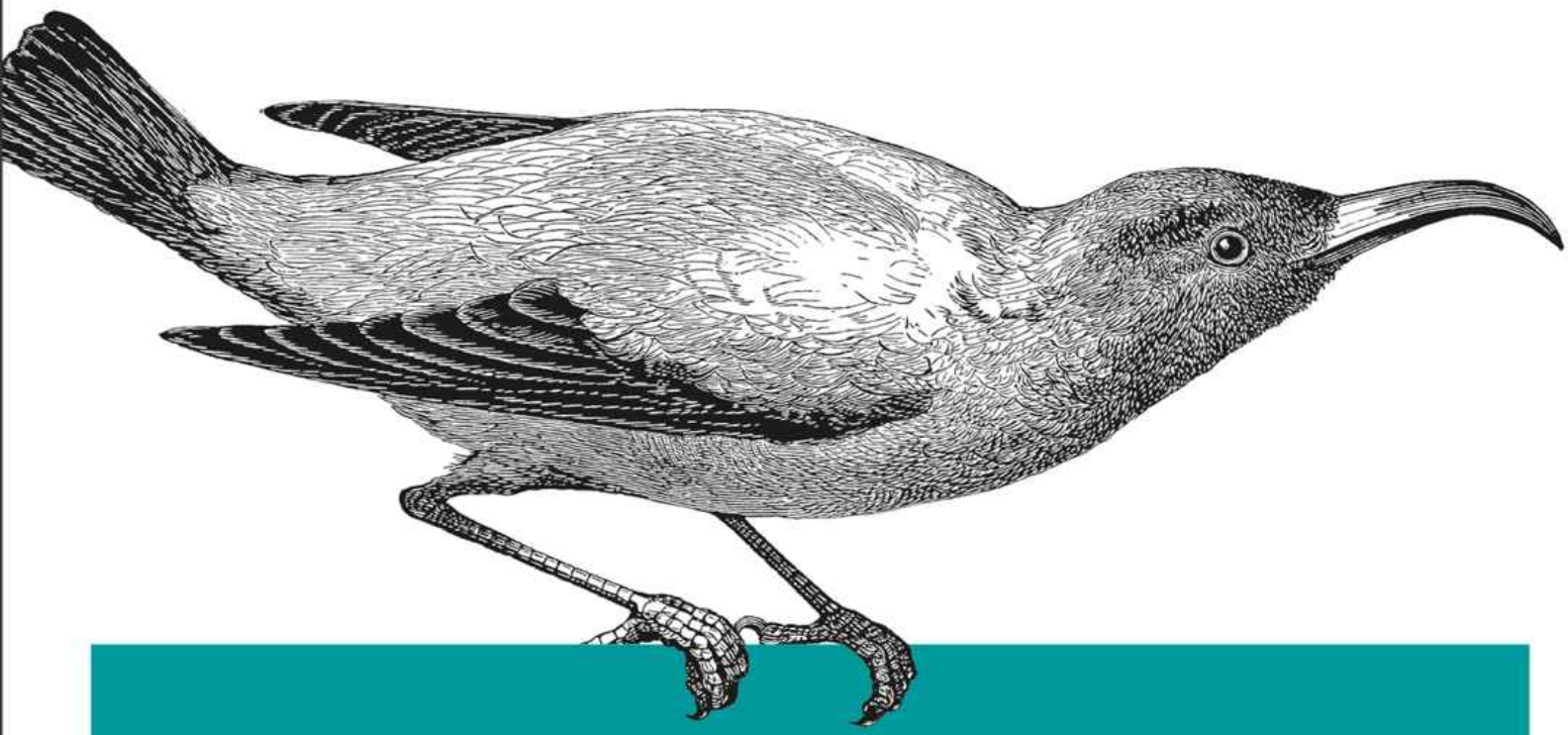
O'REILLY®



Primeiros passos com React

CONSTRUINDO APLICAÇÕES WEB

novatec

Stoyan Stefanov

Stoyan Stefanov

Novatec
São Paulo | 2019

Tradução em português autorizada da edição em inglês da obra React: Up & Running ISBN 9781491931820 © 2016 Stoyan Stefanov. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

Editor: Rubens Prates
Tradução: Lúcia A. Kinoshita
Revisão gramatical: Smirna Cavalheiro
Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-766-4

Histórico de edições impressas:

Setembro/2016 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Para Eva, Zlatina e Nathalie

Sumário

Sobre o autor

Colofão

Prefácio

Outra noite quente e maravilhosa na Califórnia. A brisa suave do oceano simplesmente contribui para você se sentir 100% “aaah!”. O lugar: Los Angeles; o ano: 2000 e alguma coisa. Eu havia acabado de me preparar para fazer um FTP de minha pequena aplicação web chamada CSSsprites.com para o meu servidor e lançá-la ao mundo. Fiquei pensando em um problema nas últimas noites que passei trabalhando na aplicação: Por que cargas d’água eu gastei 20% dos esforços para concluir o “cerne” da aplicação e, então, 80% brigando com a interface de usuário? Quantas outras ferramentas eu poderia ter criado se não tivesse que usar `getElementById()` o tempo todo e me preocupar com o estado da aplicação? (O usuário acabou de fazer uploading? O que houve, um erro? Esse diálogo continua ativo?) Por que o desenvolvimento de UI consome tanto tempo? E o que dizer de todos os navegadores diferentes? Lentamente, meu “aaah” começou a se transformar em um “aarrggh!”.

Vamos avançar rapidamente para março de 2015 na conferência F8 do Facebook. A equipe da qual faço parte está pronta para anunciar uma reescrita completa de duas aplicações web: nosso produto para comentários de terceiros e uma ferramenta de moderação para acompanhá-lo. Em comparação com minha pequena aplicação CSSsprites.com, essas eram aplicações web completas, com muito mais recursos, bem mais eficazes e com volumes insanos de tráfego. Apesar disso, o desenvolvimento foi feito com muito prazer. Colegas de equipe, a quem a aplicação era uma novidade (e algumas pessoas para quem até mesmo JavaScript e CSS eram novidade), puderam chegar e contribuir com um recurso aqui e uma melhoria ali, ganhando agilidade de forma rápida e tranquila. Como disse um dos membros da equipe, “Ah-ha, agora vejo o porquê de

tanto amor!”.

O que aconteceu no caminho? A React.

A React é uma biblioteca para construção de UIs – ela ajuda você a definir a UI de uma vez por todas. Então, quando o estado da aplicação mudar, a UI será reconstruída de modo a *reagir* à alteração, e você não precisará fazer nada adicional. Afinal de contas, você já definiu a UI. Definiu? Está mais para *declarou*. Você utiliza *componentes* pequenos e administráveis para construir uma aplicação eficaz de grande porte. Nada de gastar metade do corpo de sua função procurando nós do DOM; tudo que você deve fazer é manter o estado (state) de sua aplicação (com um velho objeto JavaScript comum), e o restante simplesmente se resolve.

Conhecer a React é um bom negócio – você aprende a usar uma biblioteca e a utiliza para criar todos os itens a seguir:

- aplicações web;
- aplicativos iOS e Android nativos;
- aplicações canvas;
- aplicações de TV;
- aplicações desktop nativas.

Você pode criar aplicações nativas com desempenho e controles nativos (controles *realmente* nativos, e não cópias com aparência nativa) usando as mesmas ideias de construção de componentes e UIs. Não se trata de “escrever uma vez, executar em todos os lugares” (nosso mercado continua falhando nesse caso); trata-se de “aprender uma vez, usar em todos os lugares).

Para resumir uma história longa: conheça a React, ganhe 80% de seu tempo de volta e concentre-se nos assuntos que realmente importam (por exemplo, o verdadeiro motivo da existência de sua aplicação).

Sobre este livro

Este livro tem como foco conhecer a React do ponto de vista do desenvolvimento web. Nos três primeiros capítulos, você começará simplesmente com um arquivo HTML em branco e continuará desenvolvendo-o a partir daí. Isso permite que você se concentre em conhecer a React, e não nas novas sintaxes ou ferramentas auxiliares.

O Capítulo 4 apresenta o JSX, que é uma tecnologia separada e opcional, geralmente usada em conjunto com a React.

A partir daí você saberá o que é necessário para desenvolver uma aplicação do mundo real e conhecerá ferramentas adicionais que poderão ajudar você no caminho. Exemplos incluem as ferramentas JavaScript de empacotamento (Browserify), testes de unidade (Jest), linting (ESLint), tipos (Flow), organização de fluxo de dados na aplicação (Flux) e dados imutáveis (Immutable.js). Todas as discussões sobre tecnologias auxiliares foram mantidas em um nível mínimo para que o foco permaneça na React; você adquirirá familiaridade com essas ferramentas e será capaz de tomar uma decisão bem fundamentada acerca de qual delas usar.

Boa sorte em sua jornada de aprendizado da React – que ela seja tranquila e produtiva!

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica termos novos, URLs, endereços de email, nomes e extensões de arquivos.

Largura constante

Usada para listagens de programas, assim como em parágrafos para se referir a elementos de programas, como nomes de variáveis ou de funções, bancos de dados, tipos de dados, variáveis de ambiente, comandos e palavras-chave.

Largura constante em negrito

Mostra comandos ou outro texto que devam ser digitados literalmente pelo usuário.

Largura constante em itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou determinados pelo contexto.

Este elemento significa uma dica ou sugestão.

Este elemento significa uma observação geral.

Este elemento significa um aviso ou uma precaução.

Uso de exemplos de código de acordo com a política da O’Reilly

Materiais suplementares (exemplos de código, exercícios etc.) estão disponíveis para download em *https://github.com/stoyan/reactbook*.

Este livro está aqui para ajudá-lo a fazer seu trabalho. De modo geral, se este livro incluir exemplos de código, você poderá usar o código em seus programas e em sua documentação. Você não precisa nos contatar para pedir permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use diversas partes de código deste livro não requer permissão. Porém vender ou distribuir um CD-ROM de exemplos de livros da O’Reilly requer permissão. Responder a uma pergunta mencionando este livro e citar o código de exemplo não requer permissão. Em contrapartida, incluir uma quantidade significativa de código de exemplos deste livro na documentação de seu produto requer permissão.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: “*React: Up & Running* de Stoyan Stefanov (O’Reilly). Copyright 2016

Stoyan Stefanov, 978-1-491-93182-0.”

Se você achar que o seu uso dos exemplos de código está além do razoável ou da permissão concedida, sinta-se à vontade em nos contatar em *permissions@oreilly.com*.

Como entrar em contato com a Novatec

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: *novatec@novatec.com.br*.

Temos uma página web para este livro na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português

  http://www.novatec.com.br/livros/primeiros-passos-com-react

- Página da edição original em inglês

  http://shop.oreilly.com/product/0636920042266.do

Para obter mais informações sobre os livros da Novatec, acesse nosso site em *http://www.novatec.com.br*.

Agradecimentos

Gostaria de agradecer a todos que leram diferentes versões preliminares deste livro e enviaram feedback e correções: Andreea Manole, Iliyan Peychev, Kostadin Ilov, Mark Duppenthaler, Stephan Alber, Asen Bozhilov.

Agradeço a todo o pessoal do Facebook que trabalha na React (ou com ela) e respondeu às minhas perguntas dia após dia. Também agradeço à comunidade estendida da React que continua gerando ótimas ferramentas, bibliotecas, artigos e padrões de uso.

Muitos agradecimentos a Jordan Walke.

Obrigado a todos da O’Reilly que possibilitaram a existência deste livro: Meg Foley, Kim Cofer, Nicole Shelby e muitos outros.

Obrigado a Yavor Vatchkov, que fez o design da UI da aplicação de

exemplo desenvolvida neste livro (você pode testá-la em *whinepad.com*).

CAPÍTULO 1

Hello World

Vamos dar início à jornada para dominar o desenvolvimento de aplicações usando a React. Neste capítulo você aprenderá a instalar a React e escrever sua primeira aplicação web “Hello World”.

Instalação

Antes de mais nada, você precisa obter uma cópia da biblioteca React. Felizmente, esse processo é extremamente simples.

Acesse *http://reactjs.com* (você deverá ser redirecionado para a página oficial do GitHub em *http://facebook.github.io/react/*); em seguida, clique no botão “Download”, depois em “Download Starter Kit” (Download do Kit de Iniciante) e você obterá a cópia de um arquivo ZIP. Descompacte e copie o diretório contido no arquivo de download para um local em que você possa encontrá-lo.

Por exemplo:

```
mkdir ~/reactbook
mv ~/Downloads/react-0.14.7/ ~/reactbook/react
```

Agora, o seu diretório de trabalho (*reactbook*) deverá ter a aparência mostrada na Figura 1.1.

