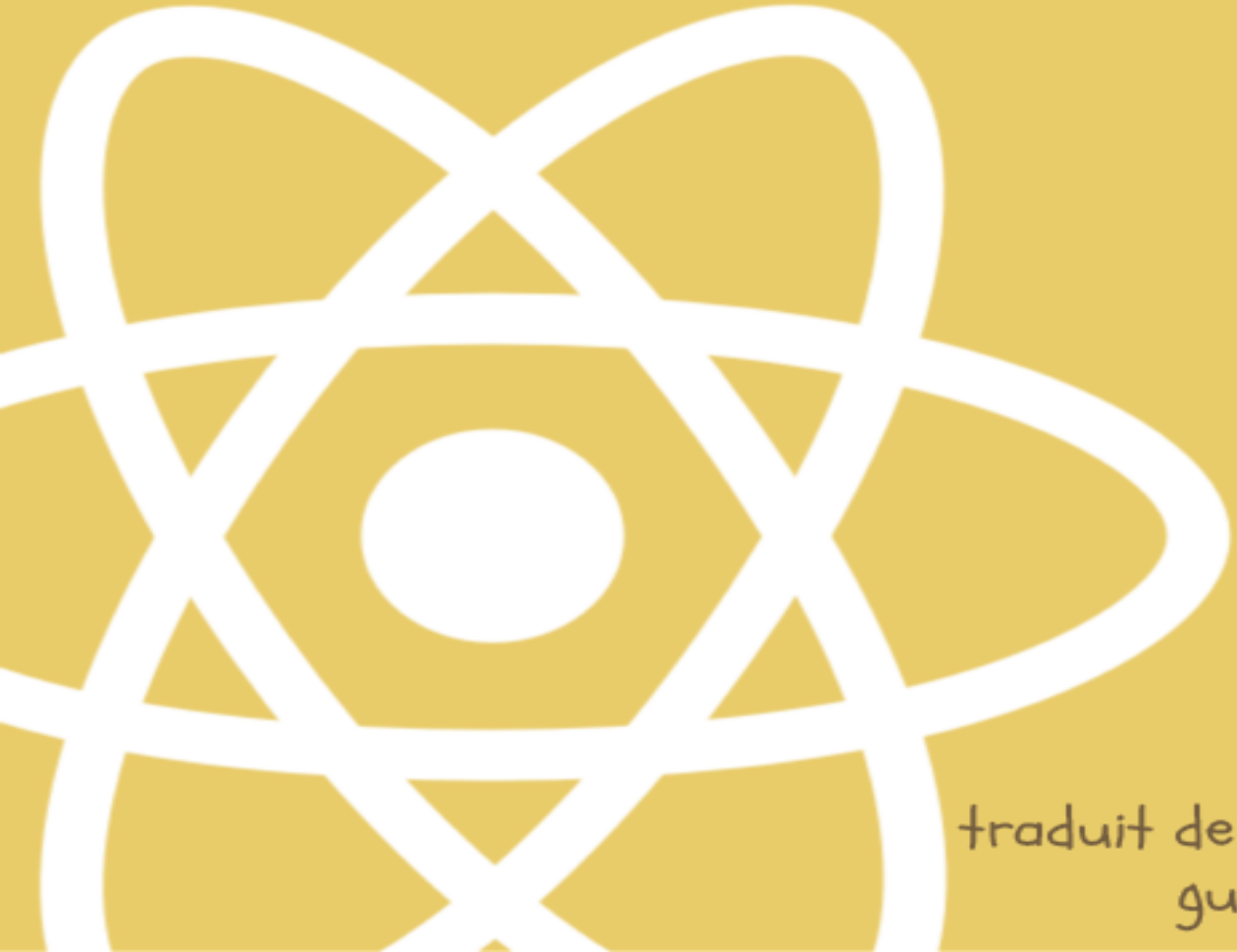
by robin wieruch

the Road to learn React



traduit de l'anglais par
guillaume borg

The Road to learn React (French)

Your journey to master plain yet pragmatic React

Robin Wieruch et Guillaume Borg

Ce livre est en vente à http://leanpub.com/the-road-to-learn-react-french

Version publiée le 2018-08-02

Ce livre est publié par Leanpub. Leanpub permet aux auteurs et aux éditeurs de bénéficier du Lean Publishing. Lean Publishing consiste à publier à l'aide d'outils très simples de nombreuses itérations d'un livre électronique en cours de rédaction, d'obtenir des retours et commentaires des lecteurs afin d'améliorer le livre.

Tweet ce livre !

S'il vous plaît aidez Robin Wieruch et Guillaume Borg en parlant de ce livre sur Twitter !

Le tweet suggéré pour ce livre est :

I am going to learn #ReactJs with The Road to learn React by @rwieruch Join me on my journey https ://roadtoreact.com

Le hashtag suggéré pour ce livre est #ReactJs.

Découvrez ce que les gens disent à propos du livre en cliquant sur ce lien pour rechercher ce hashtag sur Twitter :

#ReactJs

Table des matières

Avant-propos

The Road to learn React vous apprend les fondamentaux de React. Vous y construirez tout au long une application concrète en pur React sans outillage superflux. Tout depuis l'installation du projet jusqu'au déploiement sur un serveur vous sera expliqué. Dans chaque chapitre, le livre embarque du contenu de lecture supplémentaire référencé et des exercices. Après lecture du livre, vous serez en mesure de construire vos propres React applications. Le contenu est mis à jour par moi, et la communauté.

Dans the Road to learn React, Je souhaite fournir une fondation avant que vous commencez à plonger à l'intérieur du très large écosystème React. Il a moins d'outillages et moins de gestion d'état externe, mais possède beaucoup d'informations à propos de React. Il explique les concepts généraux, les patrons de conceptions et les meilleurs pratiques au sein d'une application React réelle.

Vous apprendrez à construire votre propre application React. Elle couvre des fonctionnalités réelles comme la pagination, le cache côté client et les intéractions telle que la recherche et le tri. De plus, tout du long vous ferez une transition de JavaScript ES5 à JavaScript ES6. J'espère que ce livre a su capturer mon enthousiasme pour React et JavaScript et vous aide à bien débuter.

A propos de l'auteur

Robin Wieruch est ingénieur logiciel et web allemand qui est dévoué à apprendre et enseigner la programmation en JavaScript. Après obtention de son diplôme de master d'université en science informatique, il a continué son apprentissage de son propre chef tous les jours. Il a gagné de l'expérience auprès de start-ups, où il a utilisé ardemment JavaScript professionnel et sur son temps libre, lui procura l'opportunité d'enseigner aux autres ces thèmes.

Pendant quelques années, Robin a étroitement travaillé avec une grande équipe d'ingénieurs dans une entreprise appelé Small Improvements sur une application de grande taille. L'entreprise construite un produit SaaS permettant aux clients de créer une culture de feedback au sein de leur entreprise. Derrière tout çà, l'application fonctionnait avec JavaScript pour le frontend et Java pour le backend. Niveau frontend, la première itération a été écrite en Java avec le Framework Wicket et jQuery. Lorsque la première génération de SPAs devint populaire, l'entreprise migra vers Angular 1.x pour la partie frontend de l'application. Après avoir utiliser Angular pendant plus de 2 ans, il devint clair qu'Angular n'était pas la meilleure solution pour travailler avec l'état intensif des applications. C'est pourquoi l'entreprise entreprit un saut final vers React et Redux qui a permis d'opérer sur une important mise à l'échelle avec succès.

Durant son passage dans l'entreprise, Robin écrivit régulièrement des articles autour du web développement sur son site personnel. Il reçut de bons retours concernant ses articles qui lui permit d'améliorer son style d'écriture et d'enseignement. Article après article, Robin améliora ses capacités à enseigner aux autres. Son premier article comprenait trop de choses ce qui était assez accablant pour les étudiants, mais il les améliora au fil du temps en se concentrant et en enseignant seulement un sujet.

De nos jours, Robin est un enseignant auto-entrepreneur. Il consacre tout son temps à voir les étudiants s'épanouir en leur donnant des objectifs clairs avec de courtes boucles de réaction. C'est quelque chose que vous apprendriez au sein d'une entreprise de feedback, n'est ce pas ? Mais sans coder soi-même, il ne serait pas possible d'enseigner des sujets. C'est pourquoi il a investit son temps restant dans la programmation. Vous pouvez trouver plus d'informations à propos de Robin et les façons de le supporter et de travailler avec lui sur son site[1].

1. https://www.robinwieruch.de/about

Témoignages

Il y a beaucoup de témoignages[2], notations[3] et avis[4] à propos du livre qui devrait confirmer sa qualité. J'en suis très satisfait, car j'attendais pas d'aussi importants retours. Si vous appréciez également le livre, j'adorerais également trouver votre avis quelque part. Cela m'aide à diffuser le livre. La suite montre un court extrait de ces avis positifs :

Muhammad Kashif[5] : "The Road to Learn React est un livre unique que je recommande à tous étudiants ou professionnels intéressés dans l'apprentissage des bases de React jusqu'aux niveaux avancées. Il embarque des astuces pertinentes et des techniques dont il est difficile de trouver ailleurs, et est remarquablement complet avec l'usage d'exemples et de références pour illustrer les problèmes, j'ai 17 ans d'expérience dans le développement d'applications desktop et web, et avant la lecture de ce livre j'avais des difficultés dans l'apprentissage de React, mais ce livre fonctionne tel de la magie."

Andre Vargas[6] : "The Road to Learn React de Robin Wieruch est un livre formidable ! La major partie de ce que j'ai appris à propos de React et de l'ES6 est dû à ce livre !"

Nicholas Hunt-Walker, Instructor of Python at a Seattle Coding School[7] : "C'est l'un des livres de développement le plus informatif et bien écrie avec lequel j'ai pu travailler. Une solide introduction à React et l'ES6."

Austin Green[8] : "Merci, j'adore le livre. Parfait mélange pour apprendre React, l'ES6 et les concepts de programmation de plus haut niveau."

Nicole Ferguson[9] : "J'ai fait le cours Road to Learn React de Robin ce weekend et je me sens coupable d'avoir eu tant d'amusements."

Karan[10] : "Je viens juste de terminer votre Road to React. Le meilleur livre pour un débutant dans le monde de React et de JS. Avec une élégante exposition à l'ES. Kudos ! :)"

Eric Priou[11] : "The Road to learn React de Robin Wieruch est un must à lire. Épuré et concis pour du React et JavaScript."

A Rookie Developer : "Je viens juste de terminer le livre en tant que développeur rookie, merci pour tout ce travail. Il était facile à suivre et je me sens confiant dans le démarrage d'une nouvelle application en partant de rien dans les jours qui viennent. Le livre était bien meilleur que le tutoriel

2. https://roadtoreact.com/
3. https://www.goodreads.com/book/show/37503118-the-road-to-learn-react
4. https://www.amazon.com/dp/B077HJFCQX
5. https://twitter.com/appsdevpk/status/848625244956901376
6. https://twitter.com/andrevar66/status/853789166987038720
7. https://twitter.com/nhuntwalker/status/845730837823840256
8. https://twitter.com/AustinGreen/status/845321540627521536
9. https://twitter.com/nicoleffe/status/833488391148822528
10. https://twitter.com/kvss1992/status/889197346344493056
11. https://twitter.com/erixtekila/status/840875459730657283

officiel de React.js que j'ai essayé plus tôt (et que je n'ai pu terminer à cause d'un manque de détail). Les exercices à la fin de chaque section étaient très enrichissants."

Student : "Le meilleur livre pour apprendre ReactJS. Le projet avance avec les concepts que nous venons d'apprendre ce qui aide à saisir le sujet. J'ai trouvé "coder et apprendre' comme la meilleure façon pour maitriser la programmation et ce livre fait exactement cela."

Thomas Lockney[12] : "Une assez solide introduction à React qui n'a pas la prétention d'être complet. Je souhaitais juste avoir un aperçu pour comprendre de quoi cela parlait et c'est exactement ce que ce livre m'a procuré. Je n'ai pas suivi toutes les petites remarques pour en apprendre plus sur les nouvelles fonctionnalités ES6 j'ai juste oublié ("Je ne dirais pas que cela m'a manqué, Bob."). Mais je suis sûr pour ceux d'entre vous qui ont pris le temps et se sont appliqués à suivre cela, vous pouvez probablement en apprendre bien plus que simplement ce que le livre enseigne."

12. https://www.goodreads.com/review/show/1880673388

Éducation pour les enfants

Le livre devrait permettre à tout un chacun d'apprendre React. Cependant, pas tout le monde à la possibilité d'utiliser ces ressources, car pas tout le monde n'a pu apprendre l'anglais en premier lieu. Ainsi je veux utiliser le projet pour supporter les projets qui enseignent aux enfants l'anglais dans les pays en voie de développement.

- Du 11 au 18 avril 2017, redistribuer, en apprenant[13]

13. https://www.robinwieruch.de/giving-back-by-learning-react/

FAQ

Comment être tenu à jour ? J'ai deux chaînes ou je partage des nouvelles à propos de mon contenu. Vous pouvez soit souscrire pour recevoir les mises à jour par email[14] ou me suivre sur Twitter[15]. Indépendamment de la chaîne, mon objectif est de partager uniquement du contenu qualitatif. Vous ne recevrez jamais de spam. Une fois que vous recevez la mise à jour comme quoi le livre a été modifié, vous pouvez télécharger sa nouvelle version.

Cela utilise-t-il la dernière version de React ? Le livre reçoit toujours une mise à jour lorsque React fait une mise à jour. Habituellement, les livres sont dépassés assez tôt après leur sortie. Puisque ce livre est autopublié, je peux le mettre à jour quand je le souhaite.

Couvre-t-il Redux ?? Il ne le couvre pas. Par conséquent, j'ai écrit un second livre. The Road to learn React vous fournit de solide fondations avant de plonger dans des sujets avancés. L'implémentation d'une application échantillon dans le livre montrera que vous n'avez pas besoin de Redux pour construire une application en React. Après que vous ayez lu le livre, vous devrez être capable d'implémenter une application complète sans Redux. Ensuite, vous pouvez lire mon second livre, Taming the State in React, pour apprendre Redux[16].

Utilise-t-il JavaScript ES6? Oui. Mais ne vous inquiétez pas. Vous n'aurez pas de problème si vous êtes familier avec JavaScript ES5. Toutes fonctionnalités JavaScript ES6, que je décris lors de ce cheminement pour apprendre React, fera une transition de ES5 à ES6. Toutes les fonctionnalités seront expliquées tout du long. Le livre n'apprend pas seulement React, mais aussi toutes les fonctionnalités JavaScript ES6 utiles pour React.

Ajouteras-tu plus de chapitres dans le futur ? Si vous avez acheté l'un des packages étendus qui vous confèrent un accès au code source des projets, à l'ensemble des captures d'écrans ou n'importe quelle autre option, vous devriez les retrouver sur votre dashboard de cours[17]. Si vous avez acheté le cours autre part que sur la plateforme officielle de cours Road to React[18], vous aurez besoin de créer un compte sur la platforme, aller sur la page Admin et à me communiquer l'un des templates d'email. Après quoi je pourrai vous enrôler au sein du cours. Si vous n'avez pas acheté l'un des packages étendus, vous pouvez me communiquer à n'importe quel moment pour mettre à jour votre contenu pour accéder au code source des projets et à l'ensemble des captures d'écrans.

Comment puis-je être aidé durant la lecture du livre ? Le livre a un groupe Slack[19] pour les personnes qui sont en train de lire le livre. Vous pouvez rejoindre la chaine et être aidé ou aider les autres. Après tout, aider les autres peut permettre d'intérioriser votre apprentissage aussi. S'il n'y a personne pour vous aider, vous pouvez toujours me contacter.

14. https://www.getrevue.co/profile/rwieruch
15. https://twitter.com/rwieruch
16. https://roadtoreact.com
17. https://roadtoreact.com/my-courses
18. https://roadtoreact.com
19. https://slack-the-road-to-learn-react.wieruch.com/

Y a-t-il des zones de danger ? Si vous rencontrez des problèmes, s'il vous plaît rejoignez le groupe Slack. De plus, vous pouvez jeter un oeil aux issues ouvertes sur GitHub[20] concernant le livre. Peut-être votre problème a déjà été mentionné et vous pouvez trouver la solution à son propos. Si votre problème n'est pas mentionné, n'hésitez pas à ouvrir un issue où vous pouvez expliquer votre problème, peut être fournir un screenshot, et quelques détails supplémentaires (ex : page du livre, version de node). Après quoi, j'essaye d'embarquer tous les fixes dans les prochaines éditions du livre.

Puis-je aider à l'améliorer ? Oui. Je serai ravi d'entendre vos retours. Vous pouvez simplement ouvrir un issue sur GitHub[21]. Cela peut être encore pour un souhait d'amélioration technique ou encore un mot écrit. Je ne suis pas un anglais natif c'est pour cela que n'importe quelle aide est appréciée. Vous pouvez ouvrir des pull requests sur la page Github également.

Y as-t-il une garantie de remboursement ? Oui, il y a une garantie de remboursement à 100% pendant deux mois, si vous estimez que cela ne convient pas. Merci de me contacter pour avoir un remboursement.

Comment supporter le projet ? Si vous croyez au contenu que j'ai créé, vous pouvez me supporter[22]. De plus, je vous serais reconnaissant si vous prêchez la bonne parole à propos de ce livre après l'avoir lu et apprécié. Enfin, j'adorerai vous avoir comme mon Patron sur Patreon[23].

Quelle est ta motivation derrière ce livre ? Je souhaite dire un mot sur ce sujet de manière cohérente. Vous trouvez souvent du contenu en ligne qui ne reçoit aucune mise à jour ou enseigne seulement une petite partie d'un sujet. Lorsque vous apprenez quelques choses de nouveau, les personnes luttent pour trouver du contenu cohérent et des ressources à jour sur lequel apprendre. Je veux vous donner cette expérience d'apprentissage consistant et à jour. De plus, j'espère que je peux supporter les minorités avec mes projets en leur donnant le contenu gratuitement ou en ayant d'autres impactes[24]. Qui plus est, récemment, j'ai trouvé mon chemin en enseignant aux autres la programmation. C'est une activité pleine de sens pour moi que je préfère à la place de n'importe quel métier de 9 à 5 au sein de n'importe quelle entreprise. C'est pourquoi j'espère poursuivre ce chemin dans le futur.

Y a-t-il un appel à l'action ? Oui. Je veux que vous preniez un moment pour penser à une personne qui pourrait être intéressée par l'apprentissage de React. La personne pourrait avoir déjà montré de l'intérêt, peut-être en plein apprentissage de React ou pourrait ne pas encore être consciente de vouloir apprendre React. Entrez en contact avec cette personne et partagez le livre. Cela signifie beaucoup pour moi. Le livre est destiné à être offert aux autres.

20. https://github.com/rwieruch/the-road-to-learn-react/issues
21. https://github.com/rwieruch/the-road-to-learn-react
22. https://www.robinwieruch.de/about/
23. https://www.patreon.com/rwieruch
24. https://www.robinwieruch.de/giving-back-by-learning-react/

Change Log

Le 10 Janvier 2017 :

- v2 Pull Request[25]
- Encore plus abordable pour les débutants
- 37% de contenu supplémentaire
- 30% de contenu amélioré
- 13 chapitres améliorés et nouveaux
- 140 pages de ressources didactiques
- + cours interactif du livre sur educative.io[26]

Le 8 Mars 2017 :

- v3 Pull Request[27]
- 20% de contenu supplémentaire
- 25% de contenu amélioré
- 9 nouveaux chapitres
- 170 pages de ressources didactiques

Le 15 Avril 2017 :

- Mise à niveau en React 15.5

Le 5 Juillet 2017 :

- Mise à niveau en node 8.1.3
- Mise à niveau en npm 5.0.4
- Mise à niveau en create-react-app 1.3.3

Le 17 Octobre 2017 :

- Mise à niveau en node 8.3.0
- Mise à niveau en npm 5.5.1
- Mise à niveau en create-react-app 1.4.1

25. https://github.com/rwieruch/the-road-to-learn-react/pull/18
26. https://www.educative.io/collection/5740745361195008/5676830073815040
27. https://github.com/rwieruch/the-road-to-learn-react/pull/34

- Mise à niveau en React 16
- v4 Pull Request[28]
- 15% de contenu supplémentaire
- 15% de contenu amélioré
- 3 nouveaux chapitres (Bindings, Event Handlers, Gestion d'erreur)
- 190+ pages de ressources didactiques
- +9 projets avec code source[29]

17. **February 2018** :

- Mise à niveau en node 8.9.4
- Mise à niveau en npm 5.6.0
- Mise à niveau en create-react-app 1.5.1
- v5 Pull Request[30]
- plus de chemins d'apprentissage
- contenu de lecture référencé supplémentaire
- 1 nouveau chapitre (Axios à la place de Fetch)

28. https://github.com/rwieruch/the-road-to-learn-react/pull/72
29. https://roadtoreact.com/course-details?courseId=THE_ROAD_TO_LEARN_REACT
30. https://github.com/the-road-to-learn-react/the-road-to-learn-react/pull/105

Comment le lire ?

Le livre est ma tentative pour enseigner React tandis que vous écrirez une application. C'est un guide pratique pour apprendre React et non un travail de référence autour de React. Vous écrirez une application Hacker News qui interagira avec une API réelle. Parmi plusieurs sujets intéressants, il couvre la gestion de l'état au sein de React, le cache et les interactions (le tri et la recherche). Tout du long vous apprendrez les meilleurs pratiques et patrons de conception de React.

De plus, le livre vous donne une transition de JavaScript ES5 vers JavaScript ES6. React adopte beaucoup de fonctionnalités JavaScript ES6 et je souhaite vous montrer comment vous pouvez les utiliser.

En général chaque chapitre du livre repose sur le chapitre précédent. Chaque chapitre vous enseignera quelque chose de nouveau. Ne vous précipitez sur le livre. Vous devez intérioriser chaque étape. Vous pouvez appliquer vos propres implémentations et en lire davantage sur le sujet. Après chaque chapitre je vous donne quelques lectures supplémentaires et implémentations ainsi que de la lecture supplémentaire à propos du sujet. Si vous voulez vraiment apprendre React, Je vous recommande chaudement de lire la documentation supplémentaire et de faire les quelques exercices pratiques. Après que vous ayez lu un chapitre, devenez à l'aise avec les acquis avant de continuer.

En fin de compte vous aurez une complète application React en production. Je suis très impatient de voir vos résultats, donc s'il vous plaît écrivez-moi lorsque vous avez fini le livre. Le chapitre final du livre vous donnera plusieurs options pour poursuivre votre voyage auprès de React. En général vous trouverez beaucoup de sujets reliés à React sur mon site web personnel[31].

Puisque vous lisez le livre, je devine que vous êtes nouveau à React. C'est parfait. À la fin j'espère avoir des retours pour améliorer le contenu pour permettre à tout à chacun d'apprendre React. Vous pouvez avoir un impact direct sur GitHub[32] ou écrivez-moi sur Twitter[33].

31. https://www.robinwieruch.de/
32. https://github.com/rwieruch/the-road-to-learn-react
33. https://twitter.com/rwieruch

Contributeurs

Plusieurs personnes ont rendu possible l'écriture et l'amélioration de *The Road to learn React* au fur et à mesure. Actuellement, c'est l'un des livres React.js les plus téléchargés. Originellement le livre a été écrit par l'Ingénieur Logiciel allemand Robin Wieruch[34]. Mais toutes les traductions du livre ne seraient pas possibles sans l'aide de collaborateurs. Cette version du livre a été traduite par l'Ingénieur Logiciel français Guillaume Borg[35] publiant du contenu portant sur des technologies Web autour du JavaScript et du W3C. Son contenu est en premier lieu francophone et accessible via sa chaîne Youtube[36], son compte Twitter[37] ainsi que son compte Github[38].

34. https://www.robinwieruch.de/
35. https://www.linkedin.com/in/guillaume-borg-0904b8a8/
36. https://www.youtube.com/c/EchyzenWebsite
37. https://twitter.com/EchyzenWebsite
38. https://github.com/GuillaumeUnice

Introduction à React

Ce chapitre fournit une introduction à React. Peut-être vous demander vous : Pourquoi devrais-je apprendre React ? L'objectif de ce chapitre est de répondre à cette interrogation. Après quoi, nous plongerons à l'intérieur de l'écosystème en débutant par votre première application React à partir de rien nécessitant zéro configuration. Durant cette phase, vous serez introduit à JSX ainsi qu'au ReactDOM. Soyez prêt pour vos premiers composants React.

Salut, mon nom est React.

Pourquoi devrez-vous, vous embêter à apprendre React Ces dernières années les applications web monopage (SPA[39]) sont devenu populaires. Les frameworks comme Angular, Ember et Backbone aident les développeurs JavaScript à construire des applications web modernes au-delà du simple usage de Vanilla JavaScript et JQuery. La liste de ces solutions populaires n'est pas exhaustive. Il existe un large panel de frameworks pour SPA. Quand on compare les dates de sorties officielles, la plupart d'entre eux font partie de la première génération de SPAs : Angular 2010, Backbone 2010, Ember 2011.

La sortie officielle de React par Facebook date de 2013. React n'est pas un framework SPA à proprement parler mais une bibliothèque gérant que la vue. C'est le "V" du fameux MVC[40] (Model View Controller). Il permet seulement de rendre des composants en tant qu'élément visionnable dans le navigateur. Naturellement, l'écosystème autour de React rend possible la création de SPAs.

Mais pourquoi vous devrez utiliser React plutôt que la première génération de frameworks SPA ? Tandis que la première génération de frameworks essayèrent de résoudre beaucoup de problématiques à la fois, React vous assiste seulement pour construire votre couche : vue. C'est une bibliothèque et non un framework. L'idée derrière cela : votre vue est une hiérarchie de composants composables.

Dans React vous pouvez conserver votre attention sur la couche : vue avant d'introduire plus de concepts pour votre application. Tous les autres concepts sont d'autres *building block* pour votre SPA. Ces *building blocks* sont essentiels pour construire une application mûre. Ils arrivent avec deux avantages.

Tout d'abord, vous pouvez apprendre les constructions de blocs étape par étape. Vous n'avez pas à vous inquiéter à propos de l'agencement de tout cela. Ceci est différent d'un framework qui vous fournit des blocs préconstruits dès le début. Ce livre se concentre sur React pour construire en tant que premier bloc de construction. Plus de blocs de construction suivront finalement. Tout d'abord, vous pouvez apprendre les *building block* étape par étape. Vous n'avez pas à vous inquiéter à propos de l'agencement de tout cela. Ceci est différent d'un framework qui vous fournit des *building block* dès le début. Ce livre se concentre sur React en tant que premier *building block*. Plus de *building block* suivront finalement.

Dans un second temps, tout *building blocks* sont interchangeable. Cela rend l'écosystème autour de React innovant. Plusieurs solutions s'affrontent entre eux. Vous pouvez dès lors récupérer la solution la plus attrayante pour vous et votre cas d'utilisation.

La première génération de frameworks SPA arrivèrent à un déploiement en entreprise. Ils sont plus rigides. Tandis que React demeure innovant et tand à être adopté par plusieurs têtes pensantes d'entreprise leader tel que Airbnb, Netflix et bien sûr Facebook[41]. Tous investissent dans le futur de React et satisfait et de react et de son écosystème.

39. https://en.wikipedia.org/wiki/Single-page_application
40. https://en.wikipedia.org/wiki/Model–view–controller
41. https://github.com/facebook/react/wiki/Sites-Using-React

De nos jours, React est probablement l'un des meilleurs choix pour construire das applications web modernes. Il délivre seulement la couche vue, mais l'écosystème React fournit un framework entièrement souple et interchangeable[42]. React dispose d'une API épuré, d'un incroyable écosystème et d'une grande communauté. Vous pouvez lire à propos de mes expériences pourquoi je suis passé d'Angular à React[43]. Je recommande d'avoir une compréhension de pourquoi vous souhaitez choisir React plutôt qu'un autre framework ou bibliothèque. Après tout, tout le monde est enjoué à l'idée d'expérimenter vers où React nous conduira dans les prochaines années.

Exercices

- lire plus à propos pourquoi je suis passé d'Angular à React[44]
- lire plus à propos l'écosystème souple de React[45]
- lire plus à propos de comment apprende un framework[46]

42. https://www.robinwieruch.de/essential-react-libraries-framework/
43. https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/
44. https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/
45. https://www.robinwieruch.de/essential-react-libraries-framework/
46. https://www.robinwieruch.de/how-to-learn-framework/

Prérequis

Quels sont les prérequis pour lire ce livre ? Tout d'abord, si vous êtes déjà familier avec les bases du web développement. Vous devriez être familier quant à l'utilisation d'HTML, CSS et JavaScript. Peut-être, le terme API[47] ne vous est pas étranger également, car nous allons utiliser des APIs dans ce livre. De plus, je vous encourage à rejoindre le [Groupe Slack] officiel (https ://slack-the-road-to-learn-react.wieruch.com/) du livre pour être aidé ou aider les autres.

Éditeur et terminal

Qu'en est-il de l'environnement de développement ? Vous aurez besoin d'un éditeur ou d'un IDE et d'un terminal (*command line tool*). Vous pouvez suivre mon guide d'installation[48]. C'est illustré pour les utilisateurs de MacOS, mais vous pouvez trouver un guide d'installation Windows pour React également. En général, il y a une pléthore d'articles qui vous montreront comment installer un environnement de développement pour le web et de façon plus élaborée pour votre OS.

Optionnellement, vous pouvez utiliser Git et Github par vous-même, pendant la réalisation des exercices du livre, pour conserver vos projets et la progression dans les dépôts de Github. Il existe un petit guide[49] sur comment utiliser ces outils. Mais encore une fois, ce n'est pas obligatoire pour le livre et peut devenir contraignant lors de l'apprentissage de tout cela en même temps à partir de rien. Donc vous pouvez sauter cela si vous êtes un novice dans le développement web pour vous concentrer sur les parties essentielles apprises dans ce livre.

Node et NPM

Dernier point, mais pas des moindres, vous aurez besoin d'une installation de node and npm[50]. Les deux sont utilisés pour gérer les bibliothèques dont vous aurez besoin tout du long. Dans ce livre, vous installerez des *external node packages* via npm (node package manager). Ces *node packages* peuvent être des bibliothèques ou des frameworks en entier.

Vous pouvez vérifier vos versions de node et npm en ligne de commande. Si vous n'obtenez aucune sortie sur le terminal, vous aurez au préalable besoin d'installer node et npm. Ce sont seulement mes versions au moment où j'écris ce livre :

47. https://www.robinwieruch.de/what-is-an-api-javascript/
48. https://www.robinwieruch.de/developer-setup/
49. https://www.robinwieruch.de/git-essential-commands/
50. https://nodejs.org/en/

Command Line

```
node --version
*v8.9.4
npm --version
*v5.6.0
```

node et npm

Cette section de chapitre vous donne un petit aperçu de node et npm. Ceci n'est pas exhaustif, mais vous aurez tous les outils nécessaires. Si vous êtes familier avec les deux technologies, vous pouvez sauter cette section.

Le **node package manager** (npm) vous permet d'installer des externes en ligne de commande. Ces packages peuvent être un ensemble de fonctions utilitaires, bibliothèques ou un framework complet. Ils sont les dépendances de votre application. Vous pouvez soit installer ces packages dans votre dossier node package globalement ou dans votre dossier de projet local.

