by robin wieruch

the Road to learn React

The Road to learn React (Português)

Sua jornada para dominar React de forma simples e pragmática

Robin Wieruch e Claudio Romero

Esse livro está à venda em http://leanpub.com/the-road-to-learn-react-portuguese

Essa versão foi publicada em 2018-07-31

Tweet Sobre Esse Livro!

Por favor ajude Robin Wieruch e Claudio Romero a divulgar esse livro no Twitter!

O tweet sugerido para esse livro é:

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

A hashtag sugerida para esse livro é #ReactJs.

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

#ReactJs

Conteúdo

Prefácio

The Road to learn React (O Caminho para aprender React, em português) ensina os fundamentos de React. Ao longo do livro, você irá construir uma aplicação de verdade em React puro, sem fazer uso de ferramentas complicadas. Tudo será explicado, da configuração do projeto, até a sua implantação em um servidor.

O livro traz referências à exercícios e materiais de leitura adicional em cada capítulo. Depois de lê-lo, você será capaz de construir suas próprias aplicações em React. Os recursos são mantidos atualizados por mim e pela comunidade.

Minha intenção neste livro é oferecer uma base sólida, antes de mergulhar em um ecossistema mais amplo. Poucas ferramentas e gerenciamento externo de estado, mas muita informação a respeito de React. Ele explica conceitos gerais, padrões e melhores práticas em uma aplicação React do mundo real.

Você irá aprender a construir sua própria versão do Hacker News. Ela cobre funcionalidades como paginação, *client-side caching* e operações de busca e ordenação. Além disso, irá paulatinamente fazer a transição de JavaScript ES5 para JavaScript ES6. Espero que este livro transmita meu entusiasmo por React e JavaScript e ajude você a ser iniciado.

Sobre o Author

Robin Wieruch é um engenheiro de software alemão, que se dedica a aprender e ensinar programação em JavaScript. Depois de obter seu mestrado em ciência da computação, ele continuou estudando por conta própria, diariamente. Ganhou experiência no mundo das *startups*, onde ele utilizou excessivamente JavaScript, não só quando estava trabalhando, mas também no seu tempo livre. Isso lhe deu a oportunidade de ensinar os outros sobre estes tópicos.

Por alguns anos, Robin trabalhou ao lado de um grande time de engenheiros, em uma companhia chamada Small Improvements[1], desenvolvendo aplicações de larga escala. Esta companhia oferece um produto no formato SaaS, que permite que clientes dêem o seu *feedback* para empresas, desenvolvida usando JavaScript na camada de *frontend* e Java na de *backend*. No *frontend*, a primeira funcionalidade foi escrita em Java com o *Wicket Framework* e jQuery. Quando a primeira geração de SPAs tornou-se popular, a companhia migrou para Angular 1.x. Depois de utilizar Angular por mais de dois anos, tornou-se claro que esta não era a melhor solução da época para se trabalhar com aplicações fortemente baseadas em estado. Por esta razão, a empresa saltou para React e Redux, o que lhe permitiu operar em larga escala com sucesso.

Durante seu tempo na empresa, Robin escreveu artigos sobre desenvolvimento *web* regularmente no seu *website*. Ele recebeu ótimo *feedback* das pessoas sobre eles e isso lhe permitiu melhorar seu estilo de escrita e ensino. Artigo após artigo, Robin evoluiu sua habilidade de ensinar para outras pessoas. O primeiro tinha sido montado com muitas coisas juntas, o que sobrecarregava os estudantes. Mas, com o tempo, ele foi aperfeiçoando-se, focando e ensinando apenas um assunto.

Atualmente, Robin é um professor autônomo. Para ele, ver seus alunos prosperarem com os objetivos claros e os ciclos curtos de *feedback*, providos por ele, é algo que lhe completa. Esta é, afinal, uma coisa que você provavelmente aprenderia em uma empresa especializada em *feedback*, não é mesmo? Mas, se ele também não estivesse codificando, não seria capaz de ensinar nada. Por isso, ele investe o tempo que lhe sobra em programação. Você pode encontrar mais informações sobre Robin e formas de apoiar o seu trabalho em seu website[2].

[1] https://www.small-improvements.com/
[2] https://www.robinwieruch.de/about

Depoimentos

Muhammad Kashif[3]: "O Caminho para Aprender React é um livro único, que eu recomendo para qualquer estudante ou profissional interessado em aprender react, do nível básico ao avançado. Ele é recheado de dicas pertinentes e técnicas difíceis de achar em qualquer outro lugar, notavelmente completo no seu uso de exemplos e referências para modelos de problemas, eu tenho 17 anos de experiência em desenvolvimento de aplicações web e desktop, e antes de ler este livro eu estava tendo problemas em aprender react, mas este livro funciona como mágica."

Andre Vargas[4]: "O Caminho para Aprender React de Robin Wieruch é um livro impressionante! Muito do que eu aprendi sobre React e até ES6 foi através dele!"

Nicholas Hunt-Walker, Instrutor de Python na Seattle Coding School[5]: "Esse é um dos livros mais bem escritos e informativos com o qual eu já trabalhei. Uma sólida introdução à React & ES6."

Austin Green[6]: "Obrigado, realmente amei o livro. Mistura perfeita para aprender React, ES6, e conceitos avançados de programação."

Nicole Ferguson[7]: "Estou fazendo o curso O Caminho para Aprender React de Robin este final de semana & eu quase que me sinto culpada por me divertir tanto."

Karan[8]: "Acabo de terminar o seu Caminho para React. Melhor livro do mundo para iniciantes em React e JS. Exposição elegante de ES. Felicitações! :)"

Eric Priou[9]: "O Caminho para aprender React de Robin Wieruch é leitura obrigatória, simples e concisa para React e JavaScript."

Desenvolvedor Novato: "Acabo de terminar o livro mesmo sendo um desenvolvedor iniciante, obrigado por ter trabalhado nisso. Foi fácil de acompanhar e eu me sinto confiante para começar um novo *app* do zero nos próximos dias. O livro se mostrou muito melhor que o tutorial oficial do React.js que eu tentei anteriormente (e que não pude completar devido à escassez de detalhes). Os exercícios ao final de cada seção foram muito gratificantes."

Estudante: "O melhor livro para começar a aprender ReactJS. O projeto acompanha os conceitos aprendidos o que ajuda a pegar o assunto. Descobri que 'Codifique e aprenda' é o melhor jeito de dominar programação e este livro faz exatamente isso."

Thomas Lockney[10]: "Introdução sólida a React que não tenta ser muito abrangente. Eu queria apenas provar para entender de que se tratava e este livro me entregou exatamente isso. Não segui todas

[3] https://twitter.com/appsdevpk/status/848625244956901376
[4] https://twitter.com/andrevar66/status/853789166987038720
[5] https://twitter.com/nhuntwalker/status/845730837823840256
[6] https://twitter.com/AustinGreen/status/845321540627521536
[7] https://twitter.com/nicoleffe/status/833488391148822528
[8] https://twitter.com/kvss1992/status/889197346344493056
[9] https://twitter.com/erixtekila/status/840875459730657283
[10] https://www.goodreads.com/review/show/1880673388

as notas de rodapé para aprender a respeito das novas características de ES6 que havia perdido ("*I wouldn't say I've been missing it, Bob.*"). Porém, tenho certeza que aqueles de vocês que pararam e são aplicados para segui-las, provavelmente aprenderão muito mais do que apenas o que o livro ensina."

Educação para Crianças

Este livro é *open source* e deveria permitir que qualquer um aprenda React. Contudo, nem todos são privilegiados para fazer uso de recursos *open source*, porque nem todo mundo é educado primariamente na língua inglesa. Mesmo que este livro seja um "pague o quanto quiser", eu gostaria de utilizá-lo para apoiar projetos que ensinam inglês para crianças no mundo em desenvolvimento.

- 11 a 18 de Abril de 2017, Giving Back, By Learning React[11]

[11] https://www.robinwieruch.de/giving-back-by-learning-react/

FAQ

Como recebo atualizações? Eu tenho dois canais onde compartilho atualizações de conteúdo. Você pode tanto subscrever à atualizações por e-mail[12], quanto seguir-me no Twitter[13]. Independente do canal escolhido, meu objetivo é compartilhar apenas conteúdo qualitativo. Você nunca receberá *spam*. Uma vez que o livro recebe uma atualização, você poderá baixá-la.

O livro usa a versão mais recente de React? O livro sempre é atualizado quando uma nova versão de React é lançada. De modo geral, livros ficam obsoletos logo após o lançamento. Mas, como este livro é de publicação própria, eu posso atualizá-lo quando quiser.

Ele aborda Redux? Não. Para isso, eu escrevi um segundo livro. "O Caminho para aprender React" deve lhe fornecer uma base sólida antes de você mergulhar em tópicos avançados. A implementação da aplicação de exemplo no livro irá mostrar que você não precisa de Redux para construir uma aplicação em React. Após ter terminado o livro, você deverá estar apto a implementar uma aplicação consistente sem Redux. Você então poderá ler meu segundo livro e aprender Redux[14].

Ele usa JavaScript ES6? Sim, mas não se preocupe. Você se sairá bem se já for familiarizado com JavaScript ES5. Todas as características de ES6, que eu descrevo nesta jornada para aprender React, irão fazer a transição de ES5 para ES6 no livro. Todas serão explicadas ao longo do caminho. O livro não ensina apenas React, mas também todas as funcionalidades de JavaScript ES6 que são úteis para React.

Como posso obter acesso ao código-fonte dos projetos e à série de *screencasts*? Se você comprou um dos pacotes estendidos, que lhe dão acesso ao código-fonte de projetos, à série de *screencasts*, ou qualquer conteúdo adicional, irá encontrá-los no seu *dashboard* do curso[15]. Se você comprou o curso em qualquer outro lugar, não na plataforma de cursos oficial Road to React[16], será preciso criar uma conta na plataforma, ir na página de *Admin* e entrar em contato comigo, utilizando um dos modelos de e-mail. Assim, eu poderei lhe matricular no curso. Se não comprou nenhum dos pacotes estendidos, pode me procurar a qualquer momento para fazer um *upgrade* de conteúdo e ganhar acesso à este materiais já citados.

Como posso obter ajuda enquanto leio o livro? Existe um Grupo no Slack[17] para pessoas que estão lendo o livro e você pode entrar no canal para obter ajuda ou ajudar outras pessoas. Afinal de contas, ajudar os outros também pode melhorar o seu aprendizado. Se ninguém estiver lá para ajudá-lo, você sempre poderá vir falar comigo.

[12] https://www.getrevue.co/profile/rwieruch
[13] https://twitter.com/rwieruch
[14] https://roadtoreact.com/course-details?courseId=TAMING_THE_STATE
[15] https://roadtoreact.com/my-courses
[16] https://roadtoreact.com
[17] https://slack-the-road-to-learn-react.wieruch.com/

Existe alguma seção de solução de problemas? Se você se deparar com problemas, por favor, junte-se ao grupo no Slack. Você também pode verificar os *issues* abertos no GitHub[18] para o livro. Talvez seu problema até já tenha sido levantado e você poderá achar a solução. Caso contrário, não hesite em abrir um novo *issue* onde você pode explicar sua dificuldade, talvez fornecer uma captura de tela e mais alguns detalhes (ex.: página do livro, versão do *node*). Eu tento enviar todas as correções nas próximas edições do livro.

Posso ajudar a melhorar o livro? Sim, eu adoraria ouvir seu *feedback*. Você pode simplesmente abrir um *issue* no GitHub[19], que pode ser uma melhoria técnica ou textual. Não sou falante nativo do inglês (idioma original do livro) e, por isso, qualquer ajuda é bem vinda. Você também pode abrir *pull requests* na página do GitHub.

Existe alguma garantia do tipo "seu dinheiro de volta"? Sim, existe a garantia de 100% do dinheiro de volta nos dois primeiros meses, caso ache que o livro não lhe serve. Por favor, entre em contato comigo (Robin Wieruch) para solicitar o reembolso.

Posso apoiar o projeto? Se você acredita no conteúdo que eu crio, pode me apoiar[20]. Fico grato também se você divulgar este livro e adoraria ter você como meu Patron no Patreon[21].

Qual é a sua motivação por trás do livro? Eu desejo ensinar sobre este tópico de forma consistente. É comum achar materiais *online* que não recebem atualizações, ou que ensinam apenas um pedaço de um tópico. Quando estão aprendendo algo novo, as pessoas têm dificuldades para achar recursos consistentes e atualizados. Eu quero dar esta experiência de aprendizado sempre atual. Além disso, eu espero poder apoiar minorias com meus projetos, dando a elas o conteúdo gratuitamente ou causando outros impactos[22]. Recentemente, tenho me sentido realizado por estar ensinando outras pessoas sobre programação. É uma atividade que significa muito para mim e que eu prefiro desempenhar, ao invés de qualquer outro emprego de 9h às 17h em alguma empresa. Por causa disso tudo, eu espero trilhar este caminho no futuro.

Sou encorajado a participar ativamente? Sim. Quero que você pare um momento para pensar em alguém que seria um bom candidato a aprender React. A pessoa pode já ter mostrado interesse, ou estar no meio do aprendizado ou ainda nem ter despertado a vontade de aprender ainda. Aborde essa pessoa e compartilhe o livro. Significaria muito para mim. Ele foi feito com a intenção de ser distribuído.

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

Registro de Mudanças

10 de janeiro de 2017:

- v2 Pull Request[23]
- ainda mais amigável para iniciantes
- 37% mais conteúdo
- 30% de conteúdo melhorado
- 13 novos (ou melhorados) capítulos
- 140 páginas de material
- + curso interativo do livro em educative.io[24]

08 de março de 2017:

- v3 Pull Request[25]
- 20% mais conteúdo
- 25% de conteúdo melhorado
- 9 novos capítulos
- 170 páginas de material

15 de abril de 2017:

- atualização para React 15.5

5 de julho de 2017:

- atualização para node 8.1.3
- atualização para npm 5.0.4
- atualização para create-react-app 1.3.3

17 de outubro de 2017:

- atualização para node 8.3.0
- atualização para npm 5.5.1
- atualização para create-react-app 1.4.1

[23] https://github.com/rwieruch/the-road-to-learn-react/pull/18
[24] https://www.educative.io/collection/5740745361195008/5676830073815040
[25] https://github.com/rwieruch/the-road-to-learn-react/pull/34

- atualização para React 16
- v4 Pull Request[26]
- 15% mais conteúdo
- 15% de conteúdo melhorado
- 3 novos capítulos (*Bindings*, Tratamento de Eventos, Tratamento de Erros)
- 190+ páginas de material
- +9 Projetos com código-fonte[27]

17 de fevereiro de 2018:

- atualização para node 8.9.4
- atualização para npm 5.6.0
- atualização pra create-react-app 1.5.1
- v5 Pull Request[28]
- mais formas de aprendizado
- material de leitura extra
- 1 novo capítulo (Axios ao invés de Fetch)

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

Como ler o livro?

Esta é uma tentativa minha de lhe ensinar enquanto você escreve uma aplicação. É um guia prático, não uma referência sobre React. Você irá escrever uma versão própria do Hacker News que interage com uma API de verdade. Entre os vários tópicos interessantes, estão o gerenciamento de estado em React, *caching* e operações como ordenações e buscas. Durante o processo, você irá aprender as melhores práticas e padrões em React.

O livro também lhe provê uma transição de JavaScript ES5 para JavaScript ES6. React adota muitas funcionalidades do JavaScript ES6 e eu gostaria de lhe mostrar como utilizá-las.

Em geral, um capítulo irá continuar de onde o anterior parou, ensinando-lhe alguma coisa nova. Não se apresse para terminar o livro. Você deve internalizar o conhecimento em cada passo. Você também pode aplicar suas próprias implementações e ler mais a respeito do tópico em questão. Ao final de cada capítulo, disponibilizo alguns exercícios e material de leitura. Se você quer aprender React, eu recomendo fortemente que leia o material extra e que "mele as mãos" com os exercícios. Sinta-se confortável com o que aprendeu antes de passar para o próximo assunto.

No fim, você irá ter uma aplicação React completa em produção e eu tenho grande interesse em ver seus resultados. Portanto, sinta-se livre para entrar em contato quando terminar. O último capítulo irá lhe prover várias opções para continuar sua jornada. Você irá encontrar muitos tópicos relacionados a React no meu website pessoal[29].

Como você está lendo este livro, suponho que seja novo em React. Isso é ótimo. Espero receber o seu *feedback* sobre como posso melhorar o material para que qualquer um possa aprender. Você pode fazer a diferença diretamente no GitHub[30] ou falar comigo no Twitter[31].

[29] https://www.robinwieruch.de/
[30] https://github.com/rwieruch/the-road-to-learn-react
[31] https://twitter.com/rwieruch

Contribuidores

Muitas pessoas tornaram possível que *The Road to learn React* fosse inscrito e melhorado recentemente. Ele é, atualmente, um dos livros sobre React.js mais baixados neste mundo. Originalmente, o livro foi escrito pelo engenheiro de software alemão Robin Wieruch[32]. Mas, suas várias traduções não teriam sido possíveis sem a ajuda de outros.

Esta versão em português foi traduzida por **Claudio R. L. Rocha**, brasileiro e também engenheiro de software. Claudio estuda e trabalha na construção de sistemas de computador há 15 anos e, recentemente, também tem tido o prazer de apoiar o ensino do desenvolvimento de aplicações React. Participa de atividades de orientação de turmas e de avaliação de projetos práticos sobre o tema, estando em contato constante com código-fonte JavaScript e React dos mais variados níveis, estilos e fontes. Ele acredita que transmitir conhecimento aos outros é a forma mais eficaz de aprender.

Siga Claudio no Twitter[33] e no Linkedin[34].

[32] https://www.robinwieruch.de/
[33] https://twitter.com/romerolrocha
[34] https://www.linkedin.com/in/claudiorlr/

Introdução a React

Você pode estar se perguntando: Por que eu deveria aprender React, pra começo de conversa? Este capítulo é uma introdução ao assunto e pode lhe dar a resposta que procura. Você irá mergulhar no ecossistema de React, montando sua primeira aplicação sem a necessidade de nenhuma configuração customizada e, durante o processo, terá uma introdução à JSX e ReactDOM. Prepare-se para seus primeiros componentes React.

Oi, meu nome é React.

Por que você deveria se incomodar em aprender React? Nos últimos anos, *single page applications* (SPA[35]) tornaram-se populares. *Frameworks* como Angular, Ember e Backbone ajudaram desenvolvedores JavaScript a construir aplicações web modernas, indo além do que já se costumava fazer com jQuery e JavaScript puro. Existe uma ampla gama de *frameworks* SPA e, olhando para suas respectivas datas de lançamento, a maioria pertence à primeira geração: Angular 2010, Backbone 2010 e Ember 2011.

A primeira versão de React foi lançada em 2013 pelo Facebook. Não como um *framework*, mas como uma biblioteca. Ele é apenas o "V" do MVC[36] (*model view controller*), que lhe permite renderizar componentes como elementos visualizáveis no *browser*. O ecossistema onde React se insere é que torna possível a construção de *single page applications*.

E por que considerar o uso de React em detrimento à primeira geração de *frameworks* SPA? Porque enquanto estes últimos tentam resolver várias coisas ao mesmo tempo, React apenas lhe ajuda a construir *views*. Uma biblioteca, não um *framework*, que segue uma simples idéia: a camada de visão é uma hierarquia de componentes.

Em React, você consegue manter o foco na criação das suas *views* antes de introduzir outros aspectos na aplicação. Cada outro aspecto é como uma peça acoplável na sua SPA. Essa divisão em partes integráveis é essencial para construir uma aplicação mais madura, trazendo duas vantagens:

Primeiro, você pode aprender passo-a-passo cada parte, sem precisar se preocupar em entender tudo de só uma vez. Bem diferente de um *framework*, que lhe entrega todas as peças para montar desde o início. Este livro foca em React como o primeiro bloco da construção da aplicação. Outros eventualmente serão adicionados.

Segundo, todas as partes são substituíveis. Isso torna o ecossistema de React um lugar de inovação. Várias soluções competem entre si e você pode escolher a que mais se aplica a você e ao contexto em que irá utilizá-la.

Frameworks SPA da primeira geração surgiram em um nível profissional e mais engessados. React permanece com um caráter mais inovador e é adotado por muitas empresas que estão na vanguarda do pensamento tecnológico, como Airbnb, Netflix e (obviamente) Facebook[37]. Todas elas investem no futuro de React e estão contentes com a biblioteca e com tudo que a cerca.

React é uma das melhores escolhas para a construção de aplicações web atualmente. Apesar de cuidar apenas da camada de visão, seu ecossistema forma um framework completo, flexível e intercambiável[38]. React tem uma API enxuta, um fantástico ecossistema e uma grande comunidade. Você pode ler sobre minhas experiências passadas em Por que saí de Angular e fui para React?[39]. Recomendo fortemente que você entenda a razão do porquê você escolheria React e não outra

[35] https://en.wikipedia.org/wiki/Single-page_application
[36] https://de.wikipedia.org/wiki/Model_View_Controller
[37] https://github.com/facebook/react/wiki/Sites-Using-React
[38] https://www.robinwieruch.de/essential-react-libraries-framework/
[39] https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/

biblioteca ou framework. Estamos todos ávidos pela experiência de seguir para onde React pode nos levar nos próximos anos.

Exercícios

- Leia mais sobre por que saí de Angular e fui para React[40]
- Leia mais sobre O ecossistema flexível de React[41]
- Leia mais sobre como aprender um *framework*[42]

[40] https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/
[41] https://www.robinwieruch.de/essential-react-libraries-framework/
[42] https://www.robinwieruch.de/how-to-learn-framework/

Pré-requisitos

Se você está migrando de um *framework* SPA ou de uma biblioteca diferente, já deve estar familiarizado com o básico de desenvolvimento para a *web*. Mas, se está começando agora, deveria pelo menos se sentir confortável com HTML, CSS e JavaScript ES5 para aprender React. Este livro irá fazer uma transição suave para JavaScript ES6 e além. Encorajo você a entrar para o Grupo no Slack[43] oficial para obter ajuda ou para ajudar outras pessoas.

Editor e Terminal

E quanto ao ambiente de desenvolvimento?

Você precisa de um editor ou IDE e uma ferramenta de linha de comando (terminal). Se quiser, siga meu guia de montagem de ambiente[44]. Ele foi feito para usuários de Mac OS, mas você pode encontrar ferramentas iguais ou equivalentes em outros sistemas operacionais. Existe também uma tonelada de artigos pela *web* que irão lhe mostrar como configurar um ambiente de desenvolvimento de uma forma mais elaborada de acordo com o seu SO.

Opcionalmente, você pode utilizar o git e o GitHub quando estiver praticando os exercícios do livro, para guardar seus projetos e monitorar o progresso em seus repositórios. Segue um pequeno guia[45] sobre como usar essas ferramentas. Mas, mais uma vez, não é obrigatório para acompanhar este guia e pode dar um pouco de trabalho caso você precise aprender tudo do começo. Se você é um novato no desenvolvimento *web*, pode pular esse passo e concentrar o foco nas partes essenciais ensinadas aqui.

Node e NPM

Por último, mas não menos importante, você precisará ter o node e o npm[46] instalados. Ambos são utilizados no gerenciamento de bibliotecas necessárias ao longo do caminho. Neste livro, você instalará pacotes node externos via npm (*node package manager*). Esses pacotes podem ser bibliotecas ou até *frameworks* completos.

É possível verificar as versões de node e npm instaladas pela linha de comando. Caso você não obtenha nenhum resultado no terminal, significa que precisará instalar ambos antes de continuar. Abaixo, minhas versões no momento em que escrevia este livro:

[43] https://slack-the-road-to-learn-react.wieruch.com/
[44] https://www.robinwieruch.de/developer-setup/
[45] https://www.robinwieruch.de/git-essential-commands/
[46] https://nodejs.org/en/

Linha de Comando

```
node --version
*v8.3.0
npm --version
*v5.5.1
```

node e npm

Esta seção do capítulo é um curso intensivo em node e npm. Ele não explora todas as funcionalidades, mas apresenta as ferramentas necessárias. Se você já está familiarizado com ambos, sinta-se livre para pular para o próximo assunto.

O **node package manager** (npm) possibilita a instalação de pacotes (**node packages**) externos pela linha de comando. Esses pacotes podem conter desde um conjunto de funções utilitárias até bibliotecas ou *frameworks* completos e, quando adicionados, tornam-se dependências da sua aplicação. Você pode instalá-los tanto globalmente (no seu diretório global de pacotes node), quanto na pasta local do seu projeto.

Pacotes node globais são acessíveis de qualquer pasta no terminal e você só precisa fazer a instalação de cada pacote uma vez. Para tanto, digite em seu terminal:

Linha de Comando

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

A *flag* `-g` diz ao npm para instalar o pacote globalmente.

Já os pacotes locais são usados apenas na sua aplicação. React, por exemplo, por se tratar de uma biblioteca, será adicionado como um pacote local que poderá ser importado para uso interno. A instalação de qualquer pacote local via terminal é feita através do comando:

Linha de Comando

```
1 npm install <package>
```

No caso específico de React, teríamos:

Linha de Comando

```
1 npm install react
```

O pacote instalado irá aparecer automaticamente em uma pasta chamada *node_modules/* e será listado no arquivo *package.json* juntamente com as outras dependências do projeto.