O único arquivo de que você precisa para começar é *~/reactbook/react/build/react.js*. Você conhecerá os demais arquivos à medida que prosseguir.

Observe que a React não impõe nenhuma estrutura de diretório; você tem a liberdade de usar um diretório diferente ou renomear *react.js* do modo que achar mais apropriado.

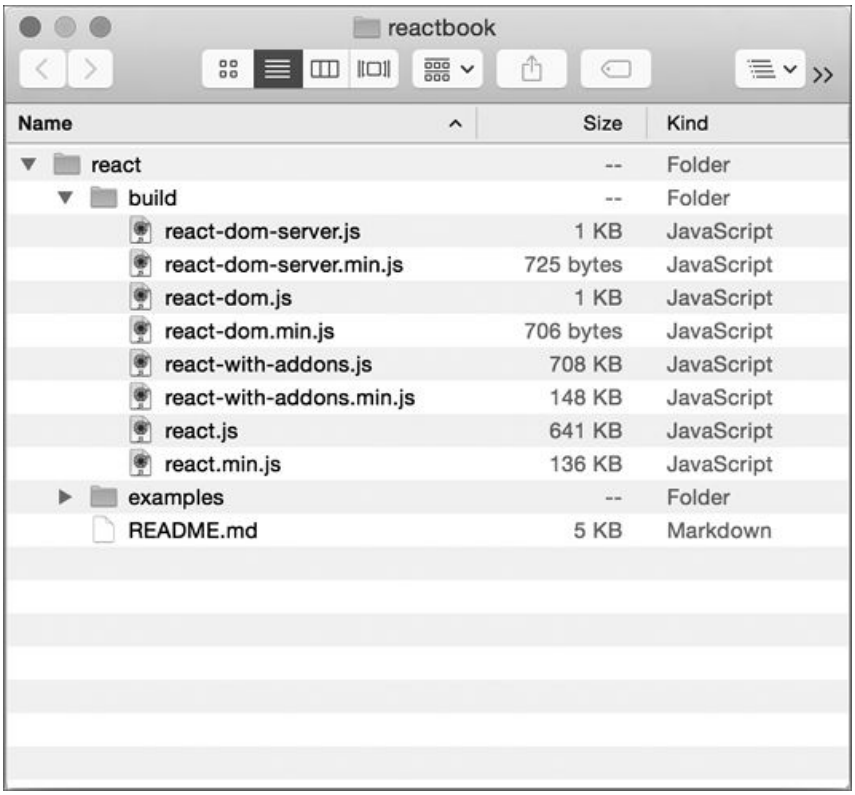


Figura 1.1 – A listagem de seu diretório da React.

Hello World com a React

Vamos começar com uma página simples em seu diretório de trabalho (*~/reactbook/01.01.hello.html*):

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello React</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="app">
      <!-- minha aplicação é renderizada aqui -->
    </div>
    <script src="react/build/react.js"></script>
    <script src="react/build/react-dom.js"></script>
    <script>
      // código de minha aplicação
    </script>
```

```
  </body>
</html>
```

Você pode encontrar todo o código deste livro no repositório que o acompanha (*https://github.com/stoyan/reactbook/*).

Há apenas dois aspectos dignos de nota acontecendo nesse arquivo:

- você incluiu a biblioteca React e seu add-on de DOM (por meio das tags `<script src>`);
- você definiu o local em que sua aplicação deve ser colocada na página (`<div id="app">`).

É sempre possível misturar conteúdo HTML comum bem como outras bibliotecas JavaScript com uma aplicação React. Você também pode ter várias aplicações React na mesma página. Tudo que você precisa é de um lugar no DOM para o qual possa apontar a React e dizer “faça sua mágica aqui”.

Vamos agora adicionar o código que diz “hello” – atualize *01.01.hello.html* e substitua `// código de minha aplicação` por:

```
ReactDOM.render(
  React.DOM.h1(null, "Hello World!"),
  document.getElementById("app")
);
```

Carregue *01.01.hello.html* em seu navegador e você verá sua nova aplicação em ação (Figura 1.2).

Parabéns, você acabou de criar sua primeira aplicação React!

A Figura 1.2 também mostra o código *gerado no* Chrome Developer Tools (Ferramentas do desenvolvedor do Chrome); nesse local, você pode ver que o conteúdo do placeholder `<div id="app">` foi substituído pelo conteúdo gerado pela sua aplicação React.

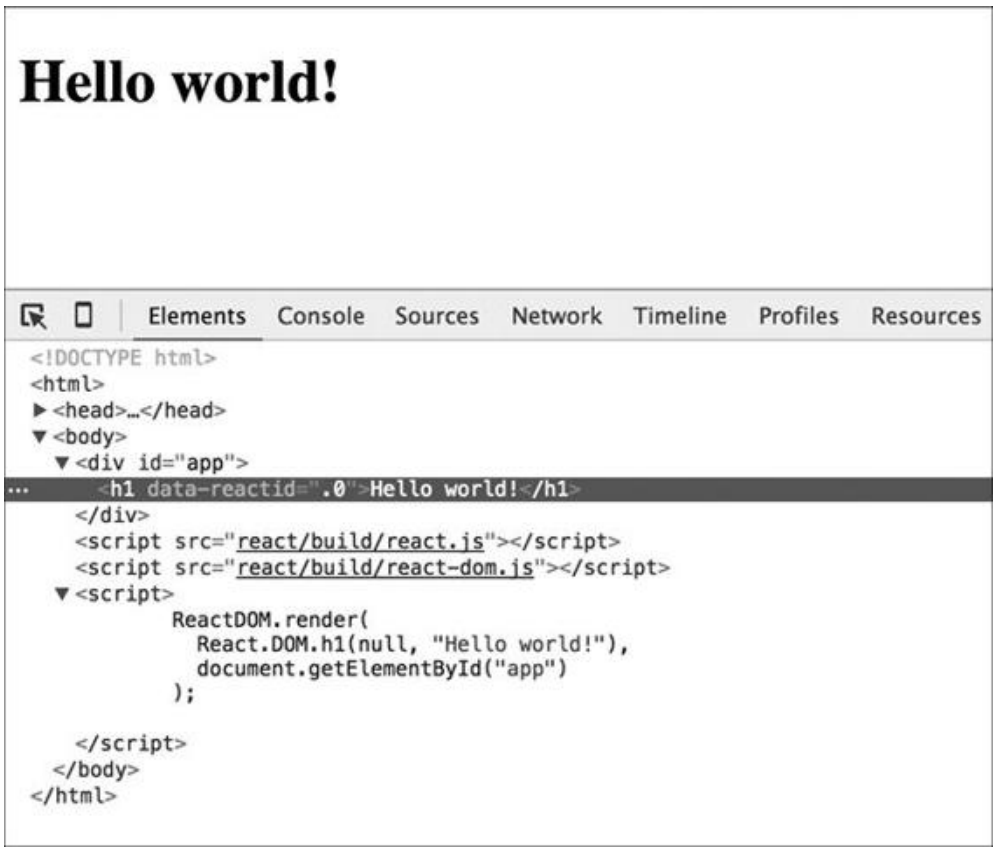


Figura 1.2 – Hello World em ação.

O que aconteceu?

Há alguns aspectos interessantes no código que fizeram sua primeira aplicação funcionar.

Inicialmente, você pode ver o uso do objeto React. Todas as APIs disponíveis a você são acessíveis por meio desse objeto. A API, na verdade, é propositalmente mínima, portanto não há muitos nomes de métodos a serem lembrados.

Você também pode ver o objeto ReactDOM. Ele tem apenas alguns métodos, entre os quais render() é o mais útil. Esses métodos anteriormente faziam parte do objeto React, porém, desde a versão 0.14, eles foram separados para enfatizar o fato de que a verdadeira renderização da aplicação é uma preocupação distinta. Você pode criar uma aplicação React para ser renderizada em ambientes diferentes – por exemplo, HTML (o DOM do navegador), canvas ou

de forma nativa no Android ou no iOS.

Em seguida, temos o conceito de *componentes.* Você constrói sua UI usando componentes e os combina de qualquer maneira que considerar apropriada. Em suas aplicações, você acabará criando seus próprios componentes personalizados, mas para que você possa começar, a React oferece wrappers em torno dos elementos HTML do DOM. Utilize os wrappers por meio do objeto `React.DOM`. Nesse primeiro exemplo, você pode ver o uso do componente `h1`. Ele corresponde ao elemento HTML `<h1>` e está disponível por meio da chamada a `React.DOM.h1()`.

Por fim, você pode ver o bom e velho acesso ao DOM com `document.getElementById("app")`. Use isso para dizer à React em que lugar a aplicação deverá estar localizada na página. Essa é a ponte que liga a manipulação de DOM como você a conhece para o território da React.

Após ter cruzado a ponte do DOM para a React, você não precisará mais se preocupar com a manipulação de DOM, pois a React faz a tradução dos componentes para a plataforma subjacente (DOM do navegador, canvas, aplicação nativa). Você *não precisa* se preocupar com o DOM, mas isso não quer dizer que não possa fazê-lo. A React oferece “pontos de escape” se você quiser voltar ao território do DOM por algum motivo necessário.

Agora que você já sabe o que cada linha faz, vamos dar uma olhada no quadro geral. O que aconteceu foi o seguinte: você renderizou um componente da React em um local do DOM de sua escolha. Você sempre renderiza um componente de nível mais alto, e ele pode ter tantos componentes filhos (e netos etc.) quantos forem necessários. De fato, mesmo nesse exemplo simples, o componente `h1` tem um filho – o texto “Hello World!”.

React.DOM.*

Como você já sabe agora, é possível usar vários elementos HTML como componentes da React por meio do objeto `React.DOM` (a Figura 1.3 mostra como obter uma lista completa usando o console de seu navegador). Vamos observar essa API com mais detalhes.

Perceba a diferença entre React.DOM e ReactDOM. O primeiro é uma coleção de elementos HTML prontos, e o segundo é uma maneira de renderizar a aplicação no navegador (pense em ReactDOM.render()).

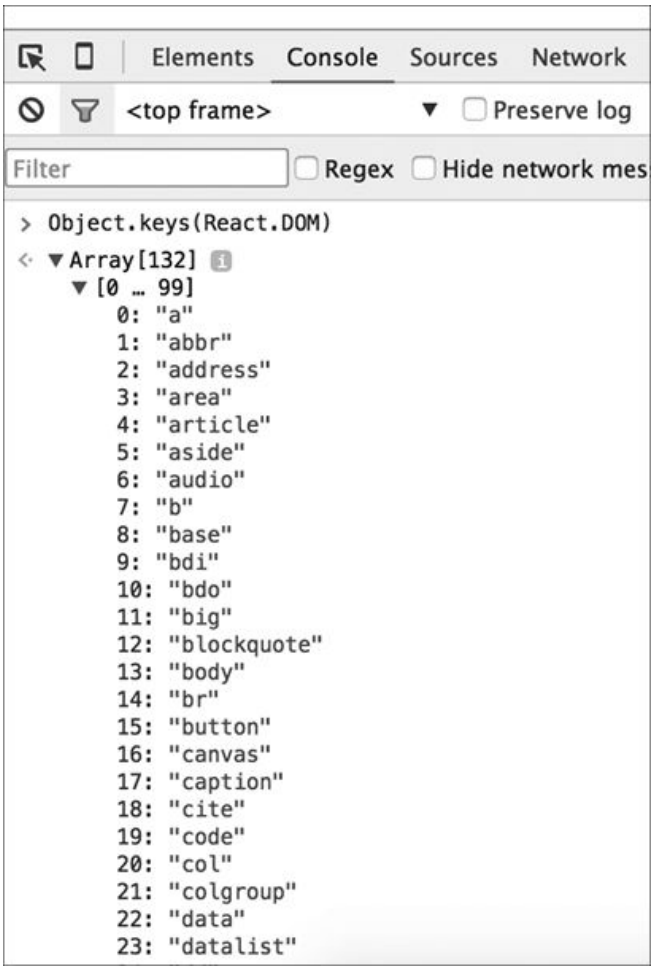


Figura 1.3 – Lista das propriedades de React.DOM.

Vamos dar uma olhada nos parâmetros que todos os métodos React.DOM.* aceitam. Lembre-se de que a aplicação “Hello World!” tinha o seguinte aspecto:

```
ReactDOM.render(
  React.DOM.h1(null, "Hello World!"),
  document.getElementById("app")
);
```

O primeiro parâmetro de h1() (null nesse caso) é um objeto que especifica qualquer propriedade (pense nos atributos de DOM) que você queira passar para o seu componente. Por exemplo, você

pode fazer o seguinte:

```
React.DOM.h1(
  {
    id: "my-heading",
  },
  "Hello World!"
),
```

A Figura 1.4 mostra o HTML gerado por esse exemplo.

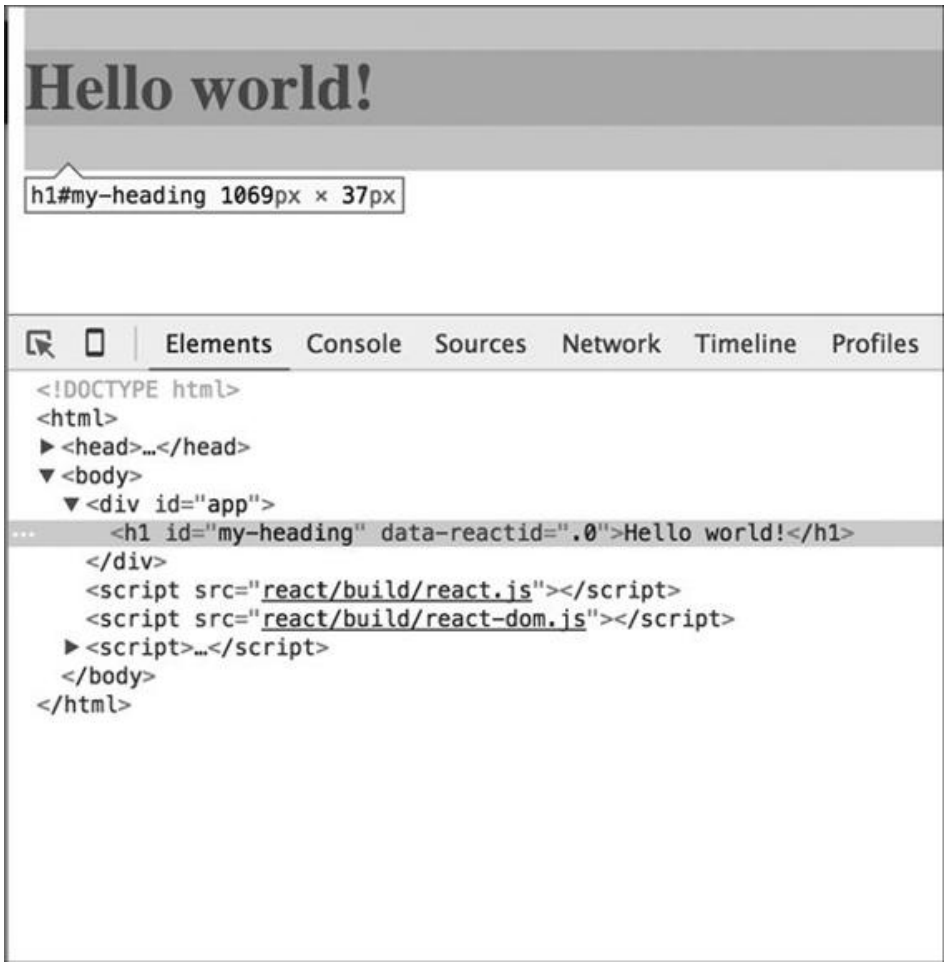


Figura 1.4 – HTML gerado por uma chamada a React.DOM.

O segundo parâmetro ("Hello World!" nesse exemplo) define um filho do componente. O caso mais simples é apenas um filho do tipo texto (um nó Text no dialeto de DOM), como você pode ver no código anterior. Entretanto, você pode ter tantos filhos aninhados quantos quiser e passá-los como parâmetros adicionais da função. Por

exemplo:

```
React.DOM.h1(
  {id: "my-heading"},
  React.DOM.span(null, "Hello"),
  " World!"
),
```

Outro exemplo, desta vez com componentes aninhados (resultado mostrado na Figura 1.5), é:

```
React.DOM.h1(
  {id: "my-heading"},
  React.DOM.span(null,
    React.DOM.em(null, "Hell"),
    "o"
  ),
  " world!"
),
```

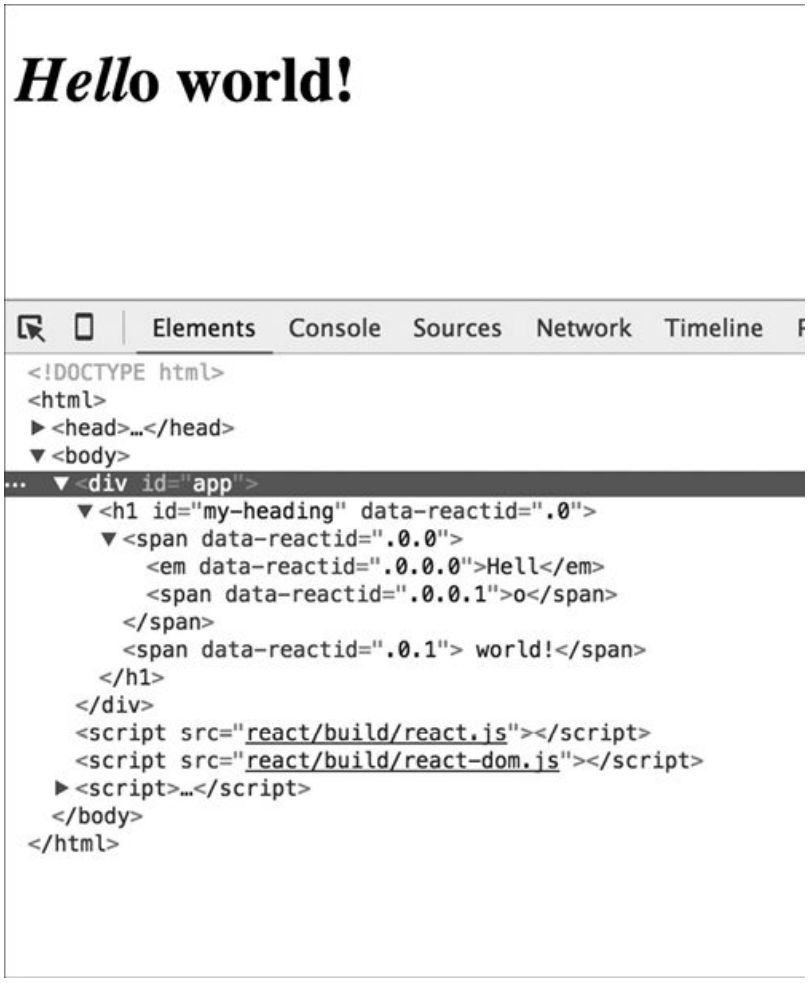


Figura 1.5 – HTML gerado com chamadas aninhadas de React.DOM.

Como podemos ver, se você começar a aninhar componentes, acabará rapidamente precisando controlar muitas chamadas de função e parênteses. Para facilitar, você pode usar a *sintaxe JSX.* JSX é um assunto para uma discussão à parte (Capítulo 4), mas, por enquanto, vamos sofrer usando a sintaxe JavaScript pura. O motivo é que o JSX é um pouco controverso: com frequência, as pessoas o acham repulsivo à primeira vista (ugh, XML em meu JavaScript!), mas o consideram indispensável depois. Somente como aperitivo, eis o trecho de código anterior com a sintaxe JSX:

```
ReactDOM.render(
<h1 id="my-heading">
  <span><em>Hell</em>o</span> world!
</h1>,
document.getElementById("app")
);
```

Atributos especiais do DOM

Alguns atributos especiais do DOM dos quais você deve estar ciente são: class, for e style.

Você não pode usar class e for porque elas são palavras reservadas em JavaScript. Em vez disso, você precisará de className e htmlFor:

```
// Contraexemplo
// isto não funciona
React.DOM.h1(
  {
    class: "pretty",
    for: "me",
  },
  "Hello World!"
);
```

```
// Exemplo apropriado
// isto funciona
React.DOM.h1(
  {
    className: "pretty",
    htmlFor: "me",
  },
  "Hello World!"
);
```

Quando se trata do atributo style, você não pode usar uma string como normalmente faria em HTML, mas precisa de um objeto JavaScript em seu lugar. Evitar strings é sempre uma boa ideia para reduzir os riscos de ataques XSS (Cross-site Scripting), portanto essa é uma mudança bem-vinda.

```
// Contraexemplo
// isto não funciona
React.DOM.h1(
  {
    style: "background: black; color: white; font-family: Verdana",
  },
  "Hello World!"
);
```

```
// Exemplo apropriado
// isto funciona
React.DOM.h1(
  {
    style: {
      background: "black",
      color: "white",
      fontFamily: "Verdana",
    }
  },
  "Hello World!"
);
```

Observe também que você deve usar os nomes da API JavaScript quando estiver lidando com propriedades CSS; em outras palavras, utilize fontFamily em oposição a font-family.

Extensão de navegador React DevTools

Se você abriu o console de seu navegador enquanto testava alguns dos exemplos deste capítulo, talvez tenha visto uma mensagem dizendo: “Download the React DevTools for a better development experience: *https://fb.me/react-devtools*” (Faça download do React DevTools para melhor experiência de desenvolvimento: *https://fb.me/react-devtools*). Ao acessar o URL, você encontrará links para instalar uma extensão de navegador que poderá se mostrar útil na depuração de aplicações React (Figura 1.6).

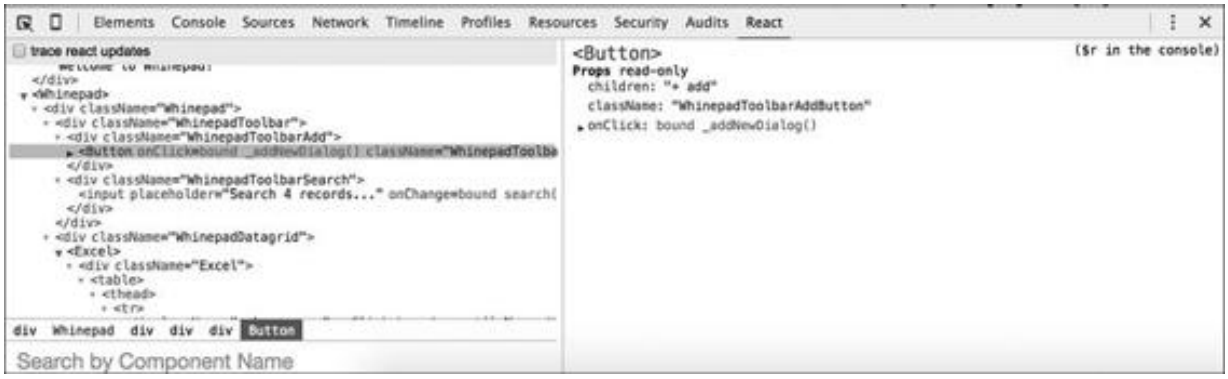


Figura 1.6 – A extensão React DevTools.

Ela pode parecer complicada à primeira vista, mas quando você

chegar ao Capítulo 4, fará todo sentido.

A seguir: componentes personalizados

A essa altura, você já conhece o esqueleto básico da aplicação "Hello World". Agora você já sabe como:

- instalar, configurar e usar a biblioteca React (é apenas uma questão de utilizar duas tags <script>);
- renderizar um componente da React em um local do DOM de sua escolha (por exemplo, React DOM.render(reactWhat, domWhere));
- usar componentes embutidos, que são wrappers em torno de elementos comuns do DOM (por exemplo, React.DOM.div(attributes, children)).

A verdadeira eficácia da React, porém, surge quando você começa a usar componentes personalizados para criar (e atualizar!) a UI de sua aplicação. Vamos aprender a fazer exatamente isso no próximo capítulo.

CAPÍTULO 2

A vida de um componente

Agora que você já sabe como usar os componentes prontos do DOM, é hora de aprender a criar alguns componentes próprios.

O mínimo necessário

A API para criar um novo componente tem o seguinte aspecto:

```
var MyComponent = React.createClass({
  /* especificações */
});
```

"especificações" é um objeto JavaScript que tem um método obrigatório chamado `render()` e alguns métodos e propriedades opcionais. Um exemplo simples pode ter o aspecto a seguir:

```
var Component = React.createClass({
  render: function() {
    return React.DOM.span(null, "I'm so custom");
  }
});
```

Como podemos ver, sua única tarefa *obrigatória* é implementar o método `render()`. Esse método deve devolver um componente React, e é por isso que vemos o `span` nesse trecho de código; você não pode simplesmente devolver um texto.

Usar seu componente em uma aplicação é semelhante a usar os componentes do DOM:

```
ReactDOM.render(
  React.createElement(Component),
  document.getElementById("app")
);
```

O resultado da renderização de seu componente personalizado está na Figura 2.1.

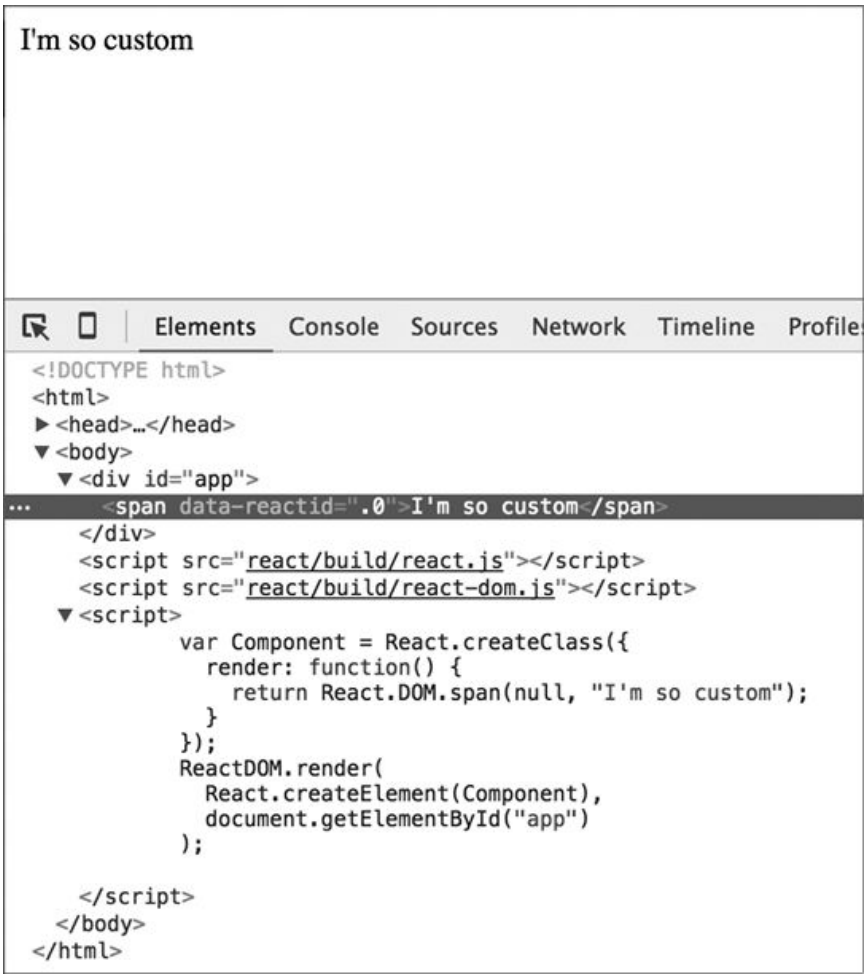


Figura 2.1 – Seu primeiro componente personalizado.

React.createElement() é uma maneira de criar uma "instância" de seu componente. Outra maneira, se você pretende criar várias instâncias, é usar uma factory (fábrica):

```
var ComponentFactory = React.createFactory(Component);

ReactDOM.render(
  ComponentFactory(),
  document.getElementById("app")
);
```

Observe que os métodos React.DOM.* que você já conhece, na verdade, são apenas wrappers convenientes em torno de React.createElement(). Em outras palavras, o código a seguir também

funciona com componentes do DOM:

```
ReactDOM.render(
  React.createElement("span", null, "Hello"),
  document.getElementById("app")
);
```

Como podemos ver, os elementos do DOM são definidos como strings em oposição a funções JavaScript como no caso dos componentes personalizados.

Propriedades

Seus componentes podem aceitar propriedades e renderizar ou se comportar de modo diferente, de acordo com os valores dessas propriedades. Todas as propriedades estão disponíveis por meio do objeto this.props. Vamos ver um exemplo:

```
var Component = React.createClass({
  render: function() {
    return React.DOM.span(null, "My name is " + this.props.name);
  }
});
```

A passagem da propriedade na renderização do componente tem o seguinte aspecto:

```
ReactDOM.render(
  React.createElement(Component, {
    name: "Bob",
  }),
  document.getElementById("app")
);
```

A Figura 2.2 mostra o resultado.

Pense em this.props como somente para leitura. As propriedades são úteis para transportar configurações dos componentes pais para os componentes filhos (e dos filhos para os pais, como veremos mais adiante no livro). Se você se sentir tentado a definir uma propriedade de this.props, use variáveis ou propriedades adicionais do objeto de especificação de seu componente (como em this.thing em oposição a this.props.thing). Com efeito, nos navegadores ECMAScript5, você não será capaz de alterar this.props, pois:

```
> Object.isFrozen(this.props) === true; // true
```

Figura 2.2 – Usando propriedades do componente.

propTypes

Em seus componentes, você pode acrescentar uma propriedade chamada propTypes para declarar a lista de propriedades que seu componente aceita e seus tipos. Eis um exemplo:

```
var Component = React.createClass({
  propTypes: {
    name: React.PropTypes.string.isRequired,
  },
  render: function() {
    return React.DOM.span(null, "My name is " + this.props.name);
  }
});
```

O uso de propTypes é opcional, porém é vantajoso de duas maneiras:

- Você declara previamente as propriedades que seu componente espera. Os usuários de seu componente não precisarão olhar o código-fonte (possivelmente longo) da função `render()` para saber quais propriedades eles podem usar para configurar o componente.
- A React faz a validação dos valores das propriedades em tempo de execução, portanto você pode escrever sua função `render()` sem ficar na defensiva (nem mesmo paranoico) acerca dos dados que seus componentes estão recebendo.

Vamos ver a validação em ação. `name: React.PropTypes.string.isRequired` claramente pede um valor não opcional de string para a propriedade `name`. Se você se esquecer de passar o valor, receberá um aviso no console (Figura 2.3):

```
ReactDOM.render(
  React.createElement(Component, {
    // name: "Bob",
  }),
  document.getElementById("app")
);
```

Figura 2.3 – Aviso ao deixar de fornecer uma propriedade necessária.

Você também receberá um aviso se fornecer um valor de tipo inválido, por exemplo, um inteiro (Figura 2.4):

```
React.createElement(Component, {
  name: 123,
})
```

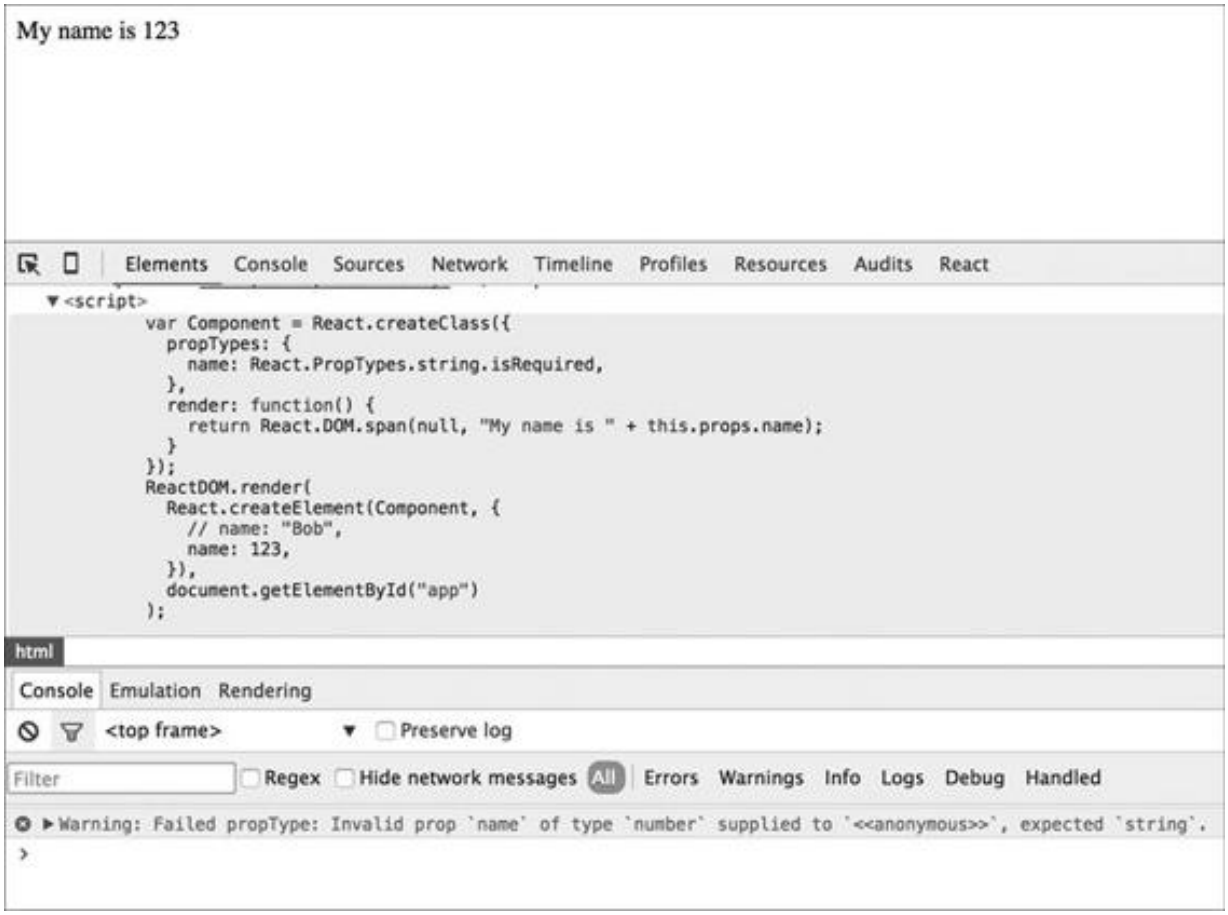


Figura 2.4 – Aviso ao fornecer um tipo inválido.

A Figura 2.5 apresenta uma amostra dos PropTypes disponíveis que você pode usar para declarar suas expectativas.

Declarar propTypes em seus componentes é opcional, o que significa também que você pode ter algumas propriedades, mas não todas, listadas aí. Você pode achar que deixar de declarar todas as propriedades não é uma boa ideia, mas tenha em mente que é possível quando estiver depurando o código de outras pessoas.

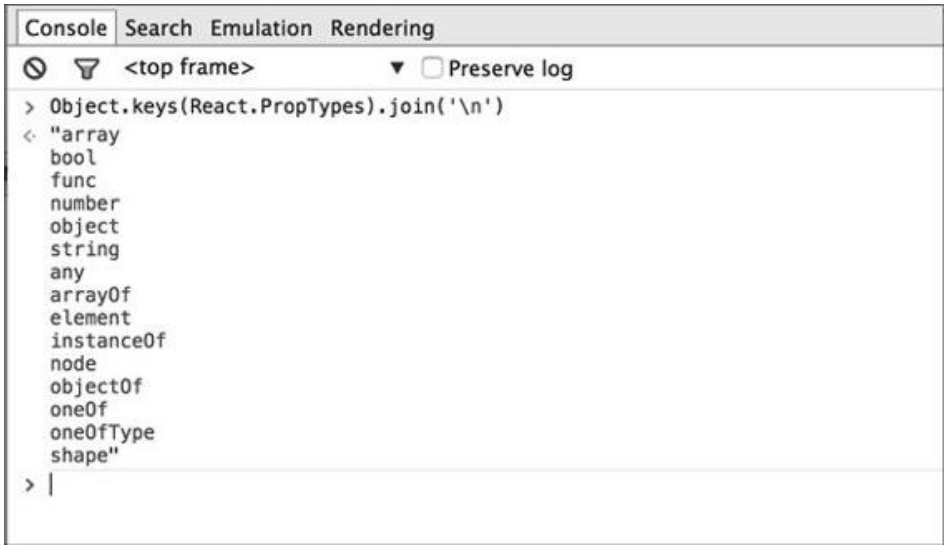


Figura 2.5 – Listando todos os React.PropTypes.

Valores default das propriedades

Quando seu componente aceitar propriedades opcionais, será preciso tomar um cuidado extra para garantir que seu componente continue funcionando quando essas propriedades não forem fornecidas. Isso, inevitavelmente, resulta em código boilerplate defensivo, como em:

```
var text = 'text' in this.props ? this.props.text : '';
```

Você pode evitar a necessidade de escrever esse tipo de código (e se concentrar nas partes mais importantes) implementando o método getDefaultProps():

```
var Component = React.createClass({
  propTypes: {
    firstName: React.PropTypes.string.isRequired,
    middleName: React.PropTypes.string,
    familyName: React.PropTypes.string.isRequired,
    address: React.PropTypes.string,
  },
  getDefaultProps: function() {
    return {
      middleName: '',
      address: 'n/a',
    };
```