Les nodes packages globaux sont accessibles depuis partout au sein du terminal et vous devez les installer seulement une seule fois dans votre répertoire. Vous pouvez installer un package global en tapant dans votre terminal :

Command Line

```
npm install -g <package>
```

Le flag `-g` demande à npm d'installer le package globalement. Les packages locaux sont utilisés dans votre application. Par exemple, React en tant que bibliothèque sera un package local qui peut être requis pour l'utilisation de votre application. Vous pouvez l'installer via le terminal en tapant :

Command Line

```
npm install <package>
```

Dans le cas de React cela pourrait être :

Command Line

```
npm install react
```

Le package installé apparaîtra automatiquement dans le dossier appelé *node_modules/* et sera listé dans le fichier *package.json* à côté de vos autres dépendances.

Mais comment initialiser pour la première fois le dossier *node_modules/* ainsi que le fichier *package.json* pour votre projet ? Il y a une commande npm pour initialiser un projet npm et ainsi un fichier *package.json*. Seulement lorsque vous avez ce fichier, vous pouvez installer des nouveaux packages locaux via npm.

Command Line

```
npm init -y
```

Le flag `-y` est un raccourci pour initialisé tout par défaut dans votre *package.json*. Si vous n'utilisez pas le flag, vous devez décider de comment configurer le fichier. Après l'initialisation pour le projet npm vous êtes prêt à installer des nouveaux packages via `npm install <package>`.

Un dernier commentaire à propos du *package.json*. Le fichier vous permet de partager votre projet avec les autres développeurs sans partager tous les nodes packages. Le fichier possède toutes les références des nodes packages utilisés dans votre projet. Ces packages sont appelés dépendances. Tout le monde peut copier votre projet sans les dépendances. Les dépendances sont des références dans le *package.json*. Quelqu'un qui copie votre projet peut simplement installer tous les packages en utilisant `npm install` en ligne de commande. Le script `npm install` prend toutes les dépendances listées dans le fichier *package.json* et les installent dans le dossier *node_modules/*.

Je veux traiter une commande npm supplémentaire :

Command Line

```
npm install --save-dev <package>
```

Le flag `--save-dev` indique que le node package est seulement utilisé dans le développement d'environnement. Il ne sera pas utilisé en production que lorsque vous déployez votre application sur un serveur. Quel type de node package peut-être concerné ? Imaginez-vous souhaiter tester votre application avec l'aide d'un node package. Vous aurez besoin d'installer le package via npm, mais vous voulez l'exclure de votre environnement de production. Les tests devront seulement s'exécuter durant la phase de développement et en aucun cas lorsque votre application est déjà lancée en production. À ce moment si, vous ne souhaitez plus la tester. Elle devrait déjà être testée et prête à l'emploi pour vos utilisateurs. C'est seulement un cas d'utilisation parmi tant d'autres où vous souhaiterez utiliser le flag `--save-dev`.

Vous rencontrerez beaucoup plus de commandes npm tout au long de votre utilisation. Mais celles-ci seront suffisants pour le moment.

Il y a une chose supplémentaire qui a le mérite d'être mentionnée. Beaucoup de persones optent pour l'utilisation d'un autre package manager afin de travailler avec les nodes packages au sein de leurs applications. **Yarn** est manageur de dépendence qui fonctionne de façon très similaire qu'**npm**. Il a ses propres commandes pour performer les mêmes tâches, mais vous aurez tout aussi bien accès au npm registry. Yarn a été conçu pour résoudre quelques problèmes qu'npm n'avez pas encore résolus. Cependant, de nos jours, les deux outils évoluent très rapidement et vous pouvez choisir celui que vous voulez.

Exercices :

- installation d'un projet via npm
  - créer un nouveau dossier avec `mkdir <folder_name>`
  - naviguer à l'intérieur du dossier avec `cd <folder_name>`
  - exécuter `npm init -y` ou `npm init`

  - installer un package local tel que React avec `npm install react`
  - regarder à l'intérieur du fichier *package.json* et du dossier *node_modules/*
  - trouver par vous-même comment désinstaller le node package *react*
- lire plus à propos de npm[51]
- lire plus à propos du yarn[52] package manager

51. https://docs.npmjs.com/
52. https://yarnpkg.com/en/docs/

Installation

Il y a plusieurs approches pour commencer une application React.

La première consiste à utiliser un CDN. Cela peut sembler plus compliqué que cela l'est. Un CDN est un content delivery network[53]. Plusieurs entreprises ont des CDNs qui hébergent des fichiers publiquement pour des personnes puissent les consommer. Ces fichiers peuvent être des bibliothèques telles que React, car après tout, la bibliothèque React empaqueté est seulement un fichier JavaScript *react.js*. Elle peut être hébergée quelque part et vous pouvez la demander au sein de votre application.

Comment utiliser un CDN pour débuter en React ? Vous pouvez utiliser la balise inline `<script>` dans votre HTML qui pointe auprès d'une url CDN. Pour débuter en React vous aurez besoin de deux fichiers (bibliothèques) : *react* et *react-dom*.

Code Playground

```
<script crossorigin src="https ://unpkg.com/react@16/umd/react.development.js"></scri\
pt>
<script crossorigin src="https ://unpkg.com/react-dom@16/umd/react-dom.development.js\
"></script>
```

Mais pourquoi devrez-vous utiliser un CDN quand vous disposez de npm pour installer des nodes packages tels que React ?

Lorsque votre application à un fichier *package.json*, vous pouvez installer *react* et *react-dom* en ligne de commande. Le prérequis est que le dossier est initialisé en tant que projet npm en utilisant `npm init -y` avec un fichier *package.json*. Vous pouvez utiliser plusieurs nodes packages en une seule ligne de commande npm.

Command Line

```
npm install react react-dom
```

Cette approche est souvent utilisée pour ajouter React à une application existante qui est gérée par npm.

Malheureusement ce n'est pas tout. Vous allez être obligéq de vous occuper de Babel[54] pour faire en sorte que votre application soit compatible au JSX (la syntaxe de React) et au JavaScript ES6. Babel transpile votre code ainsi les navigateurs peuvent interpréter le JavaScript ES6 et JSX. Pas tous les navigateurs sont capables d'interpréter la syntaxe. L'installation inclut beaucoup de configuration et d'outillage. Cela peut être accablant pour les nouveaux utilisateurs de React de s'importuner avec toute la configuration.

53. https://en.wikipedia.org/wiki/Content_delivery_network
54. http://babeljs.io/

À cause de cela, Facebook introduit *create-react-app* en tant que solution React zéro configuration. Le prochain chapitre vous montrera comment installer votre application en utilisant cette outil d'initialisation.

Exercices :

- lire plus à propos des installations React[55]

55. https://reactjs.org/docs/getting-started.html

L'installation zéro configuration

In the Road to learn React, vous utiliserez create-react-app[56] pour initialiser votre application. C'est un kit de démarrage orienté avec zéro configuration pour React introduit par Facebook en 2016. 96% des personnes souhaitent le recommander auprès des débutants[57]. Dans *create-react-app* l'outillage et la configuration évolue en arrière-plan tandis que l'attention est portée sur l'implémentation de l'application.

Pour démarrer, vous devrez installer le package dans votre global node packages. Après quoi, vous l'aurez toujours disponible en ligne de commande pour démarrer de nouvelles applications React.

Command Line

```
npm install -g create-react-app
```

Vous pouvez vérifier la version de *create-react-app* pour vérifier que l'installation a réussi à l'aide de la ligne de commande :

Command Line

```
create-react-app --version
*v1.5.1
```

Maintenant vous pouvez initier votre première application React. Nous l'appelons *hackernews*, mais vous pouvez choisir un autre nom. L'initialisation prend quelques secondes. Ensuite, naviguer simplement à l'intérieur du dossier :

Command Line

```
create-react-app hackernews
cd hackernews
```

Maintenant vous pouvez ouvrir l'application dans votre éditeur. La structure de dossier suivant, ou à quelques variations prêtes dépendant de la version *create-react-app*, devrait vous être présentée :

56. https://github.com/facebookincubator/create-react-app
57. https://twitter.com/dan_abramov/status/806985854099062785

Folder Structure

```
hackernews/
  README.md
  node_modules/
  package.json
  .gitignore
  public/
    favicon.ico
    index.html
    manifest.json
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    registerServiceWorker.js
```

Une petite pause sur les dossiers et les fichiers. C'est normal si vous ne les comprenez pas tous au début.

- **README.md** : L'extension .md indique que le fichier est un fichier Markdown. Le Markdown est utilisé comme un langage de balisage allégé avec une syntaxe de formatage en texte brut. Plusieurs code source de projets arrivent avec un fichier *README.md* pour donner des instructions initiales à propos du projet. Finalement lors de l'envoi de votre projet auprès d'une plateforme telle que Github, le fichier *README.md* montrera son contenu de manière évidente lorsque vous accédez au dépôt. Comme vous avez utilisé *create-react-app*, votre *README.md* devrait être le même que celui affiché dans le dépôt officiel dépôt GithHub create-react-app[58].
- **node_modules/** : Le dossier a tous les nodes packages qui sont installés via npm. Puisque vous avez utilisé *create-react-app*, iL devrait déjà avoir plusieurs nodes modules installés pour vous. Habituellement, vous ne toucherez jamais à ce dossier, mais seulement installerez et désinstallerez des nodes packages avec npm depuis la ligne de commande.
- **package.json** : Le fichier vous affiche une liste de dépendance de nodes packages et d'autre configurations de projet.
- **yarn.lock** : Par défaut, si Yarn est installé, create-react-app utilise Yarn en tant que package manager. Si vous n'avez pas installé Yarn alors create-react-app utilisera le Node package manager. si vous disposez de Yarn et Node en package manager d'installé vous pouvez utiliser le Node package manager en ajoutant –use-npm à la commande create-react-app. Vous n'avez pas besoin de prêter attention à ce fichier pour l'heure. Si vous voulez en apprendre plus à

58. https://github.com/facebookincubator/create-react-app

propos de la migration du Node package manager vers le Yarn package manager, se conférer https ://yarnpkg.com/en/docs/migrating-from-npm.

- **.gitignore** : Le fichier indique tous les fichiers et dossiers qui ne devront pas être ajoutés à votre dépôt git lors de l'utilisation de git. Ils devraient seulement être présent au sein de votre projet local. Le dossier *node_modules/* en est un exemple d'utilisation. En effet, c'est suffisant de partager le fichier *package.json* avec vos paires pour leur permettre d'installer toutes les dépendances par eux-mêmes sans partager la totalité du dossier de dépendance.
- **public/** : Le dossier possède tous vos fichiers racines, tel que *public/index.html*. Cet index est celui affiché lors du développement de votre app sur localhost :3000. Le boilerplate prend soin de relier cet index avec l'ensemble des scripts présents dans *src/*.
- **build/** Le dossier sera créé lors de la phase de build du projet pour une mise en production. Il possède tous nos fichiers de production lors de la création de votre projet pour une mise en production. Tout votre code écrit dans les dossiers *src/* et *public/* est empaqueté au sein de quelques fichiers lors de la phase de build de votre projet et placé au sein du dossier de build.
- **manifest.json** et **registerServiceWorker.js** : ne prêtez pas attention à ce que font ces fichiers pour le moment, nous n'en auront pas l'utilité dans ce projet.

Malgré tout, vous n'aurez pas la nécessité de toucher les fichiers et dossiers mentionnés. Pour démarrer la seule chose dont vous aurez besoin est localisé dans le dossier *src/*. L'attention principale se porte sur le fichier *src/App.js* pour implémenter les composants React. Il sera utilisé pour implémenter votre application, mais plus tard vous pourriez vouloir séparer vos composants en plusieurs fichiers tandis que chaque fichier maintient un seul ou quelques composants.

De plus, vous trouverez un fichier *src/App.test.js* pour vos tests et un fichier *src/index.js* en tant que point d'entrée au monde de React. Vous serez amené à connaitre les deux fichiers dans un prochain chapitre. En plus, il y a un fichier *src/index.css* est un fichier *src/App.css* pour styliser votre application de façon générale et vos composants. Les composants arrivent tous avec un fichier de style par défaut lorsque vous les ouvrez.

L'application *create-react-app* est un projet npm. Vous pouvez utiliser npm pour installer et désinstaller des nodes packages de votre projet. De plus, il embarque les scripts npm suivants disponibles en ligne de commande :

Command Line

```
# Lance l'application sur http ://localhost :3000
npm start

# Lance les tests
npm test

# Construit l'application pour la mise en production
npm run build
```

Les scripts sont définis dans votre *package.json*. Votre application React passe-partout est maintenant initialisée. La partie existante arrive avec les exercices pour finalement lancer votre application initialisée au sein du navigateur.

Exercices :

- `npm start` pour démarrer votre application et visionner l'application dans votre navigateur (vous pouvez arrêter la commande en tapant Ctrl + C)
- lancer le script `npm test` interactif
- lancer le script `npm run build` et vérifier qu'un dossier *build/* a été ajouté à votre projet (vous pouvez de nouveau le supprimer par la suite ; noter que le dossier build peut être utilisé plus tard pour déployer votre application[59])
- vous familiarisez avec la structure de dossier
- vous familiarisez avec le contenu des fichiers
- lire plus à propos des scripts npm et de create-react-app[60]

59. https://www.robinwieruch.de/deploy-applications-digital-ocean/
60. https://github.com/facebookincubator/create-react-app

Introduction au JSX

Maintenant vous allez faire connaissance avec le JSX. C'est une syntaxe utilisée dans React. Comme mentionné avant, *create-react-app* a déjà mis en place un boilerplate d'application pour vous. L'ensemble des fichiers arrivent avec des implémentations par défaut. Il est temps de plonger dans le code source. Le seul fichier que nous manipulerons pour débuter sera le fichier *src/App.js*.

src/App.js

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

Ne vous laissez distraire par les déclarations d'import/export et de class. Ce sont des fonctionnalités de JavaScript ES6. À ce titre, nous les réétudierons dans un prochain chapitre.

Dans le fichier vous avez une **React ES6 class component** avec le nom de l'application. C'est une déclaration de composant. Globalement après vous avez déclaré un composant, vous pouvez utiliser ce dernier en tant qu'élément depuis partout dans votre application. Il produira une **instance** de votre **component** ou en d'autres termes : le composant sera instancié.

L'**element** retourné est spécifié dans la méthode `render()`. Les éléments sont la résultant de pourquoi les composants sont créés. C'est essentiel de comprendre la différence entre un composant, une instance d'un composant et un élément.

Prochainement, vous verrez où l'App component est instancié. Sans quoi vous ne pourriez apprécier

le rendu en sortit du navigateur, n'est-ce pas ? L'App component est seulement la déclaration, mais non son utilisation. Vous instancierait le composant quelque part dans votre JSX avec `<App />`.

Le contenu dans le bloc de rendu apparaît assez similaire à de l'HTML, mais c'est du JSX. JSX vous permet de mélanger HTML et JavaScript. C'est puissant encore un peu déroutant quand on avait pour habitude de séparer l'HTML du JavaScript. C'est pourquoi, un bon point pour débuter consiste à utiliser de l'HTML basique à l'intérieur de votre JSX. Au tout début, ouvrez le fichier `App.js` et supprimez tout le code HTML inutile comme illustré ci-dessous.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h2>Welcome to the Road to learn React</h2>
      </div>
    );
  }
}

export default App;
```

Maintenant, vous retournez seulement de l'HTML dans votre méthode `render()` sans JavaScript. Laissez-moi définir le "Welcome to the Road to learn React" en tant que variable. Une variable peut être utilisée dans votre JSX en utilisant les accolades.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    var helloWorld = 'Welcome to the Road to learn React';
    return (
      <div className="App">
        <h2>{helloWorld}</h2>
      </div>
    );
  }
}
```

```
export default App;
```

Cela devrait fonctionner quand vous démarr er votre application à l'aide de la commande `npm start` de nouveau.

Qui plus est vous devriez avoir remarqué l'attribut `className`. Il réfléchit l'attribut HTML standard `class`. Due à des raisons techniques, JSX a remplacé une poignée d'attributs HTML internes. Vous pouvez trouver tous les attributs HTML supportés au sein de la documentation React[61]. Ils suivent tous la convention camelCase. Dans votre apprentissage à React, vous rencontrerez quelques attributs spécifiques JSX supplémentaire.

Exercices :

- Définir plus de variables et les rendre au sein de votre JSX
  - utiliser un objet complexe pour représenter un utilisateur avec prénom et nom
  - rendre les propriétés de l'utilisateur dans votre JSX
- Lire plus à propos de JSX[62]
- Lire plus à propos des composants React, éléments et instances[63]

61. https://reactjs.org/docs/dom-elements.html#all-supported-html-attributes
62. https://reactjs.org/docs/introducing-jsx.html
63. https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html

ES6 const et let

Je devine que vous avez remarqué que nous avons déclaré la variable `helloWorld` avec une déclaration `var`. JavaScript ES6 arrive avec deux alternatives supplémentaires pour déclarer vos variables : `const` et `let`. En JavaScript ES6, vous trouverez rarement `var` dorénavant.

Une variable déclarée avec `const` ne peut être réassignée ou redéclarée. Il ne peut être muté (changé, modifié). Vous forcez les structures de données immutables en utilisant cela. Une fois la structure de données défini, vous ne pouvez en changer.

Code Playground

```
// pas permis
const helloWorld = 'Welcome to the Road to learn React';
helloWorld = 'Bye Bye React';
```

Une variable déclarée avec `let` peut être modifiée.

Code Playground

```
// permis
let helloWorld = 'Welcome to the Road to learn React';
helloWorld = 'Bye Bye React';
```

Vous l'utiliserez quand vous aurez besoin de réassigner une variable.

Cependant, vous devez être vigilant avec `const`. Une variable déclarée avec `const` ne peut pas être modifiée. Mais quand la variable est un tableau d'objet, la valeur contenu peut-être modifié. La valeur contenue n'est pas immutable.

Code Playground

```
// permis
const helloWorld = {
  text : 'Welcome to the Road to learn React'
};
helloWorld.text = 'Bye Bye React';
```

Mais quand utiliser chaque déclaration ? Il y a différents avis à propos de leur utilisation. Je suggère d'utiliser `const` à chaque fois vous le pouvez. Cela indique que vous souhaiter conserver votre structure de données immutable même si les valeurs des objets et tableaux peuvent être modifiés. Si vous voulez modifier votre variable préconisée `let`.

L'immutabilité est adoptée par React et son écosystème. C'est pourquoi `const` devra être votre choix par défaut quand vous définissez une variable. Encore que, dans les objets complexes les valeurs à l'intérieur peuvent être modifiées. Soyez attentif à propos de ce comportement.

Dans votre application, vous devriez utiliser `const` plutôt `var`.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    const helloWorld = 'Welcome to the Road to learn React';
    return (
      <div className="App">
        <h2>{helloWorld}</h2>
      </div>
    );
  }
}

export default App;
```

Exercices :

- lire plus à propos ES6 const[64]
- lire plus à propos ES6 let[65]
- En savoir plus à propos des structures de données immutables
  - Pourquoi cela a du sens dans la programmation en général
  - Pourquoi sont-elles utilisées dans React et son écosystème

64. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
65. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

ReactDOM

Avant de continuer avec l'App component, vous devriez vouloir voir où c'est utilisé. C'est localisé dans votre point d'entrée du monde de React : le fichier *src/index.js*.

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';

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

Globalement `ReactDOM.render()` utilise un noeud du DOM dans votre HTML pour le remplacer avec votre JSX. C'est comme cela que vous pouvez facilement intégrer React dans toutes applications étrangères. Il n'est pas interdit d'utiliser `ReactDOM.render()` plusieurs fois au travers de votre application. Vous pouvez l'utiliser à différents endroits pour initialiser de la simple syntaxe JSX, un composant React, plusieurs composants React ou toute une application. Mais pour une simple application React vous l'utiliserez seulement une seule fois pour initialiser l'entièreté de votre arbre de composant.

`ReactDOM.render()` attend deux arguments. Le premier argument est du JSX qui sera rendu. Le second argument spécifie la localisation où l'application React prendra place dans votre HTML. Il attend un élément avec un `id='root'`. Vous pouvez dès lors ouvrir votre fichier pour trouver *public/index.html* l'attribut id.

L'implémentation de `ReactDOM.render()` prend déjà votre App component. Cependant, ce serait intéressant de passer du JSX le plus simplifié tant que cela reste du JSX. Cela n'a pas nécessité d'être une instanciation de composant.

Code Playground

```
ReactDOM.render(
  <h1>Hello React World</h1>,
  document.getElementById('root')
);
```

Exercices :

- ouvrir *public/index.html* pour voir où les applications React prennent place à l'intérieur de votre HTML

- lire plus à propos du rendu des éléments dans React[66]

66. https://reactjs.org/docs/rendering-elements.html

Hot Module Replacement

Il y a une chose que vous pouvez faire au sein du fichier *src/index.js* pour améliorer votre expérience de développement en tant que développeur. Mais c'est optionnel et vous ne devrez pas vous en préoccuper au début de votre apprentissage avec React.

create-react-app dispose déjà de l'avantage que le navigateur rafraichisse automatiquement la page quand vous modifiez le code source. Essayez en changeant la variable `helloWorld` dans votre fichier *src/App.js*. Le navigateur devrait rafraichir la page. Cependant il y a une meilleure façon de le faire.

Le Hot Module Replacement (HMR) est un outil pour recharger votre application dans le navigateur. Le navigateur n'a pas nécessité de performer un rafraichissement de page. Vous pouvez facilement l'activer dans create-react-app. *Dans votre *src/index.js*, votre point d'entrée pour React, vous devez ajouter une petite configuration.

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';

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

if (module.hot) {
  module.hot.accept();
}
```

C'est tout ! Essayez de nouveau de changer la variable `helloWorld` dans votre fichier *src/App.js*. Le navigateur ne dervrait pas performer de rafraichissement de page, mais l'application se recharge et affiche la nouvelle sortie correctement. HMR vient avec de nombreux avantages :

Imaginez-vous êtes en train de debugger votre code avec des déclarations de `console.log()`. Ces déclarations resteront dans votre console développeur, même si vous modifiez votre code, parce que le navigateur ne rafraichit plus la page maintenant. Cela peut être un atout à des fins de debug.

Dans une application grandissant le rafraichissement de page ralenti votre productivité. Vous êtes obligé d'attendre que la page charge. Un rechargement de page peut prendre quelques secondes au sein d'une importante application. HMR repousse ce désavantage.

Le plus grand bénéfice c'est que vous pouvez conserver l'état de l'application avec HMR. Imaginez-vous avez une boîte de dialogue dans votre application avec plusieurs étapes et vous êtes à l'étape 3. Globalement c'est un assistant. Sans HMR vous souhaiteriez changer le code source et votre

navigateur rafraichirait la page. Vous devriez rouvrir la boîte de dialogue encore une fois et naviguer de l'étape 1 à 3. Avec HMR votre boîte de dialogue reste ouverte à l'étape 3. Il conserve l'état de l'application même si le code source change. L'application elle-même se recharge, mais pas la page.

Exercises :

- Modifier votre *src/App.js* code source plusieurs fois et vérifier l'HMR en action
- Regarder les 10 premières minutes du Live de React : Hot Reloading avec Time Travel[67] par Dan Abramov

67. https://www.youtube.com/watch?v=xsSnOQynTHs

JavaScript complex en JSX

Maintenant retourner à votre App component. Jusqu'à présent, vous rendez quelques variables primitives dans votre JSX. Maintenant vous allez commencer par rendre une liste d'objets. Pour débuter, la liste sera un échantillon de données, mais plus tard vous aller chercher les données depuis une API[68] externe. Cela sera beaucoup plus passionnant.

Tout d'abord vous devez définir une liste d'objets.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

const list = [
  {
    title : 'React',
    url : 'https ://reactjs.org/',
    author : 'Jordan Walke',
    num_comments : 3,
    points : 4,
    objectID : 0,
  },
  {
    title : 'Redux',
    url : 'https ://redux.js.org/',
    author : 'Dan Abramov, Andrew Clark',
    num_comments : 2,
    points : 5,
    objectID : 1,
  },
];

class App extends Component {
  ...
}
```

L'échantillon de données reflétera les données que nous irons chercher plus tard depuis l'API. Un objet dans la liste possède un titre, une url et un auteur. Qui plus est l'objet dispose d'un identificateur, de points (indique comment un article est populaire) et d'un nombre de commentaires.

Maintenant vous pouvez utiliser la fonctionnalité native `map` dans votre JSX. Cela permet d'itérer sur votre liste d'objets et de les afficher. Encore une fois vous utiliserez les accolades pour encapsuler

68. https://www.robinwieruch.de/what-is-an-api-javascript/

les expressions JavScript dans votre JSX.

src/App.js

```
class App extends Component {
  render() {
    return (
      <div className="App">
        {list.map(function (item) {
          return <div>{item.title}</div>;
        })}
      </div>
    );
  }
}

export default App;
```

Le fait d'utiliser le JavaScript dans l'HTML est assez puissant en JSX. Habituellement, il vous est conseillé d'utiliser `map` pour convertir une liste d'objets en une autre liste d'objets. Cette fois vous utilisez `map` pour convertir une liste d'objets en éléments HTML.

Jusqu'à maintenant, seul le `title` a été affiché à chaque fois. Essayons d'afficher un peu plus de propriétés de l'objet.

src/App.js

```
class App extends Component {
  render() {
    return (
      <div className="App">
        {list.map(function (item) {
          return (
            <div>
              <span>
                <a href={item.url}>{item.title}</a>
              </span>
              <span>{item.author}</span>
              <span>{item.num_comments}</span>
              <span>{item.points}</span>
            </div>
          );
        })}
      </div>
    );
```

```
  }
}

export default App;
```

Vous pouvez remarquer la fonction `map` est simplement en ligne dans votre JSX. Chaque propriété de l'objet est affichée dans une balise `<span>`. De plus, la propriété url de l'objet est utilisée dans l'attribut `href` de la balise ancre.

React fera tout le travail pour vous et affichera chaque objet. Mais vous devez ajouter un helper pour que React atteigne son plein potentiel et qu'il améliore ses performances. Vous allez devoir assigner un attribut clé à chaque élément de la liste. Ainsi React sera capable d'identifier les éléments ajoutés, changés et supprimés quand la liste change. L'échantillon d'éléments arrive d'ores et déjà avec un identifieur.

src/App.js

```
{list.map(function (item) {
  return (
    <div key={item.objectID}>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
    </div>
  );
})}
```

Vous devrez vous assurer que l'attribut `key` est un identifiant stable. Ne faites pas l'erreur d'utiliser l'index de l'objet dans un tableau. L'index de tableau n'est pas stable du tout. Par exemple, quand la liste change son ordonnancement, React aura des difficultés à identifier les objets correctement.

src/App.js

```
// ne pas faire cela
{list.map(function (item, key) {
  return (
    <div key={key}>
      ...
    </div>
  );
})}
```

Vous affichez les deux listes d'objets maintenant. Vous pouvez démarrer votre application, ouvrir votre navigateur et voir les deux objets de la liste affichés.

Exercices :

- lire plus à propos des listes et clés de React[69]
- Récapitulatif des fonctionnalités natives et standard du tableau en JavaScript[70]
- Utiliser plus d'expression JavaScript de vous-même en JXS

69. https://reactjs.org/docs/lists-and-keys.html
70. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/

ES6 les fonctions fléchées

JavaScript ES6 introduit les fonctions fléchées. Une expression de fonction fléchée est plus courte qu'une expression de fonction.

Code Playground

```
// function expression
function () { ... }

// arrow function expression
() => { ... }
```

Mais vous devez rester conscient de ces fonctionnalités. L'un d'eux dispose d'un comportement différent avec l'objet `this`. Une expression de fonction définie toujours son propre objet `this`. Tandis que les expressions de fonction fléchée possèdent encore l'objet `this` du contexte parent. Ne soyez pas désorientés lorsque vous utiliserez `this` à l'intérieur d'une fonction fléchée.

Il y a un autre point intéressant à propos des fonctions fléchées concernant les parenthèses. Vous pouvez supprimer les parenthèses quand la fonction possède seulement un argument, mais vous êtes obligés de les conserver le cas contraire.

Code Playground

```
// permis
item => { ... }

// permis
(item) => { ... }

// pas permis
item, key => { ... }

// permis
(item, key) => { ... }
```

Cependant, jetons à coup d'oeil à la fonction `map`. Vous pouvez l'écrire de façon plus concise avec la fonction fléchée ES6.

src/App.js

```
{list.map(item => {
  return (
    <div key={item.objectID}>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
    </div>
  );
})}
```

De plus, vous pouvez supprimer le *block body*, c'est-à-dire les accolades, de la fonction fléchée ES6. Dans un *concise body* un implicite `return` est attaché. Ainsi vous pouvez supprimer la déclaration `return`. Cela s'utilisera assez souvent dans le livre, donc soyez sûre de bien comprendre la différence entre un *block body* et un *concise body* lorsque vous utilisez des fonctions fléchées.

src/App.js

```
{list.map(item =>
  <div key={item.objectID}>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
  </div>
)}
```

Votre JSX apparaît plus concis et lisible maintenant. Il omet la déclaration de `function`, les accolades et la déclaration du `return`. Au lieu de cela le développeur peut rester concentré sur les détails d'implémentation.

Exercices :

- lire plus à propos des fonctions fléchées ES6[71]

71. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions

ES6 Classes

JavaScript ES6 introduit les classes. Une classe est communément utilisé dans des langages de programmation orientés objet. JavaScript était et est très souple dans ses paradigmes de programmation. Vous pouvez faire de la programmation fonctionnelle et orientée objet côte à côte avec leur cas d'utilisation particulier.

Même si React adhère à la programmation fonctionnelle, par exemple au travers des structures de données immutables, les classes sont aussi utilisées pour déclarer des composants. Elles sont appelées les composants de classe ES6. React mélange les meilleurs des deux paradigmes.

Laissons nous considérer la classe Developer suivante pour examiner une classe JavaScript ES6 sans penser composant.

Code Playground

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  getName() {
    return this.firstname + ' ' + this.lastname;
  }
}
```

Une classe dispose d'un constructeur pour la rendre instanciable. Le constructeur peut prendre plusieurs arguments pour les assigner à l'instance de la classe. De plus, une classe peut définir des fonctions. Du fait que la fonction est associée à une classe, elle est appelée une méthode. Souvent elle est référencée en tant que méthode de classe.

La classe Developer est seulement une déclaration de classe. Vous pouver créer plusieurs instances de classe en l'invoquant. C'est similaire pour une classe ES6 de composant, qui a une déclaration, mais vous devez l'utiliser autres parts pour l'instancier.

Regardons comment vous devez instancier la classe et comment vous pouvez utiliser ses méthodes.

Code Playground

```
const robin = new Developer('Robin', 'Wieruch');
console.log(robin.getName());
// output : Robin Wieruch
```

React utilise les classes JavaScript ES6 pour les classes de composants ES6. Vous utilisez d'ores et déjà la classe de composant ES6.

src/App.js

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

...

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

La classe App étend `Component`. Globalement, vous déclarez le composant App, mais il s'étend d'un autre composant. Qu'est-ce que le terme étendre signifie-t-il ? En programmation orientée objet vous avez le principe d'héritage. C'est utilisé pour passer des fonctionnalités d'une classe à l'autre.

