Dans cette exmple notre appli gérre 3 adresses url sans paramètres dans l'url (nous reviendrons sur ce point plus tard).

```
import React from "react";
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from "react-router-dom";

export default function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
            <li>
              <Link to="/users">Users</Link>
            </li>
          </ul>
        </nav>

        {/* A <Switch> looks through its children <Route>s and
            renders the first one that matches the current URL. */}
        <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/users">
            <Users />
```

```
            </Route>
            <Route path="/">
              <Home />
            </Route>
          </Switch>
        </div>
      </Router>
    );
  }

  function Home() {
    return <h2>Home</h2>;
  }

  function About() {
    return <h2>About</h2>;
  }

  function Users() {
    return <h2>Users</h2>;
  }
```

Dans cette exemple, apparait une balise Router qui contient tout le rendu du composant. dans ce routeur, il faut identifié 2 choses :

- les liens

    Identifés par la balise Link qui est un composant de React qui fabriquera la balise <a> correspondante. On peut aussi observer l'attribut to qui définit le liens de destination dans l'application

```
<Link to="/users">Users</Link>
```

- la gestion des routes

    Identifés par la balise ** Route ** qui est un composant de React qui fabriquera le contenu quelle contient sur la régle de la route est bien celle présente dans le navigateur

```
  <Route path="/users"><Users/></Route>
```

Un path est une regex, autrement dit beaucoup de fantaisies sont possible sur ce selecteur d'url. De ce fait, il est possible de créer une route qui gerra les "404" en définissant un sélécteur generique "*". il sera biensur placé en dernier dans notre balise switch pour être sur quelle ne se déclenche que si aucune autre correspondance etais valide. Il vas de soit qu'une fonction ou un composant est deja pret a etre mise en place pour l'affichage

```
    <Route path="*"><Page404/></Route>
```

Gestion des paramétres de route

```
//pour l'expression de la route
<Route path={`/topic/:topicId`}>[...]</Route>

//pour la recuperation du parametre
function Topic() {
  let { topicId } = useParams();
  return <h3>Requested topic ID: {topicId}</h3>;
}
```

ORSYS
formation

à bientôt.

Annexes

redux vs flux conférence

Vidéo conférence en anglais sur redux et flux