E como inicializar o projeto com a pasta *node_modules/* e o arquivo *package.json*? Existe um comando npm para isso também. Quando executado, *package.json* é criado e somente com a existência desse arquivo no projeto é que é possível a instalação de novos pacotes locais.

Linha de Comando

```
npm init -y
```

A *flag* `-y` é um atalho para inicializar seu *package.json* com as configurações padrão. Caso não a use, você terá que informar alguns dados adicionais para configurar o arquivo. Feita a inicialização, você está apto a instalar novos pacotes com o comando `npm install <package>`.

Mais uma coisa sobre o *package.json*: O arquivo habilita você a compartilhar o projeto com outros desenvolvedores sem ter que incluir todos os pacotes node. Ele contém todas as referências dos pacotes utilizados no projeto, chamados de dependências. Todos podem copiar seu projeto sem baixar as dependências e simplesmente instalar todos os pacotes utilizando `npm install` na linha de comando. O *script* considera todas as entradas listadas no *package.json* e instala cada uma na pasta *node_modules/*.

Existe mais um comando npm sobre o qual eu gostaria de falar a respeito:

Linha de Comando

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

A *flag* `--save-dev` indica que o pacote node é usado apenas em ambiente de desenvolvimento, ou seja, não será usado em produção quando você implantar a aplicação no servidor. Mas, que tipo de pacote se enquadraria neste caso?

Imagine que você queira instalar um pacote para lhe ajudar a testar sua aplicação. Você precisa instalá-lo via npm, mas deseja evitar que seja adicionado no seu ambiente de produção. Atividades de teste devem acontecer durante o processo de desenvolvimento, não quando a aplicação já foi disponibilizada para o usuário e já deveria ter sido testada e estar funcionando plenamente. Este é apenas um dos casos onde você desejaria utilizar a flag `--save-dev`.

Encontraremos mais comandos npm pelo caminho. Mas, por enquanto, é suficiente.

Uma última coisa, que considero importante mencionar: Muitas pessoas optam por utilizar outro gerenciador de dependências para trabalhar com pacotes *node* em suas aplicações. **Yarn** é um gerenciador que funciona de maneira muito similar ao **npm**. Ele tem a sua própria lista de comandos para executar as mesmas tarefas, mas você continua tendo acesso ao arquivo de pacotes do npm. Yarn nasceu para resolver alguns problemas que o npm não podia. Contudo, atualmente as duas ferramentas estão evoluindo realmente muito rápido e você pode escolher a que achar melhor.

Exercícios:

- Configurando um projeto npm qualquer:
  - Crie uma nova pasta com `mkdir <nome_da_pasta>`
  - Navegue para ela com `cd <nome_da_pasta>`

  - Execute `npm init -y` ou `npm init`
  - Instale um pacote local como React com `npm install react`
  - Olhe os conteúdos do arquivo *package.json* e da pasta *node_modules/*
  - Descubra você mesmo como desinstalar o pacote *react*
- Leia mais sobre npm[47]
- Leia mais sobre yarn[48]

[47] https://docs.npmjs.com/
[48] https://yarnpkg.com/en/docs/

Instalação

Existem muitas formas de começar a trabalhar com uma aplicação React.

A primeira delas é usar um CDN. Isso pode soar mais complicado do que realmente é, mas CDN é apenas a sigla para content delivery network[49]. Muitas empresas possuem CDNs que hospedam arquivos publicamente para que as pessoas possam consumi-los. Esses arquivos podem ser de bibliotecas como React, já que toda a biblioteca é empacotada em um simples arquivo JavaScript *react.js*. Ele pode ser hospedado em algum lugar e você pode requisitá-lo em sua aplicação.

Como usar um CDN para começar a trabalhar com React? Simples. Você pode adicionar a tag `<script>` *inline* no seu HTML, apontando para a url do CDN. Serão precisos dois arquivos (bibliotecas): *react* e *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>
```

Mas, eis a questão: Por que você deveria usar um CDN quando tem o npm para instalar pacotes como React?

Quando sua aplicação possui um arquivo *package.json* em uma pasta inicializada como projeto npm usando `npm init -y`, você pode instalar *react* e *react-dom* pelo terminal. É possível, inclusive, instalar múltiplos pacotes em apenas uma linha com o npm.

Linha de Comando

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

Essa é uma abordagem frequentemente utilizada para adicionar React a uma aplicação existente, caso esta seja gerenciada com npm.

Mas, infelizmente, isso não é o bastante. Você teria que configurar também o Babel[50] para fazer com que sua aplicação seja capaz de reconhecer JSX (a sintaxe React) e JavaScript ES6. O Babel "*transpila*" (tradução informal adotada pela comunidade para *transpile*) o seu código para que navegadores possam interpretar ES6 e JSX, pois nem todos conseguem fazê-lo naturalmente. Isso demanda muita configuração e uso de ferramentas, podendo ser aterrador para iniciantes em React lidarem com tudo isso.

Por esta razão, o Facebook introduziu *create-react-app* como uma solução de trabalho com React sem a necessidade de escrever configurações. O próximo capítulo lhe mostrará como montar sua aplicação utilizando essa ferramenta de inicialização (*bootstrap tool*).

[49] https://en.wikipedia.org/wiki/Content_delivery_network
[50] http://babeljs.io/

- **Nota do Tradutor**: De agora em diante, ao longo do livro original, aparecerão cada vez mais termos como *bootstraping* e *transpile*, que não possuem uma tradução exata na língua portuguesa. Além disso, em alguns casos, apesar de existir a palavra, o resultado da tradução de um termo técnico também pode soar estranho para estudantes e profissionais da área. Sendo assim, quando convir, manterei a palavra original e explicarei entre parênteses, se for necessário. Quando for possível e soar naturalmente, irei traduzir a sentença para algo de mesmo sentido no português.

Exercícios:

- Leia mais a respeito de instalações de React[51]

[51] https://facebook.github.io/react/docs/installation.html

Setup sem nenhuma configuração

Em "The Road to learn React", você usará create-react-app[52] para montar a estrutura inicial da sua aplicação. Trata-se de um *kit* de inicialização sem necessidade de configuração, de uso opcional, introduzido pelo Facebook em 2016. Cerca de 96% dos usuários de React recomendam-no para iniciantes[53]. Com o *create-react-app*, a configuração e a instrumentação se desenvolvem automaticamente em segundo plano, enquanto o seu foco permanece na implementação da aplicação.

Para começar, será necessário fazer a instalação global do pacote. Assim, ele sempre estará disponível na linha de comando para criar suas aplicações React.

Linha de Comando

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

Confira a versão do *create-react-app* para ter certeza de que a instalação ocorreu com sucesso:

Linha de Comando

```
create-react-app --version
*v1.4.1
```

Agora você pode criar sua primeira aplicação. Iremos chamá-la de *hackernews*, mas você pode escolher um nome diferente. O processo leva alguns segundos e, logo após terminar, navegue para a pasta:

Linha de Comando

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

Você pode abrir a aplicação no editor de sua escolha. A estrutura a seguir (ou uma variação, dependendo da versão do *create-react-app*) lhe será apresentada:

[52] https://github.com/facebookincubator/create-react-app
[53] https://twitter.com/dan_abramov/status/806985854099062785

Estrutura de pastas

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

Não tem problema se você não entender tudo desde o início. Eis uma descrição curta dos arquivos e pastas:

- **README.md**: A extensão .md indica que o arquivo é do tipo *markdown*, uma linguagem de marcação mais leve com uma sintaxe de formatação de texto. Muitos projetos com código-fonte incluem um arquivo *README.md* para passar as instruções iniciais. Quando eventualmente você sincronizar seu projeto em uma plataforma como o GitHub, o conteúdo do *README.md* será exibido na página inicial do repositório. Por ter usado *create-react-app*, seu *README.md* terá conteúdo igual ao do repositório do create-react-app no GitHub[54].
- **node_modules/**: Contém todos os pacotes node instalados via npm pois, uma vez que foi utilizado o *create-react-app*, alguns módulos já foram instalados para você. Normalmente, você nunca irá manipular diretamente o conteúdo desta pasta, devendo instalar e desinstalar pacotes usando npm na linha de comando.
- **package.json**: Mostra uma lista de dependências de pacotes node e mais algumas outras configurações de projeto.
- **.gitignore**: Este arquivo indica todos os arquivos e pastas que não deveriam ser adicionados ao seu repositório quando utilizando git e que existirão apenas no seu projeto local. Um exemplo é a pasta *node_modules/*. É suficiente compartilhar apenas o *package.json* com outras pessoas envolvidas para que elas sejam capazes de instalar sozinhas todas as dependências.
- **public/**: A pasta contém todos os arquivos do projeto quando este é preparado para produção. Todo o código que você escreveu na pasta *src/* será empacotado em um ou dois arquivos durante o *building* e colocado na pasta *public*.

[54] https://github.com/facebookincubator/create-react-app

- **build/**: Esta pasta será criada quando você preparar o projeto para produção, contendo todos os arquivos que serão utilizados. Neste processo de preparação (*build*), todo o seu código nas pastas *src/* e *public/* será empacotado em alguns arquivos e colocados neste diretório.
- **manifest.json** e **registerServiceWorker.js**: Não se preocupe com estes arquivos por enquanto, pois não serão necessários neste projeto.

No fim das contas, você não tem que alterar os arquivos e pastas mencionados acima. De início, tudo que você precisa está localizado na pasta *src/*. O foco principal fica com o arquivo *src/App.js*, que será usado para implementar sua aplicação. Mais tarde, você irá querer estruturá-la em múltiplos arquivos, cada um contendo seu próprio componente (ou mais de um).

Você encontrará um arquivo *src/App.test.js* (para seus testes) e um *src/index.js* como ponto de entrada para o "mundo React". Encontrará também os arquivos *src/index.css* e *src/App.css* para aplicar estilos aos componentes e à aplicação de uma forma geral. Se abrir qualquer dos dois, verá que ambos já trazem as definições de estilos padrão.

A aplicação *create-react-app* também é um projeto npm. Além de permitir que utilizemos npm para instalar e desinstalar pacotes node ao projeto, traz os seguintes *scripts* npm para serem usados na linha de comando:

Linha de Comando

```
// Roda a aplicação em http://localhost:3000
npm start

// Executa os testes
npm test

// Pepara a aplicação para produção
npm run build
```

Os *scripts* são definidos no *package.json*. Seu "esqueleto" de aplicação React está agora criado. A parte excitante vem a seguir, com os exercícios, finalmente rodando sua aplicação no *browser*.

Exercícios:

- Inicie sua aplicação com `npm start` e abra-a em seu navegador.
- Rode o *script* interativo `npm test`
- Verifique o conteúdo da sua pasta *public/*, execute o *script* `npm run build` e olhe novamente a pasta para ver quais arquivos foram adicionados (você pode removê-los se quiser, mas eles não causam nenhum problema)
- Familiarize-se com a estrutura de pastas
- Faça o mesmo com o conteúdo de cada arquivo
- Leia mais a respeito de npm scripts e create-react-app[55]

[55] https://github.com/facebookincubator/create-react-app

Introdução à JSX

Chegou o momento de você conhecer JSX, a sintaxe React. Como foi dito antes, *create-react-app* já montou uma estrutura inicial de aplicação para você. Todos os arquivos contêm alguma implementação *default*.

Vamos ao código-fonte. Inicialmente, você irá trabalhar apenas com o arquivo *src/App.js*:

src/App.js

```
1  import React, { Component } from 'react';
2  import logo from './logo.svg';
3  import './App.css';
4
5  class App extends Component {
6    render() {
7      return (
8        <div className="App">
9          <header className="App-header">
10           <img src={logo} className="App-logo" alt="logo" />
11           <h1 className="App-title">Welcome to React</h1>
12         </header>
13         <p className="App-intro">
14           To get started, edit <code>src/App.js</code> and save to reload.
15         </p>
16       </div>
17     );
18   }
19 }
20
21 export default App;
```

Para que não se confunda com as declarações *import/export* e com a palavra *class*, saiba que essas já são funcionalidades de JavaScript ES6. Iremos falar sobre isso mais tarde, neste mesmo capítulo.

No arquivo você tem uma ** classe de componente React ES6** (do inglês, um *class component*) de nome App. É a declaração de um componente. Basicamente, depois de o ter declarado, você poderá usá-lo como um elemento em qualquer lugar da sua aplicação. Será produzida uma **instância** do seu **componente**, ou, em outras palavras: o componente é instanciado.

O **elemento** retornado é especificado no método `render()`. Componentes são feitos de elementos. É importante entender as diferenças entre componente, instância e elemento.

Logo você verá que o componente App é instanciado, pois, se não o fosse, você não seria capaz de vê-lo renderizado no navegador. O componente é apenas a declaração, mas sem utilidade por si só. Você deve instanciá-lo em algum lugar usando JSX, assim: `<App />`.

O conteúdo do bloco *render* se parece muito com HTML, mas é JSX. JSX lhe permite misturar HTML e JavaScript. É algo poderoso, mas confuso quando você já está acostumado a separar os dois. Por esta razão, é mais fácil começar com JSX usando apenas HTML básico, removendo qualquer outro conteúdo que possa ser uma distração no arquivo.

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;
```

Pronto. O método `render()` agora retorna apenas um HTML simples, sem JavaScript. Vamos definir o texto "*Welcome to the Road to learn React*" como uma variável, que pode ser usada dentro do seu JSX entre chaves.

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;
```

Deverá funcionar, quando você levantar sua aplicação novamente com `npm start` na linha de comando.

Você também deve ter notado o atributo `className`. Ele espelha o atributo `class` padrão de HTML. Por razões técnicas, JSX teve que substituir um punhado de atributos HTML. Você pode ver a lista completa em atributos HTML suportados na documentação de React[56]. Eles seguem a convenção *camelCase*. No seu caminho aprendendo React, você irá se deparar com mais atributos específicos de JSX.

Exercícios:

- Defina mais variáveis e as adicione ao seu código JSX
  - Use um objeto para representar um usuário com nome e sobrenome
  - Adicione as propriedades do objeto ao seu código JSX
- Leia mais sobre JSX[57]
- Leia mais sobre componentes, elementos e instâncias em React[58]

[56] https://facebook.github.io/react/docs/dom-elements.html
[57] https://facebook.github.io/react/docs/introducing-jsx.html
[58] https://facebook.github.io/react/blog/2015/12/18/react-components-elements-and-instances.html

ES6 const e let

Suponho que você tenha notado que declaramos a variável `helloWorld` com a palavra-chave `var`. JavaScript ES6 nos traz mais duas opções para declararmos variáveis: `const` e `let`. De agora em diante, você raramente irá encontrar `var` novamente.

Uma variável declarada com `const` não pode ter um novo valor atribuído a ela nem ser novamente declarada. Não pode ser modificada. Você adota o uso de estruturas de dados imutáveis quando utiliza `const`. Uma vez que sua estrutura de dados é definida, você não pode mais modificá-la.

Code Playground

```
// não permitido
const helloWorld = 'Welcome to the Road to learn React';
helloWorld = 'Bye Bye React';
```

Uma variável definida com `let` pode ser modificada.

Code Playground

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

Você pode usar `let` quando achar de poderia precisar atribuir novo valor à variável.

Contudo, você deve ter cuidado com `const`. Uma variável declarada com `const` não pode ser modificada, mas, quando esta se trata de um *array* ou objeto, os valores que ela armazena não são imutáveis e podem ser atualizados.

Code Playground

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

Então quando usar cada tipo de declaração? Existem diferentes opiniões sobre o melhor uso. Eu sugiro usar `const` sempre que possível, indicando que você quer manter sua estrutura de dados imutável, mesmo que os valores em objetos e *arrays* possam ser modificados. Se você deliberadamente deseja modificar sua variável, use `let`.

Imutabilidade é uma característica que foi abraçada em React e no seu ecossistema. É por este motivo que `const` deveria ser sua escolha padrão quando definindo uma variável. Mais uma vez, os valores em objetos complexos ainda podem ser modificados. Tenha cuidado com esse comportamento.

Na sua aplicação, dê preferência a `const` sobre `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;
```

Exercícios:

- Leia mais sobre ES6 const[59]
- Leia mais sobre ES6 let[60]
- Pesquise sobre imutabilidade de estruturas de dados
  - Descubra o porquê disso fazer sentido em geral em programação
  - Descubra porque é uma prática em React e em seu ecossistema

[59] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
[60] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

ReactDOM

Antes de continuar trabalhando no componente App, você deve querer ver onde ele é utilizado, não é mesmo? Ele está localizado dentro do seu ponto de entrada no mundo React: o arquivo *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')
);
```

Basicamente, `ReactDOM.render()` usa um *DOM node* no HTML e o substitui com o seu JSX. Dessa forma, você pode facilmente integrar React em qualquer aplicação estranha à sua.

Não é proibido utilizar `ReactDOM.render()` muitas vezes na aplicação, você pode fazê-lo para habilitar o uso da sintaxe JSX, de um componente React, de múltiplos componentes React ou até uma aplicação inteira. Mas, numa aplicação React pura, você só usará este método uma vez, para carregar toda a sua árvore de componentes.

`ReactDOM.render()` espera dois argumentos. O primeiro é o código JSX que será renderizado. O segundo argumento especifica o lugar onde a aplicação React irá se acomodar em seu HTML. Ele espera um elemento com um `id='root'`. Abra o arquivo *public/index.html* você encontrará esse id como atributo de uma tag.

Na nossa implementação, `ReactDOM.render()` já recebe seu componente App. Contudo, não haveria problema se, no lugar, passássemos um simples código JSX, não sendo obrigatório que o argumento passado seja um componente instanciado.

Code Playground

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

Exercícios:

- Abra o arquivo *public/index.html* e veja onde a aplicação React será alocada em seu HTML
- Leia mais a respeito da renderização de elementos em React[61]

[61] https://facebook.github.io/react/docs/rendering-elements.html

Hot Module Replacement

Existe algo que você pode fazer no arquivo *src/index.js* para melhorar sua experiência de desenvolvimento. É opcional e não é necessário "enfiar goela abaixo" essa prática quando se está começando a aprender React.

Em uma aplicação criada com *create-react-app*, o navegador atualiza a página exibida quando você altera o código fonte e isso já é uma vantagem. Faça o teste, alterando a variável `helloWorld` no seu arquivo *src/App.js*. A página será recarregada. Mas, existe uma maneira ainda melhor de fazê-lo.

Hot Module Replacement - algo como "recarregamento de módulos em tempo real" - (ou HRM) é uma ferramenta que permite a atualização da aplicação em seu navegador, sem que este faça o recarregamento da página. Você pode facilmente ativar esse recurso, adicionando uma pequena configuração ao seu *src/index.js*:

src/index.js

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import App from './App';
4  import './index.css';
5
6  ReactDOM.render(
7    <App />,
8    document.getElementById('root')
9  );
10
11 if (module.hot) {
12   module.hot.accept();
13 }
```

É só isso. Faça o teste novamente, alterando a variável `helloWorld` em seu *src/App.js*. O navegador não irá recarregar toda a página, mas a aplicação irá ser atualizada e mostrar a saída correta.

HRM nos traz muitas vantagens:

Imagine que você está depurando o código fazendo chamadas `console.log()`. Como o navegador não atualiza mais a página todas as vezes que você altera e salva o código, as chamadas anteriores irão permanecer no console até que você não queira mais. Isso pode ajudar bastante no processo de depuração.

Em uma aplicação que já está ficando grande, recarregamentos de página podem tirar sua produtividade. O tempo todo você tem que esperar que a página carregue novamente e isso pode demorar vários segundos em um app um pouco maior. HMR remove essa desvantagem.

Mas o maior benefício de usar HMR é o de que você consegue conservar o estado da aplicação por mais tempo. Imagine que você tem uma caixa de diálogo com uma sequência de passos e você está

no passo 3 (procedimento bastante conhecido como um *wizard*). Sem HMR, ao realizar alterações no código-fonte, seu navegador automaticamente recarregará a página. Você terá que reiniciar o procedimento do passo 1 e navegar até o passo 3 para ver a modificação. Com HMR, sua janela permanece ativa no passo 3, mantendo o estado da aplicação mesmo depois das mudanças de código fonte. A aplicação em si recarrega, mas a página não.

Exercícios:

- Mude o código-fonte do seu *src/App.js* algumas vezes para testar o uso de HMR
- Assista aos primeiros 10 minutos da apresentação Live React: Hot Reloading with Time Travel[62], com Dan Abramov

[62] https://www.youtube.com/watch?v=xsSnOQynTHs

JavaScript dentro do código JSX

Voltemos ao componente App. Até então, você renderiza algumas variáveis primitivas em seu código JSX. Agora você irá começar a trabalhar com listas de itens. Inicialmente, a lista virá de uma amostra local de dados, mas depois você irá consultar os dados usando uma API[63] externa, o que é muito mais empolgante.

Primeiro você precisa definir uma lista de itens.

src/App.js

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

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

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

Esses dados irão refletir o modelo daqueles que iremos consultar mais tarde com a API. Um item da lista possui um título, uma url e um autor. Ele tem identificador, pontos (que indicam o quão popular é um artigo) e também um contador de comentários.

Com a lista em mãos, você agora pode usar a funcionalidade `map`, nativa de JavaScript, em seu código JSX. Ela lhe possibilita iterar sobre sua lista de itens e exibir seu conteúdo. Mais uma vez, você usará chaves para encapsular a expressão JavaScript no JSX.

[63] https://www.robinwieruch.de/what-is-an-api-javascript/

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;
```

O uso de JavaScript dentro do HTML é algo muito poderoso em JSX. Normalmente, você teria utilizado `map` para converter uma lista de itens em outra lista de itens. Mas aqui você usa para converter uma lista de itens em elementos HTML.

Até então, apenas o `title` é exibido para cada item. Vamos adicionar mais propriedades:

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>
    );
  }
}
```

```
21
22 export default App;
```

É possível enxergar como a função map é simplesmente invocada *inline* no código JSX. Cada propriedade de item é exibida em uma tag `<span>`, com exceção da url, que colocamos no `href` da tag `<a>`.

React irá fazer todo o trabalho de exibir cada item. Contudo, você deve ajudá-lo a atingir todo o seu potencial e a ter uma melhor performance. Você deve dar um atributo `key` a cada elemento da lista. Essa é a forma de identificar que itens foram adicionados, modificados ou removidos quando a lista muda. Os itens do exemplo possuem um identificador que pode ser utilizado.

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>
  );
})}
```

Você deve se certificar de que o atributo key é um identificador único válido. Não cometa o erro de usar o índice do item no array, por exemplo. O índice não é um identificador estável, pois, quando a lista é reordenada, ficará difícil para o React identificar propriamente cada item.

src/App.js

```
// não faça isso
{list.map(function(item, key) {
  return (
    <div key={key}>
      ...
    </div>
  );
})}
```

Agora você está exibindo ambos os itens da lista. Inicie sua aplicação, abra o navegador e veja os dois sendo mostrados.

Exercícios:

- Leia mais sobre listas e *keys* em React[64]
- Revise as funcionalidades padrão de *arrays* em JavaScript[65]
- Use mais expressões JavaScript no seu código JSX

[64] https://facebook.github.io/react/docs/lists-and-keys.html
[65] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

ES6 Arrow Functions

JavaScript ES6 introduziu *arrow functions*. Uma expressão com *arrow function* é mais curta do que uma expressão com uma função convencional (utilizando a palavra `function`).

Code Playground

```
// declaração com function
function () { ... }

// declaração com arrow function
() => { ... }
```

Contudo, você precisa estar ciente das funcionalidades que essa sintaxe agrega. Uma delas é um comportamento diferente com o objeto `this`. Uma função convencional sempre define seu próprio objeto `this`. *Arrow functions* têm o objeto `this` do contexto que as contêm. Fique esperto quando utilizar `this` em funções definidas dessa forma.

Existe outro fato importante sobre *arrow functions* com relação a parênteses. Você pode removê-los quando a função recebe apenas um argumento, mas precisa mantê-los quando recebe vários.

Code Playground

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

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

// proibido
item, key => { ... }

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

Olhemos a função `map`. Você pode reescrevê-la de forma mais concisa com uma *arrow function* de 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>
  );
})}
```

Você também pode remover as chaves que delimitam o corpo da *arrow function*, pois este é conciso e o retorno é implícito. Se o fizer, você deve remover também o `return`. Essa prática irá se repetir mais vezes pelo livro, então certifique-se que aprendeu a diferença entre um bloco de código como corpo da função e um corpo conciso de função quando estiver utilizando *arrow functions*.