La classe App étend les fonctionnalités à partir de la classe `Component`. Pour être plus précis, elle hérite des fonctionnalités de la classe `Component`. La classe `Component` est utilisée pour étendre une classe ES6 basique en classe composant ES6. Elle possède toutes les fonctionnalités qu'un composant React a besoin d'avoir. La méthode `render` est une de ces fonctionnalités que vous avez déjà utilisées. Vous apprendrez un peu plus tard les autres méthodes de la classe `Component`.

La classe `Component` encapsule tous les détails d'implémentation du composant React. Permettant aux développeurs d'utiliser des classes en tant que composants dans React.

Les méthodes qu'un React `Component` expose sont l'interface publique. L'une de ces méthodes a besoin d'être surchargé, les autres n'ont pas la nécessité. Vous en apprendrez davantage à propos de cette dernière quand le livre abordera les méthodes du cycle de vie dans un prochain chapitre. La méthode `render()` à besoin d'être surchargé, parce qu'elle définit la sortie d'un `Component` React. Elle doit être définie.

Maintenant vous connaissez les bases autour des classes JavaScript ES6 et de comment elles sont utilisées dans React pour les étendre à des composants. Vous en apprendrez plus au sujet des méthodes de composant quand le livre décrira les méthodes du cycle de vie de React.

Exercices :

- lire plus à propos des classes ES6[72]
- lire plus à propos des fondamentaux de JavaScript nécessaire avant l'apprentissage de React[73]

72. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes
73. https://www.robinwieruch.de/javascript-fundamentals-react-requirements/

Vous avez appris à initier votre propre application React ! Récapitulons les derniers chapitres :

- React
  - create-react-app démarre une application React.
  - JSX mélange totalement l'HTML et le JavaScript dans le but de définir la sortie des composants React dans leurs méthodes render.
  - Composants, instances et éléments sont différentes choses dans React
  - `ReactDOM.render()` est un point d'entrée pour une application React pour attacher React à l'intérieur du DOM
  - Les fonctionnalités natives de JavaScript peuvent être utilisés en JSX
    * map peut être utilisé pour rendre une liste d'objet en tant qu'éléments HTML
- ES6
  - Les déclarations de variables avec `const` et `let` peuvent être utilisées pour des cas d'utilisation spécifiques
    * Utiliser const par-dessus let dans les applications React
  - Les fonctions fléchées peuvent être utilisées pour conserver vos fonctions concises
  - Les classes sont utilisées pour définir les composants dans React en les étendant

Il peut être intéressant de faire une pause à ce niveau-ci. Intérioriser les acquis et les appliquer de façon autonome. Vous pouvez trouver le code source dans le dépôt officiel[74].

74. https://github.com/rwieruch/hackernews-client/tree/4.1

Les bases de React

Ce chapitre vous guidera au travers des bases de React. Il couvre l'état et les interactions au sein des composants, car les composants statiques sont un peu moroses, n'est ce pas ? De plus, vous apprendrez les différentes façons de déclarer un composant et comment conserver les composants composables et réutilisables. Soyez prêt à faire vivre vos composants.

État interne de composant

L'état interne du composant, aussi connus comme état local, permet de sauvegarder, modifier et supprimer des propriétés qui sont stockées dans votre composant. Le composant de classe ES6 peut utiliser un constructeur pour initialiser l'état interne du composant nous le verrons ultérieurement. Le constructeur est appelé seulement une seule fois quand le composant est initialisé.

Introduisons le constructeur de classe.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);
  }

  ...

}
```

Le composant App est une sous-classe de `Component` : d'où l'`extends Component` dans votre déclaration du composant App. Vous apprendrez ultérieurement plus à propos des composants de classe ES6.

C'est obligatoire d'appeler `super(props);` : il crée `this.props` dans votre constructeur dans le cas où vous souhaitez y accéder depuis le constructeur. Autrement, lors de l'accès à `this.props` dans votre constructeur, ils seront `undefined`. Vous en apprendrez plus sur les propriétés de composant React ultérieurement.

Maintenant, dans votre cas, l'état initial dans votre composant devrait être un échantillon d'une liste d'éléments.

src/App.js

```
const list = [
  {
    title : 'React',
    url : 'https ://reactjs.org/',
    author : 'Jordan Walke',
    num_comments : 3,
    points : 4,
    objectID : 0,
  },
  ...
```

```
];

class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list : list,
    };
  }

  ...

}
```

L'état est lié à la classe via l'objet `this`. Ainsi, vous pouvez accéder à l'état local dans tout votre composant. Par exemple, il peut être utilisé dans la méthode `render()`. Précédemment, vous avez établi une liste d'éléments dans votre méthode `render()` qui était défini à l'extérieur de votre composant. Maintenant vous êtes à même d'utiliser la liste à partir de votre état local à l'intérieur de votre composant.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
          </div>
        )}
      </div>
    );
```

```
  }
}
```

La liste fait partie de votre composant maintenant. Elle y réside dans l'état interne du composant. Vous pouvez ajouter, modifier ou supprimer des éléments à l'intérieur et depuis votre liste. Chaque fois vous changez votre état de composant, la méthode `render()` de votre composant sera éxécuté de nouveau. C'est comme cela que vous pouvez simplement modifier votre état interne de composant et être sûre que le composant rerendra et affichera les données correctes qui sont issues de l'état local.

Mais soyez vigilant. N'établissez pas l'état directement. Vous aurez la nécessité d'appeler la méthode nommée `setState()` pour modifier votre état. Vous apprendrez à la connaître dans le prochain chapitre.

Exercices :

- expérimenter l'état local
  - définir plus de données pour l'état initial à l'intérieur du constructeur
  - utiliser et accéder à l'état dans votre méthode `render()`
- lire plus à propos du constructeur de classe ES6[75]

75. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes#Constructor

ES6 Object Initializer

En JavaScript ES6, vous pouvez utiliser la syntaxe d'abréviation de propriété pour initialiser vos objets plus concisément. Imaginez l'initialisation de l'objet suivant :

Code Playground

```
const name = 'Robin';

const user = {
  name : name,
};
```

Quand le nom de la propriété de votre objet est le même que vote nom de variable, vous pouvez faire ceci :

Code Playground

```
const name = 'Robin';

const user = {
  name,
};
```

Vous pouvez faire de même, dans votre application. Le nom de la variable liste et le nom de la propriété de l'état partagent le même nom.

Code Playground

```
// ES5
this.state = {
  list : list,
};

// ES6
this.state = {
  list,
};
```

Un autre élégant *helper* est les noms de méthodes abrégés. En JavaScript ES6, vous pouvez initialiser les méthodes dans un objet plus concisément.

Code Playground

```
// ES5
var userService = {
  getUserName : function (user) {
    return user.firstname + ' ' + user.lastname;
  },
};

// ES6
const userService = {
  getUserName(user) {
    return user.firstname + ' ' + user.lastname;
  },
};
```

Enfin mais non moins important, il vous est permis d'utiliser les noms de propriétés interprétées en JavaScript ES6.

Code Playground

```
// ES5
var user = {
  name : 'Robin',
};

// ES6
const key = 'name';
const user = {
  [key] : 'Robin',
};
```

Peut-être les noms de propriétés interprétées n'ont pas encore de sens. Pourquoi devriez vous avoir besoin de cela ? Dans un futur chapitre, vous arriverez à un point où vous pourrez l'utiliser pour allouer des valeurs par clé de façon dynamique dans un objet. C'est propre pour générer des tables de recherche en JavaScript.

Exercices :

- expérimenter avec l'initialiseur d'objet ES6
- lire à propos de l'initialiseur d'objet ES6[76]

76. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Object_initializer

Flux de données unidirectionnel

Maintenant vous avez quelques états internes dans votre composant App. Cependant, vous n'avez pas encore manipulé l'état local. L'état est statique et ainsi le composant l'est tout autant. Un bon moyen pour expérimenter la manipulation d'état est d'avoir quelques interactions de composant.

Ajoutons un bouton pour chaque élément dans la liste affichée. Le bouton écrira "Dismiss" et supprimera l'élément de la liste. Cela pourrait être utile finalement lorsque vous souhaitez seulement conserver une liste d'éléments non lus et de rejeter les éléments dont vous n'êtes pas intéressés.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

La méthode de classe `onDismiss()` n'est pas encore définie. Nous le ferons dans un moment, mais pour l'heure concentrons-nous sur l'*handler* `onClick` de l'élément bouton. Comme vous pouvez le

voir, la méthode `onDismiss()` dans l'*handler* `onClick` est comprise dans une autre fonction. C'est une fonction fléchée. Ainsi, vous pouvez vous glisser dans la propriété `objectID` de l'objet `item` pour identifier l'objet que vous supprimerez. Une alternative serait de définir la fonction à l'extérieur de l'*handler* `onClick` et seulement passer la fonction définie à l'*handler*. Un prochain chapitre abordera le sujet des *handlers* dans les éléments avec plus en détail.

Avez-vous remarqué les multiples lignes pour l'élément bouton ? Noter que les éléments avec plusieurs attributs peuvent être assez en désordre sur une seule ligne à certains endroits. C'est pourquoi l'élément bouton est utilisé sur plusieurs lignes et des indentations pour garder de la lisibilité. Mais ce n'est pas obligatoire. C'est seulement une recommandation de style de code dont je recommande chaudement.

Maintenant vous devez implémenter la fonction `onDismiss()`. Elle prend un id pour identifier l'élément à supprimer. La fonction est liée à la classe et ainsi devient une méthode de classe. C'est pourquoi vous y accédez avec `this.onDismiss()` et non `onDismiss()`. L'objet `this` est votre instance de classe. Dans le but de définir l'`onDismiss()` comme méthode de classe, vous devez la lier dans le constructeur. Les liens seront expliqués plsu tard dans un autre chapitre.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

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

  render() {
    ...
  }
}
```

Dans la prochaine étape, vous devrez définir ses fonctionnalités, la logique métier, au sein de votre classe. Les méthodes de classe peuvent être définies de la façon suivante.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

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

  onDismiss(id) {
    ...
  }

  render() {
    ...
  }
}
```

Maintenant vous êtes capable de définir ce qui se produit à l'intérieur de la méthode de classe. Globalement vous souhaitez supprimer de la liste l'élément identifié par l'id et stocker une liste mise à jour pour votre état local. Après quoi, la liste mise à jour sera utilisé pour relancer la méthode `render()` afin de l'afficher. L'élément retiré ne doit plus apparaître dès lors.

Vous pouvez retirer un élément de la liste en utilisant la fonctionnalité JavaScript natif *filter*. La fonction *filter* prend une fonction en entrée. La fonction a accès à chaque valeur de la liste, parce qu'elle itère par-dessus la liste. De cette façon, vous pouvez évaluer chaque élément dans la liste basée sur la condition de filtre. Si l'évaluation pour un élément est true, l'élément reste dans la liste. Autrement elle sera filtrée de la liste. De plus, il est bon de savoir que la fonction retourne une nouvelle liste et ne mute pas l'ancienne liste. Cela respecte la convention de React d'avoir des structures de données immutables.

src/App.js

```
onDismiss(id) {
  const updatedList = this.state.list.filter(function isNotId(item) {
    return item.objectID !== id;
  });
}
```

Dans la prochaine étape, vous pouvez extraire la fonction et la passer dans la fonction de filtre.

src/App.js

```
onDismiss(id) {
  function isNotId(item) {
    return item.objectID !== id;
  }

  const updatedList = this.state.list.filter(isNotId);
}
```

De plus, vous pouvez faire plus concis en utilisant de nouveau la fonction fléchée de JavaScript ES6?

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
}
```

Vous pourrez même le faire en ligne de nouveau, comme vous avez fait avec l'*handler* `onClick` du bouton, mais cela pourrait devenir moins lisible.

src/App.js

```
onDismiss(id) {
  const updatedList = this.state.list.filter(item => item.objectID !== id);
}
```

Maintenant, la liste supprime l'élément cliqué. Cependant l'état n'est pas encore mis à jour. Par conséquent vous pouvez finalement utiliser la méthode de classe `setState()` pour mettre à jour la liste dans l'état interne du composant.

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
  this.setState({ list : updatedList });
}
```

Maintenant lancer une nouvelle fois l'application et essayait le bouton "Dismiss". Il devrait fonctionner. Ce que vous avez expérimenté est l'**unidirectional data flow** au sein de React. Vous déclenchez une action dans votre vue avec `onCLick()`, une fonction ou méthode de classe modifie l'état interne du composant et la méthode `render()` du composant est lancée une nouvelle fois pour updater la vue.

Exercices :

- lire plus à propos de l'état et du cycle de vie de React[77]

77. https://reactjs.org/docs/state-and-lifecycle.html

Bindings

C'est important d'apprendre le *bindings* de classes en JavaScript quand on utilise les composants de classe de React ES6. Dans le précédent chapitre, vous avez lié votre méthode de classe `onDismiss()` dans le constructeur.

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      list,
    };

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

  ...
}
```

Pourquoi vouloir faire cela en premier lieu ? L'étape de binding est nécessaire, car les méthodes de classe ne sont pas automatiquement liées au `this` de l'instance de la classe. Démontrons cela à l'aide du composant de classe ES6 suivant.

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

Le composant est bien rendu, mais quand vous cliquez sur le bouton, vous obtiendrez `undefined` dans votre console log. C'est l'une des principales sources de bug quand on vient à utiliser React, car si vous souhaitez accéder à `this.state` dans votre méthode de classe, il ne peut être retrouvé car `this` est `undefined`. Donc dans le but de rendre accessible `this` dans vos méthodes de classe, vous devez lier les méthodes de classe avec `this`.

Dans la composant de classe suivante la méthode de classe est correctement lié dans le constructeur de la classe.

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

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

  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

Lorsque vous essayez de nouveau le bouton, l'objet `this`, pour être plus précis l'instance de classe, devrait être définie et vous serez en mesure d'accéder à `this.state`, ou comme vous l'apprendrez plus tard `this.props`.

Le binding de la méthode de classe peut aussi apparaître ailleurs. Par exemple, elle peut apparaître dans la méthode de classe `render()`.

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe.bind(this)}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

Mais vous devriez éviter cela, car il liera la méthode de classe à chaque fois que la méthode `render()` est lancée. Globalement, il le lance à chaque fois que votre composant est mis à jour ce qui a des implications de performance. Lorsque l'on lie la méthode de classe dans le constructeur, vous la liez seulement une seule fois au commencement lorsque le composant est instancié. C'est la meilleure approche.

Un autre point, les développeurs parfois imagine définir la partie métier de leurs méthodes de classe à l'intérieur du constructeur.

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.onClickMe = () => {
      console.log(this);
    }
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
```

```
        Click Me
      </button>
    );
  }
}
```

Vous devriez l'éviter également, car cela va mettre le désordre dans votre constructeur avec le temps. Le constructeur est seulement ici pour instancier votre classe avec l'ensemble des propriétés. C'est pour cela que la partie logique métier des méthodes de classe ne devrait pas être définie à l'extérieur du constructeur.

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.doSomething = this.doSomething.bind(this);
    this.doSomethingElse = this.doSomethingElse.bind(this);
  }

  doSomething() {
    // faire quelque chose
  }

  doSomethingElse() {
    // faire autre chose
  }

  ...
}
```

Enfin et surtout, il est utile de mentionner que les méthodes de classe peuvent être automatiquement autoliées sans le binding explicite en utilisant les fonctions fléchées JavaScript ES6.

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe = () => {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

Si le binding répétitif dans le constructeur vous ennuie, vous pouvez à la place y aller avec cette approche. La documentation officielle de React reste conformé à l'utilisation du binding de méthode de classe à l'intérieur du constructeur. C'est pour cela que ce livre le sera tout autant.

Exercices :

- essayer les différentes approches de bindings et console log l'objet `this`

Event Handler

Le chapitre devrez-vous procurer une compréhension plus global des *event handlers* dans les éléments. Dans votre application, vous allez utiliser l'élément bouton suivant pour retirer l'élément de la liste.

src/App.js

```
...

<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

C'est déjà un cas d'utilisation complexe, car vous êtes obligé de passer une valeur à la méthode de classe et donc vous devez l'englober à l'intérieur d'une autre fonction (fléchée). Donc essentiellement, il a besoin d'être une fonction qui est passé à l'*event handler*. Le code suivant ne fonctionnera pas, parce que la méthode de classe devrez être executée immédiatement quand vous ouvrez l'application dans le navigateur.

src/App.js

```
...

<button
  onClick={this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

Quand on utilise `onClick={doSomething()}`, la fonction `doSomething()` sera exécutée immédiatement lorsque vous ouvrez l'application dans le navigateur. L'expression dans l'*handler* est évaluée. Puisque la valeur retournée n'est plus une fonction, plus rien se produira lorsque vous cliquerez sur le bouton. Mais quand on utilise `onClick={doSomething}` alors que `doSomething` est une fonction, elle sera exécutée que lors du click sur le bouton. Le même principe s'applique pour la méthode de classe `onDismiss()` qui est utilisée dans votre application.

Cependant, l'utilisation d'`onClick={this.onDismiss}` n'est pas suffisant, car d'une manière ou d'une autre la propriété `item.objectID` a besoin d'être passé à la méthode de classe pour identifier l'élément qui sera sur le point d'être enlevé. C'est pourquoi elle doit être englobée à l'intérieur d'une autre fonction dans le but d'y glisser la propriété à l'intérieur. Le concept est appelé en JavaScript les fonctions d'ordre supérieur et sera expliqué brièvement plus tard.

src/App.js

```
...

<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

Une solution de contournement serait de définir la fonction englobante quelque part à l'extérieur et seulement transmettre la fonction définie dans l'*handler*. Puisqu'il a besoin d'accéder à un élément individuel, il doit être présent à l'intérieur du bloc de la fonction *map*.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item => {
          const onHandleDismiss = () =>
            this.onDismiss(item.objectID);

          return (
            <div key={item.objectID}>
              <span>
                <a href={item.url}>{item.title}</a>
              </span>
              <span>{item.author}</span>
              <span>{item.num_comments}</span>
              <span>{item.points}</span>
              <span>
```

```
                <button
                  onClick={onHandleDismiss}
                  type="button"
                >
                  Dismiss
                </button>
              </span>
            </div>
          );
        }
        )}
      </div>
    );
  }
}
```

Après tout, il a besoin d'être une fonction qui est passé à l'*handler* de l'élément. En tant qu'exemple, essayer à la place ce code :

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
            ...
            <span>
              <button
                onClick={console.log(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Cela se lancera lorsque vous ouvrez l'application dans le navigateur mais pas lorsque vous cliquez sur le bouton. Tandis que le code suivant s'exécutera seulement lorsque vous cliquez sur le bouton. C'est une fonction qui est exécutée lorsque vous déclenchez l'*handler*.

src/App.js

```
...

<button
  onClick={function () {
    console.log(item.objectID)
  }}
  type="button"
>
  Dismiss
</button>

...
```

Dans le but de rester concis, vous pouvez la transformer de nouveau en une fonction fléchée de JavaScript ES6. C'est ce que nous faisons également avec la méthode de classe `onDismiss()`.

src/App.js

```
...

<button
  onClick={() => console.log(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

Souvent les arrivants à React ont des difficultés au sujet de l'utilisation des fonctions au sein des *handlers* d'évènements. C'est pourquoi j'ai essayé de l'expliquer avec le plus de détails ici. En fin de compte, vous devez vous retrouver avec le code suivant dans votre bouton avec une fonction fléchée JavaScript ES6 concise qui a accès à la propriété `objectID` de l'objet `item`.

src/App.js

```
class App extends Component {
  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            ...
            <span>
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Un autre sujet pertinent de performance, qui est souvent mentionné, sont les implications de l'utilisation des fonctions fléchées dans les *handlers* d'évènement. Par exemple, l'handler `onCLick` pour la méthode `onDismiss()` englobe la méthode dans une autre fonction fléchée pour être capable de transmettre l'identifieur d'objet. Donc à chaque fois que la méthode `render()` se lance, l'*handler* instancie une fonction fléchée d'ordre supérieur. Il *peut* avoir un impact sur la performance de votre application, mais la plupart du temps vous ne remarquerez rien. Imaginez-vous avez une importante table de données avec 1000 éléments et chaque ligne ou colonne a une telle fonction fléchée dans un *handler* d'évènement. Alors, cela vaut le coup de penser aux implications de performance et par conséquent vous devez implémenter un composant de Bouton dédié pour lier la méthode à l'intérieur du constructeur. Mais avant que cela se produise c'est une optimisation prématurée. Il y plus d'intérêt à se concentrer sur Rect lui-même.

Exercices :

- essayer les différentes approches en utilisant les fonctions dans l'handler `onCLick` de votre bouton.

Interactions avec les Forms et Events

Ajoutons une autre interaction pour l'application dans le but d'expérimenter les évènements dans React. L'interaction est une fonctionnalité de recherche. L'entrée du champ de recherche devrait être utilisée pour filtrer temporairement votre liste basée sur la propriété titre d'un élément.

Dans la première étape, vous allez définir un formulaire avec un champ d'entrée dans votre JSX.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input type="text" />
        </form>
        {this.state.list.map(item =>
          ...
        )}
      </div>
    );
  }
}
```

Dans le scénario suivant vous allez taper à l'intérieur du champ d'entrée et filtrer temporairement la liste par le terme de recherche qui est utilisé dans le champ d'entrée. Pour être capable de filtrer la liste basée sur la valeur du champ d'entrée, vous avez besoin de stocker la valeur du champ d'entrée dans votre état local. Mais comment accèdez vous à la valeur ? Vous pouvez utiliser **synthetic events** dans React pour accéder au contenu de l'évènement.

Définisssons un *handler* `onChange` pour le champ d'entrée.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        ...
      </div>
    );
  }
}
```

La fonction est liée au composant de nouveau par une méthode de classe. Vous êtes dans l'obligation de lier et définir la méthode.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  onSearchChange() {
    ...
  }
```

```
  ...
}
```

Lors de l'utilisation d'un handler dans votre élément, vous avez accès à l'évènement synthétique de React dans votre signature de fonction callback.

src/App.js

```
class App extends Component {

  ...

  onSearchChange(event) {
    ...
  }

  ...
}
```

L'évènement a la valeur du champ d'entrée dans son objet `target`. Ainsi vous serez en mesure de mettre à jour l'état local avec le terme de recherche de nouveau en utilisant `this.setState()`.

src/App.js

```
class App extends Component {

  ...

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

  ...
}
```

De plus, vous ne devez pas oublier de définir l'état initial pour la propriété `searchTerm` dans le constructeur. Le champ d'entrée devrait être vide au démarrage et ainsi la valeur devrait être une chaine de caractères vide.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
      searchTerm : '',
    };

    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...
}
```

Maintenant vous stockez la valeur de l'entrée vers votre état interne du composant chaque fois que la valeur dans le champ d'entrée change.

Un aparté à propos de la mise à jour de l'état local dans un composant React. Il serait envisageable de supposer que lors de la mise à jour du `searchTerm` avec `this.setState()` la liste a besoin d'être passée également pour la conserver. Mais ce n'est pas le cas. `this.setState()` de React est une fusion superficielle. Il conserve les propriétés soeurs dans l'objet état quand on met à jour une seule propriété à l'intérieur. Ainsi l'état de la liste, même si vous avez déjà rejeté un objet de celle-ci, restera la même lors de la mise à jour de la propriété `searchTerm`.

Retournons à votre application. La liste n'est pas encore filtrée en fonction de la valeur du champ d'entrée qui est stocké dans l'état local. Simplement vous devez filtrer temporairement la liste basée sur le `searchTerm`. Vous disposez de tous ce dont vous avez besoin pour filtrer. Donc comment maintenant la filtrer de façon temporaire ? Dans votre méthode `render()`, avant votre *map* par-dessus la liste, vous pouvez appliquer un filtre par-dessus. Le filtre devra seulement évaluer si le `searchTerm` correspond avec la propriété *title* de l'objet. Vous avez déjà utilisé la fonctionnalité de filtre du JavaScript natif, donc faisons-le de nouveau. Vous pouvez glisser la fonction de filtre avant la fonction *map*, car la fonction de filtre retourne un nouveau tableau et ainsi la fonction *map* peut l'utiliser c'est une façon très commode.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(...).map(item =>
          ...
        )}
      </div>
    );
  }
}
```

Abordons la fonction *filter* cette fois-ci de manière différente. Nous souhaitons définir l'argument de filtre, à la fonction qui est passé à la fonction *filter*, depuis l'extérieur du composant de classe ES6. Là nous n'avons pas accès à l'état du composant et ainsi nous n'avons aucun accès à la propriété `searchTerm` pour évaluer la condition de filtre. Nous devons transmettre le `searchTerm` à la fonction de filtre et nous devons retourner une nouvelle fonction, pour évaluer la condition. Ceci est appelé une fonction d'ordre supérieur.

Normalement je ne souhaite pas mentionner les fonctions d'ordre supérieur, mais dans un livre React cela fait totalement sens. Cela fait sens de connaitre les fonctions d'ordre supérieur, car React gère un concept de composants d'ordre supérieur. Vous ferez connaissance de ce concept plus tard dans le livre. Maintenant, concentrons-nous sur la fonctionnalité de filtre.

Premièrement, vous devez définir la fonction d'ordre supérieur à l'extérieur de votre composant App.

src/App.js

```
function isSearched(searchTerm) {
  return function (item) {
    // certaines conditions qui retournent true ou false
  }
}

class App extends Component {

  ...

}
```

La fonction prend le `searchTerm` et retourne une autre fonction, car après tout, la fonction *filter* prend une fonction en entrée. La fonction retournée a accès à l'objet d'élément car c'est la fonction qui est passée auprès de la fonction *filter*. De plus, la fonction retournée sera utilisée pour filtrer la liste basée sur la condition définie dans la fonction. Définissons la condition.

src/App.js

```
function isSearched(searchTerm) {
  return function (item) {
    return item.title.toLowerCase().includes(searchTerm.toLowerCase());
  }
}

class App extends Component {

  ...

}
```

La condition dit que vous associez le motif entrant du `searchTerm` avec la propriété *title* de l'élément de votre liste. Vous pouvez faire cela avec la fonctionnalité JavaScript intégrée `includes`. Seulement quand le motif correspond, vous retournez true et l'élément reste dans la liste. Lorsque le motif ne correspond pas l'élément est supprimé de la liste. Soyez attentif à la correspondance du motif : vous ne devez pas oublier de mettre en minuscule les deux chaînes de caractères. Autrement il y aura une non-correspondance entre le terme de recherche 'redux' et un titre d'élément 'Redux'. Puisque nous sommes en train de travailler sur une liste immutable et que nous retournons une nouvelle liste en utilisant la fonction *filter*, la liste originale dans l'état interne ne sera en aucun cas modifiée.

Il reste une chose à mentionner : Nous trichons un petit peu en utilisant la fonctionnalité JavaScript intégrée `includes`. C'est d'ores et déjà une fonctionnalité ES6. À quoi cela ressemblerait en JavaScript

ES5? Vous devriez utiliser la fonction `indexOf()` pour obtenir l'index de l'élément dans la liste. Lorsque l'élément est dans la liste,`indexOf()`retournera son index du tableau.

Code Playground

```
// ES5
string.indexOf(pattern) !== -1

// ES6
string.includes(pattern)
```

Une autre façon soignée de refactorer peut être réalisé de nouveau avec la fonction fléchée d'ES6. Cela rend la fonction plus concise :

Code Playground

```
// ES5
function isSearched(searchTerm) {
  return function (item) {
    return item.title.toLowerCase().indexOf(searchTerm.toLowerCase()) !== -1;
  }
}

// ES6
const isSearched = searchTerm => item =>
  item.title.toLowerCase().includes(searchTerm.toLowerCase());
```

Chacun défendra quelle fonction est la plus lisible. Personnellement je préfère la seconde. L'écosystème React utilise beaucoup de concept de programmation fonctionnelle. Il se produira souvent que vous utiliserez une fonction qui retourne des fonctions (fonction d'ordre supérieur). En JavaScript ES6, vous pouvez les exprimer de façon plus concise avec les fonctions fléchées.

Dernier point mais pas des moindres, vous êtes obligé d'utiliser la fonction `isSearched()` définie pour filtrer votre liste. Vous lui transmettez la propriété `searchTerm` de votre état local, elle retourne la fonction d'entrée du filtre, et filtre votre liste en se basant sur la condition de filtre. Après quoi, il dresse la liste filtrée par-dessus pour afficher un élément UI pour chacun des éléments de la liste.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(isSearched(this.state.searchTerm)).map(item =>
          ...
        )}
      </div>
    );
  }
}
```

La fonctionnalité de recherche devrait fonctionner maintenant. Essayer par vous-même dans le navigateur.

Exercices :

- lire plus à propos des évènements React[78]
- lire plus à propos des fonctions d'ordre supérieur[79]

78. https://reactjs.org/docs/handling-events.html
79. https://en.wikipedia.org/wiki/Higher-order_function

ES6 Destructuring

Il y a une certaine façon en JavaScript ES6 pour facilité l'accès aux propriétés dans les objets et les tableaux. Cela s'appelle la décomposition. Comparez les bouts de code suivant en JavaScript ES5 et ES6.

Code Playground

```
const user = {
  firstname : 'Robin',
  lastname : 'Wieruch',
};

// ES5
var firstname = user.firstname;
var lastname = user.lastname;

console.log(firstname + ' ' + lastname);
// output : Robin Wieruch

// ES6
const { firstname, lastname } = user;

console.log(firstname + ' ' + lastname);
// output : Robin Wieruch
```

Tandis que vous êtes obligé d'ajouter des lignes supplémentaires à chaque fois que vous voulez accéder à la propriété d'un objet en JavaScript ES5, vous pouvez le faire en une seule ligne en JavaScript ES6. Une bien meilleure pratique pour la lisibilité est l'utilisation de plusieurs lignes lorsque vous décomposez un objet en plusieurs propriétés.

Code Playground

```
const {
  firstname,
  lastname
} = user;
```

Idem pour les tableaux. Vous pouvez tout autant les décomposer. Encore une fois, le multiligne conservera votre code plus lisible et visionnable.

Code Playground

```
const users = ['Robin', 'Andrew', 'Dan'];
const [
  userOne,
  userTwo,
  userThree
] = users;

console.log(userOne, userTwo, userThree);
// output : Robin Andrew Dan
```

Peut-être avez-vous remarqué que l'objet de l'état local dans le composant App peut être décomposé de la même façon. Vous pouvez raccourcir la ligne de code du *filter* et de la *map*.

src/App.js

```
  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        ...
        {list.filter(isSearched(searchTerm)).map(item =>
          ...
        )}
      </div>
    );
```

Vous pouvez le faire à la manière ES5 ou ES6 :

Code Playground

```
// ES5
var searchTerm = this.state.searchTerm;
var list = this.state.list;

// ES6
const { searchTerm, list } = this.state;
```

Mais puisque ce livre utilise JavaScript ES6 la plupart du temps, vous devrez vous y conformer.

Exercices :

- lire plus à propos décomposition ES6[80]

80. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Les composants contrôlés

Vous avez déjà appris des choses au sujet du flux de données unidirectionnel en React. Le même principe est appliqué pour les champs d'entrée, qui mettent à jour l'état local avec le `searchTerm` dans le but de filtrer la liste. Quand l'état change, la méthode `render()` se relance de nouveau et utilise le nouveau `searchTerm` à partir de l'état local pour appliquer la condition de filtre.

Mais n'avons-nous pas oublié quelque chose dans l'élément input ? Une balise HTML arrive avec un attribut `value`. L'attibut `value` habituellement possède la valeur qui est affichée dans le champ d'entrée. Dans ce cas ce serait la propriété `searchTerm`. Cependant, il semble que nous en ayons pas besoin en React.

C'est faux. Les éléments de formulaire tels que `<input>`, `<textarea>` et `<select>` tiennent leur propre état en HTML pur. Ils modifient la valeur intérieurement une fois que quelqu'un la change depuis l'extérieur. Dans React ceci est appelé un **uncontrolled component**, car il gère son propre état. En React, vous devez être sûre de faire de ces éléments des **controlled components**.

Comment devez vous faire cela ? Vous devez seulement définir la valeur de l'attribut du champ d'entrée. La valeur est déjà sauvegardé dans la propriété d'état `searchTerm`. Donc pourquoi ne pas y accéder par là ?

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <form>
          <input
            type="text"
            value={searchTerm}
            onChange={this.onSearchChange}
          />
        </form>
        ...
      </div>
    );
  }
}
```

C'est tout. La boucle de flux de données unidirectionnel pour le champ d'entrée est auto-contenue dorénavant. L'état interne du composant est le seul et unique source de vérité pour le champ d'entrée.