```
    },
    render: function() {/* ... */}
});
```

Como você pode ver, `getDefaultProps()` devolve um objeto que fornece valores saudáveis para cada propriedade opcional (aqueles sem `.isRequired`).

Estado

Os exemplos até agora foram bem estáticos (ou “sem estado”, isto é, “stateless”). O objetivo era apenas dar uma ideia dos blocos de construção quando o assunto é compor a sua UI. Porém, o cenário em que a React realmente se destaca (e no qual a manipulação tradicional de DOM dos navegadores e a manutenção realmente se complicam) é aquele em que os dados de sua aplicação mudam. A React tem o conceito de *estado* (state), que são os dados que seu componente utiliza para se renderizar. Quando o estado muda, a React reconstrói a UI sem que você tenha que fazer nada. Assim, depois de ter construído sua UI inicialmente (em seu `render()`), tudo com que você precisa se preocupar é com a atualização de seus dados. Você não precisa se preocupar nem um pouco com as mudanças de UI. Afinal de contas, o seu método `render()` já forneceu o esquema de como deve ser a aparência do componente.

As atualizações de UI após a chamada a setState() são feitas usando um sistema de fila que reúne as alterações em lote de modo eficiente, portanto atualizar this.state diretamente pode resultar em um comportamento inesperado, e você não deve fazer isso. Assim como no caso de this.props, considere o objeto this.state como somente de leitura, não só porque do ponto de vista semântico não é uma boa ideia atualizá-lo diretamente, como também porque ele pode agir de modo inesperado. De modo semelhante, jamais chame this.render() por conta própria – em vez disso, deixe a React reunir as alterações em lote, determinar a quantidade mínima de mudanças e chamar render() quando e se for apropriado.

De modo semelhante à forma como as propriedades são acessíveis por meio de `this.props`, você pode acessar o estado com o objeto `this.state`. Para atualizar o estado, utilize `this.setState()`. Quando `this.setState()` é chamado, a React chama o seu método `render()` e

atualiza a UI.

A React atualiza a UI quando setState() é chamado. Esse é o cenário mais comum, mas há um ponto de escape, como veremos mais adiante. Você pode evitar que a UI seja atualizada devolvendo false em um método especial de “ciclo de vida” chamado shouldComponentUpdate().

Um componente textarea com estado

Vamos criar um novo componente – uma textarea que mantém um contador do número de caracteres digitados (Figura 2.6).

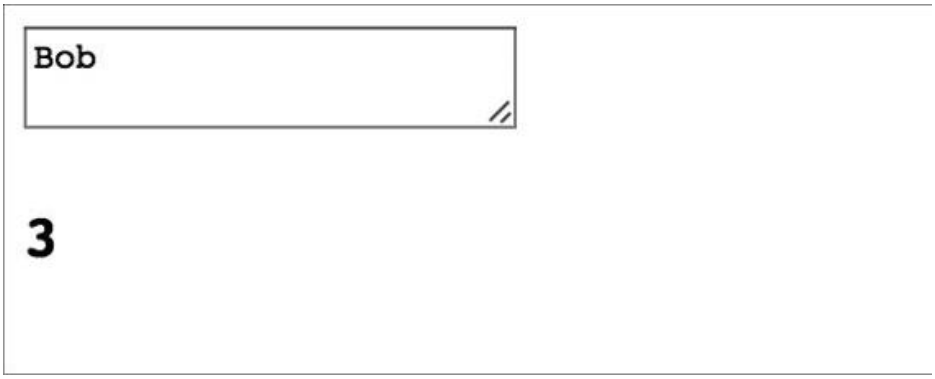


Figura 2.6 – O resultado final do componente textarea personalizado.

Você (assim como outros clientes desse componente reutilizável) pode usar o novo componente da seguinte forma:

```
ReactDOM.render(
  React.createElement(TextAreaCounter, {
    text: "Bob",
  }),
  document.getElementById("app")
);
```

Vamos agora implementar o componente. Comece inicialmente criando uma versão “sem estado” que não trata atualizações, pois isso não é muito diferente de todos os exemplos anteriores:

```
var TextAreaCounter = React.createClass({
  propTypes: {
    text: React.PropTypes.string,
  },

  getDefaultProps: function() {
```

```
    return {
      text: '',
    };
  },
  render: function() {
    return React.DOM.div(null,
      React.DOM.textarea({
        defaultValue: this.props.text,
      }),
      React.DOM.h3(null, this.props.text.length)
    );
  }
});
```

Talvez você tenha notado que a textarea no trecho de código anterior aceita uma propriedade defaultValue em oposição a um filho do tipo texto, conforme você está acostumado em um HTML comum. Isso ocorre porque há pequenas diferenças entre a React e o HTML tradicional quando se trata de formulários. Elas serão discutidas no Capítulo 4, mas fique tranquilo, pois não há muitas diferenças. Além do mais, você verá que essas diferenças fazem sentido e facilitarão sua vida como desenvolvedor.

Como podemos ver, o componente aceita uma propriedade opcional `text` do tipo string e renderiza uma textarea com o valor dado, assim como um elemento `<h3>` que simplesmente exibe o `length` da string (Figura 2.7).

O próximo passo consiste em transformar esse componente *sem estado* (stateless) em um componente *com estado* (stateful). Em outras palavras, vamos fazer o componente guardar alguns dados (estado), usar esses dados para se renderizar inicialmente e, mais tarde, atualizar-se (renderizar novamente) quando os dados mudarem.

Implemente um método chamado `getInitialState()` em seu componente para que você tenha certeza de que sempre trabalhará com dados saudáveis:

```
  getInitialState: function() {
    return {
      text: this.props.text,
    };
```

```
  },
```

O dado mantido por esse componente é simplesmente o texto da textarea, portanto o estado tem apenas uma propriedade chamada `text`, acessível por meio de `this.state.text`. Inicialmente (em `getInitialState()`), você deve apenas copiar a propriedade `text`. Depois, quando os dados mudarem (o usuário está digitando na textarea), o componente atualizará seu estado usando um método auxiliar:

```
  _textChange: function(ev) {
    this.setState({
      text: ev.target.value,
    });
  },
```

Figura 2.7 – O componente TextAreaCounter em ação.

Sempre atualize o estado com this.setState(), que aceita um objeto e o combina com os dados já existentes em this.state. Como você deve ter adivinhado, _textChange() é um listener de eventos que aceita um objeto de evento ev e acessa seus dados internos para obter o texto de entrada da textarea.

A última tarefa é atualizar o método render() de modo a usar this.state no lugar de this.props e configurar o listener de eventos:

```
render: function() {
  return React.DOM.div(null,
    React.DOM.textarea({
```

```
        value: this.state.text,
        onChange: this._textChange,
      }),
      React.DOM.h3(null, this.state.text.length)
    );
  }
```

A partir de agora, sempre que o usuário digitar na textarea, o valor do contador se atualizará para refletir o conteúdo (Figura 2.8).

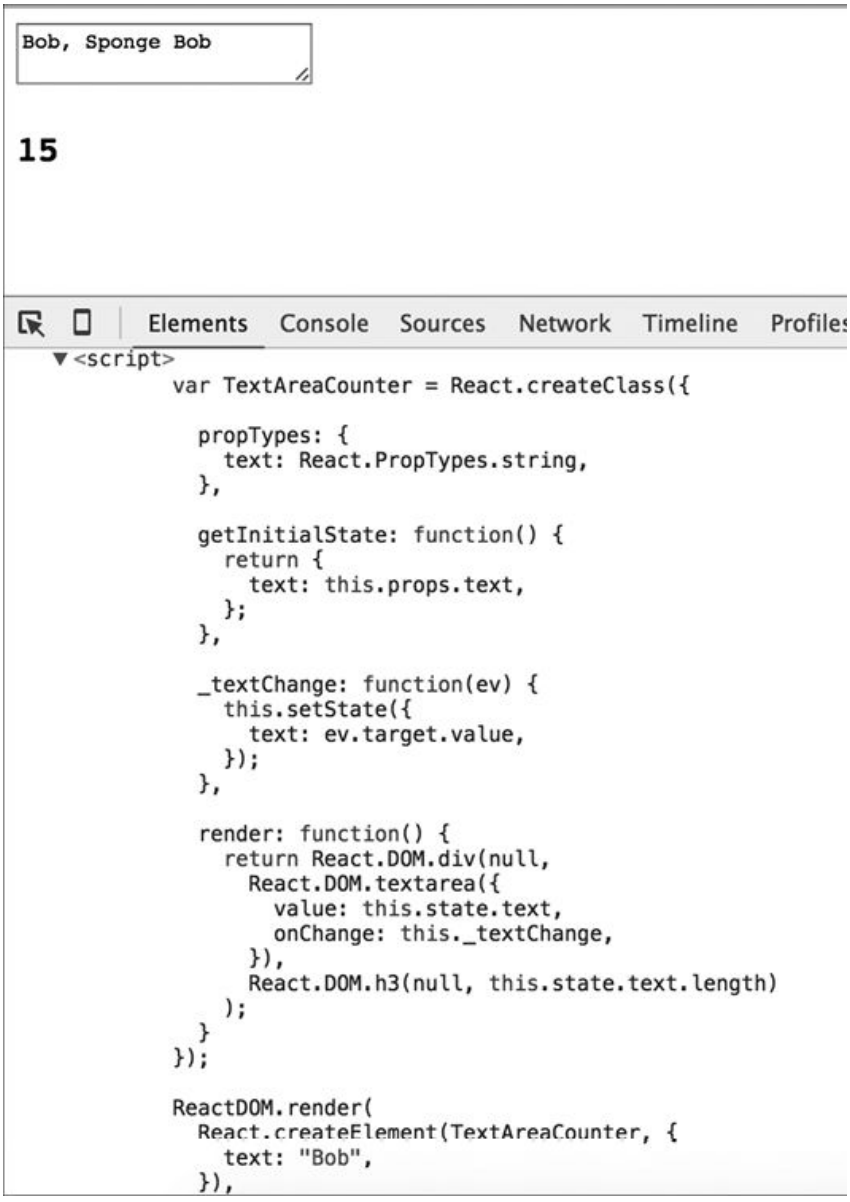


Figura 2.8 – Digitando na textarea.

Uma observação sobre eventos do DOM

Para evitar qualquer confusão, alguns esclarecimentos são necessários em relação a esta linha:

```
onChange: this._textChange
```

A React utiliza seu próprio sistema de eventos *sintéticos* por questões de desempenho, assim como de conveniência e sanidade. Para ajudar a entender o porquê, é preciso considerar de que modo a situação é tratada no mundo DOM puro.

Tratamento de eventos nos velhos tempos

É muito conveniente usar event handlers (manipuladores de eventos) *inline* para tarefas como:

```
<button onclick="doStuff">
```

Embora sejam convenientes e fáceis de ler (o listener de eventos está logo ali, na UI), ter muitos listeners de eventos espalhados dessa forma não é eficiente. Também é difícil ter mais de um listener no mesmo botão, especialmente se esse botão estiver no "componente" ou na biblioteca de outra pessoa e você não quer acessá-lo para "corrigir" ou fazer um fork (bifurcação) de seu código. É por isso que no mundo do DOM as pessoas usam `element.addEventListener` para configurar listeners (o que, agora, resulta em ter código em dois ou mais lugares) e *delegação de eventos* (event delegation, para tratar problemas de desempenho). A delegação de eventos implica que você ouve os eventos em algum nó-pai, por exemplo, em uma `<div>` que contém muitos botões, e configura um listener para todos os botões.

Com a delegação de eventos, você fará algo como:

```
<div id="parent">
  <button id="ok">OK</button>
  <button id="cancel">Cancel</button>
</div>