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>
)}
```

Seu código JSX parece mais conciso e legível agora. Ele omite a palavra-chave `function`, as chaves e o `return`. O desenvolvedor pode focar melhor nos detalhes da implementação.

Exercícios:

- Leia mais a respeito de ES6 *arrow functions*[66]

[66] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Classes ES6

JavaScript ES6 introduziu classes, que são normalmente utilizadas em linguagens de programação orientadas a objetos. JavaScript sempre foi e ainda é muito flexível quanto aos seus paradigmas de programação. Você pode usar programação funcional e programação orientada a objetos lado a lado, quando mais apropriado for.

Apesar de React adotar programação funcional, por exemplo com estruturas de dados imutáveis, classes são usadas para declarar componentes. Elas são chamadas de componentes de classe de ES6 (*ES6 class components*). React mistura as boas partes de ambos os paradigmas de programação.

Consideremos a classe `Developer` a seguir, para que possamos examinar uma classe JavaScript ES6 sem ter que pensar sobre componentes.

Code Playground

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

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

Uma classe tem um construtor para torná-la instanciável e ele pode receber argumentos para atribuí-los à instância da classe. Além disso, uma classe pode definir funções que, por estarem associadas, são chamadas de métodos. Geralmente, são chamados de métodos de classe.

`Developer` é apenas a declaração da classe. Você pode criar múltiplas instâncias invocando-a. É o mesmo raciocínio ao componente de classe de ES6, que tem uma declaração, mas que você precisa usá-lo em algum outro lugar para instanciá-lo.

Vejamos como você pode instanciar a classe e como pode utilizar seus métodos.

Code Playground

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

React usa classes de JavaScript ES6 para componentes de classe ES6 e você já utilizou um.

src/App.js

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

...

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

A classe App herda de `Component`. Basicamente você declara o componente App, mas ele estende outro componente. O que isso significa? Em programação orientada a objetos, você tem o princípio da herança. Ele é usado para repassar funcionalidades de uma classe para outra.

App herda funcionalidades da classe `Component`. Ela é utilizada para transformar uma classe básica de ES6 em uma classe de componente ES6. Ela tem todas as funcionalidades que um componente em React precisa ter. O método render é uma dessas funcionalidades que você já utilizou. Você irá aprender sobre outros métodos de classes de componentes mais tarde.

A classe `Component` encapsula todos os detalhes de implementação de um componente React. Ela torna os desenvolvedores capazes de usar classes como componentes em React.

Os métodos que um `Component` React expõe são a sua interface pública. Um desses métodos tem que ser sobrescrito, os outros não. Você aprenderá sobre os outros futuramente quando o livro chegar nos métodos de ciclo de vida em capítulos posteriores. Mas o método `render()` deve ser sobrescrito, porque ele define a saída de um `Component` React.

Agora você sabe o básico sobre classes JavaScript de ES6 e como elas são usadas em React para tornarem-se componentes. Você aprenderá mais sobre os métodos de Component quando o livro for descrever os métodos de ciclo de vida de React.

Exercícios:

- Leia mais sobre classes ES6[67]

[67] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes

Você aprendeu a criar a estrutura inicial da sua própria aplicação React! Vamos recapitular os últimos tópicos:

- React
  - create-react-app inicializa a estrutura de uma aplicação React
  - JSX mistura HTML e JavaScript para definir a saída de componentes React em seus métodos render
  - Componentes, instâncias e elementos são coisas diferentes em React
  - `ReactDOM.render()` é o ponto de entrada de uma aplicação React e o gancho para o DOM
  - Funcionalidades nativas de JavaScript podem ser utilizadas em JSX
  - map pode ser usada para renderizar uma lista de itens como elementos HTML
- ES6
  - Declarações de variáveis com `const` e `let` podem ser usadas para casos de uso específicos
    * dê preferência ao uso de const ao invés de let em aplicações React
  - *Arrow functions* podem ser usadas para manter a declaração de funções mais concisas
  - Classes são utilizadas para definir componentes em React através de herança

É prudente fazer um intervalo agora. Internalize o conhecimento adquirido e aplique-o por sua conta. Você pode brincar com o código fonte que escreveu até agora.

O código-fonte está disponível no repositório oficial[68].

[68] https://github.com/rwieruch/hackernews-client/tree/4.1

React Básico

Este capítulo irá lhe apresentar os conceitos básicos de React. Trataremos de assuntos como estado e interações, pois componentes estáticos são um pouco maçantes, não acha? Você também irá aprender os diferentes modos de declarar um componente e como fazer para mantê-los fáceis de reusar e de compor uns com os outros. Prepare-se para dar vida à eles.

Estado Interno do Componente

O estado local, também chamado de estado interno do componente, lhe permite salvar, modificar e apagar propriedades que nele são armazenadas. Componentes de classe inicializam seu estado interno utilizando um construtor. Ele é chamado apenas uma vez (quando o componente é inicializado).

Abaixo, introduzimos um construtor de classe:

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);
  }

  ...

}
```

Quando seu componente possui um construtor, torna-se obrigatória a chamada de `super();`, porque o componente App é uma subclasse de `Component` (`class App extends Component`). Mais tarde, você aprenderá mais sobre componentes de classe em ES6. Você também pode invocar `super(props);` para definir `this.props` no contexto do seu construtor. Caso contrário, se tentar acessar `this.props`, receberá o valor `undefined`. Futuramente, estudaremos mais sobre as *props* de um componente React.

A essa altura, o estado inicial do seu componente é composto por apenas uma lista de itens:

src/App.js

```
const list = [
  {
    title: 'React',
    url: 'https://facebook.github.io/react/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  ...
];

class App extends Component {

```

```
  constructor(props) {
    super(props);

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

  ...

}
```

O estado local está amarrado à classe através do objeto `this`. Dessa forma, você pode acessá-lo em qualquer lugar do componente. Por exemplo, no método `render()`. Anteriormente, você usou `map` com uma lista estática de itens (definida fora do componente) em seu método `render()`. Agora, você irá usar a lista obtida do seu estado local.

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>
    );
  }
}
```

A lista agora é parte do componente, residindo em seu estado interno. Você pode adicionar, alterar ou remover itens. Todas as vezes que o estado do seu componente mudar, o método `render()` será

chamado novamente. Você simplesmente altera o estado interno, sabendo que o componente será de novo renderizado exibindo os dados corretos.

Mas, tenha cuidado. Não altere o estado diretamente, use um método chamado `setState()` para mudá-lo. Você verá mais a respeito no próximo capítulo.

Exercícios:

- Experimente trabalhar com o estado local
  - Defina mais dados iniciais no estado em seu construtor
  - Use o estado no seu método `render()`
- Leia mais sobre o construtor de classe ES6[69]

[69] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes#Constructor

Inicializando Objetos em ES6

Em JavaScript ES6, você pode usar uma sintaxe abreviada para inicializar seus objetos de forma mais concisa. Imagine o seguinte:

Code Playground

```
const name = 'Robin';

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

Nesse caso, em que a propriedade do objeto e a variável são igualmente chamadas de `name`, você poderia fazer desta forma:

Code Playground

```
const name = 'Robin';

const user = {
  name,
};
```

Aplicando o mesmo raciocínio na sua aplicação, com a variável `list` e a propriedade do estado local que compartilha do mesmo nome:

Code Playground

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

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

Um outro atalho elegante é a declaração concisa de métodos em JavaScript ES6.

Code Playground

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

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

Por último, mas não menos importante, o uso de nomes computados de propriedades é permitido em JavaScript ES6.

Code Playground

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

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

Talvez isso ainda não faça tanto sentido para você. Por que você utilizaria nomes computados? Em um capítulo mais adiante, iremos nos deparar com a situação em que poderemos utilizá-los para alocar valores por chave, de uma forma dinâmica em um objeto. É uma forma elegante em JavaScript de gerar *lookup tables* (um tipo de estrutura de dados).

Exercícios:

- Experimente trabalhar com inicialização de objetos com ES6
- Leia mais sobre inicialização de objetos em ES6[70]

[70] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Object_initializer

Fluxo Unidirecional de Dados

Você tem agora em mãos um componente App com estado interno, que ainda não foi mexido. O estado é estático e, consequentemente, o seu componente também. Uma boa maneira de experimentar manipular o estado é criando alguma interação entre componentes.

Vamos adicionar um botão para cada item da lista que é exibida. O botão tem o rótulo "*Dismiss*" (dispensar) e irá remover o item, sendo útil quando você quer manter uma lista constando apenas itens ainda não lidos e dispensar os que não tem interesse, por exemplo.

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>
    );
  }
}
```

O método de classe `onDismiss()` ainda não foi definido, nós o faremos daqui a pouco. Vamos primeiro focar no tratamento do `onClick` do elemento `button`. Como você pode ver, o método

`onDismiss()` no `onClick` está encapsulado por uma *arrow function*. Nela, você tem acesso à propriedade `objectID` do objeto `item`, que identifica qual item será removido. Uma alternativa à isso seria definir a função fora e apenas passá-la para o `onClick`. Outro capítulo irá explicar em maiores detalhes o tópico de tratamento de eventos em elementos.

Você viu que o elemento button é declarado em múltiplas linhas? Note que, à medida que a quantidade de atributos cresce, deixar tudo em uma única linha pode acabar gerando uma bagunça. Por esse motivo, o elemento button foi escrito em múltiplas linhas e indentado. Não é obrigatório, mas é um estilo de código que eu recomendo fortemente.

Chegou a hora de implementar o comportamento de `onDismiss()`. A função recebe um id, para identificar qual item será removido e, por ser amarrada à classe, é um método de classe. Sendo assim, deve ser acessada com `this.onDismiss()` e não apenas `onDismiss()`. O objeto `this` é a instância da sua classe.

Para definir o `onDismiss()` como um método de classe, você precisa usar `bind` no construtor. *Bindings* serão explicados mais adiante, em outro capítulo.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

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

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

O próximo passo é definir a funcionalidade em si, ou a lógica de negócio do método em sua classe, da seguinte maneira:

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

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

  onDismiss(id) {
    ...
  }

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

Feito isso, você pode escrever o que acontece dentro do método. Basicamente, você quer remover da lista o idem identificado pelo id e armazenar a lista atualizada no seu estado local. A nova lista será usada no método `render`, que será novamente chamado, para ser exibida. O item removido não irá mais aparecer.

É possível remover um item de uma lista usando a funcionalidade nativa de JavaScript *filter*, que é uma função que recebe outra função como entrada. Esta, por sua vez, tem acesso a cada valor na lista, pois *filter* está iterando sobre ela, permitindo-lhe checar item a item na lista baseado na condição fornecida. Se o resultado da avaliação for *true*, o item permanece na lista. Caso contrário, será filtrado dela. Além disso, é bom saber que a função *filter* retorna uma nova lista e não mexe no estado da antiga. Ela atende à convenção em React de manter estruturas de dados imutáveis.

src/App.js

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

No próximo passo, você pode extrair a função e passá-la como argumento para *filter*.

src/App.js

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

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

Ainda pode fazê-lo de forma mais concisa, usando novamente uma *arrow function* de JavaScript.

src/App.js

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

Pode até colocá-la *inline* novamente, como fez no `onClick` do botão, mas pode ser que assim ela fique menos legível.

src/App.js

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

A lista agora remove o item clicado. Entretanto, o estado local ainda não foi atualizado. Finalmente você pode usar o método `setState()` para atualizar a lista no estado interno do componente.

src/App.js

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

Rode novamente sua aplicação e experimente clicar no botão "Dismiss". Provavelmente, irá funcionar e você estará testemunhando nesse momento o **fluxo unidirecional de dados** em React. Você dispara uma ação em sua *view* com `onClick()`, uma função ou método de classe muda o estado interno do componente e `render()` é de novo executado para atualizar a *view*.

Exercícios:

- Leia mais sobre o ciclo de vida do estado de componentes em React[71]

[71] https://facebook.github.io/react/docs/state-and-lifecycle.html

Bindings

É importante aprender sobre *bindings* em classes JavaScript quando se vai trabalhar com componentes de classe em React. No capítulo anterior, você os utilizou para ligar seu método `onDismiss()` à classe em seu construtor.

src/App.js

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

    this.state = {
      list,
    };

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

  ...
}
```

Em primeiro lugar: Por que você teve que fazer isso? Esse passo é necessário porque a amarração do `this` com a instância de classe não é feita automaticamente pelos métodos. Vamos demostrar isso com a ajuda do componente a seguir:

Code Playground

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

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

O componente renderiza normalmente, mas quando você clica no botão, obtém `undefined` no console. Essa é uma das maiores fontes de *bugs* quando se usa React. Você deseja usar `this.state` em um método de classe e ele não está acessível, porque `this` é `undefined`. Para corrigir isso, você precisa criar o *binding* entre o método e `this`.

No exemplo a seguir, o método é propriamente vinculado a `this` dentro do construtor da 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>
    );
  }
}
```

Executando novamente o teste, o objeto `this` (ou, sendo mais específico, a instância de classe) está definido nesse contexto e você tem acesso a `this.state`, ou `this.props`, que você conhecerá depois.

O *binding* de métodos também poderia ser feito em outros lugares, como no `render()`, por exemplo.

Code Playground

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

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

Apesar de possível, esta prática deve ser evitada, porque a vinculação do método seria feira todas as vezes que `render()` for chamado. Como basicamente ele roda todas as vezes que seu componente é atualizado, isso poderia trazer implicações de performance. Quando o *binding* ocorre no construtor, o processo só ocorre uma vez: quando o componente é instanciado. Essa é a melhor abordagem a ser escolhida.

Outra coisa que, uma vez ou outra, alguém acaba fazendo, é definir a lógica de negócios dos métodos de classe dentro do construtor.

Code Playground

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

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

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

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

Esta outra prática também deveria ser evitada, porque ela irá transformar seu construtor em uma bagunça ao longo do tempo. A finalidade do construtor é instanciar sua classe com todas as propriedades. Por isso, a lógica de negócio dos métodos de classe deve ser definida fora dele.

Code Playground

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

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

  doSomething() {
    // do something
  }

  doSomethingElse() {
    // do something else
  }

  ...
}
```

Por fim, devo mencionar que métodos de classe podem ser automaticamente vinculados sem fazê-lo explicitamente, com o uso de *arrow functions*.

Code Playground

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

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

Se não lhe agrada repetidamente fazer o *binding* no construtor, você pode seguir nesta abordagem. Como a documentação oficial de React continua utilizando *bindings* no construtor, o livro irá seguir fazendo-o também.

Exercícios:

- Experimente diferentes abordagens de *binding* e veja o valor de `this` no console.

Tratamento de Eventos

Nota do tradutor: De agora em diante, neste livro, usaremos "*event handler*" e "tratamento de evento" com o mesmo significado. Apesar de essa última não ser a tradução literal da primeira, é a forma mais conhecida para tal na língua portuguesa. Soaria estranho se usássemos "tratador de eventos", ou até "manipulador de eventos".

Este capítulo deve lhe dar um melhor entendimento sobre tratamento de eventos em elementos. Na sua aplicação, você usa o elemento `button` (a seguir) para remover um item da lista:

src/App.js

```
...

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

...
```

Este exemplo é um tanto quanto complexo. Você tem que passar um valor para o método de classe e, para tal, precisa encapsulá-lo em outra função (uma *arrow function*). Basicamente, o que precisa ser passado como argumento para o *event handler* é uma função, não sua chamada. O código a seguir não funcionaria, porque o método de classe seria imediatamente executado quando você abrisse sua aplicação no navegador.

src/App.js

```
...

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

...
```

Quando usamos `onClick={executarAlgo()}`, a função `executarAlgo()` seria executada imediatamente após a aplicação abrir no *browser*. A expressão passada para o *handler* é avaliada e,

como o valor retornado não é uma função, nada irá acontecer quando você clicar no botão. Mas, quando fazemos `onClick={executarAlgo}`, onde `executarAlgo` é o nome de uma função, essa só será executada quando o botão for clicado. A mesma regra se aplica para o método `onDismiss` usado em sua aplicação.

Entretanto, não é suficiente declarar `onClick={this.onDismiss}`, porque precisamos passar a propriedade `item.objectID` para o método, para identificar qual item será removido. Por esse motivo, usamos uma *arrow function* como *wrapper*, utilizando o conceito conhecido em JavaScript como *high-order function* , que será brevemente explicado mais tarde.

src/App.js

```
...

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

...
```

Uma outra alternativa seria definir a função *wrapper* em algum outro lugar e passá-la como argumento para o tratamento do evento. Uma vez que precisa ter acesso ao item, ela deve residir dentro do bloco da função *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>
```

```
17              <span>{item.author}</span>
18              <span>{item.num_comments}</span>
19              <span>{item.points}</span>
20              <span>
21                <button
22                  onClick={onHandleDismiss}
23                  type="button"
24                >
25                  Dismiss
26                </button>
27              </span>
28            </div>
29          );
30        }
31        )}
32      </div>
33    );
34  }
35}
```

No fim das contas, o *event handler* do elemento precisa receber uma função. Como exemplo, faça o teste com esse código, que faz o contrário:

src/App.js

```
1 class App extends Component {
2
3   ...
4
5   render() {
6     return (
7       <div className="App">
8         {this.state.list.map(item =>
9             ...
10            <span>
11              <button
12                onClick={console.log(item.objectID)}
13                type="button"
14              >
15                Dismiss
16              </button>
17            </span>
18          </div>
19        )}
```

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

A função é executada assim que você abre a aplicação no navegador, mas não quando você clica no botão. Enquanto que o código a seguir só roda no momento do clique. É uma função que é executada quando você dispara o evento.

src/App.js

```
...

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

...
```

Visando manter o código conciso, você pode transformá-la de volta em uma *arrow function*, fazendo o mesmo que fizemos com o método de classe `onDismiss()`.

src/App.js

```
...

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

...
```

Frequentemente, novatos em React têm dificuldades com este tópico do uso de funções no tratamento de eventos. Por este motivo, eu me alonguei tentando explicá-lo em maiores detalhes. Depois de tudo, você deve ter o seguinte código no botão, com uma *arrow function inline* , concisa, que tem acesso à propriedade `objectID` do objeto 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>
    );
  }
}
```

Outro tópico relevante que sempre é mencionado, com relação à performance, é o das implicações do uso de *arrow functions* em *event handlers*. Por exemplo, tomemos o caso do `onClick` com uma *arrow function* envolvendo o `onDismiss`. Todas as vezes que o método `render()` for executado, o *event handler* irá instanciar a função. Isso *pode* ter um certo impacto na performance da sua aplicação. Na maioria dos casos, porém, você não irar notar a diferença.

Imagine que você tem uma enorme tabela de dados com 1000 itens e cada linha ou coluna tem uma *arrow function* sendo definida no *event handler*. Nesse caso, sim, é válida a preocupação a respeito da performance e você poderia implementar um componente dedicado `Button` com o *biding* ocorrendo no construtor. Mas, preocupar-se com isso agora significa otimização prematura. É mais benéfico focar em aprender apenas React.

Exercícios:

- Experimente as diferentes abordagens de uso de funções no `onClick` do botão em sua aplicação.

Interação com Forms e Eventos

Adicionemos outra interação à aplicação, tendo uma experiência com *forms* e eventos em React. A interação em questão é uma funcionalidade de busca. O valor de entrada no campo de busca será usado para filtrar temporariamente sua lista, baseado na propriedade *title* de cada item.

O primeiro passo é definir um *form* com um campo de *input* em seu JSX.

src/App.js

```
class App extends Component {

  ...

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

Você irá digitar no *input* e filtrar a lista por esse termo de busca. Para tanto, você precisa armazenar o valor digitado em seu estado local. Mas, como acessar o valor? É possível utilizar **synthetic events** em React para acessar os detalhes do evento.

Vamos definir um *event handler* `onChange` para o campo *input*.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
```

```
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        ...
      </div>
    );
  }
}
```

A função está vinculada ao componente e, portanto, novamente temos um método de classe. Você ainda precisa do *binding* e definir o método em si.

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() {
    ...
  }

  ...
}
```

Utilizando um *event handler* em seu elemento, você ganha acesso ao *synthetic event* de React na assinatura da função que utilizou como *callback*.

src/App.js

```
class App extends Component {

  ...

  onSearchChange(event) {
    ...
  }

  ...
}
```

O evento tem o *value* do campo *input* no seu objeto *target*. Consequentemente, você consegue atualizar o estado local com o termo da busca utilizando `this.setState()` novamente.

src/App.js

```
class App extends Component {

  ...

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

  ...
}
```

Ademais, você não deve se esquecer de definir o estado inicial para a propriedade `searchTerm` em seu construtor. O campo *input* estará vazio de início e, portanto, o valor deveria ser uma *string* vazia.

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);
  }

  ...
}
```

Agora, todas as vezes que o valor no campo de *input* muda, você está armazenando o valor digitado no estado interno do seu componente.

Um breve comentário a respeito da atualização do estado local em um componente React: Seria normal se achássemos que, quando atualizamos `searchTerm` com `this.setState`, também deveríamos informar o valor de `list`. Mas não é o caso. O método `this.setState()` de React faz o que chamamos de *shallow merge*. Ele preserva o valor das outras propriedades do objeto de estado, quando apenas uma delas é atualizada. O estado da lista permanecerá o mesmo, inclusive sem o item que você removeu, quando apenas a propriedade `searchTerm` for alterada.

Voltemos à sua aplicação. A lista ainda não é temporariamente filtrada com base no valor do campo *input* que está armazenado no seu estado local, mas você já tem em mão tudo o que precisa para fazê-lo. Como? No seu método `render()`, antes de iterar sobre a lista usando *map*, você pode aplicar um filtro à ela, que apenas avaliaria se `searchTerm` coincide com o a propriedade *title* do item. Vamos usar a funcionalidade *filter*, nativa de JavaScript e já demonstrada anteriormente. Como *filter* retorna um novo *array*, você pode convenientemente chamá-la antes de *map*.

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>
    );
```

```
19    }
20  }
```

Desta vez, iremos adotar uma abordagem diferente sobre a função *filter*. Queremos definir o seu argumento (outra função) fora do componente. Lá, não temos acesso ao estado do componente e, por consequência, à propriedade `searchTerm` para avaliar a condição de filtragem. Teremos que passar o `searchTerm` como argumento e retornar uma nova função, que avalia a condição. Esse tipo de função retornada por outra função é chamada de *high-order function*.

Normalmente eu não mencionaria *higher-order functions*, mas faz todo o sentido em um livro sobre React. Faz sentido porque React trabalha com um conceito chamado de *high-order components*. Mais tarde, você aprenderá mais sobre isso. Vamos focar agora na funcionalidade *filter*.

Primeiro, você terá que definir a *high-order function* fora do componente App.

src/App.js

```
function isSearched(searchTerm) {
  return function(item) {
    // some condition which returns true or false
  }
}

class App extends Component {

  ...

}
```

A função recebe o `searchTerm` e retorna outra função, porque é o que *filter* espera como entrada. A função retornada terá acesso ao objeto item, pois será argumento da função *filter*. A filtragem será feita baseada na condição definida nela, que é o que faremos agora.

src/App.js

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

class App extends Component {

  ...

}
```

A condição diz que devemos comparar o padrão recebido em `searchTerm` com a propriedade `title` do item da lista. Você pode fazê-lo utilizando `includes`, funcionalidade nativa de JavaScript. Quando o padrão coincide, você retorna *true* e o item permanece na lista. Senão, o item é removido. Mas, tenha cuidado com comparações de padrões: Você não pode esquecer de formatar ambas as *strings*, transformando seus caracteres em minúsculas. Caso contrário, o título "Redux" e o termo de busca "redux" serão considerados diferentes. Uma vez que estamos trabalhando com listas imutáveis e uma nova lista é retornada pela função *filter*, a lista original permanecerá sem ser modificada.

Uma última coisa a mencionar: "Apelamos" um pouco, utilizando o recurso *includes* já de ES6. Como faríamos o mesmo, em JavaScript ES5? Você usaria a função `inderOf()` para pegar o índice do item na lista.

Code Playground

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

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

Outra refatoração elegante pode ser feita utilizando-se novamente uma *arrow function*, tornando a função mais concisa:

Code Playground

```
// ES5
function isSearched(searchTerm) {
  return function(item) {
    return item.title.toLowerCase().includes(searchTerm.toLowerCase());
  }
}

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

Qual das funções é mais legível, vai depender de cada um. Pessoalmente, eu prefiro a segunda opção. O ecossistema de React usa muitos conceitos de programação funcional. Corriqueiramente, você irá utilizar uma função que retorna outra função (*high order function*). Em JavaScript ES6, você pode expressá-las de forma mais concisa com *arrow functions*.

Por fim, você tem que usar a função definida `isSearched()` para filtrar sua lista. Você passa a propriedade `searchTerm` do seu estado local para a função, ela retorna outra função como *input* de *filter* e filtra sua lista baseada na condição descrita. Depois disso tudo, iteramos sobre a lista filtrada usando *map* para exibir um elemento para cada item da lista.

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>
    );
  }
}
```

A funcionalidade de buscar deve funcionar agora. Tente você mesmo, no seu navegador.