La gestion totale de l'état interne et le flux de données unidirectionnel pourraient être nouveaux pour vous. Mais une fois vous l'aurez utilisé, il deviendra un flux naturel pour implémenter des choses en React. En général, React apporte un nouveau patron de conception avec le flux de données unidirectionnel dans le monde des *single page applications*. C'est d'ores et déjà adopté par plusieurs frameworks et bibliothèques.

Exercices :

- lire plus à propos des formulaires React[81]
- lire plus à propos des différents composants contrôlés[82]

81. https://reactjs.org/docs/forms.html
82. https://github.com/the-road-to-learn-react/react-controlled-components-examples

Diviser les composants

Maintenant, vous avez un imposant composant App. Il continue de grandir et peut finalement devenir confus. Commençons par diviser en plusieurs morceaux de composants plus petits, la création de composants séparés pour le champ de recherche et pour la liste des éléments.

Commençons par utiliser un composant pour l'entrée de recherche et un composant pour la liste d'éléments.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search />
        <Table />
      </div>
    );
  }
}
```

Vous pouvez passer ces propriétés de composants pour qu'ils puissent fonctionner d'eux-mêmes. Dans le cas du composant App il a besoin de passer les propriétés gérées dans l'état local et ses méthodes de classe.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        />
```

```
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Maintenant vous pouvez définir les composants à côté de votre composant App. Ces composants seront également des composants de classe ES6. Ils rendent les mêmes éléments comme auparavant.

Le tout premier est le composant Search.

src/App.js

```
class App extends Component {
  ...
}

class Search extends Component {
  render() {
    const { value, onChange } = this.props;
    return (
      <form>
        <input
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
```

Le second est le composant Table.

src/App.js

```
...

class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        {list.filter(isSearched(pattern)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <button
                onClick={() => onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Maintenant vous avez trois composants de classe ES6. Peut-être avez-vous remarquez l'objet `props` qui est accessible via l'instance de classe en utilisant `this`. Les *props*, version raccourcie pour *properties*, possèdent toutes les valeurs que vous avez transmises au composant lorsque vous les utilisées dans votre composant App. De cette manière, les composants peuvent passer les propriétés vers le bas de l'arbre de composant.

En extractant ces composants du composant App, vous serez en mesure de les réutiliser ailleurs. Puisque les composants obtiennent leur valeurs en utilisant l'objet `props`, vous pouvez transmettre à tout moment les différentes propriétés à vos composants lorsque vous les utilisez ailleurs. Ces composants sont devenu réutilisables.

Exercices :

- envisager les composants que vous pourrez diviser comme cela a été le cas de Search et Table
  - mais ne le faite pas maintenant, sinon, vous rencontrerez des conflits dans les prochains chapitres

Composants composables

Il y a une petite propriété qui est accessible dans l'objet `props` : la propriété `children`. Vous pouvez l'utiliser pour transmettre des éléments à vos composants depuis le dessus, qui sont inconnus pour le composant lui-même, mais rendant possible la composition de composant entre les uns et les autres. Regardons à quoi cela ressemble lorsque vous passez seulement un texte (string) en tant qu'enfant au composant Search.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        >
          Search
        </Search>
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Maintenant le composant Search peut décomposer la propriété `children` de l'objet `props`. Alors il peut spécifier où le `children` devrait être affiché.

src/App.js

```
class Search extends Component {
  render() {
    const { value, onChange, children } = this.props;
    return (
      <form>
        {children} <input
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
```

Maintenant, le texte “Search” peut être visible juste à côté de votre champ d’entrée. Lorsque vous utilisez le composant Search ailleurs, vous pouvez choisir un texte différent si vous le souhaitez. Après tout, vous pouvez transmettre autre chose que tu texte en tant que children. Vous pouvez transmettre un élément et des arbres d’éléments (qui peuvent être encapsulé par des composants de nouveau) en tant qu’enfant. La propriété children rend possible le tissage de composants entre eux.

Exercices :

- lire plus à propos du modèle de composition de React[83]

83. https://reactjs.org/docs/composition-vs-inheritance.html

Composants réutilisables

Les composants réutilisables et composables vous responsabilisent pour fournir des hiérarchies de composants compétents. Il y a la couche de vue fondatrice de React. Les derniers chapitres mentionnèrent le terme de réutilisabilité. Vous pouvez réutiliser les composants Table et Search dorénavant. Même le composant App est réutilisable, car vous pouvez l'instancier ailleurs de nouveau.

Définissons un composant encore plus réutilisable, un composant Button, qui sera finalement réutilisé plus souvent.

src/App.js

```
class Button extends Component {
  render() {
    const {
      onClick,
      className,
      children,
    } = this.props;

    return (
      <button
        onClick={onClick}
        className={className}
        type="button"
      >
        {children}
      </button>
    );
  }
}
```

Cela peut sembler redondant de déclarer un tel composant. Vous utiliserez un composant `Button` au lieu de l'élément `button`. Il vous épargne seulement le `type="button"`. Hormis pour le type de l'attribut vous avez à définir tout le reste lorsque vous voulez utiliser le composant Button. Mais vous êtes obligé de penser sur de l'investissement à long terme ici. Imaginez-vous avez plusieurs boutons dans votre application, mais vous désirez modifier un attribut, le style ou un comportement pour le bouton. Sans le composant vous devriez refactorer tous les boutons. Au lieu de cela le composant Button assure d'avoir seulement une seule source vérité. Un Button pour refactorer tous les boutons en une seule fois. Un Buton pour gouverner tous les autres.

Puisque vous avez déjà un élément bouton, vous pouvez utiliser le composant Button à la place. Il omet l'attribut type, car le composant Button le spécifie.

src/App.js

```
class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        {list.filter(isSearched(pattern)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <Button onClick={() => onDismiss(item.objectID)}>
                Dismiss
              </Button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Le composant Button attend une `props className` dans les propriétés. L'attribut `className` est un autre dérivé React pour l'attribut HTML `class`. Mais nous ne transmettons aucun `className` lorsque le Button était utilisé. Dans le code il devrait être plus explicite que la `className` dans le composant Button soit optionnelle. Ainsi, vous pouvez assigner une valeur par défaut dans votre objet de décomposition.

Par conséquent, vous pouvez utiliser le paramètre par défaut qui est une fonctionnalité de JavaScript ES6.

src/App.js

```
class Button extends Component {
  render() {
    const {
      onClick,
      className = '',
      children,
    } = this.props;

    ...
  }
}
```

Maintenant, quand il n'y a pas de propriété `className` spécifiée lors de l'utilisation du composant Button, la valeur sera une chaine de caractères au lieu d'`undefined`.

Déclarations de composant

Maintenant vous avez quatre composants de classe ES6. Mais vous pouvez faire mieux. Laissez-moi vous présenter les composants fonctionnels stateless comme alternative de vos composants de classe ES6. Avant que vous refactorez vos composants, introduisons les différents types de composants React.

- **Functional Stateless Components** : Ces composants sont des fonctions qui ont une entrée et retourne une sortie. Les entrées sont les propriétés. La sortie est une instance de composant en JSX pur. Pour l'instant c'est assez similaire à une classe de composant ES6. Cependant, les composants fonctionnels stateless sont des fonctions (functional) et ils n'ont pas d'état local (stateless). Vous ne pouvez accéder ou mettre à jour l'état avec `this.state` ou `this.setState()` car il n'y a pas d'objet `this`. De plus, ils ne possèdent pas les méthodes de cycle de vie. Vous n'avez pas encore appris les méthodes de cycle de vie, mais vous en avez déjà utilisé deux : `constructor()` et `render()`. Tandis que le constructeur se lance une seule fois dans le temps de vie d'un composant, la méthode de classe `render()` se lance une première fois à l'initialisation et à chaque fois que le composant se met à jour. Garder en mémoire que les composants fonctionnels stateless n'ont pas de méthodes de cycle de vie, pour lorsque vous arriverez plus tard au chapitre sur les méthodes du cycle de vie.
- **ES6 Class Components** : Vous avez déjà utilisé ce type de déclaration de composant dans vos quatre composants. Dans la définition de classe, ils étendent le composant React. l'`extend` fournit tous les méthodes de cycle de vie, disponible au sein de l'API de composant React. De cette façon, vous serez en mesure d'utiliser la méthode de classe `render()`. De plus, vous pouvez stocker et manipuler l'état à l'intérieur des composants de classe ES6 en utilisant `this.state` et `this.setState()`.
- **React.createClass** : La déclaration de composant été utilisée dans les anciennes versions de React et encore dans les applications React ES5. Mais Facebook la déclaré comme dépréciée[84] en faveur du JavaScript ES6. Ils ont même ajouté un warning de dépréciation dans la version 15.5[85]. Vous ne les l'utiliserez pas dans le livre.

Donc globalement il reste plus que deux déclarations de composant. Mais quand utiliser les composants fonctionnels stateless au lieu des composants de classe ES6? Une règle générale est d'utiliser les composants fonctionnels stateless lorsque vous n'avez pas besoin de l'état local ou des méthodes de cycle de vie du composant. Habituellement vous débutez par implémenter vos composants en tant que composants fonctionnels stateless. Une fois que vous avez besoin d'accéder à l'état ou aux méthodes de cycle de vie, vous êtes obligé de le refactorer en un composant de classe ES6. Dans notre application, nous avons commencé dans le sens inverse pour les besoins de l'apprentissage de React.

Retournons à notre application. L'app component utilise l'état interne. C'est pourquoi il doit rester en tant que composant de classe ES6. Mais les trois autres composants de classes ES6 sont stateless. Ils

84. https://reactjs.org/blog/2015/03/10/react-v0.13.html
85. https://reactjs.org/blog/2017/04/07/react-v15.5.0.html

n'ont pas besoin d'accéder à `this.state` ou `this.setState()`. Encore plus important, ils n'ont pas de méthodes de cycle de vie. Refactorons ensemble le composant Search en un composant fonctionnel stateless. Le refactor du composant Table et Button restera comme étant votre exercice.

src/App.js

```
function Search(props) {
  const { value, onChange, children } = props;
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

C'est globalement tout. Les `props` sont accessibles dans la signature de fonction et la valeur de retour est du JSX. Mais vous pouvez faire du code plus judicieux dans un composant fonctionnel stateless. Vous connaissez déjà la décomposition ES6. La meilleure pratique est de l'utiliser dans une signature de fonction pour décomposer les `props`.

src/App.js

```
function Search({ value, onChange, children }) {
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

Mais on peut faire mieux. Vous connaissez déjà les fonctions fléchées ES6 permettant de conserver vos fonctions concises. Vous pouvez supprimer le *block body* de la fonction. Dans un corps concis un retour implicit est attaché ainsi vous pouvez supprimer la déclaration du `return`. Puisque votre composant fonctionnel stateless est une fonction, vous pouvez également la conserver concise.

src/App.js

```
const Search = ({ value, onChange, children }) =>
  <form>
    {children} <input
      type="text"
      value={value}
      onChange={onChange}
    />
  </form>
```

La dernière étape était spécifiquement utile pour garantir d'avoir seulement les propriétés en entrée et le JSX comme sortie. Rien d'autre entre. Pourtant, vous pouvez *do something* entre en utilisant un bloc dans votre fonction fléchée ES6.

Code Playground

```
const Search = ({ value, onChange, children }) => {

  // do something

  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

Mais pour maintenant vous n'en aurez pas besoin. C'est pourquoi vous pouvez conserver la précédente version sans le *block body*. Lors de l'utilisation des *block bodies*, les personnes ont souvent tendance de faire trop de choses dans la fonction. En excluant le *block body*, vous pouvez vous concentrer sur l'entrée et la sortie de votre fonction.

Maintenant vous avez un composant fonctionnel stateless allégé. Une fois que vous aurez besoin d'accéder à son état interne de composant ou à ses méthodes de cycle de vie, vous le refactorerez en un composant de classe ES6. En plus vous voyez comment le JavaScript ES6 peut être utilisé dans les composants React pour les rendre plus concis et élégant.

Exercices :

- refactorer le composant Table et Button en composants fonctionnels stateless

- lire plus à propos des composants de classe ES6 et des composants fonctionnels stateless[86]

86. https://reactjs.org/docs/components-and-props.html

Styliser les composants

Ajoutons quelques basiques styles pour votre application et vos composants. Vous pouvez réutiliser les fichiers *src/App.css* et *src/index.css*. Ces fichiers doivent toujours être dans votre projet puisque que vous l'avez initié avec *create-react-app*. Ils devraient aussi être importés dans vos fichiers *src/App.js* et *src/index.js*. J'ai préparé quelques CSS que vous pouvez simplement copier et coller vers ces fichiers, mais n'hésitez pas à utiliser votre propre style.

Tout d'abord, le style pour l'ensemble de votre application.

src/index.css

```
body {
  color : #222;
  background : #f4f4f4;
  font : 400 14px CoreSans, Arial, sans-serif;
}

a {
  color : #222;
}

a :hover {
  text-decoration : underline;
}

ul, li {
  list-style : none;
  padding : 0;
  margin : 0;
}

input {
  padding : 10px;
  border-radius : 5px;
  outline : none;
  margin-right : 10px;
  border : 1px solid #dddddd;
}

button {
  padding : 10px;
  border-radius : 5px;
  border : 1px solid #dddddd;
```

```
  background : transparent;
  color : #808080;
  cursor : pointer;
}

button :hover {
  color : #222;
}

* :focus {
  outline : none;
}
```

Deuxièmement, le style pour vos composants dans le fichier App.

src/App.css

```
.page {
  margin : 20px;
}

.interactions {
  text-align : center;
}

.table {
  margin : 20px 0;
}

.table-header {
  display : flex;
  line-height : 24px;
  font-size : 16px;
  padding : 0 10px;
  justify-content : space-between;
}

.table-empty {
  margin : 200px;
  text-align : center;
  font-size : 16px;
}

.table-row {
```

```
  display : flex;
  line-height : 24px;
  white-space : nowrap;
  margin : 10px 0;
  padding : 10px;
  background : #ffffff;
  border : 1px solid #e3e3e3;
}

.table-header > span {
  overflow : hidden;
  text-overflow : ellipsis;
  padding : 0 5px;
}

.table-row > span {
  overflow : hidden;
  text-overflow : ellipsis;
  padding : 0 5px;
}

.button-inline {
  border-width : 0;
  background : transparent;
  color : inherit;
  text-align : inherit;
  -webkit-font-smoothing : inherit;
  padding : 0;
  font-size : inherit;
  cursor : pointer;
}

.button-active {
  border-radius : 0;
  border-bottom : 1px solid #38BB6C;
}
```

Maintenant vous pouvez utiliser le style dans certains de vos composants. N'oubliez pas d'utiliser `className` de React plutôt que l'attribut HTML `class`.

Premièrement, appliquez-le dans votre composant de classe ES6 App.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
          >
            Search
          </Search>
        </div>
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Deuxièmement, appliquez-le dans votre composant fonctionnel stateless Table.

src/App.js

```
const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    {list.filter(isSearched(pattern)).map(item =>
      <div key={item.objectID} className="table-row">
        <span>
          <a href={item.url}>{item.title}</a>
        </span>
        <span>{item.author}</span>
        <span>{item.num_comments}</span>
        <span>{item.points}</span>
        <span>
          <Button
```

```
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Dismiss
          </Button>
        </span>
      </div>
    )}
  </div>
```

Maintenant que vous avez stylisé votre application et vos composants avec du CSS basique. Il devrait avoir un rendu décent. Comme vous savez, JSX mélange l'HTML et le JavaScript. Maintenant une personne peut défendre d'également ajouter du CSS dans tout çà. C'est appelé de l'*inline style*. Vous pouvez définir des objets JavaScript et les transmettre à l'attribut style d'un élément.

Gardons la largeur de colonne de Table adaptable en utilisant l'*inline style*.

src/App.js

```
const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    {list.filter(isSearched(pattern)).map(item =>
      <div key={item.objectID} className="table-row">
        <span style={{ width : '40%' }}>
          <a href={item.url}>{item.title}</a>
        </span>
        <span style={{ width : '30%' }}>
          {item.author}
        </span>
        <span style={{ width : '10%' }}>
          {item.num_comments}
        </span>
        <span style={{ width : '10%' }}>
          {item.points}
        </span>
        <span style={{ width : '10%' }}>
          <Button
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Dismiss
          </Button>
        </span>
      </div>
```

```
    )}
  </div>
```

Le style est en ligne maintenant. Vous pouvez définir les objets de style à l'extérieur de votre élément pour rendre le tout plus propre.

Code Playground

```
const largeColumn = {
  width : '40%',
};

const midColumn = {
  width : '30%',
};

const smallColumn = {
  width : '10%',
};
```

Après quoi vous les utiliserez dans vos colonnes `<span style={smallColumn}>`.

En général, vous trouverez des opinions et des solutions différentes pour le style au sein de React. Pour le moment, vous utilisez du pur CSS et de l'*inline style*. Cela est suffisant pour débuter.

Je ne veux pas vous orienter ici, mais je veux vous donner un peu plus de posibilités d'options. Vous pouvez les lire et les appliquer de votre propre chef. Mais si vous êtes nouveau à React, Je vous recommande pour le moment de rester sur du CSS pur et en de l'*inline style*.

- styled-components[87]
- CSS Modules[88]

87. https://github.com/styled-components/styled-components
88. https://github.com/css-modules/css-modules

Vous avez appris les bases pour écrire votre propre application React ! Récapitulons les derniers chapitres :

- React
  - utiliser `this.state` et `setState()` pour gérer votre état interne dun composant
  - passer des fonctions ou des méthodes de classe à votre element handler
  - utiliser les formulaires et évènements dans React pour ajouter des interactions
  - le flux de données unidirectionnel est un concept important dans React
  - favoriser les composants contrôlés (ou controlled components)
  - composer les composants avec children et des composants réutilisables
  - utilisation et implémentation des composants de classe ES6 et des composants fonctionnels stateless
  - les approches pour styliser vos composants
- ES6
  - les fonctions qui sont liées à une classe sont des méthodes de classe
  - décomposition d'objets et de tableaux
  - paramètres par défaut.
- Générale
  - les fonctions d'ordre supérieur

Encore une fois il fait sens de prendre une pause. Intérioriser les acquis et appliquer les de votre propre initiative. Vous pouvez expérimenter avec le code source que vous avez écrit jusqu'à présent. Vous pouvez trouver le code source dans le dépôt officiel[89].

89. https://github.com/the-road-to-learn-react/hackernews-client/tree/5.2

Getting Real with an API

Maintenant il est temps d'obtenir quelques choses de concret avec une API, car il peut devenir très vite ennuyant de ne traiter qu'un échantillon de données.

Si vous n'êtes pas familier avec les APIs, Je vous encourage à à lire mon périple où j'ai appris les APIs[90].

Connaissez-vous la plateforme Hacker News[91] ? C'est un agrégateur de news génial à propos de sujets technologiques. Dans ce livre, vous utiliserez l'API d'Hacker News pour aller chercher les articles en tendance au sein de la plateforme. Il y a une API basique[92] et une de recherche[93] pour obtenir les données de la plateforme. La dernière API à du sens dans le cadre de cette application dans le but de rechercher les histoires sur Hacker News. Vous pouvez visiter la spécification de l'API pour avoir une meilleure compréhension de la structure de données.

90. https://www.robinwieruch.de/what-is-an-api-javascript/
91. https://news.ycombinator.com/
92. https://github.com/HackerNews/API
93. https://hn.algolia.com/api

Les méthodes du cycle de vie

Vous aurez besoin d'apprendre les méthodes de cycle de vie de React avant de commencer à aller chercher les données dans votre composant en utilisant une API. Ces méthodes sont un crochet à l'intérieur du cycle de vie d'un composant React. Ils peuvent être utilisés dans des composants de classe ES6, mais en aucun cas dans des composants fonctionnels stateless.

Vous rappelez vous quand dans un précédent chapitre vous avez appris à propos des classes ES6 et de comment ils sont utilisés dans React ? Hormis la méthode `render()`, il y a plusieurs méthodes qui peuvent être surchargées dans un composant de classe React ES6. Toutes sont des méthodes du cycle de vie. Concentrons-nous sur ces derniers :

Vous connaissez dors et déjà deux méthodes du cycle de vie qui peuvent être utilisées dans le composant de classe ES6 : `constructor()` and `render()`.

Le constructeur est appelé seulement une seule fois quand une instance du composant est créée et insérée dans le DOM. Le composant est instancié. Ce procédé est appelé le montage du composant (*mounting of the component*).

La méthode `render()` est aussi appelée durant le procédé de montage, mais aussi lors de la mise à jour du composant. Chaque fois que l'état ou les propriétés du composant changent, la méthode `render()` du composant est appelée.

Maintenant vous en savez plus à propos des méthodes du cycle de vie et de quand elles sont appelées. D'ailleurs, vous les utilisez d'ores et déjà. Mais il y en a encore plus parmi eux.

Le montage d'un composant a deux méthodes de cycle de vie supplémentaires : `getDerivedStateFromProps()` et `componentDidMount()`. Le constructeur est appelé en premier, `getDerivedStateFromProps()` sera appelée avant la méthode `render()` et `componentDidMount()` est apellée après la méthode `render()`.

Dans l'ensemble le processus de montage a 4 méthodes de cycle de vie. Elles sont invoquées dans l'ordre suivant :

- constructor()
- getDerivedStateFromProps()
- render()
- componentDidMount()

Mais qu'advient-il du cycle de vie de la mise à jour d'un composant qui se produit lorsque l'état ou les propriétés changent ? Dans l'ensemble il a 5 méthodes de cycle de vie dans l'ordre suivant :

- getDerivedStateFromProps()
- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()

- componentDidUpdate()

Dernier point, mais pas des moindres, il y a le cycle de vie de démontage. Il a seulement une méthode de cycle de vie : `componentWillUnmount()`.

Après tous, vous n'avez pas besoin de savoir toutes ces méthodes de cycle pour débuter. Cela peut-être encore un peu intimidant vous ne les utiliserez pas toutes. Même dans une grande application React vous utiliserez seulement une petite partie d'entre eux à l'exception du `constructor()` et de la méthode `render()`. Pourtant, il est bon de connaître que chaque méthode de cycle de vie peut être utilisé pour des cas d'utilisation spécifiques :

- **constructor(props)** - C'est appelé lorsque le composant est initialisé. Vous pouvez donner un état initial au composant et lier les méthodes de classe dans cette méthode du cycle de vie.
- **static getDerivedStateFromProps(props, state)()** - C'est appelé avant la méthode `render()` du cycle de vie, à la fois lors du montage initial et lors des mis à jours ultérieurs. Il devrait retourner un objet pour mettre à jour l'état, ou null en cas de non mis à jour. Il existe pour de **rares** cas d'utilisation où l'état dépend des changements au fil du temps. Il est important de comprendre que c'est une méthode statique et qu'il n'a pas accès à l'instance du composant.

, both on the initial mount and on the subsequent updates. It should return an object to update the state, or null to update nothing. It exists for **rare** use cases where the state depends on changes in props over time. It is important to know that this is a static method and it doesn't have access to the component instance.

C'est pourquoi il peut être utilisé pour définir l'état interne du composant, car il ne déclenchera pas un second rendu du composant. Généralement c'est recommandé d'utiliser le `constructor()` pour mettre en place l'état initial.

- **render()** - Cette méthode du cycle de vie est obligatoire et retourne les éléments en tant que sortie du composant. La méthode devra être pure et par conséquent ne devra pas modifier l'état du composant. Il a une entrée comme propriétés et état et retourne un élément.
- **componentDidMount()** - Elle est appelée une seule fois lorsque le composant a été monté. C'est le moment parfait pour faire des requêtes asynchrones dans le but d'aller chercher des données depuis une API. Les données rapportées seront stockées dans l'état interne du composant pour être affichées dans la méthode du cycle de vie `render()`.
- **shouldComponentUpdate(nextProps, nextState)** - C'est aussi appelée lors de la mise à jour du composant à cause d'un changement du state ou des propriétés. Vous l'utiliserez dans des applications React mûrs pour des optimisations de performance. Dépendant d'un boolean que vous retournez depuis cette méthode du cycle de vie, le composant et tous ses enfants seront rendus ou pas durant un cycle de vie de mise à jour. Vous pouvez empêcher la méthode de rendu du cycle de vie d'un composant.
- **componentWillUpdate(nextProps, nextState)** - La méthode de cycle de vie est immédiatement invoqué avant la méthode `render()`. Vous avez toujours les prochaines propriétés et le

prochain état à votre disposition. Vous pouvez utiliser la méthode en dernier recours pour performer des préparatifs avant que la méthode de rendu soit éxecutée. Noter que vous ne pouvez plus déclencher `setState()`. Si vous voulez calculer l'état en fonction des prochaines propriétés, vous êtes obligé d'utiliser `componentWillReceiveProps()`.

- **getSnapshotBeforeUpdate(prevProps, prevState)** - Cette méthode du cycle de vie est invoquée juste avant que la plus récent rendu en sortie soit renvoyé vers le DOM. Dans de rares cas d'utilisation, le composant a besoin de capturer l'information depuis le DOM avant qu'il soit potentiellement changé. Cette méthode du cycle de vie permet au composant de le faire. Une autre méthode (`componentDidUpdate()`) recevra toutes valeurs retournées par `getSnapshotBeforeUpdate()` comme un paramètre.
- **componentDidUpdate(prevProps, prevState, snapshot)** La méthode de cycle de vie est immédiatement invoquée après que des mises à jours se produisent, mais pas lors du rendu initial. Vous pouvez l'utiliser comme opportunité pour performer des opérations du DOM ou pour performer des requêtes asynchrones supplémentaires. Si votre composant implémente la méthode `getSnapshotBeforeUpdate()`, la valeur que ce dernier retourne sera reçu comme paramètre de `snapshot`.
- **componentDidUpdate(prevProps, prevState, snapshot)** - The lifecycle method is immediately invoked after updating occurs, but not for the initial render. You can use it as opportunity to perform DOM operations or to perform further asynchronous requests. If your component implements the `getSnapshotBeforeUpdate()` method, the value it returns will be received as the `snapshot` parameter.
- **componentWillUnmount()** - Elle est appelée avant que vous détruisez votre composant. Vous devez utiliser cette méthode du cycle de vie pour performer toutes tâches de nettoyage.

Vous utilisez déjà les méthodes du cycle de vie `constructor()` et `render()`. Ce sont les méthodes du cycle de vie communément utilisées pour des composants de classe ES6. En fait, la méthode `render()` est requise, autrement vous ne seriez pas en mesure de retourner une instance de composant.

Il y a plus de méthode du cycle de vie : `componentDidCatch(error, info)`. Cela a été introduit dans [React 16] (https ://www.robinwieruch.de/what-is-new-in-react-16/) et est utilisée pour attraper les erreurs dans le composant. Par exemple, afficher la liste d'échantillon de votre application fonctionne bien. Mais cela pourrait y avoir un cas lorsque la liste dans l'état local est établie à `null` par accident (ex : lors de la recherche d'une liste depuis une API externe, mais que la requête échoue et que vous écrivez l'état local de la liste à null). Après quoi, il ne serait plus possible de filtrer et mapper la liste, car c'est `null` et non une liste vide. Le composant sera cassé et l'entièreté de l'application échouera. Maintenant, en utilisant `componentDidCatch()`, vous pouvez attraper l'erreur, la stocker dans votre état local, et montrer un message optionnel à votre utilisateur de l'application qu'une erreur c'est produite.

Exercices :

- lire plus à propos des méthodes du cycle de vie dans React[94]

94. https://reactjs.org/docs/react-component.html

- lire plus à propos de l'état relié aux méthodes du cycle de vie dans React[95]
- lire plus à propos de la gestion d'erreur dans les composants[96]

95. https://reactjs.org/docs/state-and-lifecycle.html
96. https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html

Aller chercher les données

Maintenant vous êtes prêt pour aller chercher les données de l'API d'Hacker News. Il y a une méthode du cycle de vie mentionné qui peut être utilisée pour aller chercher les données `componentDidMount()`. Vous utiliserez l'API native *fetch* de JavaScript pour performer la requête.