Exercícios:

- Leia mais sobre React events[72]
- Leia mais sobre higher order functions[73]

[72] https://facebook.github.io/react/docs/handling-events.html
[73] https://en.wikipedia.org/wiki/Higher-order_function

ES6 *Destructuring*

Existe um jeito, em JavaScript ES6, de acessar propriedades de objetos mais facilmente: É chamado de *destructuring*. Compare os seguintes trechos de código a seguir, em JavaScript ES5 e 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
```

Enquanto que, em JavaScript ES5, você precisa de uma instrução de código a mais todas as vezes que quer acessar uma propriedade de um objeto, em ES6 você pode fazê-lo de uma só vez, em uma única linha.

Uma boa prática, visando legibilidade, é fazer *destructuring* de múltiplas propriedades quebrando-o em várias linhas:

Code Playground

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

O mesmo vale para *arrays*, que também podem sofrer *destructuring*.

Code Playground

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

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

Talvez você tenha notado que o objeto do estado local do componente App pode ser "desestruturado" da mesma forma (iremos nos alternar entre "desestruturação" e o original "*destructuring*" no texto, uma vez que é importante saber o termo amplamente conhecido na comunidade). A linha de código com *filter* e *map* ficará menor.

src/App.js

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

Novamente, o jeito ES5 e o ES6:

Code Playground

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

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

Uma vez que o livro usa JavaScript ES6 a maior parte do tempo, é aconselhável que você também faça esta opção.

Exercícios:

- leia mais a respeito de ES6 destructuring[74]

[74] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Componentes Controlados

Você já tomou conhecimento do fluxo unidirecional de dados em React. A mesma lógica se aplica para o campo de *input*, que atualiza o estado local com o `searchTerm` para filtrar a lista. Quando o estado é alterado, o método `render()` é executado novamente e utiliza o `searchTerm` mais recente do estado local para aplicar a condição de filtragem.

Mas, não teríamos esquecido de alguma coisa no elemento *input*? A *tag* HTML "input" possui um atributo `value`. Este, por sua vez, geralmente contém o valor que é mostrado no campo. Neste caso, a propriedade `searchTerm`. Acho que ficou a impressão de que não precisamos disso em React.

Errado. Elementos de *forms* como `<input>`, `<textarea>` e `<select>` possuem seu próprio estado em HTML puro. Eles modificam o valor internamente quando alguém de fora do componente o muda. Em React, isso é chamado de um **componente não controlado**, porque ele gerencia seu próprio estado. Você deve garantir-se de que os transformou em **componentes controlados**.

Mas, como fazê-lo? Você só precisa definir o atributo "*value*" de um campo de *input*. O valor aqui, neste exemplo, já está salvo na propriedade `searchTerm` do estado do componente. Por que não acessá-lo, então?

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>
    );
  }
}
```

É só isso. O *loop* do fluxo de dados unidirecional do campo *input* torna-se auto-contido. O estado interno do componente é a única fonte confiável de dados (*single source of truth*) para o *input*.

Toda essa história de gerenciamento de estado interno e fluxo unidirecional de dados deve ser nova para você. Mas, uma vez acostumado, será o seu jeito natural de implementar coisas em React. Em geral, React trouxe um interessante novo padrão (com o fluxo de dados unidirecional) para o mundo das SPA (*single page applications*). Agora, este padrão é adotado por diversos *frameworks* e bibliotecas no momento.

Exercícios:

- Leia mais sobre React forms[75]

[75] https://facebook.github.io/react/docs/forms.html

Dividindo componentes

Você tem em mãos um componente *App* de tamanho considerável. Ele continua crescendo e, eventualmente, pode tornar-se confuso. É possível dividi-lo partes, ou seja, componentes menores.

Comecemos usando um componente para o *input* de busca e um componente para a lista de itens.

src/App.js

```
class App extends Component {

  ...

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

Você pode passar, para estes componentes, propriedades que eles podem utilizar. No caso, o componente *App* precisa passar as propriedades gerenciadas no seu estado local e os seus métodos 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>
    );
  }
}
```

Agora, você pode definir os componentes ao lado de App. Estes também serão classes ES6. Eles irão renderizar os mesmos elementos de antes.

O primeiro é o componente *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>
    );
  }
}
```

O segundo é o componente *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>
    );
  }
}
```

Agora que você tem esses três componentes, talvez tenha notado o objeto `props`, que é acessível na instância de classe com o uso de `this`. As `props` (diminutivo de propriedades) têm todos os valores que você passou para os componentes quando os utilizou dentro de *App*. Desta forma, componentes podem passar propriedades para os níveis abaixo, na árvore de componentes.

Tendo extraído esses componentes de *App*, você estaria apto a reutilizá-los em qualquer outro lugar. Uma vez que componentes recebem valores através do objeto *props*, você pode passar valores diferentes para seu componente a cada vez que utilizá-lo.

Exercícios:

- imagine outros componentes que você poderia separar, como fez com *Search* e *Table*.

– entretanto, não o faça agora, na prática, para não ter conflitos com o que faremos nos capítulos seguintes.

Componentes Integráveis

Existe ainda uma pequena propriedade que pode ser acessada no objeto de *props*: a *prop* `children`. Você pode usá-la para passar elementos dos componentes acima na hierarquia para os que estão abaixo, tornando possível integrar componentes com outros. Vejamos como isso é feito, quando você manda apenas um texto (*string*) como *child* para o componente *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>
    );
  }
}
```

Agora, o componente *Search* pode desestruturar a propriedade *children* do objeto *props*. Aí, poderá especificar onde *children* deverá ser mostrado.

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>
    );
  }
}
```

O texto "Search" deverá estar visível próximo do campo de *input* agora. Quando você utilizar o componente *Search* em algum outro lugar, você poderá escolher um texto diferente, se assim quiser. No fim das contas, não é só texto que pode ser passado assim. Você pode enviar um elemento (e árvores de elementos, encapsuladas por outros componentes) como *children*. Esta propriedade permite entrelaçar componentes.

Exercícios:

- Leia mais sobre o modelo de composição de React[76]

[76] https://facebook.github.io/react/docs/composition-vs-inheritance.html

Componentes Reutilizáveis

Ter componentes reutilizáveis e combináveis lhe dá o poder de produzir hierarquias de componentes com competência. Eles são a base da camada de visão do React. Os últimos capítulos mencionaram o termo reusabilidade e, agora, você pode reutilizar os componentes *Table* e *Search*. Até mesmo o componente *App* é reutilizável, porque você poderia instanciá-lo em algum outro lugar.

Vamos definir mais um componente reutilizável, *Button*, que irá, eventualmente, ser utilizado com mais frequência.

src/App.js

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

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

Pode parecer redundante declarar um componente como este. Você irá utilizar um componente `Button` ao invés do elemento `button`, poupando apenas o `type="button"`. Exceto por este atributo de tipo, quando você decide usar o componente `Button`, você precisará definir todo o resto. Mas, você tem que pensar em termos de um investimento de longo prazo. Imagine que possui vários botões em sua aplicação e quer mudar um atributo, estilo ou comportamento do botão. Sem o componente recém criado, você teria que manualmente refatorar cada botão. O componente *Button* garante que existirá uma referência única, um *Button* para refatorar todos os botões de uma só vez. "*One Button to rule them all.*"

Uma vez que você já tem um elemento de botão, substitua-o pelo componente *Button*. Note que ele omite o atributo *type*, já especificado dentro do próprio componente.

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>
    );
  }
}
```

O componente espera receber uma propriedade `className` via props. O atributo `className` é mais uma especificidade React, derivando do atributo HTML `class`. Mas, nós não passamos este atributo quando utilizamos *Button* e, sendo assim, deveria ser mais explícito no código do componente que `className` é opcional.

Portanto, você deveria definir um valor padrão para o parâmetro (mais uma funcionalidade de JavaScript ES6).

src/App.js

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

    ...
  }
}
```

Agora, sempre que não houver propriedade `className` especificada, quando o componente *Button* for utilizado, o valor do atributo será uma *string* vazia, ao invés de `undefined`.

Exercícios:

- Leia mais a respeito de parâmetros *default* em ES6[77]

[77] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Default_parameters

Declarações de Componentes

No momento, você tem quatro componentes implementados com classes ES6. É possível melhorar ainda mais este cenário. Deixe-me introduzir para você os **componentes funcionais sem estado** (*stateless funcional componentes*), como uma alternativa a componentes de classe. Vamos apresentar os diferentes tipos de componentes em React, antes de partir para a refatoração.

- ***Functional Stateless Components***: Esses componentes são funções que recebem uma entrada e retornam uma saída. As *props* do componente são a entrada. A saída é uma instância de componente (ou seja, puro e simples JSX). Em termos, é bem semelhante a componentes de classe. Contudo, eles são funções (*functional*) e não possuem estado local (*stateless*). Você não consegue acessar ou atualizar o estado com `this.state` ou `this.setState()`, porque não existe o objeto `this` aqui. Além disso, eles não possuem métodos de ciclo de vida (*lifecycle methods*). Apesar de não ter explicitamente aprendido a respeito ainda, você já utilizou dois: `constructor()` e `render()`. Ao passo que o construtor roda apenas uma vez durante todo o tempo de vida de um componente, o método `render()` é executado uma vez no início e também todas as vezes que o componente é atualizado. Tenha em mente este detalhe dos *stateless funcional components* (ausência de métodos de ciclo de vida), para quando chegarmos a este assunto posteriormente.
- **Componentes de Classe ES6**: Você já utilizou este tipo de declaração de componente nos quatro que construiu até aqui, extendendo o componente React. O `extends` atrela ao componente todos os métodos de ciclo de vida, disponíveis na API de componentes React. Assim, você pôde utilizar o método `render()`. Além disso, é possível armazenar e manipular o estado através de `this.state` e `this.setState()`.
- **React.createClass**: Esta forma era utilizada nas versões mais antigas de React e ainda é, em aplicações React que utilizam JavaScript ES5. Mas, o Facebook a declarou como *deprecated*[78], preferindo o uso de JavaScript ES6. Um deprecation warning foi adicionado na versão 15.5[79]. Ela não será utilizada no livro.

Assim, basicamente, sobram-nos apenas duas maneiras de declarar componentes. Mas, quando usar cada uma? Uma regra simples é: use *functional stateless components* quando você não precisa de estado ou de métodos de ciclo de vida. Geralmente, comece implementando seu componente desta forma e, uma vez que você precisa acessar o estado ou métodos de ciclo de vida, você pode refatorá-lo para um componente de classe ES6. Fizemos o inverso aqui no livro, apenas para fins de aprendizado.

Voltemos para sua aplicação. O componente App usa diretamente o seu estado interno. Por este motivo, ele tem que permanecer escrito como um componente de classe. Mas, os outros três componentes não precisam acessar `this.state` ou `this.setState()`, muito menos possuem algum método de ciclo de vida. Vamos, juntos, refatorar o componente *Search* para um *stateless functional component*. *Table* e *Button* ficarão como exercício para você.

[78] https://facebook.github.io/react/blog/2015/03/10/react-v0.13.html
[79] https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html

src/App.js

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

Basicamente, é isso: as props estão acessíveis na assinatura da função e o seu retorno é código JSX. Mas, você pode melhorar ainda mais o código. Da mesma forma que você utilizou *destructuring* antes, pode fazê-lo novamente no parâmetro *props* da assinatura da função.

src/App.js

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

De novo, pode melhorar. Você ja sabe que *arrow functions* permitem que você escreva funções mais concisas. Você pode remover os caracteres de declaração de bloco no corpo da função. Ocorre um retorno implícito e, desta forma, você pode remover também a instrução *return*.

Uma vez que seu *stateless functional component* é uma função, você pode escrevê-lo da forma concisa também.

src/App.js

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

Este último passo foi especialmente útil: ele forçou que a função tenha apenas *props* como entrada e JSX como saída. Contudo, se você realmente precisar fazer alguma coisa (representada aqui pelo comentário *do something*) além, sempre é possível devolver as declarações de bloco de corpo e o retorno da função.

Code Playground

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

  // do something

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

Não precisa fazê-lo agora. A motivação para manter a versão concisa é: Quando utilizando a versão mais explícita, as pessoas geralmente tendem a fazer muitas coisas em uma função. Deixando a função com retorno implícito, você pode focar nos seus *inputs* e *outputs*.

Agora, você tem um *stateless functional component* super enxuto. Se, em algum momento, você precisar acessar seu estado local ou métodos de ciclo de vida, bastará refatorar seu código para um componente de classe ES6. Ademais, vimos aqui como ES6 pode ser utilizado em componentes React para torná-los mais concisos e elegantes.

Exercícios:

- Refatore *Table* e *Button* para *stateless functional compotes*

- Leia mais a respeito de componentes de classe ES6 e functional stateless components[80]

[80] https://facebook.github.io/react/docs/components-and-props.html

Estilizando Componentes

Adicionemos alguns estilos básicos à nossa aplicação e aos nossos componentes. Você pode reutilizar os arquivos *src/App.css* e *src/index.css*. Estes arquivos já devem estar presentes em seu projeto, uma vez que você o criou utilizado o *create-react-app*. Devem ter sido importados em seus arquivos *src/App.js* e *src/Index.js*, respectivamente. Eu preparei alguns estilos que você pode simplesmente copiar e colar nesses arquivos. Mas, sinta-se livre para user seus próprios CSS, se quiser.

Primeiramente, estilizando sua aplicação de um modo geral:

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;
}
```

Em seguida, estilos para seus componentes no arquivo *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;
}
```

Agora, você pode utilizá-los em alguns dos seus componentes. Não se esqueça de usar, como atributo HTML, `className` (de React) no lugar de `class`.

Primeiramente, aplicaremos os estilos no componente *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>
    );
  }
}
```

Depois, faça o mesmo para o componente *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>
```

Você estilizou sua aplicação e componentes com CSS básico. Agora, ela deve ter um visual minimamente decente. Como você bem sabe, JSX mistura HTML e JavaScript. Sendo assim, alguém pode argumentar que é perfeitamente aceitável misturar CSS também (*inline styles*). Você pode definir objetos JavaScript e passá-los para o atributo *style* de um elemento.

Vamos definir uma coluna de largura dinâmica em Table, utilizando um *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>
```

```
26      )}
27    </div>
```

Se quiser, você pode definir seus objetos de estilos fora dos elementos, mantendo-os mais limpos.

Code Playground

```
1  const largeColumn = {
2    width: '40%',
3  };
4
5  const midColumn = {
6    width: '30%',
7  };
8
9  const smallColumn = {
10   width: '10%',
11 };
```

Finalmente, você pode utilizá-los em suas colunas: `<span style={smallColumn}>`.

Em geral, você irá encontrar diferentes opções e soluções para aplicar estilos em React. Utilizamos CSS puro e *inline styles*, até agora. É o suficiente para iniciar.

Não quero ser dogmático aqui e irei lhe deixar mais algumas opções. Você pode ler a respeito delas e aplicá-las por sua própria conta. Mas, se você é novo em React, eu recomendaria que permaneça com CSS puro e estilos *inline*, por enquanto.

- styled-components[81]
- CSS Modules[82]

[81] https://github.com/styled-components/styled-components
[82] https://github.com/css-modules/css-modules

Você aprendeu o básico para escrever sua própria aplicação React! Recapitulemos os últimos capítulos:

- React
  - use `this.state` e `setState()` para gerenciar o estado interno do seu componente
  - como passar funções ou métodos de classe para *element handlers*
  - utilizando *forms* e eventos em React para adicionar interações
  - o fluxo de dados unidirecional unidirectional é um importante conceito em React
  - adote a prática de componentes controlados
  - integre componentes com componentes filhos reutilizáveis
  - implementação e uso de componentes de classe ES6 e *stateless functional components*
  - abordagens para estilizar seus componentes
- ES6
  - funções que são atreladas à uma classe são métodos de classe
  - *destructuring* de objetos e arrays
  - parâmetros *default*
- Geral
  - *higher order functions*

Mais uma vez, é recomendável fazer um intervalo na leitura. Internalize o que aprendeu de novo e pratique. Faça experiências com o código-fonte que você escreveu até agora. Ademais, você pode também ler mais a respeito dos assuntos aqui discorridos na documentação oficial[83].

Você pode encontrar o código-fonte deste capítulo no repositório oficial[84].

[83] https://facebook.github.io/react/docs/installation.html
[84] https://github.com/rwieruch/hackernews-client/tree/4.2

Familiarizando-se com uma API

Chegou o momento de "melar as mãos" com uma API, uma vez que é entediante lidar sempre com dados estáticos.

Se você não está familiarizado, lhe encorajo a ler a minha jornada onde tomei conhecimento de APIs[85].

Conhece a plataforma Hacker News[86]? É uma grande agregadora de tópicos de tecnologia. Neste livro, você irá usar a API do Hacker News para consultar assuntos populares da plataforma. Existe uma API básica[87] e uma outra de buscas[88], ambas para recuperar dados da plataforma. Esta última faz mais sentido para ser utilizada em nossa aplicação, consultando discussões no Hacker News. Você pode visitar a especificação da API, para obter um melhor entendimento da estrutura de dados.

[85] https://www.robinwieruch.de/what-is-an-api-javascript/
[86] https://news.ycombinator.com/
[87] https://github.com/HackerNews/API
[88] https://hn.algolia.com/api

Métodos de Ciclo de Vida

Será preciso entender sobre métodos de ciclo de vida de React antes de começar a obter dados em seus componentes utilizando uma API. Estes métodos são um gancho para o ciclo de vida de um componente. Eles podem ser utilizados em um componente de classe ES6, mas não em *stateles functional components*.

Lembra-se de quando, em um capítulo anterior, você aprendeu sobre classes de JavaScript ES6 e como elas são utilizadas em React? Além do método `render()`, existem vários outros que podem ser sobrescritos em um componente de classe. Todos eles são chamados de métodos de ciclo de vida e mergulharemos neles agora.

Você já conhece dois deles: `constructor()` e `render()`.

O método `constructor` só é chamado quando uma instância de componente é criada e inserida no DOM. Este processo é chamado de **montagem de um componente**.

O método `render()` também é chamado durante o processo de montagem e quando o componente é atualizado. Cada vez que o estado interno ou as *props* de um componente mudam, o método `render()` é disparado.

Agora você já sabe um pouco mais sobre dois métodos de ciclo de vida e quando eles são chamados, além de já os ter utilizado. Mas, existem outros.

A montagem de um componente tem mais dois métodos de ciclo de vida: `getDerivedStateFromProps()` e `componentDidMount()`. O construtor é chamado primeiro, `getDerivedStateFromProps()` chamado antes de `render()` e `componentDidMount`é invocado depois do método `render()`.

Em resumo, o processo de montagem tem 4 métodos de ciclo de vida. Eles são invocados na seguinte ordem:

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

Mas, e quanto ao ciclo de vida de atualização de um componente? O que acontece quando o *state* ou as *props* mudam? No total, existem 5 métodos, chamados na seguinte ordem:

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

Por último, mas não menos importante, existe o ciclo de vida de desmontagem de um componente. Ele possui apenas um método, `componentWillUnmount()`.

No fim das contas, você não precisa conhecer todos esses métodos desde o começo. Pode ser um pouco intimidador ter que decorar e você provavelmente nem usará todos eles. Mesmo em uma aplicação React de maior escala, você usará apenas um subconjunto de métodos além de `render()` e `constructor`. Mas, é legal saber que existem outros, que podem ser usados em casos específicos:

- **constructor(props)** -É chamado quando o componente é inicializado. Você pode definir um estado inicial do componente e realizar o *binding* de métodos de classe aqui.
- **static getDerivedStateFromProps(props, state)** - Invocado antes do método `render()`, tanto na montagem inicial do componente, quanto em atualizações subsequentes. Ele deve retornar um objeto para atualização do estado (ou *null* para não atualizar nada). Este método existe para ser usado em **raras** situações onde o estado depende das mudanças de *props* ao longo do tempo. Importante notar que este é um método estático e não tem acesso à instância do componente.
- **render()** - Este método de ciclo de vida é mandatório e retorna os elementos como saída do componente. Deve ser puro e, logo, não deve modificar o estado do componente. Ele recebe como entrada o *state* e as *props* e retorna um elemento.
- **componentDidMount()** - É chamado apenas uma vez, quando o componente é montado. É o momento perfeito para fazer qualquer requisição assíncrona para obter dados de uma API. Os dados obtidos serão armazenados no estado interno do componente, para serem mostrados via `render()`.
- **shouldComponentUpdate(nextProps, nextState)** - É sempre chamado quando o componente atualiza devido a mudanças de `state` ou `props`. Você utilizará este método em uma aplicação React mais madura, visando otimizações de performance. Dependendo do valor booleano retornado aqui, o componente e todos os seus filhos irão ser novamente renderizados no ciclo de atualização. Você pode evitar que o método `render()` seja invocado.
- **getSnapshotBeforeUpdate(prevProps, prevState)** - Este método de ciclo de vida é invocado imediatamente antes da renderização mais recente ser aplicada no DOM. Em raros casos, quando o componente precisa capturar a informação do DOM antes que ela seja alterada, este método lhe permite fazê-lo. Outro método de ciclo de vida (`componentDidUpdate`) irá receber como um parâmetro qualquer valor retornado por `getSnapshotBeforeUpdate()`.
- **componentDidUpdate(prevProps, prevState, snapshot)** - Este método é chamado imediatamente depois de uma atualização do componente, com exceção da renderização inicial. Você pode utilizá-lo como uma oportunidade de realizar operações no DOM ou de efetuar requisições assíncronas depois do componente já estar montado. Se `getSnapshotBeforeUpdate()` tiver sido implementado no seu componente, o valor por ele retornado será recebido aqui, através do parâmetro `snapshot`.
- **componentWillUnmount()** - Chamado antes de você destruir seu componente. Você pode usá-lo para realizar qualquer tarefa de limpeza, por exemplo.

Os métodos de ciclo de vida `constructor()` e `render()` já foram utilizados por você, uma vez que são comumente utilizados em componentes de classe ES6. O método `render()` é, na verdade, obrigatório, caso contrário você não conseguiria retornar uma instância do componente.

Existe ainda um outro método de ciclo de vida em componentes React: `componentDidCatch(error, info)`. Ele foi introduzido no React 16[89] e é utilizado para capturar erros em componentes. No nosso contexto, a lista de dados da sua aplicação está funcionando muito bem. Mas, pode existir o caso da lista no estado local ser definida, por acidente, com o valor nulo (por exemplo, quando os dados são obtidos de uma API externa, mas a requisição falha e você define o valor do estado local para `null`). Ocorrido esta situação, não será mais possível filtrar ou mapear a lista, porque ela tem valor `null` ao invés de ser uma lista vazia. O componente quebraria e a aplicação inteira poderia falhar. Agora, com o uso do `componentDidCatch()`, você pode capturar o erro, armazená-lo no estado local e, opcionalmente, exibir uma mensagem para o usuário informando que algo de errado ocorreu.

Exercícios:

- Leia mais sobre métodos de ciclo de vida em React[90]
- Leia mais sobre o estado e sua relação com métodos de ciclo de vida[91]
- Leia mais sobre tratamento de erros em componentes[92]

[89] https://www.robinwieruch.de/what-is-new-in-react-16/
[90] https://reactjs.org/docs/react-component.html
[91] https://reactjs.org/docs/state-and-lifecycle.html
[92] https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html

Obtendo Dados

Você está preparado para obter dados da API Hacker News. Um método de ciclo de vida foi mencionado como apropriado para este uso: `componentDidMount()`. Você usará a API **fetch**, nativa de JavaScript, para efetuar a requisição.

Antes que possamos fazê-lo, vamos configurar as constantes de URL e parâmetros *default*, para decompor a requisição em pedaços menores.

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=';

...
```

Em JavaScript ES6, você pode utilizar template strings[93] para concatenar *strings*. Você fará isso aqui, para combinar sua URL para o *endpoint* da 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);
// saída: https://hn.algolia.com/api/v1/search?query=redux
```

Este formato irá manter a composição da sua URL flexível no futuro.

Agora, vejamos a requisição à API, onde você usará esta URL. O processo inteiro de obtenção dos dados será apresentado de uma só vez, mas cada passo será explicado posteriormente.

[93] 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);
  }

  ...
}
```

Um monte de coisas aconteceu neste código. Eu até pensei em quebrá-lo em pedaços menores. Mas, de novo, ficaria difícil de compreender as relações entre as partes. Deixe-me explicar cada passo em detalhes.

Primeiro, você pode remover a lista de itens criada como amostra de dados, porque a API do Hacker News retorna uma lista real, a amostra não será mais utilizada. O estado inicial do seu componente tem agora um `result` vazio e um termo de busca (`searchTerm`) padrão. O mesmo termo de busca é usado no campo *input* do componente *Search* e na sua primeira requisição.

Segundo, você utiliza o método `componentDidMount` para obter os dados depois que o componente é montado. Na primeira chamada de *fetch*, o termo de busca *default* do estado local é utilizado. Ela irá buscar discussões relacionadas com "redux".

Terceiro, a API nativa *fetch* é utilizada. O uso de *template strings* de JavaScript ES6 possibilita a composição da URL com o `searchTerm`. Ela (a URL) é o argumento para a função nativa *fetch*. A resposta então precisa ser transformada em uma estrutura de dados JSON, um passo mandatório neste caso (*fetch* nativo lidando com estruturas de dados JSON). E, finalmente, a resposta pode ser atribuída a *result* no estado interno do componente. Ademais, o block `catch` é utilizado em caso de um erro ocorrer. O fluxo será desviado para o bloco *catch* ao invés do *then*. Em um capítulo futuro, você irá incluir o tratamento para os erros.

Por último, não se esqueça de fazer o *binding* do novo método no seu construtor.

Você agora consegue utilizar os dados obtidos ao invés da amostra inicial. Contudo, tenha cuidado. O resultado não é apenas uma lista de dados. É um objeto complexo com *metadados* e uma lista de itens que são, no nosso caso, as discussões[94]. Você pode imprimir o estado interno com `console.log(this.state);` no seu método `render()`, a fim de visualizá-lo melhor.

(**Nota do Tradutor**: No texto original, o autor refere-se às discussões como *stories*. Iremos usar as formas "discussões" e "posts" aqui).

No passo seguinte, você irá renderizar o valor em *result*. Iremos evitar que qualquer coisa seja renderizada, retornando *null* quando não existir um resultado. Uma vez que a requisição à API foi bem sucedida o resultado é salvo em *state* e o componente *App* será re-renderizado, desta vez com o estado atualizado.

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}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
```

[94] https://hn.algolia.com/api

```
18        </div>
19      );
20    }
21  }
```

Vamos recapitular o que acontece durante o ciclo de vida do componente. Ele é inicializado pelo seu construtor. Depois, ele é renderizado pela primeira vez. Mas, você evitou que ele tentasse exibir qualquer coisa, porque `result`, no estado local, é *null*. Sim, é permitido a um componente retornar o valor *null* para que nada seja exibido. Então, o método `componentDidMount` é executado. Neste método, você obtém os dados da API do Hacker News de forma assíncrona. Uma vez que os dados chegam, o estado interno do componente é alterado por `setSearchTopStories()`. Por causa disto, o ciclo de vida de atualização entra em cena. O componente executa o método `render()` novamente, mas desta vez com o resultado populado no estado interno. O componente atual e, consequentemente, o componente *Table* e seu conteúdo são renderizados novamente.

Você utilizou a função *fetch* que é suportada pela maioria dos *browsers* para realizar uma requisição assíncrona para uma API. A configuração do *create-react-app* assegura que ela é suportada por todos eles. Existem, todavia, pacotes *node* de terceiros que podem ser utilizados como substitutos à API nativa *fetch*: superagent[95] e axios[96].

Este livro se baseia fortemente na notação JavaScript para checagem de dados. No exemplo anterior, `if (!result)` foi utilizado no lugar de `if (result === null)`. O mesmo se aplica para outros casos ao longo do livro. Por exemplo, `if (!list.length)` é usado ao invés de `if (list.length === 0)` ou `if (someString)` no lugar de `if (someString !== '')`. Recomendo ler mais sobre este tópico, se não estiver bem familiarizado com ele.

De volta à sua aplicação: A lista de resultados deve ser visível agora, Entretanto, existem dois *bugs* que foram deixados para trás. Primeiro, o botão "Dismiss" está quebrado. Ele não conhece o objeto complexo de resultado e ainda funciona baseado na lista simples da amostra de dados. Segundo, quando a lista com dados do servidor é exibida e você tenta buscar por outra coisa, a filtragem ocorre apenas do lado do cliente. O comportamento esperado é que outra consulta à API ocorra. Ambos os *bugs* serão corrigidos nos capítulos seguintes.

Exercícios:

- Leia mais sobre ES6 template strings[97]
- Leia mais sobre a API nativa *fetch*[98]
- Leia mais sobre como obter dados com React[99]

[95] https://github.com/visionmedia/superagent
[96] https://github.com/mzabriskie/axios
[97] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals
[98] https://developer.mozilla.org/en/docs/Web/API/Fetch_API
[99] https://www.robinwieruch.de/react-fetching-data/

ES6 e o Operador *Spread*

O botão "*Dismiss*" não funciona, porque o método `onDismiss()` não conhece o objeto do resultado exibido. Ele ainda o trata como se fosse a lista da amostra que era armazenada no estado local. Vamos mudar a função, para que opere sobre o objeto ao invés da lista.

src/App.js

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

Mas o que acontece no `setState()` agora? Infelizmente, o resultado é um objeto complexo. A lista de resultados (aqui chamados de *hits*) é apenas uma das múltiplas propriedades do objeto. Contudo, somente a lista é atualizada quando um item é removido e as outras propriedades devem permanecer as mesmas.

Uma possível abordagem seria a de mudar os *hits* no objeto *result*, como demonstro abaixo. Mas, não será a que adotaremos.

Code Playground

```
// não faça isso
this.state.result.hits = updatedHits;
```

React realmente abraça a imutabilidade de estruturas de dados. Sendo assim, você não deveria modificar diretamente um objeto. Uma abordagem melhor é a de gerar um novo objeto, baseado na informação que você tem. Nenhum dos objetos é realmente manipulado, você irá manter as estruturas de dados imutáveis, sempre retornando um novo objeto.

Você pode usar `Object.assign()`, de JavaScript ES6, que recebe como primeiro argumento o objeto alvo. Todos os outros argumentos são objetos fontes de dados, que são combinados no objeto alvo que, por sua vez, pode ser um objeto vazio. A imutabilidade é preservada aqui, uma vez que nenhum dos objetos originais é modificado. Seria algo mais ou menos assim:

Code Playground

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

Quando objetos de origem possuírem propriedades de mesmo nome, as do objeto que aparecer primeiro como argumento serão sobrescritas por aquelas do que aparecer depois (sobrescritas no objeto de destino). Apliquemos o mesmo raciocínio para o método `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 })
  });
}
```

Esta poderia muito bem ser a solução. Mas, existe um jeito ainda mais simples, em JavaScript ES6 e *releases* futuras. Gostaria de lhe introduzir o operador *spread*, que consiste apenas em "três pontos": `...`. Quando utilizado, todos os valores de um array ou objeto são copiados para outro array ou objeto.

Apesar de não precisar dele ainda, vamos examinar como funciona o operador *spread* de um **array**.

Code Playground

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

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

A variável `allUsers` é um *array* completamente novo. As outras variáveis `userList` e `additionalUser` permanecem as mesmas. Você poderia combinar até dois *arrays* em um novo, da mesma forma.

Code Playground

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

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

Olhemos, agora, o mesmo operador *spread* com objetos. Neste caso, não é JavaScript ES6, é apenas uma proposta para a próxima versão de JavaScript[100]. Mas, mesmo assim, já utilizada pela comunidade React. Por causa disto, *create-react-app* incorporou essa funcionalidade em sua configuração.

Basicamente, é o mesmo que o operador *spread* de array, só que com objetos. Ele copia cada par chave-valor em um novo objeto.

[100] https://github.com/sebmarkbage/ecmascript-rest-spread

Code Playground

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

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

Também a exemplo de *array*, múltiplos objetos podem ser combinados.

Code Playground

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

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

No fim das contas, ele pode ser utilizado no lugar de `Object.assign()`.

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 }
  });
}
```

Agora, o botão "Dismiss" voltará a funcionar, uma vez que `onDismiss()` saberá como atualizar o objeto complexo em *result*, quando um item da lista for removido.

Exercícios:

- Leia mais a respeito de ES6 Object.assign()[101]
- Leia mais sobre o operador *spread* de *array* em ES6[102]
  - O operador *spread* de objetos é brevemente mencionado.

[101] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign
[102] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator

Renderização Condicional

Renderização condicional é algo que, logo cedo, aparece em aplicações React. No livro, ainda não apareceu um caso em que fosse necessária. A renderização condicional acontece quando você quer decidir, em tempo de execução, entre renderizar um elemento ou outro. Algumas vezes, renderizar um elemento ou nada. Ela pode ser expressada como uma declaração if-else em JSX.

O objeto `result` no estado interno do componente é, inicialmente, `null`. Até então, o componente App não retornou nenhum elemento quando `result` não contém nada avindo da API. Não deixa de ser uma renderização condicional, uma vez que você opta pelo *return* mais cedo no `render()`, dada uma condição. O component App renderiza seus elementos ou então **nada**.

Mas, vamos dar um passo adiante. Faz mais sentido envolver o componente *Table*, que é um componente que depende de `result`, em uma condição independente de renderização. Todo o resto deve ser exibido, mesmo que ainda não haja nada em `result`. Você pode simplesmente usar um operador ternário em seu código 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>
```

```
      );
    }
}
```

Esta é sua segunda opção para expressar uma renderização condicional. Uma terceira opção é o operador lógico `&&`. Em JavaScript, um `true && 'Hello World'` sempre resulta em 'Hello World'. Um `false && 'Hello World'` sempre resulta em *false*.

Code Playground

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

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

Em React, você pode tomar proveito deste comportamento. Se a condição é verdadeira, a expressão após o operador lógico `&&` será a saída. Se a condição é falsa, React ignora e descarta a expressão. Isso pode ser aplicado no caso da renderização condicional de *Table*, porque o retorno será *Table* ou nada.

src/App.js

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

Estas foram algumas abordagens de como implementar a renderização condicional em React. Você poderá ler mais alternativas nesta exaustiva lista de exemplos de renderização condicional[103]. Você irá conhecer seus diferentes casos de uso e quando aplicá-los.

Finalmente, você estará vendo os dados obtidos em sua aplicação. Tudo, menos *Table*, é exibido quando a consulta dos dados ainda está pendente. Uma vez que a requisição é completada, o resultado é armazenado no estado local e *Table* é exibido, porque o método `render()` é novamente invocado e a condição avaliada resulta em seu favor.

[103] https://www.robinwieruch.de/conditional-rendering-react/

Exercícios:

- Leia mais sobre diferentes formas de renderizações condicionais[104]
- Leia mais sobre renderização condicional em React[105]

[104] https://www.robinwieruch.de/conditional-rendering-react/
[105] https://reactjs.org/docs/conditional-rendering.html

Efetuando consultas do lado do cliente ou do servidor

Atualmente, quando você usa o componente *Search* com o seu campo *input*, você irá filtrar a lista, embora isso esteja ocorrendo apenas no lado do cliente. Agora, você irá usar a API do Hacker News para realizar a busca do lado do servidor. Caso contrário, você só teria o primeiro resultado de chamada à API, feita no `componentDidMount()` com o termo de busca *default* como parâmetro.

Você pode definir um método `onSearchSubmit()` no seu componente *App*, que obtém resultados da API do Hacker News quando se está executando uma busca no componente *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;
  }

  ...
}
```

O método `onSearchSubmit()` poderia utilizar a mesma funcionalidade que `componentDidMount()`, porém com o termo de busca que foi substituído no estado local e não com o termo *default* inicial. Sendo assim, você pode extrair a funcionalidade como um método de classe reutilizável.

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);
  }

  ...
}
```

Agora, é necessário que um novo botão seja adicionado no componente *Search*. Este botão tem que, explicitamente, disparar a requisição de consulta. Caso contrário, os dados da API Hacker News seriam obtidos todas as vezes que o campo *input* mudasse. O que você espera é que este comportamento aconteça explicitamente em um tratamento do evento `onClick()`.

Uma alternativa seria o uso de um *debounce* (um técnica de "atraso" na ação) na função `onChange()`, evitando a necessidade do botão, mas isto adicionaria mais complexidade no momento e, talvez, não teria o efeito desejado. Vamos manter as coisas simples, sem o *debounce* por enquanto.

Primeiramente, passe o método `onSearchSubmit()` ao seu componente `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>
    );
  }
}
```

Segundo, crie um botão no componente *Search*. O botão possui um `type="submit"` e o *form* irá utilizar o atributo `onSubmit` para com o método `onSubmit()`. Você pode reutilizar a propriedade *children*, mas desta vez será usada como conteúdo de *button*.

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>
```

Em *Table*, você pode remover a funcionalidade *filter*, porque não mais existirá um filtro (busca) do lado do cliente. Não esqueça de remover a função `isSearched()` também, uma vez que não será mais utilizada. O resultado virá diretamente da API Hacker News, depois que o botão *"Search"* for clicado.

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>
```

Quando tentar realizar uma consulta agora, você irá notar que o *browser* irá recarregar o conteúdo. Este é um comportamento natural do navegador para uma função de *callback* do *submit* em um *form* HTML. Em React, frequentemente você irá suprimir este comportamento nativo através do método `preventDefault()`.

src/App.js

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

Enfim, você deverá conseguir consultar diferentes discussões na API Hacker News, sem mais haver filtragem no lado do cliente.

Exercícios:

- Leia mais sobre *SyntheticEvent* em React[106]
- Faça experiências com Hacker News API[107]

[106] https://reactjs.org/docs/events.html
[107] https://hn.algolia.com/api

Paginação de dados

Você já deu uma olhada na estrutura de dados retornada? A API Hacker News[108] retorna mais do que uma simples lista de resultados. Mais precisamente, ela retorna uma lista paginada. A propriedade *page*, que é `0` na primeira resposta à requisição, pode ser utilizada para consultar mais sublistas paginadas como resultado. Você precisa apenas passar a próxima página com o mesmo termo de busca (*search term*) para a API.

Vamos estender a lista de constantes de API, para que seja possível lidar com dados paginados.

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=';
```

Agora, você pode utilizar a nova constante para adicionar o parâmetro *page* à sua requisição de API.

Code Playground

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

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

O método `fetchSearchTopStories()`irá receber a página como o segundo argumento. Se você não o fornecer ao método, ele irá retornar a página `0` como resultado da requisição. Sendo assim, os métodos `componentDidMount()` e `onSearchSubmit()` consultam a primeira página na primeira requisição. Cada consulta adicional deve buscar a próxima página, fornecendo o segundo argumento.

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())
```

[108] https://hn.algolia.com/api

```
      .then(result => this.setSearchTopStories(result))
      .catch(error => error);
  }

  ...

}
```

O argumento que representa a página usa um parâmetro *default* de JavaScript ES6, para definir o valor de página `0` no caso de nenhum outro ser fornecido para a função.

Você pode agora usar a página atual do resultado obtido no `fetchSearchTopStories()`. Use este método em um botão, para obter mais discussões em um tratamento de evento `onClick`. Utilizaremos *Button* para obter mais dados paginados da API Hacker News, sendo necessário apenas definir o tratamento do `onClick()`, que recebe o termo de busca atual e a próxima página (página atual + 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>
    );
  }
}
```

Além disso, no seu método `render()`, você deverá assegurar, por padrão, a página `0` quando ainda não existir nenhum resultado. Lembre-se que o método `render()` é chamado antes que os dados sejam, de forma assíncrona, obtidos em `componentDidMount()`.

Falta ainda um passo. Você obteve sua próxima página de dados, mas ela irá sobrescrever os dados da página anterior. Seria ideal concatenar as duas listas (a antiga e a nova) do estado local e do novo objeto resultante. Vamos ajustar a funcionalidade, para que os novos dados sejam adicionados, ao invés de sobrescritos.

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 }
  });
}
```

Mais coisas acontecem em `setSearchTopStories()`. Primeiro, você recebe, no resultado, os itens (ou *hits*) e a página.

Segundo, você checa se já existiam *hits* de consultas anteriores. Quando a página é `0`, significa que é uma nova consulta feita em `componentDidMount()` ou `onSearchSubmit()`. O valor em *hits* é vazio. Mas, quando você clica no botão "More", visando obter mais dados paginados, o valor de *page* não é `0`, mas o da próxima página. Os itens do resultado anterior já se encontram armazenados em seu estado e, desta forma, podem ser utilizados.

Terceiro, você não deseja sobrescrever o valor antigo em *hits*. Você pode fundi-lo com os novos, da requisição mais recente à API. A junção das duas listas pode ser feita através do operador *spread* de *arrays* em JavaScript ES6.

Quarto, você define o novo estado local do componente, com os *hits* combinados e o valor da página.

Um último ajuste deve ser feito. Quando você testa o botão "More", ele obtém apenas alguns itens da lista. A URL da API pode ser editada para obter mais itens a cada requisição. Novamente, você pode adicionar mais constantes aqui.

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=';
```

Pronto, você pode utilizar as constantes para estender a URL da API.

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);
}
```

Assim, a requisição à API Hacker News irá obter mais itens de uma vez, em relação ao que fazia antes. Como você pode ver, uma API tão poderosa como esta lhe dá uma rica variedade de alternativas para realizar experimentos com dados do mundo real. Faça uso dela quando estiver aprendendo algo novo, de um jeito mais excitante. Veja como eu aprendi sobre os "poderes" que APIs provêem[109] quando se está aprendendo uma nova linguagem de programação ou biblioteca.

Exercícios:

- Leia mais sobre parâmetros *default* ES6[110]
- Experimente as possibilidades dos parâmetros da Hacker News API[111]

[109] https://www.robinwieruch.de/what-is-an-api-javascript/
[110] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Default_parameters
[111] https://hn.algolia.com/api

Cache do Cliente

Cada *submit* de consulta faz uma requisição à API Hacker News. Você poderia buscar por "redux", depois por "react" e, eventualmente, "redux" novamente. No total, 3 requisições. Mas, você buscou por "redux" duas vezes e, em ambas, todo o caminho de requisição assíncrona para obter dados foi percorrido. Em uma implementação de *cache* do lado do cliente, você armazena cada resultado. Quando uma requisição à API é feita, ela antes checa se o resultado já existe. Se sim, o *cache* é utilizado. Caso contrário, o processo completo é seguido.

A fim de ter um *cache* no cliente para cada resultado, você terá que armazenar múltiplos `results` ao invés de um só, no estado interno do seu componente. O objeto *results* consistirá em um mapa com o termo de busca como chave e o *result* como valor. Cada resultado da API será salvo com este conjunto chave-valor.

No momento, o resultado no estado local se encontra da seguinte forma:

Code Playground

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

Imaginando que você tenha feito duas requisições, uma para o termo "redux" e outra para "react", o objeto *results* deverá se parecer com o seguinte:

Code Playground

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

Vamos implementar a lógica do *cache* do cliente com `setState()`. Primeiro, renomeie o objeto `result` para `results` no estado inicial do componente. Segundo, defina uma `searchKey` temporária, que é utilizada para armazenar cada `result`.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

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

    ...

  }

  ...

}
```

A `searchKey` deve ser definida antes de cada requisição ser feita. Ela reflete o `searchTerm`. Você deve estar se perguntando: Por que não utilizar `searchTerm` diretamente? Esta é uma parte crucial a ser entendida, antes de continuar com a implementação. O `searchTerm` é uma variável **instável**, porque ele é alterado todas as vezes que você digita no campo *input* do *Search*. Entretanto, você precisará de uma variável mais **estável** para determinar o termo de busca recentemente submetido à API para recuperar o resultado correto do mapa de resultados. Ela é um ponteiro para seu resultado atual no *cache* e, desta forma, pode ser utilizado para exibi-lo no método `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();
}
```

Com isso, você também precisa ajustar a funcionalidade onde o resultado é armazenado no estado interno do componente. Ela agora deve gravar cada resultado por `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 }
      }
    });
  }

  ...

}
```

A `searchKey` será usada para salvar os *hits* e página atualizados em um mapa de `results`.

Primeiro, você deve recuperá-la do estado do componente. Lembre-se de que `searchKey` tem o valor definido em `componentDidMount()` e em `onSearchSubmit()`.

Segundo, os resultados anteriores precisam ser combinados com os novos, como já ocorria antes. Mas, desta vez, os valores antigos são recuperados do mapa com a chave `searchKey`.

Terceiro, um novo resultado pode ser colocado em `results` no estado do componente. Observemos o objeto `results` dentro do `setState()`.

src/App.js

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

A parte inferior garante que o resultado atualizado é armazenado por `searchKey` no mapa. O valor é um objeto com propriedades *hits* e *page*. A `searchKey` é o termo de busca. Você já aprendeu o que significa a sintaxe `[searchKey]: ...` em ES6: é uma propriedade com nome computado. Ela lhe ajuda a alocar valores dinamicamente em um objeto.

A parte superior precisa copiar todos os outros resultados no state, por `searchKey`, utilizando o operador *spread* de objetos. Caso contrário, você perderia tudo o que foi armazenado anteriormente.

Todos os resultados são agora armazenados por termo de busca. Este foi o primeiro passo para habilitar o comportamento de *cache*. No passo seguinte, você pode recuperar o resultado através da variável `searchKey` no mapa de resultados. Este é o principal motivo pelo qual `searchKey` teve que ser definido como uma variável mais estável. Se não fosse assim, o processo de recuperação em *cache* nem sempre funcionaria, uma vez que o valor em `searchTerm` muda frequentemente enquanto você utiliza o componente *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>
    );
  }
}
```

Uma vez que você definiu uma lista vazia como padrão, quando não existe resultado para a `searchKey`, você pode poupar a renderização condicional no componente *Table*. Você também irá precisar passar a `searchKey`, ao invés do `searchTerm`, para o botão "More". Caso contrário, sua consulta paginada dependerá deste último que, como já falamos, varia de uma forma instável. O `searchTerm` será utilizado com o campo *input* do componente "Search".

A funcionalidade de consulta deve voltar a funcionar. Ela armazena localmente todos os resultados da API Hacker News.

Ademais, o método `onDismiss()` precisa ser melhorado. Isto porque ele ainda trabalha com o objeto `result`. Agora ele deverá saber lidar com múltiplos resultados no objeto `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 }
    }
  });
}
```

Pronto. O botão "Dismiss" deverá voltar a funcionar.

Entretanto, não existe nada que impeça a aplicação de fazer uma requisição à API a cada *submit* de busca. Mesmo que já exista um resultado, a requisição será feita assim mesmo. Precisamos completar o comportamento da funcionalidade de *cache*, que já mantém os resultados, mas ainda não toma proveito deles. Resumindo, o último passo é: evitar uma nova requisição à API, caso já exista um resultado disponível em *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();
```

```
30    }
31
32    ...
33
34  }
```

Agora, seu cliente fará apenas uma requisição à API, quando você buscar pelo mesmo termo duas ou mais vezes. Até mesmo dados paginados, com diversas páginas, são armazenados em *cache* desta forma, porque você sempre salva a última página exibida, para cada resultado, no mapa `results`. Esta é uma abordagem muito arrojada de incluir *caching* na sua aplicação, não acha? A API Hacker News provê tudo o que você precisa para fazer *cache* até mesmo de dados paginados de forma efetiva.

Tratamento de Erros

Tudo está pronto para que você interaja com a API do Hacker News. Você até mesmo introduziu um jeito muito elegante de manter *cache* dos seus resultados e implementou paginação para obter listas de infinitos resultados de discussões da API. Mas, ainda falta uma peça no quebra-cabeças. Infelizmente, uma peça que geralmente tem faltado em aplicações desenvolvidas nos dias de hoje: tratamento de erros. É muito fácil implementar o "caminho feliz" sem a preocupação sobre erros que poderiam ocorrer no processo.

Neste capítulo, você irá incluir uma solução eficiente de tratamento de erros em sua aplicação, para o caso de problemas em uma requisição à API. Você já aprendeu tudo sobre as partes necessárias para adicionar tratamento de erros em React: estado local e renderização condicional. Basicamente, um erro é apenas mais um estado em React. Quando ele ocorre, você irá guardá-lo em um estado local e exibi-lo através de uma renderização condicional em seu componente. E só. Vamos implementar este tratamento no componente *App*, porque ele é o componente utilizado na consulta de dados da API. Primeiro, você tem que incluir o estado local `error`, inicializado com *null*, para receber um objeto em caso de erro.

src/App.js

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

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

    ...
  }

...

}
```

Segundo, você pode utilizar o bloco `catch`, da função nativa *fetch*, para chamar `setState()` e guardar o objeto de erro no estado local. Todas as vezes que uma requisição à API não é bem sucedida, o bloco *catch* será executado.

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 }));
  }

  ...

}
```

Terceiro, você pode recuperar este objeto do estado local no método `render()` e exibir uma mensagem em caso de erro, utilizando a renderização condicional 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>
    );
  }
}
```

É isto. Se quiser testar que o seu tratamento de erros está funcionando, você pode mudar a URL da API para algum valor inexistente.

src/App.js

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

Se o fizer, você deverá receber uma mensagem de erro, ao invés do conteúdo da aplicação. Fica a seu critério o local onde deseja colocar a renderização condicional para a mensagem de erro. No caso aqui, a aplicação inteira não é mais exibida, o que não seria exatamente a melhor experiência de usuário. Que tal, então, exibir a mensagem apenas no lugar do componente *Table*? O restante da aplicação ainda será visível em caso de erro.

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
    ) || [];

```

```
25      return (
26        <div className="page">
27          <div className="interactions">
28            ...
29          </div>
30          { error
31            ? <div className="interactions">
32              <p>Something went wrong.</p>
33            </div>
34            : <Table
35              list={list}
36              onDismiss={this.onDismiss}
37            />
38          }
39          ...
40        </div>
41      );
42    }
43  }
```

Depois de feitos os testes, não se esqueça de restaurar a URL original da API.

src/App.js

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

Sua aplicação estará funcionando, com a adição de um tratamento de erros em caso de problemas com a API.

Exercícios:

- Leia mais sobre tratamento de erros em componentes React[112]

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

Axios no lugar de *Fetch*

Em um capítulo anterior, você utilizou as funções nativas de *fetch* para realizar requisições à plataforma do Hacker News. A maior parte dos navegadores lhe permitem fazê-lo. Mas, alguns, especialmente os mais antigos, não oferecem suporte à essa API nativa de JavaScript. Além disso, uma vez que você começar a testar sua aplicação com ambientes que simulam um *browser*, também pode se deparar com problemas em relação ao uso de *fetch*. Existem algumas maneiras de contornar estes problemas, fazendo com que *fetch* funcione tanto em navegadores antigos (polyfills), quando em testes (isomorphic-fetch[113]), mas nós não entraremos em maiores detalhes sobre isso aqui.

Uma solução alternativa é substituir o *fetch* por uma biblioteca mais estável como a axios[114]. Axios é dedicada a resolver apenas um problema, mas o faz com alta qualidade: efetuar requisições assíncronas à APIs remotas. Por este motivo, você irá utilizá-la neste livro. Em termos práticos, este capítulo deve lhe mostrar como substituir uma biblioteca pela outra. Em um nível mais conceitual, ele também lhe mostra como sempre é possível achar uma solução para esses pequenos caprichos, existentes no desenvolvimento *web*. Nunca pare de buscar soluções para obstáculos que venham a aparecer no seu caminho.

Vejamos como *fetch* pode ser substituído por *axios*. Tudo que foi falado até então parece ser mais difícil do que realmente é. Primeiramente, você tem que instalar a biblioteca *axios* via linha de comando:

npm install –save axios

Segundo, você irá importar *axios* no seu componente *App*:

src/App.js

```
1 import React, { Component } from 'react';
2 import axios from 'axios';
3 import './App.css';
4
5 ...
```

E por último, mas não menos importante, você usará a biblioteca, de forma quase idêntica à API nativa *fetch*. Ela recebe a URL como argumento e retorna uma *promise*. Você não precisa transformar a resposta para JSON, no entanto. *Axios* faz isto para você e transforma o resultado em um objeto JavaScript chamado `data`. Certifique-se de que adaptou seu código à estrutura de dados retornada.

[113] https://github.com/matthew-andrews/isomorphic-fetch
[114] https://github.com/axios/axios

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 }));
  }

  ...

}
```

É somente isto, no que se refere a substituir *fetch* por *axios* neste capítulo. No seu código, você está chamando `axios()`, que usa por padrão uma requisição HTTP GET. Você pode fazer uma requisição GET explícita chamando `axios.get()`. Outros métodos HTTP como POST podem ser usados com `axios.post()`. Neste ponto, você já deve conseguir enxergar como a biblioteca *axios* é poderosa. Eu sempre recomendo que seja utilizada no lugar de *fetch* quando suas requisições se tornarem muito complexas ou quando você tem que lidar com caprichos do desenvolvimento *web*. Em adição a isto, em um capítulo mais na frente, você incluirá testes na sua aplicação. Não precisará, então, se preocupar mais com navegadores ou ambientes que simulam eles.