Avant que nous puissions l'utiliser, établissons les constantes d'URL et les paramètres par défaut pour décomposer la requête d'API en morceaux.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';

...
```

En JavaScript ES6, vous pouvez utiliser les template strings[97] pour concaténer les chaînes de caractères. Vous l'utiliserez pour concaténer votre URL pour le point d'entrée de l'API.

Code Playground

```
// ES6
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT_QUERY}`;

// ES5
var url = PATH_BASE + PATH_SEARCH + '?' + PARAM_SEARCH + DEFAULT_QUERY;

console.log(url);
// output : https://hn.algolia.com/api/v1/search?query=redux
```

Cela conservera votre composition d'URL souple pour le futur.

Mais passons à la requête de l'API où vous utiliserez l'URL. Le processus de recherche sera présenté en une fois, mais chaque étape sera expliquée par la suite.

97. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals

src/App.js

```
...

class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result : null,
      searchTerm : DEFAULT_QUERY,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  setSearchTopStories(result) {
    this.setState({ result });
  }

  componentDidMount() {
    const { searchTerm } = this.state;

    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(error => error);
  }


  ...
}
```

Beaucoup de choses dans ce code. Je pensais le casser en plusieurs petits morceaux. Mais encore une fois il serait difficile de saisir les relations de chacun des morceaux. Laissez-moi expliquer chaque étape en détail.

Premièrement, vous pouvez supprimer la liste d'objets, car vous retournez une liste réelle depuis l'API d'Hacker News. Les données d'échantillon ne sont plus utilisées maintenant. L'état initial de votre composant possède un résultat vide tout comme le terme de recherche par défaut. Le même

terme de recherche par défaut est utilisé dans le champ d'entrée du composant Search et dans votre première requête.

Deuxièmement, vous utilisez la méthode du cycle de vie `componentDidMount()` pour aller chercher les données après que le composant est été monté. Dans le tout premier *fetch*, le terme de recherche par défaut issue de l'état local est utilisé. Il ira chercher les sujets reliés à "redux", car c'est le paramètre par défaut.

Troisièmement, l'API native *fetch* est utilisée. Les *templates strings* du JavaScript ES6 permettent de composer l'URL avec `searchTerm`. L'URL est l'argument pour la fonction d'API native *fetch*. La réponse a besoin d'être transformée en structure de données JSON, qui est une étape obligatoire pour une fonction native *fetch* lorsque l'on traite avec des structures de données JSON, et peut finalement être écrit comme résultat dans l'état interne du composant. De plus, le *catch block* est utilisé dans le cas d'une erreur. Si une erreur se produit durant la requête, la fonction lancera l'intérieur du *catch block* au lieu du *then block*. Dans un prochain chapitre du livre, vous inclurez la gestion d'erreur.

Dernier point, mais pas des moindres, n'oubliez pas de lier votre nouvelle méthode de composant dans le constructeur.

Maintenant vous pouvez utiliser les données obtenues à la place de la liste d'objets. Cependant, vous devez être de nouveau prudent. Le résultat n'est pas seulement une liste de données. C'est un objet complexe avec des metas informations et une liste de succès qui sont dans notre cas les articles[98]. Vous pouvez afficher l'état interne avec un `console.log(this.state);` dans votre méthode `render()` pour le visualiser.

Dans la prochaine étape, vous utiliserez le résultat pour le rendu. Mais nous empêcherons de rendre n'importe quoi, ainsi nous retournerons null, lorsqu'il n'y a pas de résultat en premier lieu. Une fois que la requête de l'API est parvenue, le résultat est sauvegardé dans l'état et le composant App sera rerendu avec l'état mis à jour.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;

    if (!result) { return null; }

    return (
      <div className="page">
        ...
        <Table
          list={result.hits}
```

98. https://hn.algolia.com/api

```
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Récapitulons ce qu'ils c'est passé durant le cycle de vie du composant. Votre composant a été initialisé par le constructeur. Après cela, il est rendu pour la première fois. Mais vous l'empêchez d'afficher n'importe quoi, car le résultat dans l'état local est null. C'est permis de retourner null pour un composant dans le but de ne rien afficher. Alors la méthode de cycle de vie `componentDidMount()` se lance. Dans cette méthode vous allez chercher de façon asynchrone les données depuis l'API d'Hacker News. Une fois les données arrivées, il change votre état interne du composant dans `setSearchTopStories()`. Après quoi, le cycle de vie de mis à jour entre en jeu car l'état local a été mis à jour. Le composant lance de nouveau la méthode `render()`, mais cette fois avec un résultat hydraté dans votre état interne du composant. Le composant et ainsi le composant Table avec son contenu seront rendus.

Vous avez utilisé l'API native *fetch* qui est supporté par la plupart des navigateurs pour performer une requête asynchrone vers une API. La configuration de *create-react-app* fait en sorte qu'elle soit supportée au sein de tous les navigateurs. Il y a des packages node tiers que vous pouvez utiliser pour substituer l'API native *fetch* : superagent[99] et axios[100].

Garder en mémoire que le livre se construit sur la notation abrégée de JavaScript pour des vérifications sûres. Dans l'exemple précédent, le `if (!result)` a été utilisé en faveur de `if (result === null)`. C'est de même pour les autres cas au travers du livre. Par exemple, `if (!list.length)` est utilisé en faveur de `if (list.length === 0)` ou `if (someString)` est utilisé en faveur de `if (someString !== '')`. Lisez à propos de ce sujet si vous n'êtes pas assez à l'aise avec.

Retour à votre application : la liste des succès devrait être visible maintenant. Cependant, il y a également deux bugs régressifs dans l'application. Premièrement, le bouton "Dismiss" est cassé. Il ne connaît plus l'objet résultat complexe et opère encore sur la simple liste de l'état local lors du rejet d'un objet. Deuxièmement, lorsque la liste est affichée mais que vous essayez de chercher quelque chose d'autres, la liste reste filtrée côté client même si la recherche initiale a été faite en cherchant les articles côté serveur. Le comportement parfait serait d'aller chercher un autre objet résultat depuis l'API lors de l'utilisation du composant Search. Les deux bugs de régression seront fixés dans le prochain chapitre.

Exercices :

- lire plus à propos des ES6 template strings[101]

99. https://github.com/visionmedia/superagent
100. https://github.com/mzabriskie/axios
101. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals

- lire plus à propos de l'API fetch native[102]
- lire plus à propos de la recherhce de données au sein de React[103]

102. https://developer.mozilla.org/en/docs/Web/API/Fetch_API
103. https://www.robinwieruch.de/react-fetching-data/

ES6 Spread Operators

Le bouton “Dismiss” ne fonctionne plus à cause de la méthode `onDismiss()` qui n’est pas consciente de l’objet résultat complexe. Il connaît seulement la liste simple dans l’état interne. Mais ce n’est plus une liste simple maintenant. Changeons cela pour opérer sur l’objet résultat au lieu de la liste elle-même.

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    ...
  });
}
```

Mais que se passe-t-il maintenant dans `setState()` ? Malheureusement le résultat est un objet complexe. La liste des hits est seulement une des multiples propriétés à l’intérieur de l’objet. Cependant, seule la liste sera mis à jour, quand un objet sera supprimé dans l’objet résultat, tandis que les autres propriétés restent les mêmes.

Une approche pourrait être d’écrire les hits dans l’objet résultat. Nous ne souhaitons pas le faire de cette façon, je le démontrerai.

Code Playground

```
// ne pas faire çà
this.state.result.hits = updatedHits;
```

React adopte les structures de données immutables. Ansi vous ne devez pas muter un objet (ou muter l’état directement). Une meilleure approche est de générer un nouvel objet basé sur les informations que vous possédez ? Ainsi aucun des objets sera altéré. Vous conserverez les structures de données immutables. Vous retournerez toujours un nouvel objet et altérerez jamais un objet.

Par conséquent vous pouvez utiliser `Object.assign()` de l’ES6 JavaScript. Il prend en premier argument un objet cible. Tous les arguments suivants sont des objets sources. Ces objets sont fusionnés à l’intérieur de l’objet cible. L’objet cible peut être un objet vide. Il adopte l’immutabilité, aucun objet source sera muté. Cela ressemblera à cela :

Code Playground

```
const updatedHits = { hits : updatedHits };
const updatedResult = Object.assign({}, this.state.result, updatedHits);
```

Les derniers objets surpasseront les premiers objets fusionnés lorsqu'ils partagent les mêmes noms de propriétés. Maintenant, appliquons à la méthode `onDismiss()` :

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    result : Object.assign({}, this.state.result, { hits : updatedHits })
  });
}
```

Ce pourrait déjà être la solution. Mais il y a un moyen plus simple en ES6 JavaScript ainsi que dans les futurs sortis JavaScript. Puis je introduire le *spread operator* pour vous ? Il est constitué de seulement trois points : . . . . Quand il est utilisé, toutes les valeurs d'un tableau ou d'un objet seront copiées dans un autre tableau ou objet.

Examinons même si vous n'en avez apas encore besoin.

Code Playground

```
const userList = ['Robin', 'Andrew', 'Dan'];
const additionalUser = 'Jordan';
const allUsers = [ ...userList, additionalUser ];

console.log(allUsers);
// output : ['Robin', 'Andrew', 'Dan', 'Jordan']
```

La variable `allUsers` est un tableau complètement nouveau. Les autres variables `userList` et `additionalUser` restent les mêmes. Vous pouvez même fusionner deux tableaux de cette façon en un nouveau tableau.

Code Playground

```
const oldUsers = ['Robin', 'Andrew'];
const newUsers = ['Dan', 'Jordan'];
const allUsers = [ ...oldUsers, ...newUsers ];

console.log(allUsers);
// output : ['Robin', 'Andrew', 'Dan', 'Jordan']
```

Maintenant regardons l'*object spread operator*. C'est n'est pas de l'ES6 JavaScript. C'est encore une proposition pour une prochaine version JavaScript[104] utilisé par la communauté React. C'est pourquoi *create-react-app* a incorporé la fonctionnalité dans la configuration.

Gloablement c'est idem qu'un *array spread operator* d'ES6 JavaScript mais avec des objets. Il copie chaque paire clé valeur à l'intérieur d'un nouvel objet.

Code Playground

```
const userNames = { firstname : 'Robin', lastname : 'Wieruch' };
const age = 28;
const user = { ...userNames, age };

console.log(user);
// output : { firstname : 'Robin', lastname : 'Wieruch', age : 28 }
```

Plusieurs objets peuvent être propagés comme dans l'exemple de l'*array spread*.

Code Playground

```
const userNames = { firstname : 'Robin', lastname : 'Wieruch' };
const userAge = { age : 28 };
const user = { ...userNames, ...userAge };

console.log(user);
// output : { firstname : 'Robin', lastname : 'Wieruch', age : 28 }
```

Après tout, il peut être utilisé pour remplacer `Object.assign()`.

104. https://github.com/sebmarkbage/ecmascript-rest-spread

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    result : { ...this.state.result, hits : updatedHits }
  });
}
```

Maintenant le bouton “Dismiss” devra fonctionner de nouveau, car la méthode `onDismiss()` est consciente de l’objet résultat complexe et comment le mettre à jour après le rejet d’un objet depuis la liste.

Exercices :

- lire plus à propos de l’ES6 Object.assign()[105]
- lire plus à propos de l’ES6 array spread operator[106]
  - le *object spread operator* est brièvement mentionné

105. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign
106. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator

Rendu conditionnel

Le rendu conditionnel a été introduit assez tôt dans les applications React. Mais pas dans le cadre de ce livre, car il n'y avait pas encore de cas d'utilisation. Le rendu conditionnel se produit quand on souhaite faire une décision pour rendre soit un élément soit un autre élément. Parfois cela signifie rendre un élément ou rien. Après tout, l'usage le plus trivial d'un rendu conditionnel peut être représenté par une déclaration *if-else* en JSX.

Au démarrage l'objet `result` dans l'état interne du composant est `null`. Jusqu'à présent, le composant App retournait aucun élément quand le `result` ne provennait pas de l'API. C'est déjà un rendu conditionnel, car pour une certaine condition vous retournez le résultat plus tôt au sein de la méthode `render()` du cycle de vie. Le composant App rend soit ses éléments soit rien.

Nous pouvons aller plus loin. Cela a plus de sens d'englober le composant Table, qui est le seul composant qui dépend du `result`, dans un rendu conditionnel indépendant. Tout le reste devrait être affiché, même s'il n'y a pas encore de `result`. Vous pouvez simplement utiliser une opération ternaire dans votre JSX.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
          >
            Search
          </Search>
        </div>
        { result
         ? <Table
            list={result.hits}
            pattern={searchTerm}
            onDismiss={this.onDismiss}
          />
          : null
        }
```

```
      </div>
    );
  }
}
```

C'est une seconde option pour représenter un rendu conditionnel. Une troisième option est un opérateur `&&` logique. En JavaScript `true && 'Hello World'` est toujours évalué à 'Hello World'. Un `false && 'Hello World'` est toujours évalué à faux.

Code Playground

```
const result = true && 'Hello World';
console.log(result);
// output : Hello World

const result = false && 'Hello World';
console.log(result);
// output : false
```

Dans React vous pouvez rendre utilise ce comportement. Si la condition est vraie, l'expression après l'opérateur `&&` logique sera la sortie. Si la condition est fausse, React l'ignore et saute l'expression. C'est applicable dans le cas du rendu conditionnel de Table, car il devrait retourner un Table ou rien.

src/App.js

```
{ result &&
  <Table
    list={result.hits}
    pattern={searchTerm}
    onDismiss={this.onDismiss}
  />
}
```

C'étaient quelques approches pour utiliser le rendu conditionnel dans React. Vous pouvez lire à propos de plus d'alternatives dans une liste exhaustive d'exemples pour des approches de rendu conditionnel[107]. De plus, vous serez amené à connaitre leur différent cas d'utilisation et quand les appliquer.

En fin de compte, vous devrez être capable de voir les données récupérées dans votre application. Tout hormis le composant Table est affiché en attendant les données récupérées. Une fois la requête résolue le résultat est stocké à l'intérieur de l'état local, le Table est affiché car la méthode `render()` se lance de nouveau et la condition dans le rendu conditionnel se résout en faveur de l'affichage du composant Table.

107. https://www.robinwieruch.de/conditional-rendering-react/

Exercices :

- lire plus à propos des différentes façons pour les rendus conditionnels[108]
- lire plus à propos du rendu conditionnel de React[109]

108. https://www.robinwieruch.de/conditional-rendering-react/
109. https://reactjs.org/docs/conditional-rendering.html

Recherche côté client ou côté serveur

Actuellement, lorsque vous utilisez le composant Search avec ses champs d'entrée, vous filtrerez la liste. Cependant cela se produit côté client. Maintenant vous allez utiliser l'API d'Hacker News pour rechercher côté serveur. Sans quoi vous traiterez uniquement la première réponse de l'API que vous obtiendriez via `componentDidMount()` avec en paramètre de terme de recherche celui par défaut.

Vous pouvez définir une méthode `onSearchSubmit()` dans votre composant App qui va chercher les résultats depuis l'API d'Hacker News lors de l'exécution d'une recherche dans le composant Search.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result : null,
      searchTerm : DEFAULT_QUERY,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...

  onSearchSubmit() {
    const { searchTerm } = this.state;
  }

  ...
}
```

La méthode `onSearchSubmit()` doit utiliser la même fonctionnalité que la méthode du cycle de vie `componentDidMount()`, mais cette fois-ci avec un terme de recherche modifiée issue de l'état local et non du terme de recherche par défaut. Ainsi vous pouvez extraire la fonctionnalité en tant que méthode de classe réutilisable.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result : null,
      searchTerm : DEFAULT_QUERY,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...

  fetchSearchTopStories(searchTerm) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(error => error);
  }

  componentDidMount() {
    const { searchTerm } = this.state;
    this.fetchSearchTopStories(searchTerm);
  }

  ...

  onSearchSubmit() {
    const { searchTerm } = this.state;
    this.fetchSearchTopStories(searchTerm);
  }

  ...
}
```

Maintenant le composant de Search doit ajouter un bouton supplémentaire. Le bouton doit déclencher explicitement la requête de recherche. Autrement vous irez constamment chercher les données depuis l'API d'Hacker News lorsque votre champ d'entrée change. À part si vous voulez le faire explicitement dans un handler `onClick()`.

En alternative vous pouvez (retarder) *debounce* la fonction `onChange()` et économiser le bouton, mais cela ajouterait plus de complexité pour l'heure et peut-être ne serait pas l'effet escompté. Restons simple sans un *debounce* pour le moment.

Premièrement, passer la méthode `onSearchSubmit()` pour votre composant Search.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
            onSubmit={this.onSearchSubmit}
          >
            Search
          </Search>
        </div>
        { result &&
          <Table
            list={result.hits}
            pattern={searchTerm}
            onDismiss={this.onDismiss}
          />
        }
      </div>
    );
  }
}
```

Deuxièmement, introduire un bouton dans votre composant Search. Le bouton possède le `type="submit"` et le formulaire utilise son attribut `onSubmit` pour passer la méthode `onSubmit()`. Vous pouvez réutiliser la propriété `children`, mais cette fois elle sera utilisée pour le contenu du bouton.

src/App.js

```
const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) =>
  <form onSubmit={onSubmit}>
    <input
      type="text"
      value={value}
      onChange={onChange}
    />
    <button type="submit">
      {children}
    </button>
  </form>
```

Dans le Table, vous pouvez supprimer la fonctionnalité de filtre, car il ne sera plus filtré côté client (search). N'oubliez pas de supprimer la fonction `isSearched()` également. Elle ne sera plus utilisée. Maintenant, le résultat arrive directement de l'API d'Hacker News après que vous ayez cliqué sur le bouton "Search".

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        ...
        { result &&
          <Table
            list={result.hits}
            onDismiss={this.onDismiss}
          />
        }
      </div>
    );
  }
```

```
}

...

const Table = ({ list, onDismiss }) =>
  <div className="table">
    {list.map(item =>
      ...
    )}
  </div>
```

Lorsque vous essayez de rechercher, vous remarquerez que le navigateur se recharge. C'est un comportement natif du navigateur pour une callback de soumission dans un formulaire HTML. Dans React vous serez souvent amené au travers de la méthode d'évènement `preventDefault()` de supprimer le comportement natif du navigateur.

src/App.js

```
onSearchSubmit(event) {
  const { searchTerm } = this.state;
  this.fetchSearchTopStories(searchTerm);
  event.preventDefault();
}
```

Maintenant vous devez être capable de rechercher différents sujets d'Hacker News. Parfait, vous interagissez avec une une véritable API. Il ne devrait plus avoir de recherche côté client dorénavant.

Exercices :

- lire plus à propos des évènements synthétiques dans React[110]
- expérimenter avec l'API d'Hacker News[111]

110. https://reactjs.org/docs/events.html
111. https://hn.algolia.com/api

Recherche paginée

Avez-vous une meilleure vision de la structure de données retournée ? L'Hacker News API[112] retourne plus qu'une liste de hits. Précisément elle retourne une liste paginée. La propriété page, qui est `0` dans la première réponse, peut être utilisé pour aller chercher encore plus de sous-listes paginées en tant que résultat. Vous avez seulement besoin de transmettre la prochaine page avec le même terme de recherche auprès de l'API.

Étendons les constantes d'API composable ainsi il peut traiter les données paginées.

src/App.js

```
const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';
```

Maintenant vous pouvez utiliser la nouvelle constante pour ajouter la paramètre page pour votre requête d'API.

Code Playground

```
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT_QUERY}&${PARAM_PAGE}`;

console.log(url);
// output : https://hn.algolia.com/api/v1/search?query=redux&page=
```

La méthode `fetchSearchTopStories()` prendra la page en tant que second argument. Si vous ne fournissez pas de second argument, il se repliera sur la page `0` pour la requête initiale. Ainsi les méthodes `componentDidMount()` et `onSearchSubmit()` iront chercher la première page de la première requête. Toutes recherches additionnelles devraient aller chercher la prochaine page en fournissant le deuxième argument.

112. https://hn.algolia.com/api

src/App.js

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(error => error);
  }

  ...

}
```

L'argument de page utilise la valeur par défaut des arguments ES6 pour introduire une solution de repli vers la page `0` au cas où aucun argument de page est fourni à la fonction.

Maintenant vous pouvez utiliser la page courante de la réponse d'API dans `fetchSearchTopStories()`. Vous pouvez utiliser cette méthode dans un bouton pour aller chercher plus de sujets au handler `onClick` du bouton. Utilisons le Button pour aller chercher plus de données paginées depuis l'API d'Hacker News. Vous aurez seulement besoin de définir l'handler `onClick()` qui prend le terme de recherche courant et la prochaine page (current page + 1).

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    const page = (result && result.page) || 0;
    return (
      <div className="page">
        <div className="interactions">
        ...
        { result &&
          <Table
            list={result.hits}
            onDismiss={this.onDismiss}
          />
```

```
        }
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopStories(searchTerm, page + 1)}>
            More
          </Button>
        </div>
      </div>
    );
  }
}
```

En plus, dans votre méthode `render()` vous devrez vous assurer par défaut d'aller chercher à la page `0` lorsqu'il n'y a pas encore de résultat. Souvenez-vous de cela la méthode `render()` est appelée avant que les données sont rapportées de façon asynchrone dans la méthode du cycle de vie `componentDidMount()`.

Il manque une étape. Vous allez chercher la prochaine page de données, mais il écartera votre précédente page de données. Il serait idéal de concaténer l'ancienne avec la nouvelle liste de hits au sein de l'état local et du nouvel objet résultat. Ajustons la fonctionnalité pour ajouter les nouvelles données plutôt que de les écraser.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  const oldHits = page !== 0
    ? this.state.result.hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    result : { hits : updatedHits, page }
  });
}
```

Maintenant plusieurs choses se produisent dans la méthode `setSearchTopStories()`. Premièrement, vous obtenez les hits et page depuis le résultat.

Deuxièmement, vous avez vérifié s'il y a déjà des vieux hits. Quand la page est à `0`, c'est une nouvelle requête de recherche issue de `componentDidMount()` ou `onSearchSubmit()`. Les hits sont vides. Mais

quand vous cliquez sur le bouton “More” pour aller chercher les données paginées la page n’est plus 0. C’est une nouvelle page. Les anciens hits sont déjà stockés dans votre état et peuvent ainsi être utilisés.

Troisièmement, vous ne voulez pas écarter les anciens hits. Vous pouvez fusionner les anciens et nouveaux hits issus de la dernière requête API. La fusion des deux listes peut être effectuée avec le *spead operator* de JavaScript ES6.

Quatrièmement, vous écrivez les hits fusionnés et la page courante dans l’état local du composant.

Vous pouvez faire un dernier ajustement. Quand vous essayez le bouton “More” il va seulement chercher quelques objets de la liste. L’URL de l’API peut être étendu pour aller chercher davantage d’objets de la liste avec chaque requête. De nouveau, vous pouvez ajouter plus de constantes d’URL composable.

src/App.js

```
const DEFAULT_QUERY = 'redux';
const DEFAULT_HPP = '100';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';
const PARAM_HPP = 'hitsPerPage=';
```

Now you can use the constants to extend the API URL.

src/App.js

```
fetchSearchTopStories(searchTerm, page = 0) {
  fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}\
}&${PARAM_HPP}${DEFAULT_HPP}`)
    .then(response => response.json())
    .then(result => this.setSearchTopStories(result))
    .catch(error => error);
}
```

Après çà, la requête auprès de l’API d’Hacker News va chercher plus d’éléments de la liste que précédemment en une seule requête. Comme vous pouvez le voir, une API puissante telle que l’API d’Hacker News vous donne plein de façons pour expérimenter des données du monde réel. Vous devez vous efforcer de faire usage de cela lors de l’apprentissage de quelque chose de nouveau et de plus excitant. Ici comment j’ai appris le pouvoir que les APIs procurent[113] lors de l’apprentissage d’un nouveau langage de programmation ou d’une bibliothèque.

113. https://www.robinwieruch.de/what-is-an-api-javascript/

Exercices :

- lire plus à propos des valeurs par défaut des arguments ES6[114]
- expérimenter avec les paramètres de l'API d'Hacker News[115]

114. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Default_parameters
115. https://hn.algolia.com/api

Cache client

Chaque recherche soumise fait une requête à l'API d'Hacker News. Vous pourriez rechercher "redux", suivi par "react" et finalement de nouveau "redux". Au total cela fait 3 requêtes. Mais vous recherchez "redux" deux fois et les deux fois prennent un aller-retour asynchrone complet pour aller chercher les données. Dans un cache côté client vous souhaiterez stocker chaque résultat. Quand une requête auprès de l'API est effectuée, il vérifie si le résultat est déjà présent. S'il l'est, le cache est utilisé. Autrement la requête API est faite pour aller chercher les données.

Dans le but d'obtenir un cache client pour chaque requête, vous avez à stocker plusieurs `results` plutôt qu'un seul `result` dans votre état interne du composant. L'objet `results` sera une *map* avec le terme de recherche en tant que clé et le résultat en tant que valeur. Chaque résultat depuis l'API sera sauvegardé par terme de recherche (key).

Pour le moment, votre résultat dans l'état local ressemble au suivant :

Code Playground

```
result : {
  hits : [ ... ],
  page : 2,
}
```

Imaginez vous devoir faire deux requêtes API. Une pour le terme "redux" et une autre pour "react". Les objets résultats devront ressembler au suivant :

Code Playground

```
results : {
  redux : {
    hits : [ ... ],
    page : 2,
  },
  react : {
    hits : [ ... ],
    page : 1,
  },
  ...
}
```

Implémentons le cache côté client avec React `setState()`. Premièrement, renommer l'objet `result` pour `results` dans l'état initial du composant. Deuxièmement, définissez un `searchKey` temporaire qui est utilisé pour stocker chaque `result`.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results : null,
      searchKey : '',
      searchTerm : DEFAULT_QUERY,
    };

    ...

  }

  ...

}
```

La `searchKey` doit être établie avant que chaque requête soit effectuée. Elle reflète le `searchTerm`. Vous pouvez vous demander : Pourquoi ne pas utiliser le `searchTerm` en premier lieu ? C'est un point crucial à comprendre avant de continuer avec l'implémentation. Le `searchTerm` est une variable changeante, car il sera changé chaque fois vous tapez dans le champ d'entrée de Search. Cependant, à la fin vous aurez besoin de variable stable. Cela détermine le terme de recherche récemment soumis auprès de l'API et il peut être utilisé pour retrouver le résultat correct depuis la *map* de results. C'est un pointeur vers notre résultat courant dans le cache et ainsi il peut être utilisé pour afficher le résultat courant dans votre méthode `render()`.

src/App.js

```
componentDidMount() {
  const { searchTerm } = this.state;
  this.setState({ searchKey : searchTerm });
  this.fetchSearchTopStories(searchTerm);
}

onSearchSubmit(event) {
  const { searchTerm } = this.state;
  this.setState({ searchKey : searchTerm });
  this.fetchSearchTopStories(searchTerm);
  event.preventDefault();
}
```

Maintenant vous devez ajuster la fonctionnalité où le résultat est stocké dans l'état interne du composant. Il devrait stocker chaque résultat par `searchKey`.

src/App.js

```
class App extends Component {

  ...

  setSearchTopStories(result) {
    const { hits, page } = result;
    const { searchKey, results } = this.state;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    this.setState({
      results : {
        ...results,
        [searchKey]: { hits : updatedHits, page }
      }
    });
  }

  ...

}
```

La `searchKey` sera utilisée en tant que clé pour sauvegarder les hits mis à jour et la page dans la *map* `results`.

Premièrement, vous devez retrouver la `searchKey` depuis l'état du composant. Souvenez-vous que la `searchKey` sera établie au `componentDidMount()` et au `onSearchSubmit()`.

Deuxièmement, comme avant les anciens hits doivent être fusionnés avec les nouveaux hits. Mais cette fois les anciens hits sont récupérés depuis la *map* `results` avec la `searchKey` en tant que clé.

Troisièmement, un nouveau résultat peut être écrit dans la *map* `results` dans l'état. Examinons l'objet `results` dans `setState()`.

src/App.js

```
results : {
  ...results,
  [searchKey] : { hits : updatedHits, page }
}
```

La dernière partie fait en sorte de stocker le résultat mis à jour par `searchKey` dans la *map* `results`. La valeur est un objet avec des hits et une propriété page. La `searchKey` est le terme de recherche. Vous avez déjà appris la syntaxe `[searchKey]: ...`. C'est un nom de propriété ES6 calculé. Cela vous aide à allouer des valeurs dynamiquement dans un objet.

La partie du haut a besoin de propager tous les autres résultats via `searchKey` dans l'état en utilisant l'*object spread operator*. Autrement vous perdrez tous les résultats que vous avec avez stockés précédemment.

Maintenant vous stockez tous les résultats par termes de recherche. C'est la première étape pour rendre disponible votre cache. Dans la prochaine étape, vous pouvez retrouver le résultat en fonction de la `searchKey` non changeante depuis votre *map* de `results`. C'est pourquoi vous devez introduire la `searchKey` en premier lieu comme variable non changeante. Autrement la récupération pourrait être rompue lorsque vous souhaiterez utiliser le `searchTerm` changeant pour retrouver le résultat courant, car cette valeur pourrait changer lorsque vous souhaiterez utiliser le composant Search.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
```

```
      results[searchKey].hits
    ) || [];

    return (
      <div className="page">
        <div className="interactions">
          ...
        </div>
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}>
            More
          </Button>
        </div>
      </div>
    );
  }
}
```

Puisque vous retournez par défaut une liste vide quand il n'y a pas de résultat via `searchKey`, vous pouvez maintenant économiser le rendu conditionnel pour le composant Table. De plus vous aurez besoin de passer la `searchKey` plutôt que le `searchTerm` auprès du bouton "More". Sinon, votre recherche paginée dépend de la valeur `searchTerm` qui est changeante. De plus soyez sûr de conserver la propriété `searchTerm` changeante pour le champ d'entrée du composant "Search".

La fonctionnalité de recherche doit fonctionner de nouveau. Il stocke tous les résultats depuis l'API Hacker News.

De plus, la méthode `onDismiss()` a besoin d'être améliorée. Elle traite encore l'objet `result`. Maintenant elle doit traiter les multiples `results`.

src/App.js

```
  onDismiss(id) {
    const { searchKey, results } = this.state;
    const { hits, page } = results[searchKey];

    const isNotId = item => item.objectID !== id;
    const updatedHits = hits.filter(isNotId);

    this.setState({
      results : {
```

```
        ...results,
        [searchKey]: { hits : updatedHits, page }
      }
    });
  }
```

Le bouton “Dismiss” doit fonctionner de nouveau.

Cependant, rien n’arrête l’application d’envoyer une requête API à chaque soumission de requête. Même s’il y aurait déjà un résultat, il n’y a aucun contrôle pour empêcher la requête. Ainsi la fonctionnalité de cache n’est pas encore achevée. Elle cache les résultats, mais elle ne les utilise pas. La dernière étape serait d’empêcher la requête d’API lorsqu’un résultat est disponible en cache.

src/App.js

```
class App extends Component {

  constructor(props) {

    ...

    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  needsToSearchTopStories(searchTerm) {
    return !this.state.results[searchTerm];
  }

  ...

  onSearchSubmit(event) {
    const { searchTerm } = this.state;
    this.setState({ searchKey : searchTerm });

    if (this.needsToSearchTopStories(searchTerm)) {
      this.fetchSearchTopStories(searchTerm);
    }

    event.preventDefault();
```

```
    }

    . . .

}
```

Maintenant votre client fait une requête auprès de l'API seulement une seule fois même si vous recherchez un terme de recherche une seconde fois. Même les données paginées avec plusieurs pages seront cachées de cette façon, car vous sauvegardez toujours la dernière page pour chaque résultat dans la *map* `results`. N'est-ce pas une approche percutante pour introduire un cache pour votre application ? L'API d'Hacker News vous fournit tout ce dont vous avez besoin pour mettre en cache les données paginées de façon effective.

Gestion des erreurs

Tout est place concernant vos interactions avec l'API d'Hacker News. Vous avez même introduit une façon élégante de mettre en cache vos résultats depuis l'API et d'employer sa fonctionnalité de liste paginée pour aller chercher une liste sans fin de sous-listes d'articles. Mais il y a une pièce manquante. Malheureusement, c'est souvent oublié lors de développements d'applications de nos jours : la gestion des erreurs. C'est tellement facile d'implémenter le cas avec succès sans se soucier des erreurs pouvant se produire tout du long.

Dans ce chapitre, vous introduirez une solution efficace pour ajouter une gestion des erreurs pour votre application en cas de requête API erronée. Vous avez déjà pris connaissance de la nécessité des *building blocks* en React pour introduire la gestion d'erreur : l'état local et le rendu conditionnel. Globalement, l'erreur est seulement un autre état dans React. Lorsqu'une erreur se produit, vous le stockerez dans l'état local et l'afficherez avec un rendu conditionnel au sein de votre composant. C'est tout. Implémentons cela dans le composant App, car c'est le composant qui est utilisé en premier lieu pour aller chercher les données depuis l'API d'Hacker News. Tout d'abord, vous devez introduire l'erreur dans l'état local. C'est inutilisé à null, mais défini avec un *error object* en cas d'erreur.

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      results : null,
      searchKey : '',
      searchTerm : DEFAULT_QUERY,
      error : null,
    };

    ...
  }

...