Eu gostaria de introduzir outra melhoria para a consulta ao Hacker News no componente *App*. Imagine que seu componente seja montado quando a página é renderizado pela primeira vez no *browser*. Em `componentDidMount()` o componente começa a fazer a requisição mas, logo depois, porque sua aplicação disponibilizou algum tipo de navegação, você sai da página atual e navega para outra. Seu componente *App* é desmontado, mas ainda existirá uma requisição pendente disparada no método e ciclo de vida `componentDidMount()`. Ela tentará, eventualmente, utilizar `this.setState()` no `then()` ou no `catch()` da *promise*. Provavelmente, pela primeira vez, você verá o seguinte *warning* na linha de comando ou no *console* do seu navegador:

Linha de Comando

```
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.
```

Você pode tratar isto abortando a requisição quando seu componente é desmontado, prevenindo a chamada de `this.setState()` em um componente nesta situação. É uma boa prática em React (mesmo que não seja seguida por muitos desenvolvedores) para preservar a aplicação limpa e sem

warnings. Contudo, a atual API de *promises* não permite abortar requisições. Você precisa se virar para fazê-lo, o que pode ser o motivo pelo qual poucos desenvolvedores têm seguido esta boa prática. A implementação a seguir se parece mais com uma solução de contorno do que uma implementação sustentável. Sabendo disso, fica em suas mãos a escolha de fazê-lo, ou não. Esteja apenas ciente do *warning*, para o caso dele aparecer novamente em algum capítulo do livro ou uma aplicação sua, no futuro. Se acontecer, você saberá tratá-lo.

Para começar, você pode adicionar uma variável de classe que mantém o estado do ciclo de vida do seu componente. Ele pode ser inicializado com o valor `false` quando o componente inicializa, mudado para `true` quando o componente é montado e, novamente, `false` quando ele é desmontado. Assim, você será capaz de rastrear o estado do ciclo de vida. Este campo nada tem a ver com o estado local, nem é acessado ou modificado com `this.state` e `this.setState()`, uma vez que você deveria poder acessá-lo diretamente na instância do componente sem depender do gerenciamento de estado de React. Ele também não leva a nenhuma nova renderização do componente quando seu valor é modificado.

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;
  }

  ...

}
```

Finalmente, você pode utilizar este conhecimento, não para abortar a requisição por si mesma, mas

para evitar que `setState()` seja chamado depois do componente ter sido desmontado, evitando o *warning* antes mencionado.

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 }));
  }

  ...

}
```

No geral, este capítulo lhe mostrou como você pode substituir uma biblioteca por outra em React. Se você se deparar com problemas, coloque a seu favor o vasto ecossistema de bibliotecas JavaScript. Você também viu uma forma de evitar que `this.setState()` seja chamado em um component React não montado. Se você investigar a biblioteca *axios* mais a fundo, encontrará uma outra forma de cancelar a requisição. Fica a seu critério se aprofundar no tópico, ou não.

Exercícios:

- Leia mais sobre porque *frameworks* importam[115]
- Leia mais sobre uma sintaxe alternativa para componentes React[116]

[115] https://www.robinwieruch.de/why-frameworks-matter/
[116] https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax

Você aprendeu a interagir com uma API em React! Vamos recapitular os últimos tópicos:

- React
  - Métodos de ciclo de vida de componentes de classe e seus diferentes casos de uso
  - componentDidMount() para interações com APIs
  - renderização condicional
  - *synthetic events* em *forms*
  - tratamento de erros
  - abortando uma requisição à uma API remota
- ES6 e além
  - *template strings*
  - operador *spread* para estruturas de dados imutáveis
  - nomes de propriedades computados
  - campos (variáveis) de classe
- Geral
  - Integração com a Hacker News API
  - API nativa *fetch*
  - buscas do lado do cliente e do servidor
  - dados paginados
  - *caching* no cliente
  - *axios* como uma alternativa à API nativa de *fetch*

Novamente, faz sentido fazer uma pausa. Internalize o aprendizado e aplique-o você mesmo. Você pode fazer experiências com o código que escreveu até agora. Você também pode achar o código-fonte no repositório oficial[117].

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

Organização do Código e Testes

O capítulo irá manter o foco em tópicos importantes para a manutenção do código em uma aplicação escalável. Você irá aprender sobre como organizá-lo, visando adotar as melhores práticas de estruturação de arquivos e pastas. Outro aspecto são os testes, muito importantes para manter seu código robusto. Finalmente, você irá aprender sobre uma ferramenta muito útil para depurar suas aplicações React. O capítulo irá deixar um pouco de lado a aplicação prática que estamos desenvolvendo e explicará alguns destes tópicos para você de forma mais conceitual.

ES6 Modules: *Import* e *Export*

Em JavaScript ES6, você pode importar e exportar funcionalidades de módulos. Estas funcionalidades podem ser funções, classes, componentes, constantes e outros. Basicamente, tudo o que pode ser atribuído à uma variável. Módulos podem ser simples arquivos ou pastas inteiras com um arquivo *index* como ponto de entrada.

No começo deste livro, depois que você inicializou sua aplicação com *create-react-app*, os arquivos gerados já tinham várias declarações de `import` e `export`. É chegada a hora de explicá-los.

As declarações de `import` e `export` facilitam o compartilhamento de código entre múltiplos arquivos. Antes, haviam diversas formas de atingir este objetivo em um ambiente JavaScript. Era uma verdadeira bagunça e desejava-se que existisse um padrão, ao invés de múltiplas abordagens para a mesma coisa. Agora, desde JavaScript ES6, este é o comportamento nativo.

Adicionalmente, elas abraçam o paradigma de *code splitting*, onde você distribui seu código por múltiplos arquivos para mantê-lo reutilizável e manutenível. Reutilizável porque você consegue importar um pedaço de código em múltiplos arquivos. Manutenível porque você tem uma única fonte onde mantém o mesmo pedaço de código.

No fim, estas declarações lhe ajudam a pensar sobre o encapsulamento de código. Nem toda funcionalidade precisa ser exportada em um arquivo, algumas deveriam ser utilizadas apenas onde foram definidas. Os *exports* de um arquivo são basicamente a sua API pública. Apenas as funcionalidades exportadas estarão disponíveis para reuso em outro lugar, seguindo assim a boa prática de encapsulamento.

Vamos colocar a mão na massa. Como `import` e `export` funcionam? Os exemplos a seguir demonstram ambas as declarações, compartilhando uma ou múltiplas variáveis entre dois arquivos. No final, esta abordagem pode escalar para múltiplos arquivos e poderia também compartilhar mais do que simples variáveis.

Você pode exportar uma ou múltiplas variáveis com o chamado "*export* nomeado" (*named export*).

Code Playground: file1.js

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

export { firstname, lastname };
```

E importá-las em outro arquivo, utilizando o caminho relativo para o primeiro.

Code Playground: file2.js

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

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

Você também pode importar, em um único objeto, todas as variáveis exportadas de outro arquivo.

Code Playground: file2.js

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

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

Imports podem ter um *alias*. Por existir a possibilidade de você importar funcionalidades exportadas com o mesmo nome, de arquivos diferentes, você pode utilizar o *alias* para fornecer "apelidos" para elas.

Code Playground: file2.js

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

console.log(foo);
// saída: robin
```

Por último, mas muito importante, existe a declaração `default`, que pode ser utilizada em alguns casos, como:

- exportar e importar uma única funcionalidade
- para destacar a funcionalidade principal sendo exportada na API em um módulo
- para ter uma funcionalidade padrão de importação

Code Playground: file1.js

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

export default robin;
```

Quando importando um módulo exportado com a declaração *default*, você pode dispensar o uso de chaves no *import*.

Code Playground: file2.js

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

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

Além disso, o nome dado para o módulo no *import* pode diferir daquele que foi exportado com *default*. Ele pode até ser usado em conjunto com as declarações nomeadas de *export* e *import*.

Code Playground: file1.js

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

const person = {
  firstname,
  lastname,
};

export {
  firstname,
  lastname,
};

export default person;
```

Code Playground: file2.js

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

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

Em *exports* nomeados, você pode poupar linhas adicionais, exportando diretamente as variáveis.

Code Playground: file1.js

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

Essas são as funcionalidades principais de *ES6 modules*. Elas lhe ajudam a organizar e manter o seu código, projetando APIs com módulos reutilizáveis. Você também pode exportar e importar funcionalidades com a finalidade de testá-las, o que será feito em um dos capítulos a seguir.

Exercícios:

- Leia mais sobre ES6 import[118]
- Leia mais sobre ES6 export[119]

[118] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import
[119] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

Organização de Código com *ES6 Modules*

Você pode estar se perguntando: Por que nós não seguimos as melhores práticas de divisão de código para o arquivo *src/App.js*? No arquivo, nós já temos múltiplos componentes que poderiam ter sido definidos em seus próprios arquivos/pastas (módulos). Para o nosso propósito de aprendizado de React, é prático manter estas coisas em um só lugar. Mas, uma vez que a sua aplicação cresce, você deve considerar dividir estes componentes em múltiplos módulos. Somente desta forma sua aplicação será escalável.

A seguir, eu irei propor várias estruturas de módulos que você **poderia** aplicar. Eu recomendo aplicá-las como um exercício, ao final do livro. Para manter o código simples, eu não irei fazer a divisão e irei continuar os capítulos seguintes com o arquivo *src/App.js*.

Uma possível estrutura de módulo seria:

Estrutura de Pastas

```
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
```

Ela separa os componentes em seus próprios arquivos, mas não aparenta ser muito promissora. Note quantos arquivos com nomes duplicados, diferindo apenas nas extensões.

Outra estrutura de módulos seria:

Estrutura de Pastas

```
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
```

Ela parece mais limpa que a anterior. Dar o nome "index" a um arquivo o coloca como o ponto de entrada de uma pasta. É uma convenção de nome comumente utilizada, mas não significa que você não possa utilizar os nomes que quiser. Nesta estrutura de módulos, um componente é definido pela sua declaração em um arquivo JavaScript, mas também pelo seu estilo e seus testes.

Outro passo que poderia ser dado é o de extrair as constantes do componente *App*. Estas constantes foram utilizadas para compor a URL da API Hacker News.

Estrutura de Pastas

```
src/
  index.js
  index.css
  constants/
    index.js
  components/
    App/
      index.js
      test.js
      index.css
    Button/
      index.js
      test.js
```

```
      index.css
    ...
```

Os módulos naturalmente se dividem em *src/constants/* e *src/components/*. O arquivo *src/constants/index.js* se pareceria com algo assim:

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=';
```

O arquivo *App/index.js* poderia importar estas variáveis para utilizá-las.

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';

...
```

Quando você utiliza a convenção de nome *index.js*, pode omitir o nome no arquivo do caminho relativo no *import*.

Code Playground: src/components/App/index.js

```
import {
  DEFAULT_QUERY,
  DEFAULT_HPP,
  PATH_BASE,
  PATH_SEARCH,
  PARAM_SEARCH,
  PARAM_PAGE,
  PARAM_HPP,
} from '../../constants';

...
```

Mas, o que estaria realmente por trás da convenção de nomes *index.js*? Ela foi introduzida no mundo *node.js*. O arquivo *index* é o ponto de entrada de um módulo, descrevendo sua API pública. Módulos externos somente são permitidos de usar o arquivo *index.js* para importar código compartilhado pelo módulo. Considere a seguinte estrutura de módulos, feita para demonstrar o que foi falado aqui:

Estrutura de Pastas

```
src/
  index.js
  App/
    index.js
  Buttons/
    index.js
    SubmitButton.js
    SaveButton.js
    CancelButton.js
```

A pasta *Buttons/* tem múltiplos componentes de botões definidos em arquivos distintos. Cada arquivo pode utilizar `export default` para tornar o componente específico disponível para *Buttons/index.js*. O arquivo *Buttons/index.js* importa todas as diferentes representações de botões e as exporta como a API pública do módulo.

Code Playground: src/Buttons/index.js

```
import SubmitButton from './SubmitButton';
import SaveButton from './SaveButton';
import CancelButton from './CancelButton';

export {
  SubmitButton,
  SaveButton,
  CancelButton,
};
```

Agora, *src/App/index.js* pode importar os botões desta API pública localizada em *index.js*.

Code Playground: src/App/index.js

```
import {
  SubmitButton,
  SaveButton,
  CancelButton
} from '../Buttons';
```

Se seguirmos esta restrição, seria então uma má prática acessar outros arquivos diretamente, sem ser através do *index.js* do módulo. Isto quebraria as regras de encapsulamento.

Code Playground: src/App/index.js

```
// má prática, não fazer
import SubmitButton from '../Buttons/SubmitButton';
```

Você agora sabe como poderia refatorar seu código-fonte em módulos, aplicando restrições de encapsulamento. Como eu disse antes, com a intenção de manter o livro simples, eu não farei estas mudanças aqui. Você deve refatorar quando terminar a leitura do último capítulo.

Exercícios:

- Refatore seu arquivo *src/App.js* em múltiplos módulos de componentes, quando terminar de ler o livro

Snapshot Tests com Jest

Este livro não irá mergulhar fundo no tópico de testes, mas estes não poderiam deixar de ser mencionados. Em programação, testar o código é essencial e deveria ser encarado como algo obrigatório. Você, com certeza, quer manter alta a qualidade do seu código e com a garantia de que tudo funciona.

Talvez você tenha ouvido falar sobre a pirâmide de testes. Existem testes de ponta a ponta, testes de integração e testes unitários. Se você não está familiarizado com eles, o livro lhe dará uma rápida e básica noção geral.

Um teste unitário é utilizado para testar isoladamente um pequeno bloco de código. Pode ser uma única função a ser testada. Contudo, às vezes as partes isoladas funcionam bem, mas não funcionam como esperado quando combinadas. Elas precisam ser testadas como um grupo de unidades. É onde entram os testes de integração, que ajudam a ir aonde os testes unitários não chegam. Por último, mas não menos importante, testes ponta a ponta são uma simulação de um cenário real de usuário. Poderia ser, por exemplo, um *setup* automatizado em um *browser*, simulando o fluxo de *login* de um usuário em uma aplicação *web*. Enquanto testes unitários são rápidos e fáceis de escrever e manter, testes de ponta a ponta estão do lado oposto deste espectro.

De quantos testes de cada tipo eu preciso? Você irá querer ter muitos testes unitários para cobrir suas funções isoladamente. Depois disso, você poderá ter vários testes de integração para cobrir o uso combinado das mais importantes funções. Por fim, você pode querer ter apenas alguns testes ponta-a-ponta para simular cenários críticos em suas aplicações web. Isso é tudo o que tínhamos para falar, na nossa excursão geral no mundo dos testes.

Então, como é que você aplica este conhecimento sobre testes em sua aplicação React? A base para testar em React são os testes de componentes, que podem ser generalizados como testes de unidade e como *snapshot tests*. Você irá criar testes de unidade para seus componentes no próximo capítulo, utilizando uma biblioteca chamada *Enzyme*. Neste capítulo, você irá focar no outro tipo: *snapshot tests*. É quando *Jest* entra em cena.

Jest[120] é um framework JavaScript de testes utilizado no Facebook. Na comunidade React, ele é usado para testar componentes. Felizmente, *create-react-app* já embute Jest e você não precisa se preocupar em configurá-lo.

Comecemos os testes dos seus primeiros componentes. Antes de fazê-lo, precisará exportar os componentes que estão em *src/App.js* e serão testados. Assim, poderá fazê-lo em um arquivo separado. Você aprendeu como fazer a exportação no capítulo sobre organização de código.

[120] https://facebook.github.io/jest/

src/App.js

```
...

class App extends Component {
  ...
}

...

export default App;

export {
  Button,
  Search,
  Table,
};
```

Em seu arquivo *App.test.js*, você encontrará um primeiro teste que veio do *create-react-app*. Ele verifica se o componente *App* é renderizado sem erros.

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);
});
```

O bloco "it" descreve um caso de teste. Ele traz uma descrição e, quando executado, pode ter sucesso ou falhar. Além disso, ele poderia ser colocado dentro de um bloco "describe", que define uma suíte de testes. Sua suíte poderia incluir um punhado de blocos "it" para um componente específico. Mais tarde, você verá mais sobre o bloco "describe". Ambos os blocos são utilizados para separar e organizar seus casos de teste.

Note que `it` é a função conhecida na comunidade JavaScript como a função onde você roda um único teste. Contudo, em Jest, ela é frequentemente encontrada sob o *alias* de `test`.

Você pode rodar seus casos de testes utilizando o script *test* do *create-react-app* na linha de comando. Irá obter a saída para todos os casos de testes nesta mesma interface de linha de comando.

Linha de Comando

```
npm test
```

Jest lhe permite escrever *snapshot tests*. Eles fazem uma fotografia (*snapshot*) do seu componente e a compara com futuras fotografias. Quando um *snapshot* futuro for diferente, você será notificado no teste. Você pode aceitar a mudança, porque realmente mudou a implementação do componente, de propósito, ou rejeitar a mudança e investigar o que causou o erro. É um ótimo complemento para testes unitários, porque você testa apenas as diferenças entre as renderizações. Não adiciona grandes custos de manutenção, porque você pode simplesmente aceitar os *snapshots* que mudaram quando você altera algo conscientemente.

Jest guarda os *snapshots* (ou fotografias) em uma pasta. Somente desta forma ele consegue validar as diferenças nas comparações com *snapshots* futuros. Isso ainda permite que eles sejam compartilhados pelo time.

Antes de escrever o seu primeiro *snapshot test* com Jest, você tem que instalar uma biblioteca utilitária:

Linha de Comando

```
npm install --save-dev react-test-renderer
```

Agora você pode extender o teste do componente *App* como seu primeiro *snapshot test*. Primeiro, importe a nova funcionalidade da biblioteca recém adicionada e envolva seu bloco "it" em um bloco "describe". Neste caso, a suíte de testes trata apenas do componente *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);
  });

});
```

Implemente agora seu primeiro *snapshot test* utilizando um bloco "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();
  });

});
```

Rode seus testes novamente e veja se eles têm sucesso ou se falham. Eles devem ter sucesso. Se você mudar a saída do bloco "render" em seu componente *App*, o *snapshot test* deve falhar. Então você pode decidir atualizar o *snapshot* ou investigar o que aconteceu em seu componente.

Basicamente, a função `renderer.create()` cria um *snapshot* do seu componente App. Ela o renderiza virtualmente e armazena uma fotografia do DOM. Depois, espera-se que essa fotografia (ou *snapshot*) coincida com outras anteriores, de quando você rodou seus *snapshot tests* da última vez. Desta forma, você pode assegurar que o seu DOM permanecerá o mesmo, sem mudanças acidentais.

Vamos adicionar mais testes para nossos componentes independentes. Primeiro, o componente *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();
  });

});
```

O componente *Search* tem dois testes similares aos do componente *App*. O primeiro simplesmente renderiza *Search* no DOM e verifica que não existe erro no processo. Se um erro ocorrer, o teste quebrará, mesmo que não exista nenhuma assertiva (*expect, match, equal*) no bloco de teste. O segundo *snapshot test* é utilizado para armazenar o *snapshot* do componente renderizado e rodá-lo em comparação aos anteriores. Ele falha quando o *snapshot* muda.

Segundo, você pode testar o componente *Button* aplicando estas mesmas regras do componente *Search*.

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();
  });

});
```

Por fim, o componente *Table*, para o qual você passa um punhado de *props*, contendo uma amostra de dados para renderizá-lo.

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' },
    ],
  };

```

```
15    it('renders without crashing', () => {
16      const div = document.createElement('div');
17      ReactDOM.render(<Table { ...props } />, div);
18    });
19
20    test('has a valid snapshot', () => {
21      const component = renderer.create(
22        <Table { ...props } />
23      );
24      const tree = component.toJSON();
25      expect(tree).toMatchSnapshot();
26    });
27
28  });
```

Snapshot tests geralmente são bem básicos. Você só quer cobrir o caso de o componente ter ou não mudado. Uma vez que mudou, você decide se aceita as mudanças. Caso não, você tem que consertar a modificação indesejada.

Exercícios:

- Veja como um *snapshot test* falha quando você muda o retorno do seu componente no método `render()`
  - Aceite ou negue a mudança de *snapshot*
- Mantenha seus *snapshot tests* atualizados quando implementar mudanças nos componentes nos próximos capítulos
- Leia mais sobre Jest em React[121]

[121] https://facebook.github.io/jest/docs/tutorial-react.html

Testes de Unidade com Enzyme

Enzyme[122] é um utilitário de testes, lançado pelo Airbnb, para percorrer, manipular e realizar assertivas com seus componentes React. Você pode utilizá-lo para conduzir testes unitários, complementando seus *snapshot tests*.

Vejamos como podemos utilizar *enzyme*. Primeiro, você precisa instalar a biblioteca, uma vez que não ela vem por padrão com *create-react-app*. Ela possui, também, uma extensão para ser utilizada em React.

Linha de Comando

```
npm install --save-dev enzyme react-addons-test-utils enzyme-adapter-react-16
```

Segundo, você precisa incluir a biblioteca no seu *setup* de testes e inicializar o seu *Adapter* para ser usado com 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() });
```

Agora, você pode escrever seu primeiro teste unitário no bloco "describe" de *Table*. Você irá usar `shallow()` para renderizar seu componente e verificar que *Table* tem dois itens, de acordo com os dois itens de lista que você passou para ele. A verificação simplesmente checa se o elemento renderizado possui dois outros elementos com a 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';

...
```

[122] https://github.com/airbnb/enzyme

```
9
10 describe('Table', () => {
11
12   const props = {
13     list: [
14       { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
15       { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
16     ],
17   };
18
19   ...
20
21   it('shows two items in list', () => {
22     const element = shallow(
23       <Table { ...props } />
24     );
25
26     expect(element.find('.table-row').length).toBe(2);
27   });
28
29 });
```

Shallow renderiza o componente sem renderizar seus componentes filhos. Assim, você pode fazer o teste verdadeiramente dedicado a ele.

Enzyme tem, ao todo, três mecanismos de renderização em sua API. Você já conhece `shallow()`, mas também existem `mount()` e `render()`. Ambos criam instâncias do componente pai e também dos seus filhos. A diferença é que `mount()` lhe dá acesso aos métodos de ciclo de vida do componente.

Quando, então, utilizar cada mecanismo de renderização? Seguem algumas regras:

- Sempre comece com um teste *shallow*
- Se `componentDidMount()` ou `componentDidUpdate()` precisam ser testados, use `mount()`
- Se você quiser testar o ciclo de vida do componente e o comportamento dos seus filhos, use `mount()`
- Se você quiser testar a renderização dos filhos de um componente, com menos *overhead* do que quando se usa `mount()`, use `render()`

Você pode continuar escrevendo testes de unidade para seus componentes. Mas, faça por onde manter os testes simples e manuteníveis. Caso contrário, você irá acabar tendo que refatorá-los todas as vezes que seus componentes mudarem. O Facebook já introduziu *snapshot tests*, com Jest, para este propósito.

Exercícios:

- Escreva um teste unitário com *Enzyme* para o seu componente *Button*
- Mantenha seus testes unitários atualizados ao longo dos próximos capítulos
- Leia mais sobre enzyme e sua API de renderização[123]
- Leia mais sobre testes de aplicações React[124]

[123] https://github.com/airbnb/enzyme
[124] https://www.robinwieruch.de/react-testing-tutorial

Interface de Componente com *PropTypes*

Você pode conhecer TypeScript[125] ou Flow[126] como meios de introduzir uma interface de tipos para JavaScript. Uma linguagem com tipos é menos propensa a erros, porque o código é validado com base no texto. Editores e outros utilitários podem capturar estes erros antes mesmo do programa ser executado e seu programa torna-se mais robusto.