}
```

Deuxièmement, vous pouvez utiliser le *catch block* dans votre *fetch* natif pour stocker l'*error object* dans l'état local en utilisant `setState()`. À chaque fois, que la requête d'API n'est pas un succès, le *catch block* sera exécuté.

src/App.js

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(error => this.setState({ error }));
  }

  ...

}
```

Troisièmement, vous pouvez retrouver l'*error object* depuis votre état local dans la méthode `render()` et afficher un message en cas d'erreur en utilisant le rendu conditionnel de React.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error
    } = this.state;

    ...

    if (error) {
      return <p>Something went wrong.</p>;
    }

    return (
      <div className="page">
        ...
```

```
      </div>
    );
  }
}
```

C'est tout. Si vous voulez tester que votre gestion d'erreur fonctionne, vous pouvez substituer l'URL de l'API par quelque chose d'inexistant.

src/App.js

```
const PATH_BASE = 'https ://hn.foo.bar.com/api/v1';
```

Après quoi, vous devrez obtenir le message d'erreur au lieu de votre application. Vous pouvez choisir où vous souhaitez placer le rendu conditionnel pour votre message d'erreur. Dans le cas présent, l'entièreté de l'application ne sera plus du tout affichée. Ce n'est surement pas la meilleure expérience utilisateur. Donc pourquoi ne pas afficher soit le composant Table ou le message d'erreur ? Le reste de l'application sera encore visible en cas d'erreur.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
    ) || [];
```

```
    return (
      <div className="page">
        <div className="interactions">
          ...
        </div>
        { error
         ? <div className="interactions">
            <p>Something went wrong.</p>
          </div>
          : <Table
            list={list}
            onDismiss={this.onDismiss}
          />
        }
        ...
      </div>
    );
  }
}
```

À la fin, n'oubliez pas de remettre l'URL d'API existante.

src/App.js

```
const PATH_BASE = 'https ://hn.algolia.com/api/v1';
```

Votre application devrait encore fonctionner, mais cette fois avec une gestion d'erreur en cas de requête d'API qui échoue.

Exercices :

- lire plus à propos de la gestion des erreurs pour les composants React[116]

116. https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html

Axios à la place de Fetch

Dans l'un des précédents chapitres, nous avons introduit l'API *fetch* native pour performer une requête auprès de la plateforme d'Hacker News. Le navigateur vous permet d'utiliser cette API *fetch* native. Cependant, pas tous les navigateurs la supportent, particulièrement les plus anciens navigateurs. De plus, une fois que vous commencer à tester votre application au sein d'un environnement de navigateur sans tête (*headless browser*) (il n'y a pas de navigateur, à la place il est seulement simulé), il peut avoir des soucis concernant l'API *fetch*. Un tel environnement *headless browser* peut se produire lors de l'écriture et de l'exécution des tests pour votre application qui ne tourne pas dans un véritable navigateur. Il y a différentes manières pour faire fonctionner le *fetch* au sein des plus anciens navigateurs (polyfills) et dans les tests (isomorphic-fetch[117]), mais dans ce livre nous ne nous aventurerons dans cette voie.

Une manière alternative pour résoudre cela serait de substituer l'API *fetch* native avec une bibliothèque stable telle qu'axios[118]. Axios est une bibliothèque qui résout seulement un problème, mais elle le résout très bien : performer des requêtes asynchrones pour des APIs distantes. C'est pourquoi vous l'utiliserez dans ce livre. Plus concrètement, le chapitre devrait vous montrer comment substituer une bibliothèque (qui est une API native du navigateur dans ce cas). De façon abstraite, le chapitre devrait vous montrer comment vous pouvez toujours trouver une solution pour les bizarreries (ex : anciens navigateurs, tests avec *headless browser*) du développement web. Donc n'arrêtez pas de chercher des solutions peu importe les embuches.

Regardons comment l'API *fetch* native peut être substitué avec *axios*. A vrai dire tout ce qui a été dit précédemment semble plus compliqué que ce qui l'en est. Premièrement, vous devez installer *axios* avec la ligne de commande :

Command Line

```
npm install --save axios
```

Deuxièmement, vous pouvez importer *axios* dans votre fichier du composant App :

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import './App.css';

...
```

Et enfin mais pas des moindres, vous pouver l'utiliser à la place de `fetch()`. Son utilisation apparaît pratiquement identique à l'API *fetch*. Elle prend l'URL en tant qu'argument et retourne une *promise*.

117. https://github.com/matthew-andrews/isomorphic-fetch
118. https://github.com/axios/axios

Vous n'êtes pas non plus obligé de transformer la réponse retournée vers du JSON. Axios le fait pour vous et englobe le résultat dans un objet `data` en JavaScript. Ainsi assurez-vous d'adapter votre code à la structure de données retournées.

src/App.js

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    axios(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(result => this.setSearchTopStories(result.data))
      .catch(error => this.setState({ error }));
  }

  ...

}
```

C'est tout concernant le remplacement de *fetch* avec *axios* dans ce chapitre. Dans votre code, vous appelez `axios()` qui utilise par défaut une requête HTTP GET. Vous pouvez faire la requête HTTP GET explicite en appelant `axios.get()`. Aussi vous pouvez à la place utiliser une autre méthode HTTP tel que le POST HTTP avec `axios.post()`. Ici vous pouvez déjà apprécier à quel point *axios* est une bibliothèque puissante pour performer des requêtes auprès d'APIs distantes. Je recommande souvent de l'utiliser à la place de l'API *fetch* lorsque vos requêtes d'API deviennent complexes ou que vous devez traiter les bizarreries du développement web avec les *promises*. De plus, dans un prochain chapitre, vous introduirez les tests dans votre application. Alors vous n'aurez plus besoin de vous inquiéter à propos d'un navigateur ou d'un environnement d'*headless browser*.

Je souhaite introduire une autre amélioration pour la requête d'Hacker News au sein du composant App. Imaginez votre composant se monte lorsque la page est rendue pour la première fois dans le navigateur. Dans `componentDidMount()` le composant débute la requête, mais ensuite, comme votre application introduit une certaine navigation, vous naviguez en dehors de cette page vers une autre page. Votre composant App se démonte, mais il y a toujours une requête en *pending* issue de votre méthode du cycle de vie `componentDidMount()`. Il tentera d'utiliser `this.setState()` au final dans le `then()` ou le `catch()` du bloc de la *promise*. Peut-être que c'est la première fois que vous verrez le warning suivant sur votre ligne de commande ou sur votre console développeur du navigateur :

Command Line

```
Warning : Can only update a mounted or mounting component. This usually means you cal\
led setState, replaceState, or forceUpdate on an unmounted component. This is a no-o\
p.
```

Vous pouvez traiter ce problème en interrompant la requête lorsque votre composant se démonte ou empêcher l'appel `this.setState()` sur un composant démonté. C'est une bonne pratique dans React, même si elle n'est pas suivi par de très nombreux développeurs, dans le but de conserver une application propre sans warnings dérangeants. Cependant, l'API *promise* courante n'implémente pas le fait d'interrompre une requête. Ainsi vous avez besoin de vous débrouiller par vous-même sur ce problème. Cela devrait aussi être ce qui fait que pas beaucoup de développeurs suivent cette bonne pratique. L'implémentation suivante semble plus une solution de contournement qu'une implémentation durable. À cause de cela, vous pouvez décider par vous-même si vous souhaitez l'implémenter pour contourner le warning dû à un composant démonté. Cependant, conserver le warning en mémoire dans le cas où il se produit dans un prochain chapitre de ce livre ou un jour dans votre propre application. Alors vous saurez comment le traiter.

Commençons à contourner cela. Vous pouvez introduire une propriété qui tient maintient l'état du cycle de votre composant. Elle peut être initialisée à `false` lorsque le composant s'initialise, changé à `true` lorsque le composant est monté, mais ensuite de nouveau établit à `false` lorsque le composant est démonté. De cette façon, vous pouvez suivre l'état du cycle de vie de votre composant. Il n'a aucun rapport avec l'état local stocké et modifié avec `this.state` et `this.setState()`, car vous devez être en mesure d'y accéder directement avec l'instance du composant sans compter sur la gestion de l'état local de React. De plus, il ne gère aucun rerendu du composant lorsque la propriété est modifiée de cette façon.

src/App.js

```
class App extends Component {
  _isMounted = false;

  constructor(props) {
    ...
  }

  ...

  componentDidMount() {
    this._isMounted = true;

    const { searchTerm } = this.state;
    this.setState({ searchKey : searchTerm });
    this.fetchSearchTopStories(searchTerm);
  }
```

```
  componentWillUnmount() {
    this._isMounted = false;
  }

  ...

}
```

Finalement, vous pouvez utiliser vos connaissances non pour interrompre la requête elle-même mais pour éviter le fait d'appeler `this.setState()` sur votre instance de composant même si le composant est déjà démonté. Cela empêchera le warning mentionné précédemment.

src/App.js

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    axios(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(result => this._isMounted && this.setSearchTopStories(result.data))
      .catch(error => this._isMounted && this.setState({ error }));
  }

  ...

}
```

De plus le chapitre vous a montré comment vous pouvez remplacer une bibliothèque par une autre bibliothèque en React. Si vous rencontrez n'importent quels problèmes, vous pouvez utiliser le vaste écosystème de bibliothèque JavaScript pour vous aider. De plus, vous avez vu une façon de comment vous pouvez éviter le fait d'appeler `this.setState()` dans React sur un composant démonté. Si vous creusez plus profondément dans la bibliothèque *axios*, vous trouverez une façon d'annuler la requête. C'est à vous de décider de faire plus de recherche sur ce sujet.

Exercices :

- lire plus à propos de pourquoi les frameworks sont importants[119]
- lire plus à propos d'une syntaxe alternative de composant React[120]

119. https://www.robinwieruch.de/why-frameworks-matter/
120. https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax

Vous avez appris à interagir avec l'API React ! Récapitulons les derniers chapitres :

- React
  - Les méthodes du cycle de vie du composant de classe ES6 pour différent cas d'utilisation
  - componentDidMount() pour les interactions d'API
  - rendus conditionnels
  - évènement synthétique concernant les formulaires
  - gestion d'erreur
  - interrompre une requête d'API distante
- ES6 et au-delà
  - template strings pour composer de chaines de caractères
  - spread operator pour des structures de données immutables.
  - noms de prorpiétés calculés
- Général
  - interaction avec l'API d'Hacker News
  - API native du navigateur *fetch*
  - recherche côté client et côté serveur
  - pagination des données
  - cache côté client

Encore une fois il y a un intérêt à faire une pause. Intérioriser les acquis et les appliquer par vous-même. Vous pouvez expérimenter avec le code source que vous avez écrit jusqu'ici. Vous pouvez trouver le code source dans le dépôt officiel[121].

121. https://github.com/the-road-to-learn-react/hackernews-client/tree/5.3.1

Organisation du code et tests

Ce chapitre se concentrera sur des sujets importants pour conserver un code maintenable dans une application scalable. Vous apprendrez à organiser le code adopter les meilleurs pratiques lors de la structuration de vos dossiers et fichiers. Un autre aspect vous apprendrez à tester, ce qui est primordial pour conserver un code robuste. Enfin, vous allez apprendre à utiliser un outil très utile pour le deboggage de vos applications React. La majeure partie du chapitre prendra du recul sur l'aspect pratique de l'application et expliquera quelques-uns de ces sujets pour vous.

ES6 Modules : Import et Export

En JavaScript ES6 vous pouvez importer et exporter les fonctionnalités de modules. Ces fonctionnalités peuvent être des fonctions, classes, composants, constantes et autres. Globalement tous ce que vous pouvez assigner à une variable. Les modules peuvent être de simples fichiers ou des dossiers complets avec un fichier d'index en tant que point d'entrée.

Au début du livre, après que vous ayez démarré votre application avec *create-react-app*, vous avez déjà plusieurs déclarations d'`import` et d'`export` au sein de vos fichiers initiaux. Maintenant il est temps de les expliquer.

Les déclarations d'`import` et d'`export` vous aident à partager le code au travers de multiples fichiers. Avant il y avait déjà plusieurs solutions pour çà au sein de l'environnement JavaScript. C'était un bazar, car vous souhaitez suivre une façon standardisée plutôt que d'avoir plusieurs approches pour la même chose. Maintenant c'est un comportement natif depuis JavaScript ES6.

De plus ces déclarations adhèrent à la séparation de code. Vous distribuez votre code au travers de multiples fichiers pour le garder réutilisable et maintenable. Réutilisable car vous pouvez importer un morceau de code dans plusieurs fichiers. Maintenable car vous avez qu'une seule source où vous maintenez le morceau de code.

Enfin mais pas des moindres, il vous aide à penser en matière d'encapsulation de code. Pas toutes les fonctionnalités ont la nécessité d'être exportées depuis un fichier. Certaines de ces fonctionnalités devrait seulement être utilisé dans le fichier où ils ont été définies. Les exports d'un fichier sont simplement l'API publique du fichier. Seules les fonctionnalités exportées sont disponibles pour être réutilisé ailleurs. Cela suit la meilleure pratique de l'encapsulation.

Mais mettons-nous à la pratique. Comment ces déclarations d'`import` et d'`export` fonctionnent-ils ? Les exemples suivants présentent les déclarations en partageant une ou plusieurs variables au travers de deux fichiers. À la fin, l'approche peut être mise à l'échelle pour de multiples fichiers et peut partager plus que de simples variables.

Vous pouvez exporter une ou plusieurs variables. C'est appelé un export nommé (named export).

Code Playground : file1.js

```
const firstname = 'robin';
const lastname = 'wieruch';

export { firstname, lastname };
```

Et les importer dans un autre fichier avec un chemin relatif au premier fichier

Code Playground : file2.js

```
import { firstname, lastname } from './file1.js';

console.log(firstname);
// output : robin
```

Vous pouvez aussi importer toutes les variables exporter depuis un autre fichier comme étant un seul objet.

Code Playground : file2.js

```
import * as person from './file1.js';

console.log(person.firstname);
// output : robin
```

Les imports peuvent avoir un alias. Cela peut arriver que vous importez des fonctionnalités de divers fichiers et qu'ils ont le même nom d'export. C'est pour cela que vous pouvez utiliser un alias.

Code Playground : file2.js

```
import { firstname as username } from './file1.js';

console.log(username);
// output : robin
```

Dernier point, mais pas des moindres il existe la déclaration `default`. Elle peut être utilisée pour plusieurs cas d'utilisation :

- exporter et importer une seule fonctionnalité
- mettre en exergue la fonctionnalité principale de l'API exporté d'un module
- avoir une fonctionnalité d'import de secours

Code Playground : file1.js

```
const robin = {
  firstname : 'robin',
  lastname : 'wieruch',
};

export default robin;
```

Vous pouvez omettre les accolades pour l'import afin d'importer le default export.

Code Playground : file2.js

```
import developer from './file1.js';

console.log(developer);
// output : { firstname : 'robin', lastname : 'wieruch' }
```

De plus, le nom d'import diffère du nom par défaut exporté. Vous pouvez aussi l'utiliser conjointement avec les exports nommés et les déclarations d'imports.

Code Playground : file1.js

```
const firstname = 'robin';
const lastname = 'wieruch';

const person = {
  firstname,
  lastname,
};

export {
  firstname,
  lastname,
};

export default person;
```

Et importer l'export nommé ou par défaut dans un autre fichier.

Code Playground : file2.js

```
import developer, { firstname, lastname } from './file1.js';

console.log(developer);
// output : { firstname : 'robin', lastname : 'wieruch' }
console.log(firstname, lastname);
// output : robin wieruch
```

Vous pouvez aussi économiser des lignes additionnelles et exporter directement les variables en export nommé.

Code Playground : file1.js

```
export const firstname = 'robin';
export const lastname = 'wieruch';
```

Ce sont les principales fonctionnalités des modules ES6. Ils vous aide à organiser votre code, maintenir votre code et concevoir des APIs de module réutilisable. Vous pouvez aussi exporter et importer des fonctionnalités pour les tester. Vous ferez cela dans l'un des chapitres suivants.

Exercices :

- lire plus à propos ES6 import[122]
- lire plus à propos ES6 export[123]

122. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import
123. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

Organisation du code avec les modules ES6

Vous pourriez vous demander : pourquoi nous n'avons pas suivi les meilleurs pratiques pour séparer notre code pour le fichier *src/App.js* ? Dans le fichier nous avons déjà plusieurs composants qui pourrait être défini dans leurs propres (modules) fichiers/dossiers. Pour l'intérêt d'apprentissage de React, c'est pratique de conserver ces choses à un seul endroit. Mais une fois que votre application React grossit, vous devez considérer le fait de séparer ces composants à l'intérieur de plusieurs modules. La seule façon pour vos applications d'être mis à l'échelle.

Par conséquent, je proposerai plusieurs structures de module que vous "pourriez" appliquer. Je vous recommande de les appliquer en tant qu'exercice à la fin du livre. Pour conserver le livre simple, j'effectuerai pas la séparation de code et continuerai les chapitres suivants avec le fichier *src/App.js*.

Une structure de module possible serait :

Folder Structure

```
src/
  index.js
  index.css
  App.js
  App.test.js
  App.css
  Button.js
  Button.test.js
  Button.css
  Table.js
  Table.test.js
  Table.css
  Search.js
  Search.test.js
  Search.css
```

Cela sépare les composants à l'intérieur de leurs propres fichiers, mais cela ne semble pas très prometteur. Vous pouvez voir beaucoup de duplication de nom et seules les extensions de fichier diffèrent. Une autre structure de module serait :

Folder Structure

```
src/
  index.js
  index.css
  App/
    index.js
    test.js
    index.css
  Button/
    index.js
    test.js
    index.css
  Table/
    index.js
    test.js
    index.css
  Search/
    index.js
    test.js
    index.css
```

Cela semble plus propre qu'avant. La dénomination *index* d'un fichier le décrit comme étant le point d'entrée du dossier. C'est juste une convention de nommage commune, mais vous pouvez utiliser votre propre nommage également. Dans cette structure de module, un composant est défini par sa déclaration de composant dans le fichier JavaScript, mais aussi par son style et ses tests.

Une autre étape pourrait être l'extraction des variables de constantes du composant App. Ces constantes étaient utilisées pour composer l'URL de l'API d'Hacker News.

Folder Structure

```
src/
  index.js
  index.css
  constants/
    index.js
  components/
    App/
      index.js
      test.js
      index.css
    Button/
      index.js
      test.js
```

```
      index.css
    // ...
```

Naturellement les modules devront être séparés à l'intérieur de *src/constants/* et *src/components/*. Maintenant le fichier *src/constants/index.js* pourrait ressembler à cela :

Code Playground : src/constants/index.js

```
export const DEFAULT_QUERY = 'redux';
export const DEFAULT_HPP = '100';
export const PATH_BASE = 'https://hn.algolia.com/api/v1';
export const PATH_SEARCH = '/search';
export const PARAM_SEARCH = 'query=';
export const PARAM_PAGE = 'page=';
export const PARAM_HPP = 'hitsPerPage=';
```

Le fichier *App/index.js* peut importer ces variables pour les utiliser.

Code Playground : src/components/App/index.js

```
import {
  DEFAULT_QUERY,
  DEFAULT_HPP,
  PATH_BASE,
  PATH_SEARCH,
  PARAM_SEARCH,
  PARAM_PAGE,
  PARAM_HPP,
} from '../../constants/index.js';

// ...
```

Lorsque vous utilisez la convention de nommage *index.js*, vous pouvez omettre le nom du fichier du chemin relatif.

Code Playground : src/components/App/index.js

```
import {
  DEFAULT_QUERY,
  DEFAULT_HPP,
  PATH_BASE,
  PATH_SEARCH,
  PARAM_SEARCH,
  PARAM_PAGE,
  PARAM_HPP,
} from '../../constants';

// ...
```

Mais qu'est-ce qui est derrière le nommage de fichier *index.js* ? La convention a été introduite dans le monde de node.js. Le fichier index est un point d'entrée auprès du module. Cela décrit l'API publique envers le module. Considérez la stucture de module suivante construite pour la démonstration :

Folder Structure

```
src/
  index.js
  App/
    index.js
  Buttons/
    index.js
    SubmitButton.js
    SaveButton.js
    CancelButton.js
```

Le dossier *Buttons/* a plusieurs composants boutons définis dans ses fichiers distincts. Chaque fichier peut `export default` un composant spécifique le rendant disponible pour *Buttons/index.js*. Le fichier *Buttons/index.js* importe tous les différentes représentation de bouton et les exporte en tant qu'API de module publique.

Code Playground : src/Buttons/index.js

```
import SubmitButton from './SubmitButton';
import SaveButton from './SaveButton';
import CancelButton from './CancelButton';

export {
  SubmitButton,
  SaveButton,
  CancelButton,
};
```

Maintenant *src/App/index.js* peut importer les boutons à partir de l'API de module public localisé dans le fichier *index.js*.

Code Playground : src/App/index.js

```
import {
  SubmitButton,
  SaveButton,
  CancelButton
} from '../Buttons';
```

En suivant cette contrainte, il serait malvenu d'atteindre d'autres fichiers que l'*index.js* dans le module. Cela casserait les règles d'encapsulation.

Code Playground : src/App/index.js

```
// Mauvaise pratique, ne faites pas çà
import SubmitButton from '../Buttons/SubmitButton';
```

Maintenant vous savez comment vous pouvez refactorer votre code source dans des modules avec les contraintes d'encapsulation. Comme je l'ai dit, dans l'intérêt de garder le livre simple je n'appliquerai pas ces changements. mais vous devrez faire le refactoring lorsque vous avez terminé la lecture du livre.

Exercices :

- refactorer votre fichier *src/App.js* en plusieurs modules de composant lorsque vous avez terminé le livre

Tests instantanés avec Jest

Le livre ne plongera pas profondément à l'intérieur du sujet des tests, mais il ne doit pas être omis. Tester votre code en programmation est essentiel et doit être perçu comme obligatoire. Vous souhaitez conserver une bonne qualité de votre code et une assurance que tout fonctionne.

Peut-être avez-vous entendu la pyramide de tests. Il y a les tests end-to-end, les tests d'intégration et les tests unitaires. Si vous n'êtes pas familier avec cela, le livre donne un rapide et basique aperçu. Un test unitaire est utilisé pour tester un bloc de code isolé et petit. Cela peut être une fonction seule qui est testée par un test unitaire. Cependant, parfois les tests unitaires fonctionnent en l'isolant mais ne fonctionnent plus en les combinant avec d'autres tests unitaires. Ils ont besoin d'être testés en tant que groupe de tests unitaires. C'est là où les tests d'intégration peuvent aider en couvrant le fait que les tests unitaires fonctionnent ensemble. Enfin, mais pas des moindres, un test end-to-end est la simulation d'un scénario utilisateur réel. Cela peut être une installation automatisée au sein du navigateur simulant les étapes d'authentification d'un utilisateur au sein d'une application web. Tandis que les tests unitaires sont rapides et faciles à écrire et maintenir, les tests end-to-end sont le total opposé.

Combien de tests de chaque type ai-je besoin ? Vous souhaitez avoir plusieurs tests unitaires pour couvrir vos fonctions isolées. Après quoi, vous pouvez avoir plusieurs tests d'intégration pour couvrir le fait que les fonctions les plus importantes fonctionnent de façons combinées comme attendues. Enfin, mais pas des moindres vous pourrez vouloir avoir seulement quelques tests end-to-end pour simuler les scénarios critiques de votre application web. C'est tout pour l'introdution générale au sein du monde des tests.

Donc comment appliquer ces connaissances de tests pour votre application React ? La base pour les tests au sein de React est les tests de composant qui peut être vulgarisé en tant que tests unitaires et une certaine partie en tant que tests instantanés. Vous mènerez les tests unitaires pour vos composants dans le prochain chapitre en utilisant une bibliothèque appelée Enzyme. Dans cette section du chapitre, vous vous concentrerez sur un autre type de tests : les tests d'instantanés. C'est là que Jest rentre en jeu.

Jest[124] est un framework JavaScript de tests qui est utilisé par Facebook. Dans la communauté React, c'est utilisé pour les tests de composant React. Heureusement *create-react-app* possède déjà Jest, donc vous n'avez pas besoin de vous préoccuper de son installation.

Démarrons le test de vos premiers composants. Avant de pouvoir faire cela, vous devez exporter les composants, que vous allez tester, depuis votre fichier *src/App.js*. Après quoi vous pouvez les tester dans un fichier différent. Vous avez appris cela dans le chapitre portant sur l'organisation du code.

124. https://jestjs.io/

src/App.js

```
// ...

class App extends Component {
  // ...
}

// ...

export default App;

export {
  Button,
  Search,
  Table,
};
```

Dans votre fichier *App.test.js*, vous trouverez un premier test qui a été constuit avec create-react-app*. Il vérifie que le composant App fera un rendu sans erreur.

src/App.test.js

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

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
  ReactDOM.unmountComponentAtNode(div);
});
```

Le block “it” décrit un cas de test. Il arrive avec une description de test et lorsque vous le testez, il peut soit réussir soit échouer. De plus, vous pouvez l’englober à l’intérieur d’un bloc “describe” qui définit votre suite de tests. Une suite de tests peut inclure un certain nombre de blocs “it” pour un composant spécifique. Vous verrez ces blocs “describe” après. Les deux blocs sont utilisés pour séparer et organiser vos cas de tests.

Notez que la fonction `it` est connu dans la communauté JavaScript comme la fonction où vous lancez un test unique. Cependant, dans Jest c’est souvent perçu comme un alias de fonction `test`.

Vous pouvez lancer vos cas de tests en utilisant le script de test interactif de *create-react-app* en ligne de commande. Vous obtiendrez la sortie pour tous les cas de tests sur votre interface en ligne de commande.

Command Line

```
npm test
```

Maintenant Jest vous permet d'écrire des tests d'instantanés. Ces tests font un instantané de votre composant rendu et lancent cet instantané par rapport à de futurs instantanés. Lorsqu'un futurs intantané change, vous serez informé dans le test. Vous pouvez soit accepter le changement d'instantané, car vous avez changé l'implémentation du composant délibérément, ou rejeter le changement et investiguer sur l'erreur. Il complète les tests unitaires très bien, car vous testez seulement les différences sur la sortie rendue. Il n'ajoute pas un important coût de maintenance, car vous pouvez simplement accepter les instantanés modifiés lorsque vous changez quelques chose délibérément pour la sortie de rendu de votre composant.

Jest stocke les instantanés dans un dossier. C'est seulement de cette façon qu'il peut valider la différence par rapport à un instantané futur.

Avant l'écriture de votre premier test instantané avec Jest, vous devez installer une bibliothèque utilitaire.

Command Line

```
npm install --save-dev react-test-renderer
```

Maintenant vous pouvez étendre le test du composant App avec votre premier test instantané. Premièrement, importer la nouvelle fonctionnalité depuis le node package et englober votre précédent bloc "it" du composant App dans un bloc de description "describe". Dans ce cas, la suite de test est consacré au composant App.

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App from './App';

describe('App', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
    ReactDOM.unmountComponentAtNode(div);
  });


});
```

Maintenant vous pouvez implémenter votre premier instantané en utilisant un bloc "test".

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App from './App';

describe('App', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <App />
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Lancez de nouveau vos tests et observez comment les tests réussissent ou échouent. Ils devraient réussir. Une fois que vous modifiez la sortie du bloc de rendu de votre composant App, le test instantané doit échouer. Alors vous pouvez décider de mettre à jour l'instantané ou investiguer dans votre composant App.

Globalement la fonction `renderer.create()` crée un instantané de votre composant App. Il le rend virtuellement et stocke le DOM à l'intérieur de l'instantané. Après quoi, l'instantané est attendu pour correspondre avec le précédent instantané au moment où vous avez lancé vos tests d'instantané. De cette façon, vous pouvez assurer que votre DOM reste le même et que rien ne change par accident.

Ajoutons plus de tests pour nos composants indépendants. Premièrement, le composant Search :

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App, { Search } from './App';

// ...

describe('Search', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Search>Search</Search>, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Search>Search</Search>
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Le composant Search a deux tests similaires au composant App. Le premier test rend simplement le composant Search auprès du DOM et vérifie qu'il n'y est pas d'erreur durant le processus de rendu. S'il y aurait une erreur, le test casserait même s'il n'y a pas d'assertion. (ex : expect, match, equal) dans le bloc de test. Le second test d'instantané est utilisé pour stocker un instantané du composant rendu et de le lancer par rapport au précédent instantané. Il échoue lorsque l'instantané a changé.

Deuxièmement, vous pouvez tester le composant Button auquel les mêmes principes de tests que le composant Search sont appliqués.

src/App.test.js

```
// ...
import App, { Search, Button } from './App';

// ...

describe('Button', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Button>Give Me More</Button>, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Button>Give Me More</Button>
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Enfin, mais pas des moindres, le composant Table auquel vous pouvez passer un ensemble de `props` initiales dans le but de le rendre avec un échantillon de liste.

src/App.test.js

```
// ...
import App, { Search, Button, Table } from './App';

// ...

describe('Table', () => {

  const props = {
    list : [
      { title : '1', author : '1', num_comments : 1, points : 2, objectID : 'y' },
      { title : '2', author : '2', num_comments : 1, points : 2, objectID : 'z' },
    ],
  };
```

```
  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Table { ...props } />, div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Table { ...props } />
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Les tests d'instantané habituellement restent assez basiques. Vous voulez seulement couvrir le fait que le composant ne change pas sa sortie. Une fois qu'il change la sortie, vous devez décider si vous acceptez les changements. Autrement vous devez réparer le composant lorsque la sortie n'est pas une sortie désirée.

Exercices :

- regarder comment un test d'instantané échoue une fois que vous changez votre valeur de retour du composant avec la méthode `render()`
  - acceptez ou rejetez le changement d'instantané
- conservez vos instantanés à jour lorsque l'implémentation de composant change dans les prochains chapitres
- lire plus à propos de Jest au sein de React[125]

125. https://jestjs.io/docs/en/tutorial-react

Les tests unitaires avec Enzyme

Enzyme[126] est un utilitaire de test d'Airbnb pour assert, manipuler et traverser vos composants React. Vous pouvez l'utiliser pour piloter vos tests unitaires dans le but de compléter vos tests d'instantanés dans React.

Regardons comment vous pouvez utiliser Enzyme. Premièrement vous devez l'installer puisqu'il n'est pas embarqué dans *create-react-app*. Il arrive avec une extension pour l'utiliser au sein de React.

Command Line

```
npm install --save-dev enzyme react-addons-test-utils enzyme-adapter-react-16
```

Deuxièmement, vous aurez besoin de l'inclure dans votre installation de test et initialiser son Adapter pour son usage au sein de React.

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
import App, { Search, Button, Table } from './App';

Enzyme.configure({ adapter : new Adapter() });
```

Maintenant, vous pouvez écrire votre premier test unitaire dans le bloc "description" de Table. Vous utiliserez `shallow()` pour rendre votre composant et affirmer que la Table possède deux objets, car vous lui passez deux éléments de liste. L'affirmation vérifie simplement si l'élément possède deux éléments avec la classe `table-row`.

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import Enzyme, { shallow } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
import App, { Search, Button, Table } from './App';

// ...
```

126. https://github.com/airbnb/enzyme

```
describe('Table', () => {

  const props = {
    list : [
      { title : '1', author : '1', num_comments : 1, points : 2, objectID : 'y' },
      { title : '2', author : '2', num_comments : 1, points : 2, objectID : 'z' },
    ],
  };

  // ...

  it('shows two items in list', () => {
    const element = shallow(
      <Table { ...props } />
    );

    expect(element.find('.table-row').length).toBe(2);
  });

});
```

Shallow rend le composant sans ses composants enfants. De cette façon, vous pouvez faire le test dévoué à un seul composant.

Enzyme a dans l'ensemble trois mécanismes de rendu dans son API ? Vous connaissez déjà `shallow()`, mais il y existe aussi `mount()` and `render()`. Les deux instancient les instances du composant parent ainsi que tous les composants enfants. De plus `mount()` vous donne un accès aux méthodes de cycle de vie du composant. Mais quand savoir utiliser quel mécanisme de rendu ? Ici quelques règles générales :

- Toujours débuter avec un test shallow
- Si `componentDidMount()` ou `componentDidUpdate()` doit être testé, utiliser `mount()`
- Si vous voulez tester le cycle de vie du composant et le comportement des enfants, utiliser `mount()`
- Si vous voulez tester un rendu des enfants du composant avec moins de contrainte que `mount()` et que vous n'êtes pas intéressé par les méthodes du cycle de vie, utilisez `render()`

Vous pouvez continuer à tester unitairement vos composants. Mais assurez-vous de conserver les tests simple et maintenable. Autrement vous devrez les refactorer une fois que vous modifiez vos composants. C'est pourquoi Facebook a introduit les tests instantanés avec Jest en premier lieu.

Exercices :

- écrire un test unitaire avec Enzyme pour votre composant Button
- conserver vos tests unitaires à jour durant le chapitre suivant
- lire plus à propos d'Enzyme et de son API de rendu[127]
- lire plus à propos du testing des applications React[128]

127. https://github.com/airbnb/enzyme
128. https://www.robinwieruch.de/react-testing-tutorial

Interface de composant avec les PropTypes

Vous pourriez connaitre TypeScript[129] ou Flow[130] pour introduire une interface de type pour JavaScript. Un langage typé est plus robuste aux errors, car le code sera validé en fonction de son propre code. Les éditeurs et autres utilitaires peuvent saisir ces erreurs avant que le programme soit lancé. Cela rend votre programme plus robuste.

Dans le livre, vous ne serez pas introduit à Flow ou TypeScript, mais à une autre façon proche de tester vos types dans les composants. React arrive avec un vérificateur de type embarqué pour empêcher les bugs. Vous pouvez utiliser *PropTypes* pour décrire votre interface de composant. Toutes les propriétés qui sont passées depuis un composant parent à un composant enfant seront validées selon l'interface *PropTypes* assignée au composant enfant.

Cette section du chapitre vous montera comment vous pouvez rendre tous vos types de composants sûrs avec *PropTypes*. J'omettrai ces changements pour les prochains chapitres, car ils ajoutent du remaniement de code inutile. Mais vous pouvez les conserver et les mettre à jour tout du long pour conserver votre type d'interface de composants sûrs.

Premièrement, vous devez installer un package séparé pour React.

Command Line

```
npm install prop-types
```

Maintenant, vous pouvez importer *les PropTypes*.

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import PropTypes from 'prop-types';
```

Débutons en assignant une interface de propriétés aux composants :

129. https://www.typescriptlang.org/
130. https://flowtype.org/

src/App.js

```
const Button = ({
  onClick,
  className = '',
  children,
}) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

Button.propTypes = {
  onClick : PropTypes.func,
  className : PropTypes.string,
  children : PropTypes.node,
};
```

Tout simplement. Vous prenez chaque argument depuis la signature de fonction et lui assigner un *PropType*. Le *PropTypes* basiques pour les primitives et les objets complexes sont :

- PropTypes.array
- PropTypes.bool
- PropTypes.func
- PropTypes.number
- PropTypes.object
- PropTypes.string

De plus vous avez deux PropTypes supplémentaires pour définir un fragment (node) présentable, ex : une string, et un élément React :

- PropTypes.node
- PropTypes.element

Vous utilisez déjà le *PropType* `node` pour le composant Button. Globalement il y a des définitions de *PropType* supplémentaires que vous pouvez étudier dans la documentation officielle de React.

Pour le moment tout les *PropTypes* définies pour le Button sont optionnelles. Les paramètres peuvent être `null` ou `undefined`. Mais pour certaines propriétés vous voudriez imposer le fait qu'elles soient définies. Vous pouvez rendre nécessaire le fait que ces propriétés sont passées au composant.

src/App.js

```
Button.propTypes = {
  onClick : PropTypes.func.isRequired,
  className : PropTypes.string,
  children : PropTypes.node.isRequired,
};
```

La `className` n'est pas requise, car elle peut par défaut être une chaine de caractère vide. Après vous définirez une interface *PropTypes* pour le composant Table :

src/App.js

```
Table.propTypes = {
  list : PropTypes.array.isRequired,
  onDismiss : PropTypes.func.isRequired,
};
```

Vous pouvez définir le contenu d'un *PropType* tableau plus explicitement :

src/App.js

```
Table.propTypes = {
  list : PropTypes.arrayOf(
    PropTypes.shape({
      objectID : PropTypes.string.isRequired,
      author : PropTypes.string,
      url : PropTypes.string,
      num_comments : PropTypes.number,
      points : PropTypes.number,
    })
  ).isRequired,
  onDismiss : PropTypes.func.isRequired,
};
```

Seul l'`objectID` est requis, car vous savez que certaines parties de votre code en dépendent. Les autres propriétés sont seulement affichées, ainsi elles ne sont pas nécessairement requises. De plus vous ne pouvez être sûr que l'API d'Hacker News a toujours une propriété définie pour chaque objet dans le tableau.

C'est tout pour les *PropTypes*. Mais il y a un autre aspect. Vous pouvez définir des propriétés par défaut dans votre composant. Prenons de nouveau le composant Button. La propriété `className` a un paramètre ES6 par défaut dans la signature du composant.

src/App.js

```
const Button = ({
  onClick,
  className = '',
  children
}) =>
  // ...
```

Vous pouvez le remplacer avec la propriété par défaut interne de React :

src/App.js

```
const Button = ({
  onClick,
  className,
  children
}) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

Button.defaultProps = {
  className : '',
};
```

Identique que le paramètre par défaut d'ES6, la propriété par défaut assure que la propriété est établie à une valeur par défaut lorsque le composant parent n'est pas spécifié. La vérification du type de *PropType* se produit après que la propriété par défaut est évaluée.

Si vous lancez de nouveau vos tests, vous pourriez voir les erreurs *PropTypes* de vos composants dans votre terminal. Elles peuvent apparaitre car vous n'avez pas défini toutes les propriétés de vos composants dans les tests qui sont définis comme requises dans votre définition de *PropType*. Cependant les tests eux-même passent correctement. Vous pouvez passer toutes les propriétés requises aux composants dans vos tests pour éviter ces erreurs.

Exercices :

- définir l'interface *PropType* pour le composant Search

- ajouter et mettre à jour les interfaces *PropType* lorsque vous ajoutez et mettez à jour des composants dans les prochains chapitres
- lire plus à propos des PropTypes de React[131]

131. https://reactjs.org/docs/typechecking-with-proptypes.html

Débuggage à l'aide du React Developer Tools

Cette dernière section vous présente un outil très utile, généralement utilisé pour inspecter et debugger les applications React. React Developer Tools[132] vous laisse inspecter la hiérarchie des composants React, les props et l'état. C'est une extension de navigateur web (pour l'heure, sur Chrome et Firefox) et une application standalone (qui fonctionne sur d'autres environnements).

Une fois installé, l'icône de l'extension s'éclairera sur les sites web utilisant React. Sur ces sites web, vous verrez un onglet appelé "React" au sein de votre browser developer tools.

Essayons cela sur notre application Hacker News. Sur la plupart des navigateurs, un moyen rapide pour faire apparaître le *dev tools* up consiste à réaliser un click droit sur la page et allez sur "Inspecter". Faites-le lorsque votre application est chargée, puis cliquez sur l'onglet "React". Vous devriez voir sa hiérarchie d'éléments, avec `<App>` l'élément racine. Si vous le développez, vous trouverez également des instances de vos composants `<Search>`, `<Table>` et `<Button>`.

L'extension montre sur le panneau latéral l'état du composant ainsi que les props de l'élément sélectionné. Par exemple, si vous cliquez sur `<App>`, vous verrez qu'il ne dispose d'aucune props, mais qu'il a un état. C'est une technique de débuggage évidente pour monitorer l'état changeant de votre application dû à l'interaction de l'utilisateur.

Premièrement, vous vérifierez l'option "Highlight Updates" (habituellement au-dessus de l'arbre des éléments). Deuxièmement, vous pouvez taper un terme de recherche différent dans le champ d'entrée de l'application. Comme vous le remarquerez, seulement `searchTerm` sera modifié au sein de l'état du composant. Vous connaissez d'ores et déjà ce qui se produit, mais maintenant vous pouvez voir que cela fonctionne comme prévu.

Finalement, vous pouvez appuyer sur le bouton "Search". L'état `searchKey` sera immédiatement changé par la même valeur que `searchTerm` et quelques secondes après l'objet de réponse sera ajouté à `results`. la nature asynchrone de votre code est maintenant visible à l'oeil.

Enfin mais pas des moindres, si vous effectuer un click droit sur n'importe quel élément, un dropdown menu vous montrera plusieurs options utiles. Par exemple, vous pouvez copier les props ou le nom de l'élément, trouver le noeud du DOM correspondant ou sauter vers le code source de l'application au sein du navigateur web. Cette dernière option est très utile pour insérer des points d'arrêt dans le but de debugger les fonctions JavaScript.

Exercises :

- Installer l'extension React Developer Tools[133] sur votre navigateur web favoris
  - lancer votre application clone d'Hacker News et l'inspecter en utilisant l'extension
  - Experimenter les changements d'état et de props
  - Observer ce qui se produit lorsque vous déclencher une requête asynchrone

132. https://github.com/facebook/react-devtools
133. https://github.com/facebook/react-devtools

  - Performer plusieurs requêtes, en en répétant certaines. Observer que le mécanisme de cache fonctionne
- lire à propos de comment débugger vos fonctions JavaScript au sein d'un navigateur web[134]

134. https://developers.google.com/web/tools/chrome-devtools/javascript/

Vous avez appris comment organiser votre votre et comment le tester ! Récapitulons les derniers chapitres :

- React
  - PropTypes vous laissent définir les vérifications de type pour les composants
  - Jest vous permet d'écrire des tests instantanés pour vos composants
  - Enzyme vous permet d'écrire des tests unitaires pour vos composants
  - React Developer Tools est un outil de débuggage très utile
- ES6
  - les déclarations d'import et d'export vous aident à organiser votre code
- Général
  - L'organisation de code vous permet de mettre à l'échelle votre application avec les meilleurs pratiques

Vous pouvez trouver le code source dans le dépôt officiel[135].

135. https://github.com/the-road-to-learn-react/hackernews-client/tree/5.4

React composants avancés

Le chapitre se concentrera sur l'implémentation de composants React avancés. Vous apprendrez à propos des composants d'ordre supérieur et comment les implémenter. De plus, vous plongerez dans des sujets plus avancés de React et implémenterez des interactions complexes avec ce dernier.

Ref un DOM Element

Parfois vous aurez besoin d'interagir avec votre noeud de DOM au sein de React. L'attribut `ref` vous donne un accès à un noeud de vos éléments. Habituellement c'est un anti-pattern en React, car vous devez l'utiliser sa manière déclarative pour faire quoi que ce soit ainsi que son flux de données unidirectionnel. Vous avez appris cela lorsque vous avez introduit votre premier champ d'entrée de recherche. Mais il y a certains cas où vous avez besoin d'accéder à un noeud de DOM. La documentation officielle mentionne trois cas :

- utiliser l'API du DOM (focus, media playback etc.)
- invoquer des animations impératives de noeuds du DOM
- intégrer une librairie tierce qui a besoin de noeuds du DOM (ex : D3.js[136])

Réalisons un exemple avec le composant Search. Lorsque l'application se rend pour la première fois, le champ d'entrée devra être en focus. C'est l'un des cas d'utilisation où vous aurez besoin d'accéder à l'API du DOM. Ce chapitre vous montre comment cela fonctionne, mais puisque c'est ne pas très utile pour l'application en soi, nous omettrons les changements dans le prochain chapitre. Néanmoins vous pouvez les conserver pour votre propre application.

En général, vous pouvez utiliser l'attribut `ref` à la fois dans les composants fonctionnels stateless et les composants de classe ES6. Dans le cas de la fonctionnalité de focus, vous aurez besoin des méthodes du cycle de vie. C'est pourquoi l'approche est en premier lieu présentée en utilisant l'attribut `ref` avec un composant de classe ES6.

L'étape initiale consiste à refactoriser le composant fonctionnel stateless en un composant de classe ES6.

src/App.js

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
```

136. https://d3js.org/

```
          onChange={onChange}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

L'objet `this` du composant de classe ES6 nous aide à référencer l'élément du DOM avec l'attribut `ref`.

src/App.js

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={el => this.input = el}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

Maintenant vous pouvez focus le champ d'entrée lorsque le composant est monté en utilisant l'objet `this`, la méthode du cycle de vie appropriée, et l'API du DOM.

src/App.js

```
class Search extends Component {
  componentDidMount() {
    if (this.input) {
      this.input.focus();
    }
  }

  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={el => this.input = el}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

Le champ d'entrée doit être focus lorsque l'application se rend. C'est essentiellement tout pour l'usage de l'attribut `ref`.

Mais comment vous aurez accès au `ref` dans un composant fonctionnel stateless sans l'objet `this` ? Le composant fonctionnel stateless suivant en fait une démonstration :

src/App.js

```
const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) => {
  let input;
  return (
    <form onSubmit={onSubmit}>
      <input
        type="text"
        value={value}
        onChange={onChange}
        ref={el => this.input = el}
      />
      <button type="submit">
        {children}
      </button>
    </form>
  );
}
```

Maintenant vous serez capable d'accéder à l'entrée de l'élément du DOM. Dans l'exemple de la fonctionnalité du focus cela ne vous aidera pas, car vous ne disposez pas des méthodes du cycle de vie dans un composant fonctionnel stateless pour déclencher le focus. Mais dans un futur vous pourrez traverser d'autres cas d'utilisation où il y aura du sens d'utiliser un composant fonctionnel stateless avec l'attribut `ref`.

Exercises

- lire plus à propos de l'usage de l'attribut ref au sein de React[137]
- lire plus à propos de l'attribut ref de façon générale au sein de React[138]

137. https://www.robinwieruch.de/react-ref-attribute-dom-node/
138. ttps://reactjs.org/docs/refs-and-the-dom.html

Chargement ...

Retournons à l'application. Vous pourriez vouloir afficher un indicateur de chargement lorsque vous soumettez une requête de recherche à l'API d'Hacker News. La requête est asynchrone et vous devrez montrer à votre utilisateur quelques retours visuels que quelque chose se produit. Définissons un composant Loading réutilisable dans votre fichier *src/App.js*.

src/App.js

```
const Loading = () =>
  <div>Loading ...</div>
```

Maintenant vous aurez besoin d'une propriété pour stocker l'état du chargement. Selon l'état du chargement vous pouvez décider de montrer le composant Loading par la suite.

src/App.js

```
class App extends Component {
  _isMounted = false;

  constructor(props) {
    super(props);

    this.state = {
      results : null,
      searchKey : '',
      searchTerm : DEFAULT_QUERY,
      error : null,
      isLoading : false,
    };

    // ...
  }

  // ...

}
```

La valeur initiale de votre propriété `isLoading` est fausse. Vous ne chargez rien avant que le composant App soit monté.

Lorsque vous faites la requête, vous mutez l'état de chargement à true. Finalement la requête réussira et vous pourrez muter l'état de chargement à false.

src/App.js

```
class App extends Component {

  // ...

  setSearchTopStories(result) {
    // ...

    this.setState({
      results : {
        ...results,
        [searchKey]: { hits : updatedHits, page }
      },
      isLoading : false
    });
  }

  fetchSearchTopStories(searchTerm, page = 0) {
    this.setState({ isLoading : true });

    axios(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(result => this._isMounted && this.setSearchTopStories(result.data))
      .catch(error => this._isMounted && this.setState({ error }));
  }

  // ...

}
```

Dernière étape, vous utiliserez le composant Loading dans votre App. Un rendu conditionnel basé sur l'état de chargement décidera si vous affichez le composant Loading ou le composant Button. Ce dernier est votre bouton pour aller chercher plus de données.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;

    // ...

    return (
      <div className="page">
        // ...
        <div className="interactions">
          { isLoading
           ? <Loading />
           : <Button
             onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}>
             More
           </Button>
          }
        </div>
      </div>
    );
  }
}
```

Initialement le composant Loading s'affichera lorsque vous démarrer votre application, car vous ferez une requête avec `componentDidMount()`. Il n'y a pas de composant Table, car le liste est vide. Lorsque la réponse revient de l'API d'Hacker News, le résultat est montré, le chargement d'état est muté à false et le composant Loading disparaît. À la place, le bouton "More" pour aller chercher plus de données apparaît. Une fois que vous recherchez plus de données, le bouton disparaîtra de nouveau et le composant Loading se révèlera.

Exercices :

- utiliser une biblohtèque telle que Font Awesome[139] pour afficher une icône de chargement au lieu du texte “Loading …”

139. https://fontawesome.io/

Composants d'ordre supérieur

Les composants d'ordre supérieur (HOC) sont un concept avancé en React. Les HOCs sont un équivalent des fonctions d'ordre supérieur. Ils prennent plusieurs entrées - la plupart du temps un composant, mais aussi des arguments optionnels - et retournent un composant en sortie. Le composant retourné est une version améliorée du composant d'entrée et peut-être utilisé dans votre JSX.

Les HOCs sont utilisés pour différents cas d'utilisation. Ils préparent les propriétés, gère l'état où altère la représentation d'un composant. Un cas d'utilisation pourrait être d'utiliser un HOC en tant qu'assistant pour un rendu conditionnel. Imaginez-vous avez un composant List qui rend une liste d'éléments ou rien, car la liste est vide ou null. L'HOC pourrait protéger le fait que la liste souhaiterait ne rien rendre lorsqu'il n'y a pas de liste. À l'inverse, le sobre composant List n'a plus besoin de s'encombrer d'une liste non existante. Il se préoccupe seulement de la liste rendue.

Faisons un simple HOC qui prend un composant en entrée et retourne un composant. Vous pouvez le placer à l'intérieur de votre fichier *src/App.js*.

src/App.js

```
function withFoo(Component) {
  return function(props) {
    return <Component { ...props } />;
  }
}
```

Une convention cohérente est de préfixer le nommage d'HOC avec `with`. Puisque vous utilisez du JavaScript ES6, vous pouvez exprimer l'HOC plus concisément avec une fonction fléchée ES6.

src/App.js

```
const withFoo = (Component) => (props) =>
  <Component { ...props } />
```

Dans l'exemple, le composant d'entrée devrait rester le même que le composant de sortie. Rien ne se produit. Il rend la même instance de composant et passe toutes les propriétés au composant de sortie. Mais c'est inutile. Améliorons le composant de sortie. Le composant de sortie devrait afficher le composant Loading, quand l'état de chargement est à true, autrement il devrait afficher le composant d'entrée. Un rendu conditionnel est un bon cas d'utilisation pour un HOC.

src/App.js

```
const withLoading = (Component) => (props) =>
  props.isLoading
    ? <Loading />
    : <Component { ...props } />
```

Basé sur la propriété de chargement vous pouvez appliquer un rendu conditionnel. La fonction retournera le composant de chargement ou le composant d'entrée.

En général il peut être efficace de diffuser un objet, tel que les propriétés d'objet dans l'exemple précédent, en tant qu'entrée pour un composant. Observez la différence avec le bout de code suivant.

Code Playground

```
// avant vous aurez déstructuré les propriétés avant de les transmettre
const { foo, bar } = props;
<SomeComponent foo={foo} bar={bar} />

// Mais vous pouvez utiliser le spread operator objet pour transmettre toutes les pr\
opriétés de l'objet.
<SomeComponent { ...props } />
```

Il y a une petite chose que vous devez éviter. Vous passez toutes les propriétés incluant la propriété `isLoading`, en diffusant l'objet, à l'intérieur du composant d'entrée. Cependant, le composant d'entrée pourrait ne pas désirer la propriété `isLoading`. Vous pouvez utiliser la décomposition du reste en ES6 pour éviter cela.

src/App.js

```
const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />
```

Cela prend une propriété à l'extérieur de l'objet, mais conserve le reste de l'objet. Cela fonctionne avec plusieurs propriétés également. Vous pourriez l'avoir déjà lu dans l'affectation par décomposition[140].

Maintenant vous pouvez utiliser l'HOC dans votre JSX. Un cas d'utilisation au sein de l'application pourrait être d'afficher soit le bouton "More" soit le composant Loading. Le composant Loading est toujours encapsulé dans l'HOC, mais un composant en entrée est manquant. Dans le cas d'utilisation consistant à montrer un composant Button ou un composant Loading, le Button est le composant d'entrée de l'HOC. Le composant de sortie améliorée est un composant ButtonWithLoading.

140. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

src/App.js

```
const Button = ({
  onClick,
  className = '',
  children,
}) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

const Loading = () =>
  <div>Loading ...</div>

const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
   ? <Loading />
    : <Component { ...rest } />

const ButtonWithLoading = withLoading(Button);
```

Maintenant tout est défini. En dernier étape, vous devez utiliser le composant ButtonWithLoading, qui reçoit l'état de chargement en tant que propriété additionnelle. Tandis que l'HOC consomme la propriété loading, tout autres propriétés seront transmises au composant Button.

src/App.js

```
class App extends Component {

  // ...

  render() {
    // ...
    return (
      <div className="page">
        // ...
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}>
```

```
            More
          </ButtonWithLoading>
        </div>
      </div>
    );
  }
}
```

Lorsque vous lancez de nouveau vos tests, vous remarquerez que votre test d'instantané pour le composant App échoue. La différence devrait ressembler à cela sur le terminal :

Command Line

```
-     <button
-       className=""
-       onClick={[Function]}
-       type="button"
-     >
-       More
-     </button>
+     <div>
+       Loading ...
+     </div>
```

Vous pouvez maintenant soit réparer le composant, lorsque vous pensez qu'il y a quelques soucis avec cela, ou vous pouvez accepter le nouvel instantané. Parceque vous avez introduit le composant Loading dans ce chapitre, vous pouvez accepter la version modifiée du test d'instantané sur le terminal dans le test interactif.

Les composants d'ordre supérieur sont une technique avancée dans React. Ils ont de multiples buts comme améliorer la réutilisabilité des composants, une meilleure abstraction, une composabilité des composants et la manipulation des propriétés ainsi que de l'état et la vue. Ne vous inquiétez pas si vous ne les comprenez pas immédiatement. Cela prend du temps à s'habituer à eux.

Je vous encourage à lire la véritable introduction aux composants d'ordre supérieur[141]. Il vous fournit une autre approche pour les apprendre, vous montre une façon élégante de les utiliser dans une approche de programmation fonctionnelle et résout spécifiquement le problème de rendu conditionnel avec les composants d'ordre supérieur.

Exercices :

- lire la véritable introduction aux composants d'ordre supérieur[142]

141. https://www.robinwieruch.de/gentle-introduction-higher-order-components/
142. https://www.robinwieruch.de/gentle-introduction-higher-order-components/

- experimenter avec l'HOC que vous avez créé
- penser à un cas d'utilisation où un autre HOC ferait sens
  - implementer l'HOC, s'l y a un cas d'utilisation

Tri avancé

Vous avez déjà implémenté une interaction de recherche côté client et serveur. Puisque vous avez un composant Table, il ferait sens d'améliorer le Table avec des interactions avancées. Pourquoi ne pas introduire une fonctionnalité de tri pour chaque colonne en utilisant les en-têtes de colonnes du Table ?

Il serait possible d'écrire votre propre fonction `sort`, mais personnellement je préfère utiliser une bibliothèque pour ce genre de cas. Lodash[143] est l'une de ces bibliothèques utiles, mais vous pouvez utiliser n'importe quelle bibliothèque qui vous correspond. Installons Lodash et utilisons la pour la fonctionnalité de tri.

Command Line

```
npm install lodash
```

Maintenant vous pouvez importer la fonctionnalité de tri de Lodash dans votre fichier *src/App.js*.

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import { sortBy } from 'lodash';
import './App.css';
```

Vous avez plusieurs colonnes dans votre Table. Il y a les colonnes title, author, comments et points. Vous pouvez définir les fonctions de tri ainsi chaque fonction prend une liste et retourne une liste d'éléments triée en spécifiant la propriété. Qui plus est, vous aurez besoin d'une fonction de tri par défaut qui ne triera pas mais seulement retournera la liste non triée. Cela sera votre état initial.

src/App.js

```
// ...

const SORTS = {
  NONE : list => list,
  TITLE : list => sortBy(list, 'title'),
  AUTHOR : list => sortBy(list, 'author'),
  COMMENTS : list => sortBy(list, 'num_comments').reverse(),
  POINTS : list => sortBy(list, 'points').reverse(),
};

class App extends Component {
```

143. https://lodash.com/

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

Vous pouvez voir qu'il y a deux fonctions de tri qui retournent une liste inversée. C'est parce que vous voulez voir les éléments avec le plus de commentaires et de points plutôt que de voir les éléments avec le total le plus faible lorsque vous triez la liste pour la première fois.

Maintenant l'objet `SORTS` vous permet de référencer toutes fonctions de tri.

Encore une fois votre composant App est responsable de stocker l'état du tri. L'état initial sera la fonction de tri initial par défaut, qui trie pas du tout mais retourne la liste d'entrée en sortie.

src/App.js

```
this.state = {
  results : null,
  searchKey : '',
  searchTerm : DEFAULT_QUERY,
  error : null,
  isLoading : false,
  sortKey : 'NONE',
};
```

Une fois que vous choisissez un `sortKey` différent, admettons la clé `AUTHOR`, vous trierez la liste avec la fonction de tri approprié issu de l'objet `SORTS`.

Maintenant vous pouvez définir une nouvelle méthode de classe dans votre composant App qui simplement mute un `sortKey` vers votre état local du composant. Après quoi, le `sortKey` peut être utilisé pour retrouver la fonction de tri pour l'appliquer sur votre liste.

src/App.js

```
class App extends Component {
  _isMounted = false;

  constructor(props) {

    // ...

    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
```

```
    this.onSort = this.onSort.bind(this);
  }

  // ...

  onSort(sortKey) {
    this.setState({ sortKey });
  }

  // ...

}
```

La prochaine étape est de passer la méthode et `sortKey` à votre composant Table.

src/App.js

```
class App extends Component {

  // ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading,
      sortKey
    } = this.state;

    // ...

    return (
      <div className="page">
        // ...
        <Table
          list={list}
          sortKey={sortKey}
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
        // ...
      </div>
```

```
    );
  }
}
```

Le composant Table est responsable de trier votre liste. Il prend une des fonctions de `SORT` à l'aide de `sortkey` et passe la liste d'entrée. Après quoi il conserve le mappage sur la liste triée.

src/App.js

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
  <div className="table">
    {SORTS[sortKey](list).map(item =>
      <div key={item.objectID} className="table-row">
        // ...
      </div>
    )}
  </div>
```

En théorie la liste devrait être triée par l'une des fonctions. Mais le tri par défaut est réglé à `NONE`, donc rien n'est encore trié. Jusqu'ici, personne éxecute la méthode `onSort()` pour changer le `sortKey`. Étendons la Table avec une ligne d'en-tête de colonne qui utilise les composants Sort dans les colonnes pour trier chaque colonne.

src/App.js

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
  <div className="table">
    <div className="table-header">
      <span style={{ width : '40%' }}>
        <Sort
          sortKey={'TITLE'}
          onSort={onSort}
        >
          Title
```

```
        </Sort>
      </span>
      <span style={{ width : '30%' }}>
        <Sort
          sortKey={'AUTHOR'}
          onSort={onSort}
        >
          Author
        </Sort>
      </span>
      <span style={{ width : '10%' }}>
        <Sort
          sortKey={'COMMENTS'}
          onSort={onSort}
        >
          Comments
        </Sort>
      </span>
      <span style={{ width : '10%' }}>
        <Sort
          sortKey={'POINTS'}
          onSort={onSort}
        >
          Points
        </Sort>
      </span>
      <span style={{ width : '10%' }}>
        Archive
      </span>
    </div>
    {SORTS[sortKey](list).map(item =>
      // ...
    )}
  </div>
```

Chaque composant Sort dispose d'une `sortKey` spécifique et de la fonction générale `onSort()`. Internement cela appelle la méthode avec la `sortKey` pour établir la clé spécifique.

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button onClick={() => onSort(sortKey)}>
    {children}
  </Button>
```

Comme vous pouvez le voir, le composant Sort réutilise votre composant Button commun. Au clic d'un bouton chaque individuelle `sortKey` passée sera mutée par la méthode `onSort()`. Maintenant vous devez être capable de trier la liste lorsque vous cliquez sur les en-têtes de colonne.

Il y a une amélioration mineure pour un visuel amélioré. Jusqu'ici, le bouton dans la colonne d'en-tête est un peu en décalé. Donnons au bouton dans le composant Sort un `className` approprié.

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button
    onClick={() => onSort(sortKey)}
    className="button-inline"
  >
    {children}
  </Button>
```

Maintenant, il doit avoir un ravissant aspect. Le prochain but serait d'également implémenter un tri inverse. La liste doit inverser le tri dès que vous cliquez deux fois sur le composant Sort. Premièrement, vous avez besoin de définir l'état inversé avec un booléen. Le tri peut être inversé ou non inversé.

src/App.js

```
this.state = {
  results : null,
  searchKey : '',
  searchTerm : DEFAULT_QUERY,
  error : null,
  isLoading : false,
  sortKey : 'NONE',
  isSortReverse : false,
};
```

Maintenant dans votre méthode de tri, vous pouvez évaluer si la liste est triée de façon inversée. C'est inversé si la `sortKey` dans l'état est la même que la `sortKey` entrante et que l'état inversé n'est pas déjà établi à true.

src/App.js

```
onSort(sortKey) {
  const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortReverse;
  this.setState({ sortKey, isSortReverse });
}
```

De nouveau vous pouvez passer la propriété reverse à votre composant Table.

src/App.js

```
class App extends Component {

  // ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading,
      sortKey,
      isSortReverse
    } = this.state;

    // ...

    return (
      <div className="page">
        // ...
        <Table
          list={list}
          sortKey={sortKey}
          isSortReverse={isSortReverse}
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
        // ...
      </div>
    );
  }
}
```

Maintenant le Table doit avoir un *block body* de fonction fléchée pour calculer les données.

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const finalSortedList = isSortReverse
   ? sortedList.reverse()
    : sortedList;

  return(
    <div className="table">
      <div className="table-header">
        // ...
      </div>
      {finalSortedList.map(item =>
        // ...
      )}
    </div>
  );
}
```

Le tri inversé doit fonctionner maintenant.

Enfin mais pas des moindres, vous devez traiter une question ouverte pour l'intérêt d'une une expérience utilisateur améliorée. Un utilisateur peut-il distinguer quelle colonne est activement trié ? Jusqu'à présent, ce n'est pas possible. Donnons un retour visuel à l'utilisateur.

Chaque composant Sort dispose déjà de sa `sortKey` spécifique. Elle pourrait être utilisée pour identifier le tri actif. Vous pouvez passer la `sortKey` depuis l'état interne du composant en tant que clé de tri actif pour votre composant Sort.