Neste livro, você não irá trabalhar com *Flow* ou *TypeScript*, mas com outra forma bastante organizada de checagem de tipos em seus componentes. React traz, nativamente, uma checagem de tipos para prevenir bugs. Você utiliza *PropTypes* para descrever a interface do seu componente. Todas as *props* passadas de um componente pai para um componente filho são validadas, com base na interface definida no componente filho.

Este capítulo irá lhe mostrar como tornar seus componentes mais seguros com *PropTypes*. Não irei manter estas mudanças nos capítulos seguintes, porque elas irão causar refatorações desnecessárias de código, durante a evolução dos exemplos. Mas, você deveria mantê-las e atualizá-las ao longo do caminho, para manter a interface dos seus componentes segura em relação à tipos.

Primeiro, você tem que instalar um pacote separado para React.

Linha de Comando

```
1 npm install prop-types
```

Agora, você pode importar *PropTypes*.

src/App.js

```
1 import React, { Component } from 'react';
2 import axios from 'axios';
3 import PropTypes from 'prop-types';
```

Vamos começar a definir uma interface de *props* nos nossos componentes:

[125] https://www.typescriptlang.org/
[126] 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,
};
```

Basicamente, é isto. Você atribui um *PropType* para cada argumento na assinatura da função. Os *PropTypes* básicos para tipos primitivos e objetos complexos são:

- PropTypes.array
- PropTypes.bool
- PropTypes.func
- PropTypes.number
- PropTypes.object
- PropTypes.string

Adicionalmente, você tem mais dois *PropTypes* que podem ser usados para definir um fragmento renderizável (node) e um elemento React.

- PropTypes.node
- PropTypes.element

Você já utilizou o *PropType* `node` para o componente *Button*. Existem, no total, mais definições de *PropTypes* que você pode ler na documentação oficial de React.

No momento, todos os *PropTypes* definidos para *Button* são opcionais. Os parâmetros podem receber *null* ou *undefined*. Mas, em alguns casos, você quer forçar a definição de algumas *props*. É possível fazer com que a passagem destas *props* para o componente seja obrigatória.

src/App.js

```
Button.propTypes = {
  onClick: PropTypes.func.isRequired,
  className: PropTypes.string,
  children: PropTypes.node.isRequired,
};
```

O parâmetro `className` não é obrigatório, porque ele pode simplesmente obter o valor *default* da *string* vazia.

O próximo passo é definir uma interface com *PropTypes* para o componente *Table*:

src/App.js

```
Table.propTypes = {
  list: PropTypes.array.isRequired,
  onDismiss: PropTypes.func.isRequired,
};
```

Você pode definir o conteúdo de um *PropType* do tipo *array* mais explicitamente:

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,
};
```

Apenas `objectID` é requerido, porque você sabe que alguma parte do seu código depende dele. As outras propriedades são apenas exibidas, não sendo necessariamente obrigatórias. Além disso, você não tem plena certeza de que a API Hacker News tem sempre uma propriedade definida para cada objeto no *array*.

Isto era, praticamente, tudo o que tínhamos para falar sobre *PropTypes*. Só falta mais um aspecto a ser mencionado: Você pode definir *props default* em seu componente. Tomemos novamente como exemplo o componente *Button*. A propriedade `className`, na assinatura do componente, tem um parâmetro *default* (de ES6).

src/App.js

```
const Button = ({
  onClick,
  className = '',
  children
}) =>
  ...
```

Você pode substituí-lo por uma *prop default* de React:

src/App.js

```
const Button = ({
  onClick,
  className,
  children
}) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

Button.defaultProps = {
  className: '',
};
```

Assim como o parâmetro *default* de ES6, a *default prop* garante que a propriedade é definida com um valor padrão quando o componente pai não informa um. A checagem de tipo de *PropType* é feita **depois** que a *default prop* é avaliada.

Se você rodar seus testes novamente, verá que erros de *PropType* aparecerão, na linha de comando, para seu componente. Isso pode ocorrer porque você não definiu todas as propriedades para seus componentes nos testes que foram escritos e algumas delas foram marcadas como obrigatórias na definição de PropTypes. Os testes, no entanto, passam sem problemas. Se quiser, você poderia passar todas as propriedades obrigatórias em seus testes, para evitar estes erros.

Exercícios:

- Defina a interface de *PropTypes* para o componente *Search*

- Adicione e atualize as interfaces de *PropTypes* quando você adicionar ou alterar componentes nos próximos capítulos
- Leia mais sobre React PropTypes[127]

[127] https://reactjs.org/docs/typechecking-with-proptypes.html

Depuração com *React Developer Tools*

Esta última seção apresenta uma ferramenta muito útil, geralmente utilizada para analisar e depurar aplicações React. **React Developer Tools** lhe permite inspecionar a hierarquia, as *props* e o estado local de componentes, estando disponível como uma extensão (para Chrome e Firefox) e como uma aplicação *standalone* (que funciona com qualquer outro ambiente). Uma vez instalada, o ícone da extensão irá acender quando você carregar *websites* que estejam utilizando React. Em páginas assim, você poderá ver uma aba chamada "React" nas ferramentas de desenvolvedor do seu navegador.

Vamos testá-la com a sua aplicação Hacker News. Na maioria dos navegadores, uma forma rápida de abrir as ferramentas de desenvolvedor é clicando com o botão direito do mouse na página e, depois, selecionar "Inspect" (ou "Inspecionar"). Faça isso com a sua aplicação carregada, depois clique na aba "React". Você deverá ver a hierarquia de elementos, sendo `App` o elemento raiz (*root element*). Se você expandir a árvore, achará também as instâncias dos componentes `Search`, `Table` e `Button`.

A extensão mostra, no painel lateral do lado direito, o estado local e as *props* de componente para o elemento selecionado. Por exemplo, se você clicar em `App`, verá que ele ainda não possui *props*, mas já possui um estado local. Uma técnica de depuração bastante simples é a de monitorar as mudanças de estado da aplicação causadas pela interação do usuário.

Primeiro, irá querer marcar a opção "Highlight Updates" (geralmente localizada acima da árvore de elementos). Segundo, você pode digitar um termo de busca diferente no campo de *input* da aplicação. Como você poderá ver, apenas `searchTerm` será modificado no estado do componentes. Você até já sabia que isto aconteceria desta forma, mas agora pode ver de verdade que ocorreu como planejado.

Finalmente, pressione o botão "Search". O estado `searchKey` irá ser imediatamente alterado para o mesmo valor de `searchTerm` e o objeto de retorno da requisição irá ser adicionado à `results` poucos segundos depois. A natureza assíncrona do seu código está agora visível aos seus olhos.

Por último, se você clicar com o botão direito do mouse em qualquer elemento, um menu de contexto irá aparecer com várias opções. Por exemplo, você pode copiar as *props* ou o nome de um elemento, achar o nó correspondente a ele no DOM ou até saltar para o código-fonte da aplicação direto no navegador. Esta última opção é bem útil, uma vez que lhe permite inserir *breakpoints* e depurar as suas funções JavaScript.

Exercícios:

- Instale a extensão React Developer Tools[128] no navegador de sua escolha
  - Rode a sua aplicação clone do Hacker News e a analise utilizando a extensão
  - Faça experiências mudando o estado e as *props* de componentes
  - Observe o que acontece quando você dispara requisições assíncronas
  - Faça múltiplas requisições, incluindo algumas repetidas. Observe o mecanismo de *cache* em funcionamento
- Leia sobre como depurar suas funções JavaScript no navegador[129]

[128] https://github.com/facebook/react-devtools
[129] https://developers.google.com/web/tools/chrome-devtools/javascript/

Você aprendeu como organizar o seu código e como testá-lo. Vamos recapitular:

- React
  - *PropTypes* lhe permite definir checagem de tipos para componentes
  - Jest lhe permite escrever *snapshot tests* para seus componentes
  - Enzyme lhe permite escrever testes unitários para seus componentes
  - React Developer Tools é uma ferramenta útil de *debug*
- ES6
  - Declarações *import* e *export* lhe ajudam a organizar o seu código
- Geral
  - Uma melhor organização de código lhe permite escalar sua aplicação com as melhores práticas

Você irá achar o código-fonte deste capítulo no repositório oficial[130].

[130] https://github.com/the-road-to-learn-react/hackernews-client/tree/5.4

Conceitos Avançados de Componentes React

Este capítulo irá focar na implementação de conceitos avançados de componentes React. Você irá aprender à respeito de *high-order components* e como implementá-los. Também irá mergulhar em outros tópicos mais avançados e implementar interações mais complexas em React.

Usando *ref* com um elemento do DOM

Às vezes, você precisa interagir com os nós da árvore do DOM (*DOM nodes*) em React. O atributo `ref` lhe dá acesso à um nó em seus elementos. Geralmente isto é um anti-padrão em React, porque você deveria utilizar a sua forma mais declarativa de fazer as coisas e também seu fluxo unidirecional de dados, como você bem aprendeu quando adicionou seu primeiro campo de *input* para consultas. Mas, existem certos casos onde você realmente precisa acessar um nó no DOM. A documentação oficial menciona três deles:

- para usar a API do DOM (focus, media playback, etc.)
- para invocar animações em nós do DOM
- para integrar-se com uma biblioteca de terceiros que precisa ter acesso à arvore de nós do DOM (exemplo: D3.js[131])

Façamos um exemplo com um componente *Search*. Quando a aplicação é renderizada pela primeira vez, o campo de *input* deveria receber o foco. Este é um caso onde você precisaria acessar a API do DOM. Este capítulo lhe mostrará como isto funciona mas, uma vez que não é muito útil para a aplicação em si, omitiremos estas mudanças em capítulos posteriores. Você pode mantê-las no seu próprio código, se quiser.

Em geral, você pode usar o atributo `ref` em ambos os tipos de componentes React (*functional stateless components* e componentes de classe ES6). No caso do foco, você precisará usar um método de ciclo de vida e, por isso, um componente de classe ES6 foi utilizado na demonstração.

O passo inicial é refatorar o componente funcional para passar a ser um componente de classe.

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}
```

[131] https://d3js.org/

```
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

O objeto `this` de um componente de classe ES6 nos ajuda a referenciar o *DOM node* com o atributo `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={(node) => { this.input = node; }}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

Agora, você pode definir o foco para o campo *input* quando o componente é montado, usando o objeto `this`, o método de ciclo de vida apropriado e a API do 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={(node) => { this.input = node; }}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

O campo *input* deverá receber o foco quando a aplicação é renderizada. Basicamente, este é o uso do atributo `ref`.

Mas, como você obteria acesso ao atributo `ref` em um *stateless functional component*, sem o objeto `this`? O código a seguir demonstra como fazê-lo:

src/App.js

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

Você agora estará apto a acessar o elemento *input* do DOM. Não funcionaria para o exemplo com o foco, pois você não tem acesso à métodos de ciclo de vida em um *stateless functional component* para disparar a ação de foco. Mas, no futuro, você pode se deparar com casos de uso onde faz sentido ter acesso ao atributo `ref` em um componente deste tipo.

Exercícios

- Leia mais sobre o uso do atributo ref em React[132]
- Leia mais sobre o atributo ref, em termos gerais, em React[133]

[132] https://www.robinwieruch.de/react-ref-attribute-dom-node/
[133] https://reactjs.org/docs/refs-and-the-dom.html

Loading ...

Voltemos à aplicação. Você pode querer exibir um indicador de carregamento (*loading*), quando faz uma requisição de busca na API Hacker News. A requisição é assíncrona e seria bom se você exibisse algum *feedback* para o usuário, demonstrando que algo ainda está para acontecer. Vamos definir um componente *Loading*, reutilizável, no seu arquivo *src/App.js*.

src/App.js

```
const Loading = () =>
  <div>Loading ...</div>
```

Você precisa agora de uma propriedade para armazenar o estado de *loading*. Baseado neste estado, você pode decidir se exibe ou não o componente *Loading*.

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,
    };

    ...
  }

  ...

}
```

O valor inicial da propriedade `isLoading` é *false*. Você não carrega nada antes do componente *App* ser montado.

Quando você faz a requisição, você altera o estado de *loading* par *true*. Eventualmente, a requisição irá ter sucesso e você poderá alterá-lo novamente para *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 }));
  }

  ...

}
```

Como último passo, você irá usar o componente *Loading* em *App*. Uma renderização condicional, baseada no estado de *loading*, irá determinar quando mostrar o componente *Loading* ou o componente *Button*. Este último é o botão que carrega mais dados.

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>
    );
  }
}
```

Inicialmente, o componente *Loading* irá aparecer quando você inicia a sua aplicação, porque você faz uma requisição em `componentDidMount()`. Não existe componente *Table*, porque a lista está vazia. Quando uma resposta é retornada da chamada à API Hacker News, o resultado é mostrado, o estado de *loading* é alterado para *false* e o componente *Loading* desaparece. No lugar dele, o botão "More" (que obtém mais dados) aparece. Uma vez que você obtém mais dados, o botão irá desaparecer novamente e *Loading* irá ser exibido.

Exercícios:

- Use uma biblioteca como Font Awesome[134] para exibir um ícone de *loading* no lugar do texto "Loading ..."

[134] http://fontawesome.io/

Higher-Order Components

Higher-order component (HOC) é um conceito avançado em React. HOCs são equivalentes a *higher-order functions*, pois recebem qualquer coisa como parâmetro de entrada - na maior parte do tempo um componente, mas também argumentos opcionais - e retornam um componente. O componente retornado é uma versão aprimorada do componente de entrada e pode ser usado em seu código JSX.

HOCs são utilizados em diferentes casos. Eles podem preparar propriedades, gerenciar estado ou alterar a representação de um componente. Um dos casos poderia ser o de utilizar o HOC como um utilitário para renderização condicional. Imagine que você tem um componente *List*, que renderiza ou uma lista de itens, ou nada (quando a lista é vazia ou tem valor *null*). O HOC poderia tratar e evitar o caso em que a lista não existe, fazendo com que o componente *List* não precise mais se preocupar com isso, focando apenas em renderizar a lista.

Vamos criar um HOC simples, que recebe um componente como entrada e retorna um outro componente. Você pode colocá-lo em seu arquivo *src/App.js*.

src/App.js

```
function withFoo(Component) {
  return function(props) {
    return <Component { ...props } />;
  }
}
```

Existe a convenção de prefixar o nome de HOCs com `with` e, uma vez que você usa JavaScript ES6, pode expressar o HOC de forma mais concisa com uma *arrow function*.

src/App.js

```
const withFoo = (Component) => (props) =>
  <Component { ...props } />
```

Neste exemplo, o componente de entrada é igual ao de saída. Nada acontece, por enquanto ele renderiza a mesma instância de componente e repassa todas as *props* para o componente de saída, o que é inútil. Vamos então aprimorar o componente retornado, que deveria mostrar o componente *Loading* quando o estado de *loading* é *true*. Caso contrário, deveria mostrar o componente que recebeu como entrada. Uma renderização condicional é um ótimo caso de uso para um HOC.

src/App.js

```
const withLoading = (Component) => (props) =>
  props.isLoading
    ? <Loading />
    : <Component { ...props } />
```

Baseado na propriedade de *loading*, você pode aplicar a renderização condicional. A função irá retornar o componente *Loading* ou o componente que recebeu via parâmetro.

Em geral, utilizar o operador *spread* em um objeto pode ser bastante eficiente, como aqui com o objeto *props*. Veja a diferença no seguinte trecho de código:

Code Playground

```
// antes, você teria que utilizar destructuring com as props, antes de passá-las
const { foo, bar } = props;
<SomeComponent foo={foo} bar={bar} />

// você pode, no entanto, utilizar o operador spread para passar todas as propriedad\
es do objeto
<SomeComponent { ...props } />
```

Existe mais uma pequena coisa que você deveria evitar. Você passa todas as propriedades para o componente de entrada, incluindo `isLoading`, utilizando operador *spread* no objeto. Contudo, o componente de *input* poderia ignorar a propriedade `isLoading`. Você pode utilizar *rest destructuring*, de JavaScript ES6, para evitar isso.

src/App.js

```
const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />
```

Note que, agora, uma propriedade é extraída explicitamente do objeto, permanecendo as outras agrupadas. Isto também funcionaria para múltiplas propriedades, inclusive. Você já deve ter lido mais à respeito de destructuring[135].

Você já pode usar o HOC em seu código JSX, no caso de exibir ou o botão "More", ou o componente *Loading*. Este último já foi encapsulado no HOC, mas falta o componente de entrada. No caso de uso de exibir um componente *Button* ou um componente *Loading*, *Button* é o componente de entrada do HOC. A saída aprimorada é um componente *ButtonWithLoading*.

[135] 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);
```

Finalmente, com tudo bem definido, damos o último passo. Você deve utilizar o componente *ButtonWithLoading*, que recebe o estado de *loading* como uma propriedade extra. Enquanto o HOC explicitamente usa esta propriedade, ele repassa todas as outras para o componente *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>
    );
  }
}
```

Quando você rodar seus testes novamente, irá notar que o *snapshot test* para o componente *App* falha. O *diff* deverá aparentar mais ou menos como mostrado a seguir, na linha de comando:

Linha de Comando

```
-     <button
-       className=""
-       onClick={[Function]}
-       type="button"
-     >
-       More
-     </button>
+     <div>
+       Loading ...
+     </div>
```

Mais uma vez, você tem a oportunidade consertar algo que possa estar errado no componente, ou aceitar o novo *snapshot*. Uma vez que você introduziu o componente *Loading* neste capítulo, você deve aceitá-lo na interface interativa apresentada na linha de comando após o teste.

O uso de *higher-order components* é uma técnica avançada em React. Eles têm múltiplos propósitos, como reuso, maior abstração, melhor composição de componentes e manipulação de *props*, estado e das suas *views*. Não se preocupe caso não entenda de imediato a utilização de HOCs. Leva algum tempo para se acostumar à eles.

Eu lhe encorajo a ler essa gentil introdução a *higher-order components*[136]. Ela lhe provê outra abordagem para aprender, mostrando um jeito elegante de usá-los com um paradigma de programação funcional e resolvendo especificamente o problema de renderização condicional com *higher-order components*.

Exercícios:

- Leia o artigo A gentle introduction to higher-order components[137]
- Faça experiências com o HOC que você criou
- Pense a respeito de algum caso onde outro HOC poderia fazer sentido
  - Implemente o HOC, se este caso existir

[136] https://www.robinwieruch.de/gentle-introduction-higher-order-components/
[137] https://www.robinwieruch.de/gentle-introduction-higher-order-components/

Ordenação Avançada

Você já implementou interações de busca do lado do cliente e do servidor. Uma vez que você tem um componente *Table*, faria sentido enriquecê-lo com interações mais avançadas. Que tal introduzir uma funcionalidade de ordenação (*sort*) para cada coluna, usando os seus cabeçalhos?

Seria possível escrever sua própria função de ordenação, mas eu pessoalmente prefiro usar uma biblioteca utilitária para tais casos. Lodash[138] é uma dessas bibliotecas, mas você pode utilizar qualquer uma que lhe convir. Vamos instalar *Lodash* e usá-la para a funcionalidade de ordenação.

Linha de Comando

```
npm install lodash
```

Você pode agora importar a funcionalidade *sortBy* de *Lodash* em seu arquivo *src/App.js*.

src/App.js

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

Existem várias colunas em *Table*. Título, autor, comentários e pontos (*title, author, comments _e _points*). Você pode definir funções de ordenação, onde cada uma recebe uma lista e retorna uma lista ordenada para uma propriedade específica. Em adição a isto, você irá precisar de uma função *default*, que não faz nenhuma ordenação, apenas retorna a lista desordenada. Este será o seu estado inicial.

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 {
  ...
}
...
```

[138] https://lodash.com/

Como você pode ver, duas das funções retornam uma lista inversamente ordenada. O motivo é que você desejará ver os items com o maior número de comentários ou de pontos no topo, ao invés de ver os valores mais baixos quando realizar a ordenação da lista.

O objeto `SORTS` lhe permite fazer referência a qualquer função de ordenação agora.

Mais uma vez, seu componente *App* é responsável por armazenar o estado da operação *sort*. O estado inicial será o *default*, que acaba não ordenando nada e devolvendo a mesma lista como saída.

src/App.js

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
  sortKey: 'NONE',
};
```

Quando você escolher uma `sortKey` diferente, digamos a chave `AUTHOR`, você irá ordenar a lista com a função mais apropriada do objeto `SORTS`.

Você pode agora definir um novo método de classe no seu componente *App*, que simplesmente define `sortKey` no estado local do seu componente. A `sortKey` poderá ser utilizada para obter a função de ordenação a ser aplicada em sua lista.

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);
  }

  ...
```

```
18
19    onSort(sortKey) {
20      this.setState({ sortKey });
21    }
22
23    ...
24
25  }
```

O próximo passo é passar o método e `sortKey` para seu componente *Table*.

src/App.js

```
1   class App extends Component {
2
3     ...
4
5     render() {
6       const {
7         searchTerm,
8         results,
9         searchKey,
10        error,
11        isLoading,
12        sortKey
13      } = this.state;
14
15      ...
16
17      return (
18        <div className="page">
19          ...
20          <Table
21            list={list}
22            sortKey={sortKey}
23            onSort={this.onSort}
24            onDismiss={this.onDismiss}
25          />
26          ...
27        </div>
28      );
29    }
30  }
```

O componente *Table* é responsável por ordenar a lista. Ele recebe uma das funções `SORT` através da `sortKey` e passa a lista como entrada. Ele então continua iterando sobre a lista ordenada com a função *map*.

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>
```

Em teoria, a lista deveria ser ordenada por alguma das funções. Mas, o valor *default* é `NONE`, logo nada acontece ainda. Até agora, ninguém executa o método `onSort()` para mudar `sortKey`. Iremos estender a tabela com uma linha de cabeçalhos de coluna, que usa componentes *Sort* para ordenar a lista em cada uma.

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>
```

Cada componente *Sort* recebe uma `sortKey` específica e a função `onSort()`. Internamente, ele chama o método com a `sortKey` para definir a chave específica.

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button onClick={() => onSort(sortKey)}>
    {children}
  </Button>
```

Como você pode ver, o componente *Sort* reutiliza seu componente genérico *Button*. Quando o botão for clicado, a `sortKey` passada irá ser definida com a ajuda do método `onSort()`. Agora você deve ser capaz de ordenar a lista quando clicar nos cabeçalhos das colunas.

Ainda é possível fazer uma pequena melhoria, para um visual melhor. Até então, o botão no *header* da coluna parece um pouco bobo. Vamos dar ao botão no componente *Sort* um `className` mais apropriado.

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button
    onClick={() => onSort(sortKey)}
    className="button-inline"
  >
    {children}
  </Button>
```

Deve parecer melhor agora. O próximo objetivo seria o de também implementar a ordenação inversa. A ordenação deveria ser invertida uma vez que você novamente clica no componente *Sort*. Primeiro, você precisa definir o estado booleano *reverse*. A ordenação poderá ser inversa, ou não.

src/App.js

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
  sortKey: 'NONE',
  isSortReverse: false,
};
```

Agora, em seu método de ordenação, você pode verificar se a lista terá a ordenação invertida. Isto irá acontecer se o estado local `sortKey` tiver o mesmo valor que a `sortKey` recebida e o estado `isSortReverse` não já estiver definido com o valor *true*.

src/App.js

```
onSort(sortKey) {
  const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortReverse;
  this.setState({ sortKey, isSortReverse });
}
```

Novamente, você pode passar a *prop* de ordenação invertida para o componente *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>
    );
  }
}
```

Table precisará ter uma um bloco de escopo de função explícito, para computar os dados agora.

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    <div className="table">
      <div className="table-header">
        ...
      </div>
      {reverseSortedList.map(item =>
        ...
      )}
    </div>
  );
}
```

A ordenação invertida deverá funcionar.

Por último, mas não menos importante, você deverá lidar com uma questão de melhoria de experiência do usuário. Poderia o usuário distinguir por qual coluna está sendo feita a ordenação no momento? Até agora, isto não é possível. Vamos fornecer este *feedback* visual.

Cada componente *Sort* já recebe sua `sortKey` específica. Ela poderia ser utilizada par identificar a ordenação corrente. Você pode passar a `sortKey` do estado interno do componente como a chave de ordenação ativa para seu componente de Sort.

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  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={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>
      {reverseSortedList.map(item =>
          ...
      )}
    </div>
  );
}
```

Agora, no seu componente *Sort*, você sabe qual ordenação está ativa, baseado na `sortKey` e na `activeSortKey`. Dê ao seu componente *Sort* um atributo extra `className`, para o caso dele estar ativo, provendo assim um *feedback* visual ao usuário.

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>
  );
}
```

A forma como definimos `sortClass` é um pouco bizarra, não acha? Existe uma pequena biblioteca que nos ajuda a consertar isto. Primeiro, você tem que instalá-la.

Linha de Comando

```
npm install classnames
```

Segundo, você precisa importá-la no topo do arquivo *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';
```

Agora será possível usá-la para definir o `className` do seu componente com classes condicionais.