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const finalSortedList = isSortReverse
   ? sortedList.reverse()
    : sortedList;

  return(
    <div className="table">
      <div className="table-header">
        <span style={{ width : '40%' }}>
          <Sort
            sortKey={'TITLE'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Title
          </Sort>
        </span>
        <span style={{ width : '30%' }}>
          <Sort
            sortKey={'AUTHOR'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Author
          </Sort>
        </span>
        <span style={{ width : '10%' }}>
          <Sort
            sortKey={'COMMENTS'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Comments
          </Sort>
        </span>
```

```
        <span style={{ width : '10%' }}>
          <Sort
            sortKey={'POINTS'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Points
          </Sort>
        </span>
        <span style={{ width : '10%' }}>
          Archive
        </span>
      </div>
      {finalSortedList.map(item =>
          // ...
      )}
    </div>
  );
}
```

Maintenant dans votre composant Sort, vous savez à l'aide de la `sortKey` et de `activeSortKey` si le tri est actif. Donnez à votre composant un attribut `className` supplémentaire, dans le cas où il est trié, pour donner à l'utilisateur un retour visuel.

src/App.js

```
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = ['button-inline'];

  if (sortKey === activeSortKey) {
    sortClass.push('button-active');
  }

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass.join(' ')}
    >
      {children}
```

```
    </Button>
  );
}
```

La manière de définir le `sortClass` est un peu malhabile, n'est ce pas ? Il y a une petite bibliothèque sympa pour se débarrasser de çà. Premièrement vous devez l'installer.

Command Line

```
npm install classnames
```

Et deuxièmement vous devez l'importer tout en haut du fichier *src/App.js*.

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import { sortBy } from 'lodash';
import classNames from 'classnames';
import './App.css';
```

Maintenant vous pouvez l'utiliser pour définir votre `className` du composant avec des classes conditionnelles.

src/App.js

```
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = classNames(
    'button-inline',
    { 'button-active' : sortKey === activeSortKey }
  );

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass}
    >
      {children}
    </Button>
  );
}
```

De nouveau, lorsque vous lancez vos tests, vous devez voir des tests d'instantanés échouant mais aussi des tests unitaires échouant pour le composant Table. Dû au fait que vous avez changé une nouvelle fois vos représentations de composant, vous pouvez accepter les tests instantanés. Mais vous devez réparer le test unitaire. Dans votre fichier *src/App.test.js*, vous avez besoin de fournir une `sortKey` et le booléen `isSortReverse` pour le composant Table.

src/App.test.js

```
// ...

describe('Table', () => {

  const props = {
    list : [
      { title : '1', author : '1', num_comments : 1, points : 2, objectID : 'y' },
      { title : '2', author : '2', num_comments : 1, points : 2, objectID : 'z' },
    ],
    sortKey : 'TITLE',
    isSortReverse : false,
  };

  // ...

});
```

De nouveau vous pourriez avoir besoin d'accepter les tests d'instantané échouant pour votre composant Table, car vous avez fourni des propriétés étendues pour le composant Table.

Enfin, votre interaction de tri avancé est maintenant terminée.

Exercises :

- utiliser une bibliothèque comme Font Awesome[144] pour indiquer le tri (inversé)
  - cela pourrait être une icône de flèche vers le haut ou de flèche vers le bas à côté de chaque en-tête Sort
- lire plus à propos de la bibliothèque classnames[145]

144. https://fontawesome.io/
145. https://github.com/JedWatson/classnames

Vous avez appris les techniques de composant avancées dans React ! Récapitulons les derniers chapitres :

- React
  - l'attribut `ref` pour référencer les éléments du DOM
  - les components d'ordre supérieur sont un moyen courant de construire des composants avancés
  - l'implementation d'interactions avancées dans React
  - les classNames conditionnels avec une élégante bibliothèque assistante
- ES6
  - la décomposition du reste pour séparer les objets et tableaux

Vous pouvez trouver le code source sur le dépôt officiel[146].

146. https://github.com/the-road-to-learn-react/hackernews-client/tree/5.5

Gestion de l’état au sein de React et au-delà

Vous avez déjà appris les basiques de la gestion d’état dans React dans les chapitres précédents. Ce chapitre creuse ce sujet un peu plus profondément. Vous apprendrez les meilleurs pratiques, comment les appliquer et pourquoi vous pouvez envisager l’usage d’une bibliothèque tierce de gestion d’état.

L'élévation de l'état

Seule le composant App est un composant ES6 stateful dans votre application. Il gère beaucoup d'aspects de l'état et de la logique de l'application au sein de ses méthodes. Peut-être avez-vous remarqué que vous passez beaucoup de propriétés à votre composant Table. La plupart de ces propriétés sont seulement utilisées dans le composant Table. En conclusion une personne pourrez soutenir que c'est un non-sens que le composant App connaisse toutes ces propriétés.

L'entièreté de la fonctionnalité de tri est seulement utilisé dans le composant Table. Vous pourriez la migrer dans le composant Table, car le composant App n'en a pas du tout besoin. Le processus de refactoring en sous-état d'un composant vers un autre est connu comme *lifting state*. Dans votre cas, vous voulez migrer l'état qui n'est pas utilisé dans le composant App à l'intérieur du composant Table. L'état est déplacé vers le bas depuis le parent vers le composant enfant.

Dans le but de traiter avec l'état et les méthodes de classe dans le composant Table, il doit devenir un composant de classe ES6. Le refactoring d'un composant fonctionnel stateless vers un composant de classe ES6 est trivial.

Voici votre composant Table comme un composant fonctionnel stateless :

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return (
    // ...
  );
}
```

Voici votre composant Table comme un composant de classe ES6 :

src/App.js

```
class Table extends Component {
  render() {
    const {
      list,
      sortKey,
      isSortReverse,
      onSort,
      onDismiss
    } = this.props;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
     ? sortedList.reverse()
      : sortedList;

    return(
      // ...
    );
  }
}
```

Puisque vous voulez traiter l'état et les méthodes dans votre composant, vous devez ajouter un constructeur et un état initial.

src/App.js

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {};
  }

  render() {
    // ...
  }
}
```

Maintenant vous pouvez migrer l'état et les méthodes de classe concernant la fonctionnalité de tri depuis votre composant App vers votre composant Table de dessous.

src/App.js

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {
      sortKey : 'NONE',
      isSortReverse : false,
    };

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

  onSort(sortKey) {
    const isSortReverse = this.state.sortKey === sortKey &&!this.state.isSortRevers\
e;
    this.setState({ sortKey, isSortReverse });
  }

  render() {
    // ...
  }
}
```

N'oubliez pas de supprimer l'état déplacé et la méthode de classe `onSort()` de votre composant App.

src/App.js

```
class App extends Component {
  _isMounted = false;

  constructor(props) {
    super(props);

    this.state = {
      results : null,
      searchKey : '',
      searchTerm : DEFAULT_QUERY,
      error : null,
      isLoading : false,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
```

```
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
  }

  // ...

}
```

De plus, vous pouvez faire une API allégée du composant Table. Supprimez les propriétés qui lui étaient passées depuis le composant App, car elles sont maintenant gérées dans le composant Table de façon interne.

src/App.js

```
class App extends Component {

  // ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;

    // ...

    return (
      <div className="page">
        // ...
        { error
         ? <div className="interactions">
            <p>Something went wrong.</p>
          </div>
          : <Table
            list={list}
            onDismiss={this.onDismiss}
          />
        }
```

```
        // ...
      </div>
    );
  }
}
```

Maintenant dans votre composant Table vous pouvez utiliser la méthode interne `onSort()` et l'état interne de Table.

src/App.js

```
class Table extends Component {

  // ...

  render() {
    const {
      list,
      onDismiss
    } = this.props;

    const {
      sortKey,
      isSortReverse,
    } = this.state;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
     ? sortedList.reverse()
     : sortedList;

    return(
      <div className="table">
        <div className="table-header">
          <span style={{ width : '40%' }}>
            <Sort
              sortKey={'TITLE'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Title
            </Sort>
          </span>
          <span style={{ width : '30%' }}>
```

```
            <Sort
              sortKey={'AUTHOR'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Author
            </Sort>
          </span>
          <span style={{ width : '10%' }}>
            <Sort
              sortKey={'COMMENTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Comments
            </Sort>
          </span>
          <span style={{ width : '10%' }}>
            <Sort
              sortKey={'POINTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Points
            </Sort>
          </span>
          <span style={{ width : '10%' }}>
            Archive
          </span>
        </div>
        { reverseSortedList.map((item) =>
          // ...
        )}
      </div>
    );
  }
}
```

Votre application doit encore fonctionner. Mais vous avez fait un refactoring crucial. Vous avez déplacé la fonctionnalité et l'état plus près dans un autre composant. Les autres composants ont été allégés. De plus l'API du composant Table a été allégé car il traite la fonctionnalité de tri de façon interne.

Le processus de *lifting state* peut aller dans l'autre sens également : depuis le composant enfant vers le composant parent. Cela s'appelle le *lifting state up*. Imaginez-vous vous occupiez également de l'état interne dans un composant enfant. Maintenant vous devriez satisfaire également un besoin de montrer l'état dans votre composant parent. Vous devriez élever l'état vers votre composant parent. Mais cela va plus loin. Imaginez vous souhaitez montrer l'état dans un composant frère de votre composant enfant. De nouveau vous devriez élever l'état vers votre composant parent. Le composant parent traite l'état interne, mais l'expose vers les deux composants enfants.

Exercises :

- lire plus à propos de l'élévation de l'état en React[147]
- lire plus à propos du lifting state dans React avant l'utilisation de Redux[148]

147. https://reactjs.org/docs/lifting-state-up.html
148. https://www.robinwieruch.de/learn-react-before-using-redux/

setState() : revisité

Jusqu'ici vous devez utiliser `setState()` de React pour gérer votre état de composant interne. Vous pouvez transmettre un objet à une fonction qui vous permet de mettre à jour partiellement l'état interne.

Code Playground

```
this.setState({ foo : bar });
```

Mais `setState()` ne prend pas seulement un objet. Dans sa seconde version, vous pouvez transmettre une fonction pour mettre à jour l'état.

Code Playground

```
this.setState((prevState, props) => {
  // ...
});
```

Pourquoi vouloir faire cela ? Il y a un des cas d'utilisation cruciaux où il fait sens d'utiliser une fonction à la place d'un objet. C'est quand vous mettez à jour l'état en fonction de votre état précédent ou des propriétés. Si vous n'utilisez pas une fonction, la gestion interne d'état peut engendrer des bugs.

Mais pourquoi cela cause des bugs d'utiliser un objet à la place d'une fonction lorsque la mise à jour dépend de l'état précédent ou des propriétés ? La méthode `setState()` de React est asynchrone. React regroupe les appelles de `setState()` et les exécute finalement. Il peut arriver que l'état précédent ou que les propriétés soient modifiés entre-temps alors vous souhaitez vous fier à eux dans votre appelle `setState()`.

Code Playground

```
const { fooCount } = this.state;
const { barCount } = this.props;
this.setState({ count : fooCount + barCount });
```

Imaginez `fooCount` et `barCount`, c'est-à-dire l'état ou les propriétés, changent autre part de façon asynchrone lorsque vous appelez `setState()`. Dans une application grandissante, vous avez plus qu'un appelle à `setState()` au travers de votre application. Puisque `setState()` s'exécute de façon asynchrone, vous pourriez dépendre dans cet exemple de valeurs périmées.

Avec l'approche par fonction, la fonction dans `setState()` est une fonction de rappel (callback) qui opère sur l'état et les propriétés au moment de l'exécution de la fonction de rappel. Bien que `setState()` soit asynchrone, avec une fonction il prend l'état et les propriétés au moment de quand c'était exécuté.

Code Playground

```
this.setState((prevState, props) => {
  const { fooCount } = prevState;
  const { barCount } = props;
  return { count : fooCount + barCount };
});
```

Maintenant, retournons à votre code pour réparer ce comportement. Ensemble nous allons le réparer à un endroit où `setState()` est utilisé et dépend de l'état et des propriétés. Après quoi, vous serez capable de l'appliquer à d'autres endroits également.

La méthode `setSearchTopStories()` dépend de l'état précédent c'est donc un parfait exemple pour utiliser une fonction à la place d'un objet dans `setState()`. Pour l'heure, cela ressemble au bout de code suivant.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  const { searchKey, results } = this.state;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    results : {
      ...results,
      [searchKey] : { hits : updatedHits, page }
    },
    isLoading : false
  });
}
```

Vous extrayez les valeurs de l'état, mais vous mettez à jour l'état en fonction de l'état précédent de façon asynchrone.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    // ...
  });
}
```

Vous pouvez déplacer le bloc en entier que vous avez déjà implémenté à l'intérieur de la fonction. Vous avez seulement à remplacer le fait que vous opérez sur le `prevState` plutôt que `this.state`.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    const { searchKey, results } = prevState;

    const oldHits = results && results[searchKey]
     ? results[searchKey].hits
     : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    return {
      results : {
        ...results,
        [searchKey]: { hits : updatedHits, page }
      },
      isLoading : false
    };
  });
}
```

Cela réparera notre problème d'état périmé. Il y a une amélioration supplémentaire. Puisque c'est une fonction, vous pouvez extraire la fonction pour un gain de lisibilité. C'est l'un des avantages d'utiliser une fonction au lieu d'un objet. La fonction peut vivre à l'extérieur du composant. Mais vous devez utiliser une fonction d'ordre supérieur pour transmettre le résultat. En fin de compte, vous souhaitez mettre à jour l'état basé sur le résultat rapporté de l'API.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  this.setState(updateSearchTopStoriesState(hits, page));
}
```

La fonction `updateSearchTopStoriesState()`retourne une fonction. C'est une fonction d'ordre supérieur. Vous pouvez définir cette fonction d'ordre supérieur à l'extérieur de votre composant App. Notez comment la signature de la fonction change légèrement.

src/App.js

```
const updateSearchTopStoriesState = (hits, page) => (prevState) => {
  const { searchKey, results } = prevState;

  const oldHits = results && results[searchKey]
   ? results[searchKey].hits
   : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  return {
    results : {
      ...results,
      [searchKey]: { hits : updatedHits, page }
    },
    isLoading : false
  };
};

class App extends Component {
  // ...
}
```

C'est tout. La fonction à la place de l'approche objet dans `setState()` résout de potentiels bugs et améliore la lisibilité et maintenabilité de votre code. De plus, elle devient testable à l'extérieur du composant App. Vous pouvez l'exporter et écrire un test pour vous exercer.

Exercise :

- lire plus à propos de l'utilisation correcte de l'état dans React[149]
- exporter updateSearchTopStoriesState du fichier
- écrire un test pour cela qui passe un *payload* (hits, page) et un état précédent crée et finalement attendre un nouvel état
- refactorer vos méthodes `setState()` utiliser une fonction
  - mais seulement quand cela fait sens, c'est-à-dire lorsqu'il dépend de l'état ou des propriétés
- lancer vos tests de nouveau et vérifier que tout est à jour

149. https://reactjs.org/docs/state-and-lifecycle.html#using-state-correctly

Apprivoiser l'état

Les chapitres précédents vous ont montrés que la gestion de l'état peut être un sujet crucial dans les applications imposantes. De façon générale, pas seulement React mais beaucoup de frameworks SPA luttent avec çà. Les applications deviennent plus complexes depuis quelques années. Un des gros challenges dans les applications web de nos jours est d'apprivoiser et contrôler l'état.

Comparé aux autres solutions, React à déjà fait un grand pas en avant. Le flux de données unidirectionnel et une API simple pour gérer l'état dans un composant sont indispensables. Ces concepts le rendent plus simple pour raisonner au sujet de votre état ainsi que de vos changements d'états. Cela rend plus simple de raisonner au niveau composant et d'un certain point au niveau applicatif.

Dans une application grandissante, cela devient plus difficile de raisonner au sujet des changements d'état. Vous pouvez introduire des bugs en agissant sur des états périmés lors de l'utilisation d'un objet à la place d'une fonction dans `setState()`. Vous devez élever l'état pour partager le nécessaire ou cacher l'état non nécessaire au travers des composants. Il peut arriver qu'un composant est besoin d'élever son état vers le haut, car ses composants frères dépendent de ce dernier. Peut-être le composant est éloigné dans l'arbre de composant et ainsi vous devrez partager l'état au travers de l'ensemble de l'arbre de composant. En conclusion les composants seront impliqués dans la gestion d'état dans de bien plus grandes mesures. Mais après tout, la principale responsabilité des composants devraient être la représentation de l'UI (interface utilisateur), n'est ce pas ?

Pour toutes ces raisons, il existe des solutions autonomes pour prendre en charge la gestion de l'état. Ces solutions ne sont pas seulement utilisées dans React. Cependant, cela fait de l'écosystème React un outil extrêmement puissant. Vous pouvez utiliser différentes solutions pour résoudre vos problèmes. Confier le problème de la mise à l'échelle de la gestion de l'état, vous pourriez avoir entendu parler des bibliohtèques Redux[150] ou MobX[151]. Vous pouvez utiliser l'une de ces solutions dans une application React. Elles arrivent avec des extensions, react-redux[152] et mobx-react[153], pour les intégrer à l'intérieur de la couche vue de React.

Redux et MobX sont en dehors du cadre de ce livre. Lorsque vous avez fini le livre, vous obtiendrez des conseils sûrs comment vous pouvez continuer à apprendre React et son écosystème. Un cheminement d'apprentissage pourrait être l'apprentissage de Redux. Avant que vous plongez dans le sujet de la gestion d'état externe, je peux vous recommander de lire cette article[154]. Il a pour but de vous donner une meilleure compréhension de comment apprendre la gestion d'état externe.

Exercices :

- lire plus à propos de la gestion d'état externe et de comment l'apprendre[155]

150. http://redux.js.org/docs/introduction/
151. https://mobx.js.org/
152. https://github.com/reactjs/react-redux
153. https://github.com/mobxjs/mobx-react
154. https://www.robinwieruch.de/redux-mobx-confusion/
155. https://www.robinwieruch.de/redux-mobx-confusion/

- regarder mon second livre à propos de la gestion de l'état dans React[156]

156. https://roadtoreact.com/

Vous avez appris la gestion d'état avancé dans React ! Récapitulons les derniers chapitres :

- React
  - élévation de la gestion de l'état haut et bas vers les composants appropriés
  - `setState()` peut utiliser une fonction pour empêcher des bugs d'état périmé
  - les solutions externes existantes qui vous aident à apprivoiser l'état

Vous pouvez trouver le code source sur le dépôt officiel[157].

157. https://github.com/the-road-to-learn-react/hackernews-client/tree/5.6

Dernière étape pour la mise en production

Les derniers chapitres vous montreront comment déployer votre application en production. Vous utiliserez le service d'hébergement gratuit Heroku. Lors du déploiement de votre application, vous en apprendrez plus sur *create-react-app*.

Eject

L'étape suivante et les connaissances liées ne sont **pas nécessaires** pour déployer votre application en production. Cependant je souhaite vous l'expliquer. *create-react-app* arrive avec une fonctionnalité qui le rend extensible mais aussi empêche une clientèle captive. Une clientèle captive se produit habituellement lorsque vous achetez une technologie mais qu'il n'y a pas de trappe d'évacuation pour son usage futur. Heureusement dans *create-react-app* vous disposez d'une telle trappe à l'aide d'**eject**.

Dans votre *package.json* vous trouverez les scripts pour *start*, *test* et *build* votre application. Le dernier script est *eject*. Vous pouvez l'essayer, mais il n'y a aucun moyen de faire chemin inverse. **C'est une opération à sens unique. Une fois que vous éjectez, vous ne pouvez revenir en arrière !** si vous venez juste de démarrer l'apprentissage de React, cela n'a aucun intérêt de quitter l'environnement commode de *create-react-app*.

Si vous souhaitez lancer `npm run eject`, la commande fera une copie de toute la configuration et des dépendances vers votre *package.json* et un nouveau dossier *config/*. Vous souhaiteriez convertir le projet complet dans une installation customisée comprenant l'outillage incluant Babel et Webpack. Après quoi, vous possèderiez un contrôle absolu sur tous ces outils.

La documentation officielle dit que *create-react-app* est adaptée pour des projets de petite à moyenne taille. Vous ne devez pas vous sentir obligé d'utiliser la commande "eject".

Exercices :

- lire plus à propos d'[eject]((https ://github.com/facebook/create-react-app/blob/master/packages/react-scripts/template/README.md#npm-run-eject)

Déployer votre application

Au bout du compte, aucune application ne devrait rester en local. Vous souhaitez la mettre en ligne. Heroku est un PaaS (Platform as a Service) où vous pouvez héberger votre application. Ils offrent une intégration transparente avec React. Pour être plus précis : il est possible de déployer une application *create-react-app* en quelques minutes. C'est un déploiement avec zéro-configuration qui suit la philosophie de *create-react-app*.

Vous avez besoin de remplir deux prérequis avant de pouvoir déployer votre application sur Heroku :

- installer le CLI d'Heroku[158]
- créer un compte Heroku gratuit[159]

Si vous avez installé Homebrew, vous pouvez installer le CLI d'Heroku en ligne de commande :

Command Line

```
brew update
brew install heroku-toolbelt
```

Maintenant vous pouvez utiliser Git et le CLI d'Heroku pour déployer votre application.

Command Line

```
git init
heroku create -b https ://github.com/mars/create-react-app-buildpack.git
git add .
git commit -m "react-create-app on Heroku"
git push heroku master
heroku open
```

C'est tout. Maintenant, j'espère que votre application a démarré et est en cours d'exécution. Si vous rencontrez des problèmes vous pouvez vérifier les ressources suivantes :

- Git et GitHub Essentials[160]
- Déploiement de React avec zéro-configuration[161]
- Heroku Buildpack pour create-react-app[162]

158. https://devcenter.heroku.com/articles/heroku-cli
159. https://www.heroku.com/
160. https://www.robinwieruch.de/git-essential-commands/
161. https://blog.heroku.com/deploying-react-with-zero-configuration
162. https://github.com/mars/create-react-app-buildpack

Plan

C'était le dernier chapitre du livre. J'espère que vous avez apprécié la lecture et qu'elle vous a aidé à saisir React. Si vous avez aimé le livre, partagez-le comme moyen d'apprendre React auprès de vos amis. Il pourrait tout à fait être un cadeau. De plus, cela représente beaucoup pour moi que vous prenez 5 minutes pour écrire un avis sur Amazon[163] ou Goodreads[164].

Mais vers où aller après avoir lu ce livre ? Vous pouvez soit approfondir l'application de vous-même ou partager votre propre projet React. Avant que vous plongez dans un autre livre, cours ou tutoriel, vous devriez créer votre propre projet React en pratique. Faite le pendant une semaine, mettez-la en production en le déployant quelque part, et contactez-moi sur Twitter[165] pour l'exposer. Je suis curieux de ce que vous allez construire après que vous ayez lu le livre.

Si vous recherchez des voies de poursuites supplémentaires pour votre application, je peux recommander quelques chemins d'apprentissage après que vous ayez utilisé seulement du simple React dans ce livre :

- **La gestion d'état** : Vous avez utilisé `this.setState()` et `this.state` de React pour manager et accéder à l'état local du composant. C'est parfait pour débuter. Cependant, dans une application plus imposante vous rencontrerez les limites de l'état local de composant React[166]. Par conséquent, vous pouvez utiliser une bibliothèque tierce de gestion d'état telle que [Redux ou MobX] (https ://www.robinwieruch.de/redux-mobx-confusion/). Sur la plateforme de cours Road to React[167], vous trouverez "Taming the State in React" qui apprend l'état local dans React de façon avancé, Redux et MobX. Le cours arrive avec un ebook également, mais je recommande tout le monde de se plonger dans le code source et aussi dans les captures d'écrans. Si vous avez aimé ce livre, vous devrez sans aucun doute regarder "Taming the State in React".
- **Se connecter à une base de données et/ou s'authentifier** : Dans une application React grandissante, vous pourrez finalement désirer persister vos données. Les données devraient être stockées dans une base de données ainsi elles peuvent survivre après la session du navigateur et être partagé auprès de divers utilisateurs utilisant l'application. La manière la plus simple d'introduire une base de données est l'utilisation de Firebase. Dans ce tutoriel abordable[168], vous trouverez un guide étape par étape sur comment utiliser l'authentification de Firebase (inscription, identification, déconnexion, ...) au sein de React. Au-delà de cela vous utiliserez la base de données en temps réel de Firebase pour enregistrer les entités des utilisateurs. Après quoi, c'est à vous de stocker plus de données issues de votre application.

163. https://www.amazon.com/dp/B077HJFCQX
164. https://www.goodreads.com/book/show/37503118-the-road-to-learn-react
165. https://twitter.com/rwieruch
166. https://www.robinwieruch.de/learn-react-before-using-redux/
167. https://roadtoreact.com/
168. https://www.robinwieruch.de/complete-firebase-authentication-react-tutorial/

- **Outillage avec Webpack et Babel** : Dans le livre vous avez utilisé *create-react-app* pour installer votre application. À certains moments, une fois que vous avez appris React, vous pourriez être amené à apprendre l'outillage autour. Cela vous permet d'installer votre propre projet sans *create-react-app*. Je peux recommander de suivre une installation minimale avec Webpack and Babel[169]. Après quoi vous pouvez appliquer de vous-même plus d'outils. Par exemple, vous pourrez utiliser ESLint[170] pour contraindre un code style unifié au sein de votre application.
- **Syntaxe de composant React** : Les possibilités et bonnes pratiques pour implémenter les composants React évoluent au fil du temps. Vous trouverez dans d'autres ressources d'apprentissages, plusieurs manières d'écrire vos composants React, en particulier les composants de classe React. Vous pouvez vous renseignez sur ce dépôt GitHub[171] pour trouver des façons alternatives d'écrire les composants de classe React. En utilisant les déclarations de champs de classe, vous pouvez les écrire encore plus concisément dans le futur.
- **Projets d'échantillon** : Après l'apprentissage en pur React, c'est toujours bon d'appliquer les connaissances en premier au sein de votre projet avant de reprendre l'apprentissage de nouveau. Vous pouvez écrire votre propre jeu tic-tac-toe ou une simple calculette en React. Il y a beaucoup de tutoriels par-ci par-là qui utilisent React pour construire des choses formidables. Regardez ma construction [d'une liste paginée à défilement infini] (https ://www.robinwieruch.de/react-paginated-list/), ou connecter votre application React à des cartes bancaires pour recharger de l'argent[172]. Expérimentez avec ces mini-applications pour être à l'aise avec React.
- **Composants UI** : Vous ne devez pas faire l'erreur d'introduire trop tôt une bibliothèque de composant UI dans votre projet. Premièrement, vous devez apprendre comment implémenter et utiliser à partir de rien un *dropdown*, une *checkbox* ou un *dialog* au sein de React avec des éléments HTML standards. La major partie de ces composants gèreront leur propre état local. Une *checkbox* doit connaître si elle est coché ou non coché. Ainsi vous devez les implémenter en tant que composants controllés. Après que vous ayez fait face à toutes les implémentations fondamentales, vous pouvez introduire une bibliohtèque de composant UI qui vous fournit des composants React *checkboxes* et *dialogs*. You shouldn't make the mistake to introduce too early a UI component library in your project. First, you should learn how to implement and use a dropdown, checkbox or dialog in React with standard HTML elements from scratch. The major part of these components will manage their own local state. A checkbox has to know whether it is checked or not checked. Thus you should implement them as controlled components. After you went through all the foundational implementations, you can introduce a UI component library which gives you checkboxes and dialogs as React components.
- **Organisation du code** : Au cours de la lecture du livre, vous avez rencontré un chapitre portant sur l'organisation du code. Vous pouvez appliquer ces changements maintenant, si vous n'avez pas encore eu l'occasion de le faire. Cela organisera vos composants au sein de fichiers et dossiers (modules) structurés. De plus, cela vous aidera à comprendre et apprendre les principes de séparation de code, réutilisabilité, maintenabilité et de conception de d'API de

169. https://www.robinwieruch.de/minimal-react-webpack-babel-setup/
170. https://www.robinwieruch.de/react-eslint-webpack-babel/
171. https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax
172. https://www.robinwieruch.de/react-express-stripe-payment/

module. Finalement, votre application grossira et vous aurez besoin de la structurer en modules. Donc il est bon pour vous de commencer dès aujourd'hui.

- **Testing** : Le livre aborde seulement en surface la phase de test. Si vous n'êtes pas famillié avec ce thème en général, vous pouvez approfondir les concepts de test unitaire et de test d'intégration, notamment dans le contexte d'application React. Sur le plan de l'implémentation, je vous recommanderai de vous en tenir à Enzyme et Jest dans le but d'améliorer votre approche des tests avec les tests unitaires et les snapshots test de React.
- **Routage** : Vous pouvez implémenter un routage pour votre application avec react-router[173]. Jusqu'ici, vous avez seulement une page dans votre application. Le React Router vous aide à obtenir plusieurs pages avec plusieurs URLs. Quand vous introduisez le routage pour votre application, vous ne réalisez aucune requête auprès de votre serveur web pour requêter la prochaine page. Le routeur fera tout pour vous côté client. Donc essayez vous et mettez vos mains dans le cambouis en introduisant le routage dans votre application.
- **Type Checking** : Dans un chapitre, vous avez utilisé React PropTypes pour définir les interfaces des composants. C'est en général une bonne pratique pour éviter les bugs. Mais les PropTypes sont vérifiés seulement à l'exécution. Vous pouvez pousser le curseur plus loin en introduisant la vérification à la compilation par typage statique. TypeScript[174] est une solution populaire. Mais dans l'écosystème React, les utilisateurs utilisent souvent Flow[175]. Je peux recommander de prêter une attention à Flow si vous êtes intéressé par l'obtention d'application plus robuste.
- **React Native** : React Native[176] amène votre application dans les périphériques mobiles. Vous pouvez appliquer vos acquis de React pour délivrer des applications IOS et Android. La courbe d'apprentissage, une fois React apprit, ne devrez pas être si abrupte pour React Native. Les deux partages les mêmes principes. Vous rencontrerez seulement différent niveau de composants sur mobile que ce que vous avez utilisé au sein d'applications web.

En général, je vous invite à visiter mon site[177] pour trouver des sujets intéressant à propos du développement web et de l'ingénierie logiciel. Vous pouvez souscrire à ma newsletter[178] pour être mis au courant sur des articles, livres et cours. De plus, la plateforme de cours Road to React[179] offre des cours plus avancés pour apprendre l'écosystème React. Foncez !

Enfin mais pas des moindres, j'espère trouver plus de Patrons[180] qui sont prêts à supporter mon contenu. Il y a beaucoup d'étudiants là-bas qui ne peuvent pas se permettre de payer du contenu éducatif. C'est pourquoi je poste beaucoup de contenus gratuits là-bas gratuitement. En me supportant dans mes actions en tant que Patron, je peux durablement continuer ces efforts d'éducation gratuitement.

Une fois encore, si vous aimez le livre, je souhaiterais que vous preniez le temps de penser à propos de potentielle personne qui pourrait entreprendre l'apprentissage de React. Entrer en contact avec

173. https://github.com/ReactTraining/react-router
174. https://www.typescriptlang.org/
175. https://flowtype.org/
176. https://facebook.github.io/react-native/
177. https://www.robinwieruch.de
178. https://www.getrevue.co/profile/rwieruch
179. https://roadtoreact.com
180. https://www.patreon.com/rwieruch

ces personnes et partager le livre. Cela représente énormément pour moi. Le livre est pensé dans un but de partage et d'ouverture. Il s'améliorera à chaque fois que des personnes le lisent et partagent leur feedback avec moi. J'espère également voir votre feedback, avis ou note !

Merci beaucoup pour avoir lu the Road to learn React.

Robin