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>
  );
}
```

Novamente, quando você rodar os seus testes, verá *snapshot tests* e agora também testes unitários falhando para o componente *Table*. Uma vez que você mudou novamente a representação do componente, pode acatar as mudanças de *snapshot*. Mas, você precisa consertar o teste unitário. Em seu arquivo *src/App.test.js*, você precisa fornecer uma `sortKey` e o valor booleano `isSortReverse` para o componente *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,
  };

  ...

});
```

Mais uma vez, você precisará aceitar as mudanças de *snapshot* para o componente *Table*, porque você forneceu novas *props* para ele.

Finalmente, sua interação avançada de ordenação está completa.

Exercícios:

- Use uma biblioteca como Font Awesome[139] para indicar a ordenação (inversa ou não)
  - Poderia ser um ícone de seta para cima ou para baixo, ao lado do cabeçalho *Sort*
- Leia mais sobre a biblioteca classnames[140]

[139] http://fontawesome.io/
[140] https://github.com/JedWatson/classnames

Você aprendeu técnicas avançadas em componentes React! Vamos recapitular:

- React
  - o atributo *ref* para referenciar *DOM nodes*
  - *higher-order components* são um jeito comum de construir componentes avançados
  - implementação de interações avançadas em React
  - *classNames* condicionais com uma biblioteca utilitária elegante
- ES6
  - uso de *rest destructuring* para separar objetos e *arrays*

Você irá encontrar o código fonte no repositório oficial[141].

[141] https://github.com/the-road-to-learn-react/hackernews-client/tree/5.5

Gerenciamento de Estado em React (e além)

Você já aprendeu o básico sobre gerenciamento de estado em React nos capítulos anteriores. Este capítulo "cava" um pouco mais fundo neste tópico. Você aprenderá as melhores práticas, como aplicá-las e por que você deveria considerar utilizar uma biblioteca de terceiros para gerenciar o estado de uma aplicação.

Realocando o Estado

Apenas *App* é um componente *stateful* (com estado) em sua aplicação. Ele lida com um bastante estado e lógica em seus métodos de classe. Talvez você tenha notado que passa muitas propriedades para o componente *Table* e muitas delas só são utilizadas nele mesmo. Podemos afirmar, então, que não faz sentido que *App* tenha conhecimento sobre elas.

A funcionalidade de ordenação, como um todo, só é utilizada no componente *Table*. Você pode movê-la para ele, *App* não precisa dela para nada. O processo de refatoração onde um pedaço do estado é movido de um componente para o outro é conhecido como realocação de estado (*lifting state*). No seu caso, você quer mover o estado que não é utilizado no componente *App* para o componente *Table*. O estado move-se para baixo, do componente pai para o filho.

Com o intuito de lidar com estado local e métodos de classe no componente *Table*, ele deverá se tornar um componente de classe ES6. A refatoração de um *stateless functional component* para um componente de classe é um processo simples.

Table como um *stateless functional component*:

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    ...
  );
}
```

Table como um componente 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 (
      ...
    );
  }
}
```

Uma vez que você deseja trabalhar com estado e métodos no seu componentes, precisa adicionar também um construtor e o estado inicial.

src/App.js

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

    this.state = {};
  }

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

Você pode agora mover o estado e os métodos de classe relacionados com a funcionalidade de ordenação do componente *App* para o componente *Table*.

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() {
    ...
  }
}
```

Não se esqueça de remover o estado que foi movido e o método `onSort()` do componente *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);
  }

  ...

}
```

Além disso, você pode fazer com que a API do componente *Table* fique mais enxuta. Remova as *props* que não são mais utilizadas, pois são valores tratados internamente no componente agora.

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>
    );
  }
}
```

Agora, você pode utilizar o método `onSort()` e o estado interno no seu componente *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'}
```

```
36                onSort={this.onSort}
37                activeSortKey={sortKey}
38              >
39                Author
40              </Sort>
41            </span>
42            <span style={{ width: '10%' }}>
43              <Sort
44                sortKey={'COMMENTS'}
45                onSort={this.onSort}
46                activeSortKey={sortKey}
47              >
48                Comments
49              </Sort>
50            </span>
51            <span style={{ width: '10%' }}>
52              <Sort
53                sortKey={'POINTS'}
54                onSort={this.onSort}
55                activeSortKey={sortKey}
56              >
57                Points
58              </Sort>
59            </span>
60            <span style={{ width: '10%' }}>
61              Archive
62            </span>
63          </div>
64          { reverseSortedList.map((item) =>
65            ...
66          )}
67        </div>
68      );
69    }
70  }
```

Sua aplicação deve continuar funcionando. Mas, você fez uma refatoração crucial, movendo funcionalidade e estado para outro componente. Outros componentes ficaram menos pesados, assim como a assinatura do componente *Table*, pois ele lida internamente com a funcionalidade de ordenação.

O processo de realocação de estado pode acontecer no sentido oposto, da mesma forma: de um componente filho, para componente pai. Podemos chamar este processo de “promoção de estado”

(*lifting state up*). Imagine que você estava lidando com o estado interno em um componente filho. Agora, você precisa atender à um requisito exibindo o estado também no componente pai. Você teria que promover este estado para o componente pai. Mas, iremos mais além. Imagine que você deseja exibir o mesmo estado em componentes "irmãos" (dois componentes com o mesmo componente pai). Mais uma vez, você terá que promover o estado para o componente pai, que irá tratar dele e expô-lo para ambos os componentes filhos.

Exercícios:

- Leia mais sobre realocação de estado em React[142]
- Leia mais sobre realocação do estado em aprenda React antes de utilizar Redux[143]

[142] https://reactjs.org/docs/lifting-state-up.html
[143] https://www.robinwieruch.de/learn-react-before-using-redux/

Revisitando: setState()

Até então, você utilizou `setState()` para gerenciar o estado interno de componentes. Você pode passar um objeto para a função, onde o estado interno pode ser parcialmente atualizado.

Code Playground

```
this.setState({ foo: bar });
```

Mas, `setState()` não recebe apenas um objeto. Em uma segunda versão, você pode passar uma função que atualiza o estado.

Code Playground

```
this.setState((prevState, props) => {
  ...
});
```

Por que você iria querer isto? Em um caso de uso crítico, onde faz sentido usar uma função ao invés de um objeto. É quando você atualiza o estado dependendo do estado ou de *props* anteriores. Se não fizer desta forma, podem ocorrer *bugs* relacionados ao gerenciamento interno do estado.

Mas e por que utilizar um objeto no lugar de uma função causaria *bugs* quando há dependência de estado e *props* anteriores? Porque o método `setState()` de React é assíncrono. React processa as chamadas de `setState()` em lote e pode acontecer de o estado ou *props* serem modificados no meio da execução da sua própria chamada de `setState()`.

Code Playground

```
const { fooCount } = this.state;
const { barCount } = this.props;
this.setState({ count: fooCount + barCount });
```

Imagine que `fooCount` e `barCount`, aqui estado e *props*, são modificados de forma assíncrona em outro ponto no momento em que você chama `setState()` aqui. Em uma aplicação que está ganhando maior escala, você tem mais de uma chamada de `setState()`. Uma vez que `setState()` é assíncrono, você fica dependendo dos valores atuais dos estados.

Com a abordagem funcional, a função em `setState()` é um *callback* que irá operar sobre o estado e as *props* que existiam no momento da sua execução. Mesmo `setState()` sendo assíncrono, este será o comportamento.

Code Playground

```
this.setState((prevState, props) => {
  const { fooCount } = prevState;
  const { barCount } = props;
  return { count: fooCount + barCount };
});
```

Agora, voltemos para o seu código, para consertar este comportamento. Vamos juntos fazê-lo em um lugar onde `setState()` é utilizado e depende do estado e das *props*. Você depois estará apto a aplicar a correção em outros lugares.

O método `setSearchTopStories()` depende do estado anterior e é, assim, um exemplo perfeito para utilizarmos uma função ao invés de um objeto em `setState()`. No momento, o trecho de código se parece com esse a seguir:

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
  });
}
```

Você extrai valores de *state*, mas o atualiza de forma assíncrona dependendo do estado anterior. É possível, então, utilizar a abordagem funcional para prevenir *bugs* causados por um estado viciado.

src/App.js

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

  this.setState(prevState => {
    ...
  });
}
```

Você pode mover o bloco inteiro implementado para dentro da função. A única modificação necessária é operar com `prevState` ao invés de `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
    };
  });
}
```

Isso irá resolver o problema do estado viciado. Existe ainda uma coisa que pode ser melhorada. Uma vez que você tem uma função, ela pode ser extraída, em nome de uma melhor legibilidade. Esta é mais uma vantagem de se utilizar uma função e não um objeto. A função pode existir até fora do componente, mas você tem que usar uma *higher-order function* para passar o resultado para ela.

No fim das contas você quer atualizar o estado com base o resultado obtido da API:

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  this.setState(updateSearchTopStoriesState(hits, page));
}
```

A função `updateSearchTopStoriesState()` precisa retornar uma função. Ela é uma *higher-order function* e pode ser definida fora do componente *App*. Note como a assinatura pode ser ligeiramente diferente agora.

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 {
  ...
}
```

É isto. A abordagem funcional de `setState()` resolve potenciais *bugs* e ainda aumenta a legibilidade e manutenabilidade do seu código. Torna-se também mais estável fora do componente *App*. Como exercício, você pode exportá-la e escrever um teste para ela.

Exercício:

- Leia mais sobre uso correto do estado em React[144]
- Exporte *updateSearchTopStoriesState* no seu arquivo
- Escreva um teste para, o qual passa o *payload* (hist, page) e um "estado anterior", com um *expect* para um novo estado
- refatore seus métodos `setState()` para a abordagem funcional
  - mas, somente quando fizer sentido, por ele depender de estado ou *props*
- Rode seus testes novamente e verifique se tudo foi atualizado

[144] https://reactjs.org/docs/state-and-lifecycle.html#using-state-correctly

Taming the State

Os capítulos anteriores lhe mostraram que o gerenciamento de estado pode ser um tópico crucial em aplicações de larga escala. Em geral, não apenas React, mas uma grande quantidade de *frameworks* SPA gastam bastante energia com o assunto. As aplicações ficaram ainda mais complexas em anos recentes e um dos maiores desafios para aplicações *web*, nos dias de hoje, é o de **domar** (do inglês "*tame*") e controlar o estado.

Comparado com outras soluções, React já deu um grande passo à frente. O fluxo unidirecional de dados e uma API simples para gerenciar estado em um componente são indispensáveis. Estes conceitos tornam mais leve a tarefa de raciocinar sobre seu estado e as mudanças sobre ele. É mais fácil pensar a nível de components e, até certo ponto, de aplicação também.

Em uma aplicação que está crescendo, fica mais difícil raciocinar sobre mudanças de estado. Você pode acabar introduzindo *bugs* operando sobre um estado viciado, quando passa um objeto para `setState()`. Você precisa ficar realocando o estado para lá e para cá, por necessidade de compartilhamento, além de ficar escondendo estados desnecessários em componentes. Pode acontecer de um componente precisar promover um estado, porque o componente irmão depende dele. Talvez este esteja muito longe na árvore de componentes e você terá que compartilhar o estado por toda ela. No fim, componentes têm um grande envolvimento no gerenciamento de estados. Mas, a responsabilidade maior de componentes deveria ser a de representar a UI, não é mesmo?

Por todos estes motivos é que existem soluções independentes para cuidar do gerenciamento de estado. Estas soluções não são utilizadas apenas em React, mas são elas que fazer do seu ecossistema um lugar tão poderoso. Você pode usar diferentes soluções para resolver seus problemas. Já pode ter ouvido falar das bibliotecas Redux[145] e MobX[146]. Você pode utilizar qualquer uma delas em sua aplicação React. Elas trazem extensões, react-redux[147] e mobx-react[148] (respectivamente) para integrá-las na camada de visão de React.

Redux e MobX estão fora do escopo deste livro. Quando você terminar de lê-lo, irá ser guiado sobre como pode continuar a aprender React e seu ecossistema. Um dos caminhos que pode ser seguido é o aprendizado de Redux. Mas, antes de você mergulhar no tópico de gerenciamento externo de estado, recomendo que leia este artigo[149]. Ele visa dar-lhe um melhor entendimento sobre como aprender este assunto.

Exercícios:

- Leia mais sobre gerenciamento externo de estado e como aprendê-lo[150]
- Confira meu segundo *e-book* sobre gerenciamento de estado em React[151]

[145] http://redux.js.org/docs/introduction/
[146] https://mobx.js.org/
[147] https://github.com/reactjs/react-redux
[148] https://github.com/mobxjs/mobx-react
[149] https://www.robinwieruch.de/redux-mobx-confusion/
[150] https://www.robinwieruch.de/redux-mobx-confusion/
[151] https://roadtoreact.com/

Você aprendeu técnicas avançadas de gerenciamento de estado em React! Vamos recapitular:

- React
  - realocação do estado para cima e para baixo, para componentes mais adequados
    * setState pode usar uma função para prevenir *bugs* relacionados a um estado viciado
    * soluções externas existentes que lhe ajudam a domar o estado (*tame the state*)

Você pode encontrar o código-fonte no repositório oficial[152].

[152] https://github.com/the-road-to-learn-react/hackernews-client/tree/5.6

Etapas Finais para Produção

Os últimos capítulos irão mostrar como implantar sua aplicação em produção. Você irá utilizar o serviço grátis de hospedagem *Heroku*. No meio do caminho, você irá aprender mais sobre o *create-react-app*.

Ejetando

O passo (e o conhecimento) a seguir **não é realmente necessário** para que sua aplicação seja implantada em produção. Mesmo assim, quero explicá-lo para você. *create-react-app* traz uma funcionalidade para deixá-lo extensível e também prevenir amarrações, que podem ocorrer quando você compra uma tecnologia que não lhe dá possibilidades de livrar-se dela no futuro. Felizmente, *create-react-app* possui está válvula de escape com "eject".

No seu arquivo *package.json*, você encontrará os *scripts* para iniciar (*start*), testar (*test*) e construir (*build*) sua aplicação. Um último *script* é *eject*. Uma vez que você use ele, não existe caminho de volta. **É uma operação definitiva**. **Uma vez que você ejetou sua aplicação**, **não é possível voltar atrás!** Se você apenas iniciou o aprendizado de React, não faz muito sentido abandonar o ambiente conveniente provido pelo *create-react-app*.

Se você executar `npm run eject`, o comando irá copiar todas as configurações e dependências para o seu *package.json* e para uma nova pasta *config/*. Você converteria o projeto inteiro em uma configuração customizada, com ferramentas que incluem *Babel* e *Webpack*. No final, você teria total controle sobre elas.

A documentação oficial recomenda *create-react-app* para projetos pequenos e médios. Não se sinta obrigado a utilizar o comando "eject" nas suas aplicações.

Exercícios:

- Leia mais sobre eject[153]

[153] https://github.com/facebookincubator/create-react-app#converting-to-a-custom-setup

Implante sua Aplicação

Nenhuma aplicação deveria permanecer para sempre em *localhost*, você deve querer publicá-la. *Heroku* é uma plataforma em forma de serviço onde você pode hospedar seus *apps*. Ele oferece uma suave integração com React, sendo possível implantar uma aplicação criada com *create-react-app* em minutos, sem configuração alguma, seguindo a mesma filosofia desta ferramenta.

Você precisa cumprir dois passos antes de poder implantar sua aplicação no *Heroku*:

- instalar a Heroku CLI[154]
- criar uma conta gratuita no Heroku[155]

Se você possui o *Homebrew* no seu computador, poderá instalar a Heroku CLI direto da linha de comando:

Linha de Comando

```
brew update
brew install heroku-toolbelt
```

Agora, você pode utilizar *git* e *Heroku CLI* para implantar a sua aplicação.

Linha de Comando

```
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
```

É o bastante. Espero que sua aplicação esteja rodando agora. Se tiver algum problema, você pode recorrer aos seguintes recursos:

- Git and GitHub Essentials[156]
- Deploying React with Zero Configuration[157]
- Heroku Buildpack for create-react-app[158]

[154] https://devcenter.heroku.com/articles/heroku-command-line
[155] https://www.heroku.com/
[156] https://www.robinwieruch.de/git-essential-commands/
[157] https://blog.heroku.com/deploying-react-with-zero-configuration
[158] https://github.com/mars/create-react-app-buildpack

Outline

Foi-se o último capítulo do livro. Espero que você tenha gostado da leitura e que ele tenha ajudado a deixar React um pouco mais popular no seu conceito. Se você gostou do livro, compartilhe-o como uma forma de aprender React com seus amigos. Ele pode ser dado a quem você quiser. Também significaria muito para mim se você puder dedicar 5 minutos do seu tempo, para escrever um *review* sobre ele na Amazon[159] ou no Goodreads[160].

Então, onde devo ir, agora que li este livro? Você pode estender a aplicação por conta própria ou até tentar construir o seu primeiro projeto React. Antes de mergulhar em outro livro, curso ou tutorial, você deveria colocar a mão na massa com um projeto. Faça-o durante uma semana, coloque-o em produção em algum lugar e me avise[161] para divulgá-lo. Eu estou curioso sobre o que você irá construir depois de ter consumido o livro.

Se você está procurando ir mais além com sua aplicação, posso recomendar vários caminhos distintos de aprendizado, uma vez que você utilizou apenas React puro neste livro:

- **Gerenciamento de Estado**: Você utilizou `setState()` e `this.state` de React para gerenciar e acessar o estado local de componentes. Este é o jeito ideal de começar. Contudo, em uma aplicação maior, você irá testar os limites do estado local de componentes React[162]. Você pode, portanto, utilizar uma biblioteca de terceiros para gerenciamento de estados como Redux ou MobX[163]. Na plataforma de cursos Road to React[164], você irá encontrar o curso "Taming the State in React", que ensina o gerenciamento avançado de estado em React, com Redux e MobX. Ele disponibiliza um e-book, mas eu recomendo a todos que mergulhem fundo no código-fonte e nos *screencasts* que o acompanham. Se você gostou deste livro, definitivamente deveria levar o Taming the State in React.
- **Conexão com um Banco de Dados e/ou Autenticação**: Em uma aplicação React que está crescendo, você pode, eventualmente, querer persistir dados. Eles devem ser armazenados em um banco de dados para que possam sobreviver entre diferentes sessões de um navegador e serem compartilhados entre diferentes usuários da sua aplicação. A forma mais simples de introduzir um banco de dados é utilizando Firebase. Neste abrangente tutorial[165], você irá encontrar um passo a passo sobre como utilizar autenticação Firebase (sign up, sign in, sign out, ...) em React. Além disso, você irá utilizar o banco de dados Firebase para armazenar entidades do usuário. Depois disso, ficará a seu critério armazenar ou não mais dados no banco de dados da sua aplicação.

[159] https://www.amazon.com/dp/B077HJFCQX
[160] https://www.goodreads.com/book/show/37503118-the-road-to-learn-react
[161] https://twitter.com/rwieruch
[162] https://www.robinwieruch.de/learn-react-before-using-redux/
[163] https://www.robinwieruch.de/redux-mobx-confusion/
[164] https://roadtoreact.com/
[165] https://www.robinwieruch.de/complete-firebase-authentication-react-tutorial/

- **Adicionando ferramentas com Webpack e Babel**: Neste livro, você utilizou *create-react-app* para configurar sua aplicação. Em algum momento futuro, quando você já tiver aprendido React, pode querer conhecer melhor o ferramental que o cerca, que lhe habilita a configurar seu próprio projeto sem o *create-react-app*. Eu recomendo seguir com um *setup* mínimo com Webpack e Babel[166]. Depois, você pode aplicar mais ferramentas por sua própria conta. Por exemplo, você poderia usar ESLint[167] para seguir um estilo de código unificado em sua aplicação.
- **Sintaxe de Componentes React**: As possibilidades e melhores práticas de implementação de componentes React evoluem com o tempo. Você encontrará formas de escrever seus componentes React, especialmente componentes de classe em outros materiais de aprendizado. Você pode baixar esse repositório do GitHub[168] para descobrir um jeito alternativo de escrever componentes de classe React. No futuro, você poderá escrevê-los ainda mais concisos utilizando declarações de campos de classe.
- **Outros Projetos**: Depois de aprender React puro, sempre é bom aplicar os conhecimentos adquiridos antes em seus projetos, antes de partir para aprender algo novo. Você poderia escrever seu próprio jogo da velha ou uma simples calculadora em React. Existe uma abundância de tutorials por aí, que usam apenas React para construir coisas excitantes. Confira os meus, sobre a rolagem de uma lista paginada infinita[169], um *showcase _ de tweets][12] ou [conectando sua aplicação React ao _Stripe* para efetuar cobranças em dinheiro[170]. Experimente essas mini aplicações para ficar confortável com React.
- **Componentes de UI**: Não cometa o erro de investir muito cedo no aprendizado de uma bibliotecas de componentes de UI no seu projeto. Primeiro, você deveria aprender como implementar e usar um *dropdown*, um *checkbox* ou uma caixa de diálogo em React com elementos de HTML puro, do zero. A maior parte desses componentes irá gerenciar o seu próprio estado local. Uma *checkbox* precisa saber quando está marcada ou não. Desta forma, você deveria implementá-los como componentes controlados. Depois que passar por todas as implementações base, você pode se iniciar no uso de uma biblioteca de componentes de UI, que lhe dá *checkboxes* e caixas de diálogo como componentes React.
- **Organização de Código**: Na sua jornada lendo o livro, você se deparou com um capítulo sobre organização de código. Você pode aplicar essas mudanças agora, se ainda não o fez. Organizar seus componentes em arquivos e pastas estruturados (módulos). Ademais, isto lhe ajudaria a entender e aprender princípios de separação, reusabilidade e manutenabilidade de código, além do projeto (*design*) de API de módulos.
- **Testes**: O livro apenas arranhou a superfície da disciplina de testes. Se você não é familiarizado com o tópico em geral, pode mergulhar mais fundo nos conceitos de testes unitários e de integração, especialmente no contexto de aplicações React. Em termos de implementação, eu recomendaria continuar com *Enzyme* e *Jest* para refinar sua técnica escrevendo testes de unidade e *snapshot tests* em React.

[166] https://www.robinwieruch.de/minimal-react-webpack-babel-setup/
[167] https://www.robinwieruch.de/react-eslint-webpack-babel/
[168] https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax
[169] https://www.robinwieruch.de/react-paginated-list/
[170] https://www.robinwieruch.de/react-express-stripe-payment/

- ***Routing***: Você pode implementar o roteamento de páginas para sua aplicação com react-router[171]. Até então, você só possui uma página na aplicação. React Router lhe ajuda a ter várias páginas através de múltiplas URLs. Quando você adiciona o roteamento de páginas à sua aplicação, nenhuma requisição à um servidor é feita para obter a próxima página. O *router* irá fazer todo o seu trabalho do lado do cliente. Suje suas mãos, então, adicionando *routing* à sua aplicação.
- **Checagem de Tipos**: Em um capítulo, você utilizou React *PropTypes* para definir interfaces de componentes. É uma boa prática comum para prevenir *bugs*. Mas, *PropTypes* só são checados em tempo de execução. Você pode ir um passo além, adicionando a checagem estática de tipos em tempo de compilação. TypeScript[172] é uma abordagem popular mas, no ecossistema de React, as pessoas geralmente utilizam Flow[173]. Eu recomento que você faça um teste com Flow, se estiver interessado a fazer com que suas aplicações sejam mais robustas.
- **React Native**: React Native[174] lhe leva ao mundo dos dispositivos móveis. Você pode aplicar seus novos conhecimentos em React para entregar aplicativos iOS e Android. A curva de aprendizado de React Native, uma vez que você já sabe React, não deve ser muito íngrime. Ambos compartilham dos mesmos princípios. Você apenas irá encontrar componentes de *layout* diferentes no mundo *mobile*, em relação aos que usa em aplicações web.

Em geral, eu o convido a visitar meu website[175], para encontrar mais tópicos interessantes sobre desenvolvimento web e engenharia de software. Você pode assinar minha Newsletter[176] para receber novidades sobre artigos, livros e cursos. Além disso, a plataforma de cursos Road to React[177] oferece mais cursos avançados sobre o ecossistema de React. Vá lá e confira!

Por último, mas não menos importante, espero encontrar mais Patrons[178] (patrocinadores) aptos a patrocinar meu conteúdo. Existem muitos estudantes por aí que não podem dar conta de pagar por conteúdo educacional. Por este motivo, eu disponibilizo boa parte do meu conteúdo de forma gratuita. Apoiando meu trabalho como um Patron, garante que eu posso continuar meus esforços em prover educação gratuita.

Mais uma vez, se você gostou do livro, peço-lhe que reserve um momento para pensar em alguém que seria uma boa indicação para aprender React. Vá até esta pessoa e compartilhe o livro, significaria muito para mim. Ele é feito para ser passado para outras pessoas e irá melhorar ao longo do tempo, à medida que mais pessoas o lêem e compartilham suas opiniões de *feedback* comigo. Espero também ver seu *feedback*, sua avaliação ou comentário, também!

Muito obrigado por ler **The Road to learn React**.

Robin

[171] https://github.com/ReactTraining/react-router
[172] https://www.typescriptlang.org/
[173] https://flowtype.org/
[174] https://facebook.github.io/react-native/
[175] https://www.robinwieruch.de
[176] https://www.getrevue.co/profile/rwieruch
[177] https://roadtoreact.com
[178] https://www.patreon.com/rwieruch