<script>
```

```
document.getElementById('parent').addEventListener('click', function(event) {
  var button = event.target;
  // Executa tarefas diferentes de acordo com o botão que foi clicado
  switch (button.id) {
    case 'ok':
      console.log('OK!');
      break;
    case 'cancel':
      console.log('Cancel');
      break;
    default:
      new Error('Unexpected button ID');
  };
});
</script>
```

Esse código funciona e tem um bom desempenho, mas há algumas desvantagens:

- a declaração do listener está longe do componente de UI, o que deixa o código mais difícil de ser achado e depurado;
- usar delegação e sempre um switching cria código boilerplate desnecessário, mesmo antes de chegar a fazer o trabalho propriamente dito (responder a um clique de botão, nesse caso);
- inconsistências entre navegadores (omitidas aqui), na verdade, exigem que esse código seja mais longo.

Infelizmente, quando se trata de ativar esse código diante de usuários reais, você precisará de mais alguns acréscimos para dar suporte a todos os navegadores:

- você precisa de `attachEvent` além de `addEventListener`;
- você precisa de `var event = event || window.event;` no início do listener;
- você precisa de `var button = event.target || event.srcElement;`.

Tudo isso é necessário e irritante o suficiente a ponto de você acabar usando algum tipo de biblioteca de eventos. No entanto, por que acrescentar outra biblioteca (e estudar mais APIs), quando a React inclui uma solução para os pesadelos de tratamento de

eventos?

Tratamento de eventos na React

A React utiliza *eventos sintéticos (*synthetic events) para encapsular e normalizar os eventos dos navegadores, o que significa que não haverá mais inconsistências entre os navegadores. Você sempre pode contar com o fato de event.target estar disponível a você em todos os navegadores. É por isso que, no trecho de código de TextAreaCounter, você só precisa de ev.target.value, e ele simplesmente funciona. Também significa que a API para cancelar eventos é a mesma em todos os navegadores; em outras palavras, event.stopPropagation() e event.preventDefault() funcionam até mesmo nos IEs antigos.

A sintaxe facilita manter juntos a UI e os listeners de eventos. É semelhante aos event handlers inline tradicionais, mas, internamente, não é. Na verdade, a React utiliza delegação de eventos por questões de desempenho.

A React usa a sintaxe camelCase para os event handlers, portanto você deve usar onClick em vez de onclick.

Se você precisar do evento original do navegador por algum motivo, ele estará disponível como event.nativeEvent, mas é improvável que você vá precisar acessá-lo.

Mais uma observação: o evento onChange (conforme usado no exemplo com textarea) comporta-se conforme esperado: ele é disparado quando um usuário digita, e não depois de ter terminado a digitação e navegado para fora do campo, que é o comportamento no DOM simples.

Propriedades *versus* estado

Agora você sabe que tem acesso a this.props e a this.state quando se trata de exibir o seu componente em seu método render(). Talvez você esteja se perguntando quando deve usar um e quando deve usar o

outro.

As propriedades são um mecanismo para o mundo externo (usuários do componente) configurar o seu componente. O estado é a manutenção de seus dados internos. Portanto, se você considerar uma analogia com a programação orientada a objetos, this.props é semelhante a todos os *argumentos passados para o construtor de uma classe*, enquanto this.state é o conjunto de suas *propriedades privadas.*

Propriedades no estado inicial: um antipadrão

Anteriormente, vimos um exemplo de uso de this.props em getInitialState():

```
getInitialState: function() {
  return {
    text: this.props.text,
  };
},
```

Isso, na verdade, é considerado um antipadrão. O ideal é que você use qualquer combinação de this.state e this.props que considerar adequada para construir sua UI em seu método render().

Porém, às vezes, você vai querer tomar um valor passado para o seu componente e usá-lo para construir o estado inicial. Não há nada de errado com isso, exceto que quem chama seu componente pode esperar que a propriedade (text, no exemplo anterior) sempre tenha o valor mais recente, e o exemplo violou essa expectativa. Para definir devidamente as expectativas, uma mudança simples de nome é suficiente – por exemplo, chamar a propriedade com um nome como defaultText ou initialValue em vez de apenas text:

```
propTypes: {
  defaultValue: React.PropTypes.string
},
getInitialState: function() {
```

```
    return {
      text: this.props.defaultValue,
    };
  },
```

O Capítulo 4 mostra como a React resolve isso em sua própria implementação das entradas e textareas, para as quais as pessoas podem ter expectativas geradas pelo seu conhecimento anterior de HTML.

Acessando o componente de fora

Nem sempre você terá o privilégio de iniciar uma aplicação React totalmente nova do zero. Às vezes, será necessário se conectar a uma aplicação ou a um site já existentes e migrar para a React por partes. Felizmente, a React foi projetada para funcionar com qualquer base de código preexistente que você possa ter. Afinal de contas, os criadores originais da React não podiam parar o mundo e reescrever uma aplicação enorme inteira (Facebook) totalmente do zero.

Uma maneira de fazer sua aplicação React se comunicar com o mundo externo é obter uma referência para um componente renderizado com `ReactDOM.render()` e usá-la de fora do componente:

```
var myTextAreaCounter = ReactDOM.render(
  React.createElement(TextAreaCounter, {
    defaultValue: "Bob",
  }),
  document.getElementById("app")
);
```

Agora você pode usar `myTextAreaCounter` para acessar os mesmos métodos e propriedades que normalmente acessaria com `this` se estivesse dentro do componente. Você pode até mesmo interagir com o componente usando o seu console JavaScript (Figura 2.9).

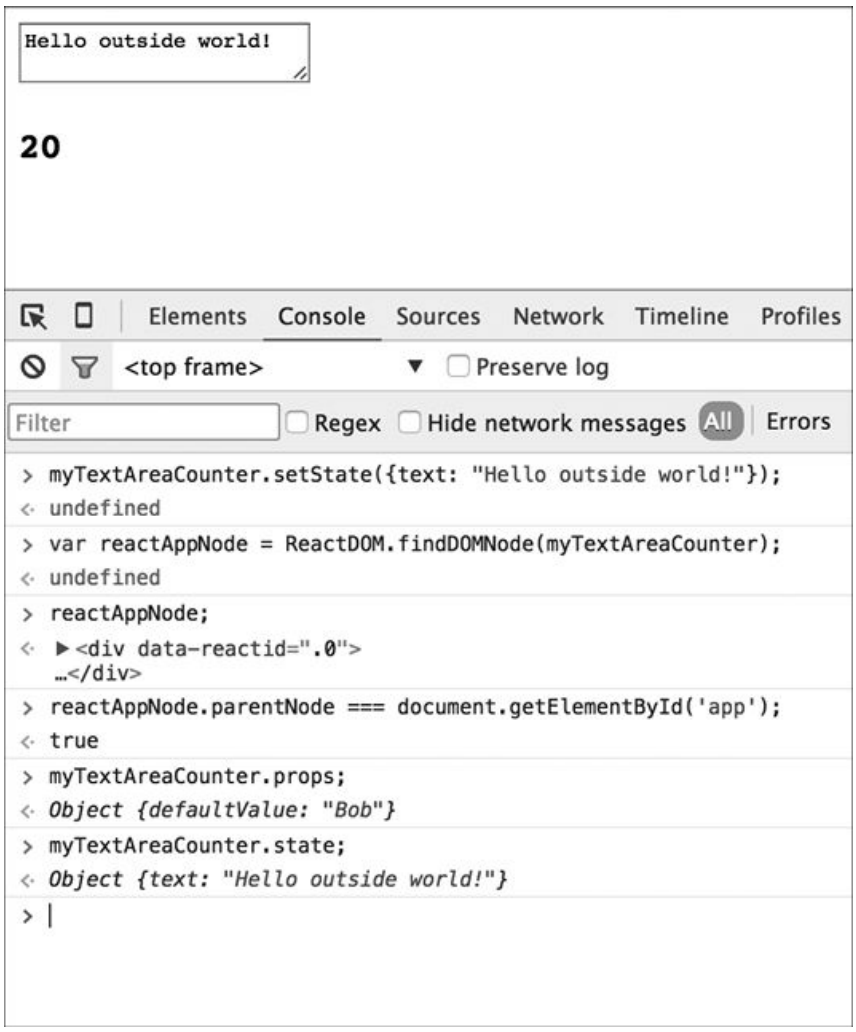


Figura 2.9 – Acessando o componente renderizado mantendo uma referência.

Esta linha define um estado novo:

```
myTextAreaCounter.setState({text: "Hello outside world!"});
```

A linha a seguir obtém uma referência para o nó-pai principal do DOM criado pela React:

```
var reactAppNode = ReactDOM.findDOMNode(myTextAreaCounter);
```

Este é o primeiro filho de , que é o local em que você disse à React para fazer a sua mágica:

```
reactAppNode.parentNode === document.getElementById('app'); // true
```

Eis o modo de acessar as propriedades e o estado:

```
myTextAreaCounter.props; // Object { defaultValue: "Bob"}
```

```
myTextAreaCounter.state; // Object { text: "Hello outside world!"}
```

Você tem acesso a toda a API do componente de fora dele. Porém, você deve usar seus novos superpoderes com parcimônia, se é que vai usá-los. Talvez você possa usar ReactDOM.findDOMNode() se precisar obter as dimensões do nó para garantir que ele caiba em sua página de modo geral, mas não deve fazer muito mais que isso. Pode ser tentador manipular o estado dos componentes que não sejam seus e “corrigi-los”, mas você estaria violando as expectativas e gerando bugs no futuro, pois o componente não prevê esse tipo de intromissão. Por exemplo, o código a seguir funciona, mas não é recomendado:

```
// Contraexemplo
myTextAreaCounter.setState({text: 'NOOOO'});
```

Alterando as propriedades durante a execução

Como você já sabe, as propriedades são uma maneira de configurar um componente. Portanto, alterar externamente as propriedades do componente depois que ele foi criado pode ser razoável. Entretanto, seu componente deve estar preparado para lidar com esse cenário.

Se observar o método `render()` dos exemplos anteriores, você verá que ele utiliza apenas `this.state`:

```
render: function() {
  return React.DOM.div(null,
    React.DOM.textarea({
      value: this.state.text,
      onChange: this._textChange,
    }),
    React.DOM.h3(null, this.state.text.length)
  );
}
```

Se você alterar as propriedades de fora do componente, isso não terá nenhum efeito na renderização. Em outras palavras, o conteúdo da textarea não mudará depois que você executar:

```
myTextAreaCounter = ReactDOM.render(
  React.createElement(TextAreaCounter, {
    defaultValue: "Hello", // anteriormente conhecido como "Bob"
  }),
  document.getElementById("app")
```

```
);
```

Mesmo que myTextAreaCounter seja reescrito por uma nova chamada a ReactDOM.render(), o estado da aplicação será mantido. A React faz a *reconciliação* da aplicação antes/depois e não limpa tudo. Em vez disso, ela aplica a quantidade mínima de alterações.

O conteúdo de this.props agora está alterado (mas a UI não):

```
myTextAreaCounter.props; // Object { defaultValue="Hello"}
```

Configurar o estado *faz* a UI ser atualizada:

```
// Contraexemplo
myTextAreaCounter.setState({text: 'Hello'});
```

Porém, essa é uma péssima ideia, pois pode resultar em um estado inconsistente em componentes mais complicados; por exemplo, pode bagunçar contadores internos, flags booleanas, listeners de eventos, e assim por diante.

Se quiser tratar intromissões externas (mudança de propriedades) de forma elegante, você poderá se preparar implementando um método chamado componentWillReceiveProps():

```
componentWillReceiveProps: function(newProps) {
  this.setState({
    text: newProps.defaultValue,
  });
},
```

Como podemos ver, esse método recebe o novo objeto de propriedades e você pode definir state de acordo com ele, assim como executar qualquer tarefa necessária para manter o componente em um estado saudável.

Métodos de ciclo de vida

O método componentWillReceiveProps() do trecho de código anterior é um dos métodos conhecidos como métodos de *ciclo de vida* (lifecycle methods) oferecidos pela React. Você pode usar os métodos de ciclo de vida para ouvir mudanças em seu componente. Outros métodos de ciclo de vida que você pode implementar incluem:

componentWillUpdate()

Executado antes do método `render()` de seu componente ser chamado novamente (como resultado de mudanças nas propriedades ou no estado).

componentDidUpdate()

Executado após o método `render()` ter sido concluído e as novas alterações no DOM subjacente terem sido aplicadas.

componentWillMount()

Executado antes de o nó ser inserido no DOM.

componentDidMount()

Executado depois que o nó é inserido no DOM.

componentWillUnmount()

Executado imediatamente antes de o componente ser removido do DOM.

shouldComponentUpdate(newProps, newState)

Esse método é chamado antes de `componentWillUpdate()` e dá a você a oportunidade de executar um `return false;` e cancelar a atualização, o que significa que o seu método `render()` não será chamado. Isso é conveniente em áreas de desempenho crítico da aplicação, quando você achar que nada interessante mudou e uma nova renderização não é necessária. Você deve tomar essa decisão com base na comparação entre o argumento `newState` e o `this.state` existente, e pela comparação entre `newProps` e `this.props`, ou simplesmente por saber que esse componente é estático e não muda. (Você verá um exemplo em breve.)

Exemplo de ciclo de vida: fazendo log de tudo

Para entender melhor a vida de um componente, vamos acrescentar um pouco de logging no componente `TextAreaCounter`. Basta fazer a implementação de modo que todos os métodos de ciclo de vida

façam log no console quando forem chamados, juntamente com qualquer argumento:

```
var TextAreaCounter = React.createClass({
  _log: function(methodName, args) {
    console.log(methodName, args);
  },
  componentWillUpdate: function() {
    this._log('componentWillUpdate', arguments);
  },
  componentDidUpdate: function() {
    this._log('componentDidUpdate', arguments);
  },
  componentWillMount: function() {
    this._log('componentWillMount', arguments);
  },
  componentDidMount: function() {
    this._log('componentDidMount', arguments);
  },
  componentWillUnmount: function() {
    this._log('componentWillUnmount', arguments);
  },
  // ...
  // outras implementações, render(), etc.
};
```

A Figura 2.10 mostra o que acontece depois que você carrega a página.

Como podemos ver, dois métodos foram chamados sem argumentos. `componentDidMount()` geralmente é o mais interessante dos dois. Você pode acessar o nó recém-montado no DOM com `ReactDOM.findDOMNode(this)` se for necessário, por exemplo, para obter as dimensões do componente. Você também pode executar qualquer tarefa de inicialização, agora que seu componente está ativo.

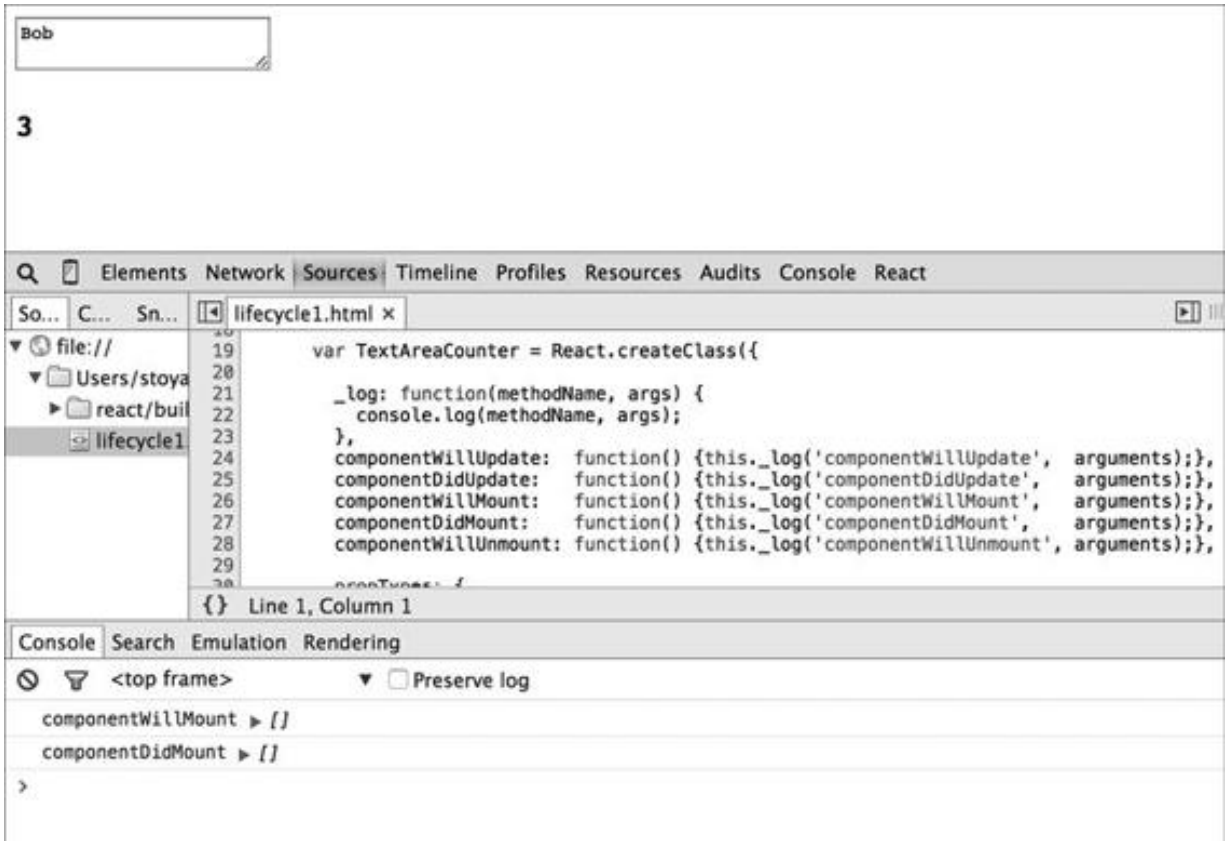


Figura 2.10 – Montando o componente.

A seguir, o que acontecerá se você digitar “s” para transformar o texto em “Bobs”? (Veja a Figura 2.11)

O método componentWillUpdate(nextProps, nextState) é chamado com os novos dados que serão usados para renderizar mais uma vez o componente. O primeiro argumento é o futuro valor de this.props (que não muda nesse exemplo), e o segundo é o futuro valor do novo this.state. O terceiro argumento é context, que não é tão interessante assim neste momento. Você pode comparar os argumentos (por exemplo, newProps) com o this.props atual e decidir se deve agir sobre ele.

Depois de componentWillUpdate(), você pode ver que componentDidUpdate(oldProps, oldState) é chamado, passando os valores de props e state anteriores à mudança. Essa é uma oportunidade para fazer algo após a mudança. Você pode usar this.setState() aqui, o que não pode ser feito em componentWillUpdate().

Suponha que você queira restringir o número de caracteres a ser digitado na textarea. Você deve fazer isso no event handler _textChange(), que é chamado enquanto o usuário digita. Mas, e se alguém (você mais jovem, mais ingênuo) chamar setState() de fora do componente? (O que, como mencionamos antes, é uma péssima ideia.)

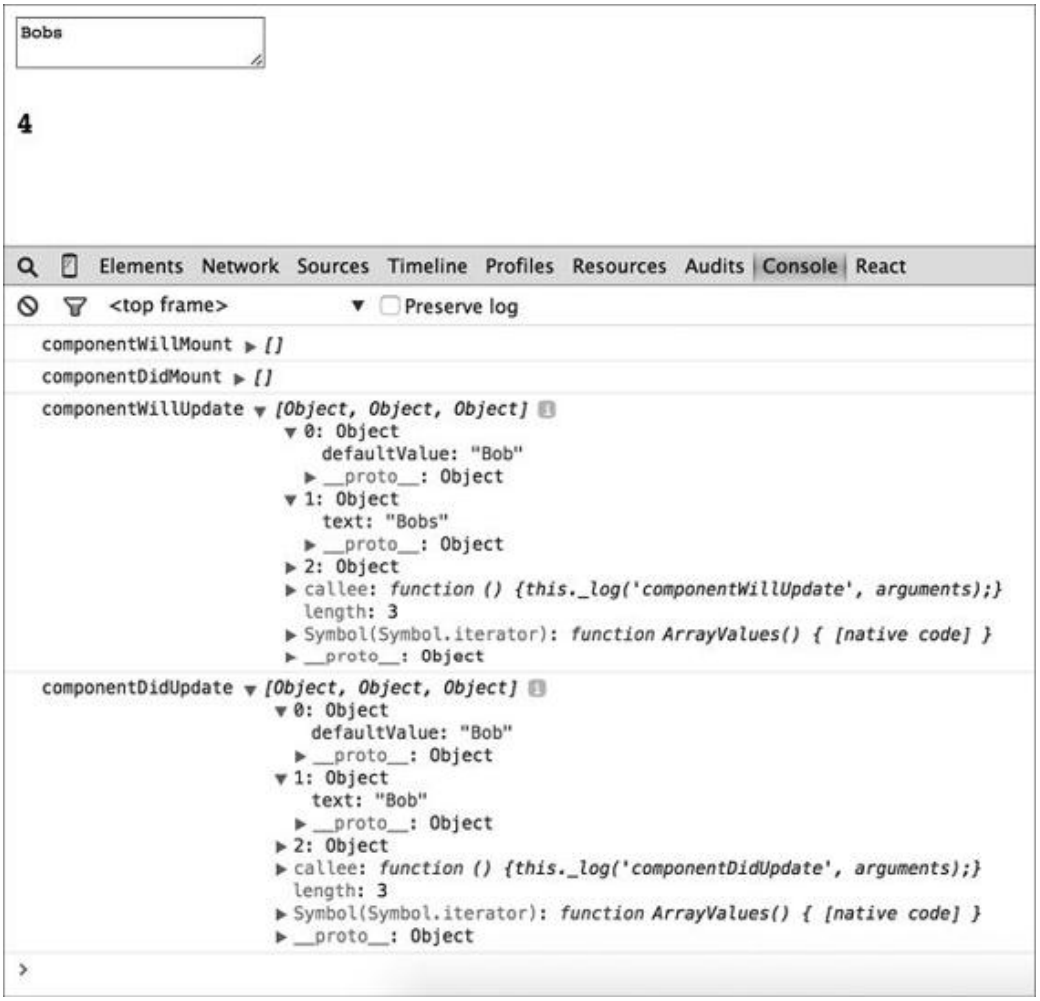


Figura 2.11 – Atualizando o componente.

Você ainda poderá proteger a consistência e o bem-estar de seu componente? Certamente. Você pode fazer a validação em componentDidUpdate() e, se o número de caracteres for maior que o permitido, faça a reversão do estado para o valor anterior. Será algo como:

```
componentDidUpdate: function(oldProps, oldState) {
```

```
    if (this.state.text.length > 3) {
      this.replaceState(oldState);
    }
  },
```

Isso pode parecer excessivamente paranoico, mas é possível de ser feito.

Observe o uso de replaceState() no lugar de setState(). Enquanto setState(obj) mescla as propriedades de obj com aquelas em this.state, replaceState() sobrescreve tudo.

Exemplo de ciclo de vida: usando uma mixin

No exemplo anterior, vimos quatro das cinco chamadas de método de ciclo de vida sendo registradas em log. A quinta chamada, componentWillUnmount(), é mais bem demonstrada quando temos componentes filhos removidos por um pai. Nesse exemplo, você quer fazer o log de todas as mudanças, tanto no filho quanto no pai. Por isso, vamos introduzir um novo conceito para reutilização de código: uma mixin.

Uma mixin é um objeto JavaScript que contém uma coleção de métodos e propriedades. Uma mixin não foi feita para ser usada sozinha, mas deve ser incluída (misturada) nas propriedades de outro objeto. No exemplo de logging, uma mixin pode ter a aparência a seguir:

```
var logMixin = {
  _log: function(methodName, args) {
    console.log(this.name + '::' + methodName, args);
  },
  componentWillUpdate: function() {
    this._log('componentWillUpdate', arguments);
  },
  componentDidUpdate: function() {
    this._log('componentDidUpdate', arguments);
  },
  componentWillMount: function() {
    this._log('componentWillMount', arguments);
  },
```

```
    componentDidMount: function() {
      this._log('componentDidMount', arguments);
    },
    componentWillUnmount: function() {
      this._log('componentWillUnmount', arguments);
    },
  };
```

Em um mundo sem React, você pode usar for-in para executar um laço e copiar todas as propriedades para um novo objeto e, desse modo, fazer o novo objeto adquirir todas as funcionalidades da mixin. No mundo da React, você tem um atalho: a propriedade mixins. Ela tem o seguinte aspecto:

```
var MyComponent = React.createClass({
  mixins: [obj1, obj2, obj3],
  // O restante dos métodos ...
};
```

Você deve atribuir um array de objetos JavaScript à propriedade mixins e a React cuidará do resto. A inclusão de logMixin em seu componente é feita assim:

```
var TextAreaCounter = React.createClass({
  name: 'TextAreaCounter',
  mixins: [logMixin],
  // Todo o restante..
};
```

Como podemos ver, esse trecho de código também adiciona uma propriedade conveniente name para identificar quem faz a chamada.

Se executar o exemplo com a mixin, você poderá ver o logging em ação (Figura 2.12).

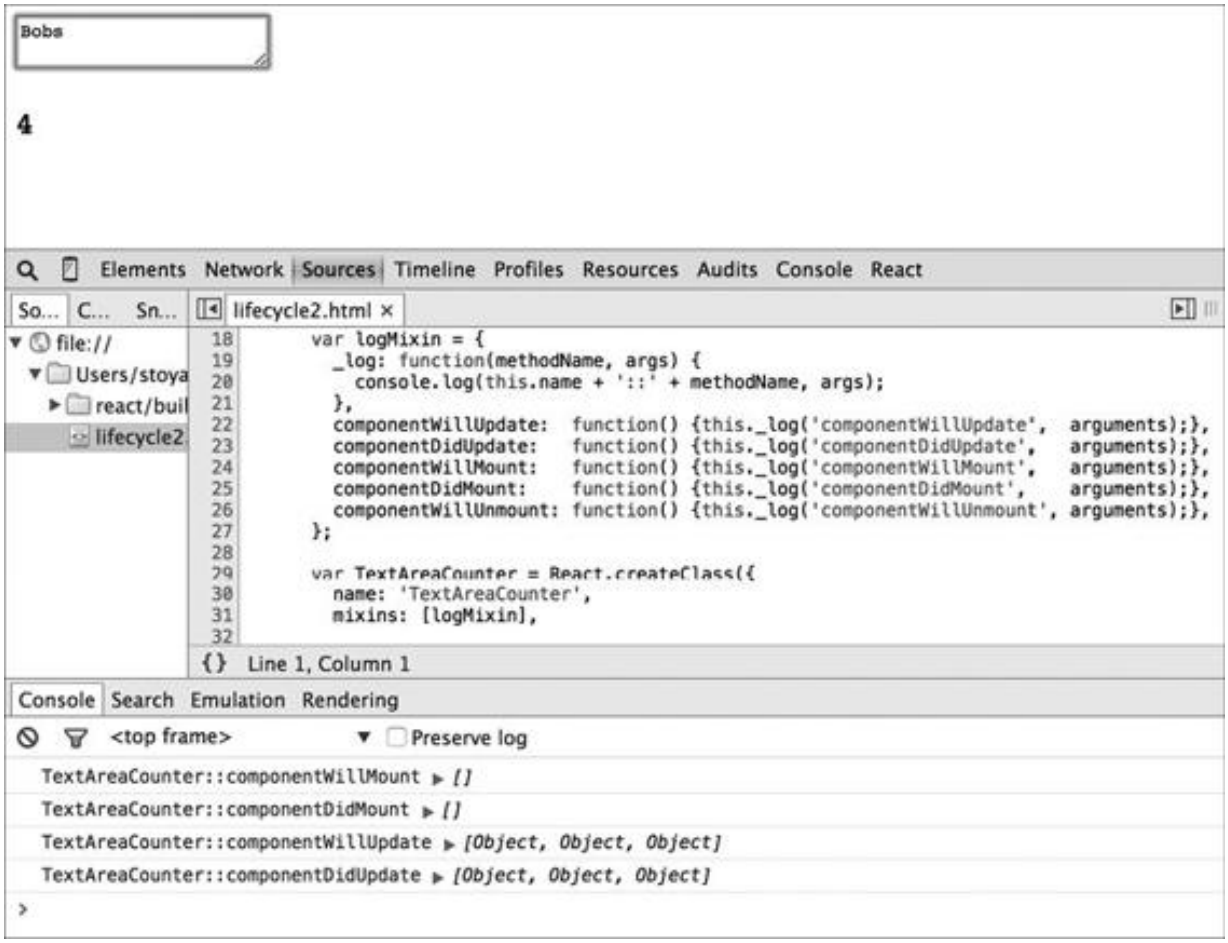


Figura 2.12 – Usando uma mixin e identificando o componente.

Exemplo de ciclo de vida: usando um componente-filho

Você sabe que pode misturar e aninhar componentes React conforme achar apropriado. Até agora, vimos apenas componentes de React.DOM (em oposição a componentes personalizados) nos métodos render(). Vamos dar uma olhada em um componente personalizado simples a ser usado como filho.

Você pode isolar a parte do contador em um componente próprio:

```
var Counter = React.createClass({
  name: 'Counter',
  mixins: [logMixin],
  propTypes: {
```

```
    count: React.PropTypes.number.isRequired,
  },
  render: function() {
    return React.DOM.span(null, this.props.count);
  }
});
```

Esse componente contém apenas a parte do contador – ele renderiza um `<span>` e não mantém um estado, mas apenas exibe a propriedade `count` fornecida pelo pai. Ele também combina `logMixin` para fazer log quando os métodos de ciclo de vida são chamados.

Vamos agora atualizar o método `render()` do componente-pai `TextAreaCounter`. Ele deve usar o componente `Counter` de forma condicional; se o contador for 0, não mostra um número:

```
render: function() {
  var counter = null;
  if (this.state.text.length > 0) {
    counter = React.DOM.h3(null,
      React.createElement(Counter, {
        count: this.state.text.length,
      })
    );
  }
  return React.DOM.div(null,
    React.DOM.textarea({
      value: this.state.text,
      onChange: this._textChange,
    }),
    counter
  );
}
```

A variável `counter` será `null` quando a textarea estiver vazia. Se houver algum texto, a variável `counter` conterá a parte da UI responsável por mostrar o número de caracteres. Não é necessário que toda a UI esteja inline na forma de argumentos do componente `React.DOM.div` principal. Você pode atribuir pequenas partes da UI a variáveis e usá-las de modo condicional.

Agora você pode observar os métodos de ciclo de vida sendo

registrados em log para os dois componentes. A Figura 2.13 mostra o que acontece quando você carrega a página e então altera o conteúdo da textarea.

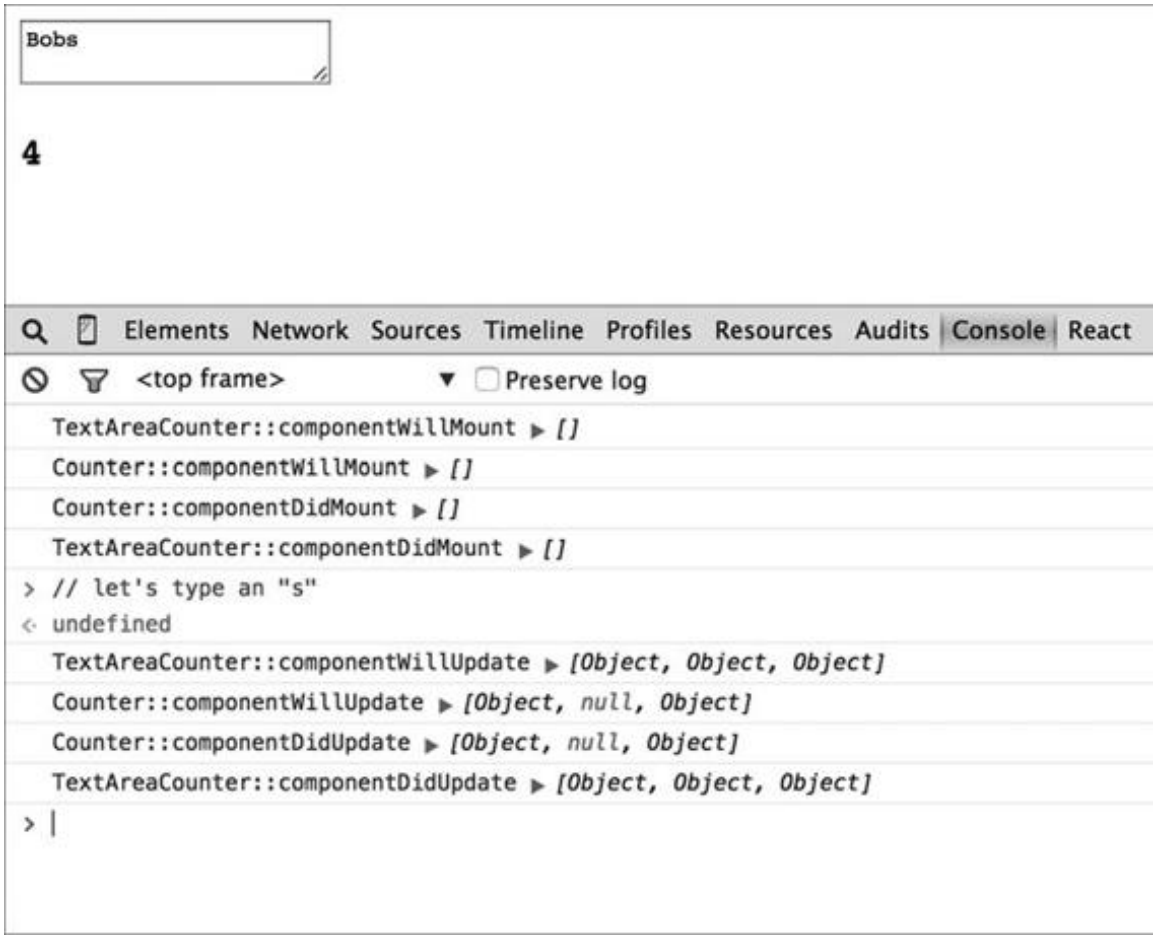


Figura 2.13 – Montando e atualizando dois componentes.

Você pode ver como o componente-filho é montado e atualizado antes do pai.

A Figura 2.14 mostra o que acontece depois que o texto é apagado da textarea e o contador se torna 0. Nesse caso, o filho Counter torna-se null e seu nó no DOM é removido da árvore, após você ser notificado por meio da callback componentWillUnmount.

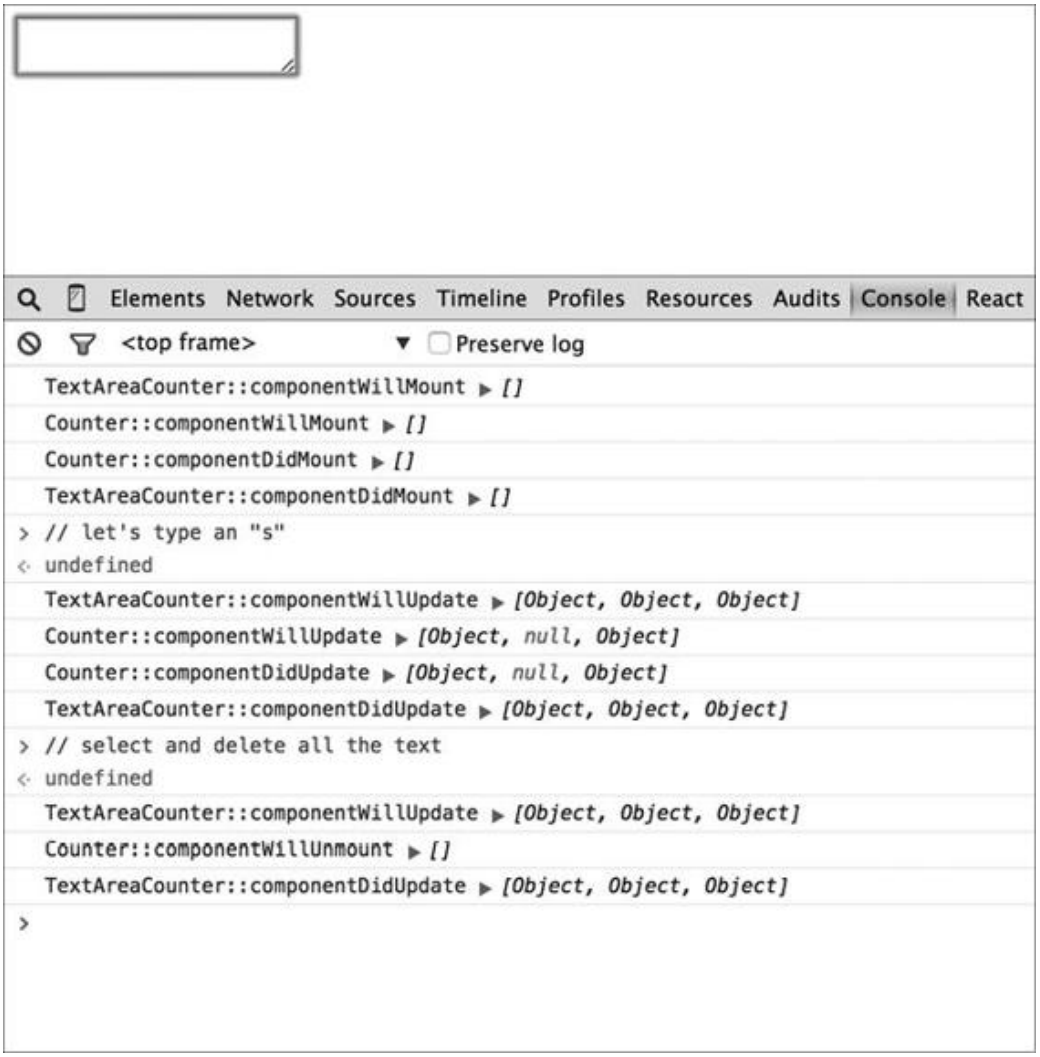


Figura 2.14 – Desmontando o componente do contador.

Ganho de desempenho: evitando atualizações de componentes

O último método de ciclo de vida que você deve conhecer, especialmente quando estiver construindo partes críticas de sua aplicação no que diz respeito ao desempenho, é o método shouldComponentUpdate(nextProps, nextState). Ele é chamado antes de componentWillUpdate() e oferece uma oportunidade para cancelar a atualização se você decidir que ela não é necessária.

Há uma classe de componentes que utiliza apenas this.props e this.state em seus métodos render(), sem nenhuma chamada de função

adicional. Esses componentes são chamados de componentes “puros”. Eles podem implementar shouldComponentUpdate() e comparar o estado e as propriedades antes e depois e, se não houver nenhuma mudança, devolvem false para economizar um pouco de capacidade de processamento. Além disso, existem componentes estáticos puros que não usam props nem state. Eles podem devolver false diretamente.

Vamos explorar o que acontece com as chamadas aos métodos render() e implementar shouldComponentUpdate() para obter ganhos no que concerne ao desempenho.

Inicialmente, tome o novo componente Counter. Remova a mixin de logging e, em seu lugar, faça log no console sempre que o método render() for chamado:

```
var Counter = React.createClass({
  name: 'Counter',
  // mixins: [logMixin],
  propTypes: {
    count: React.PropTypes.number.isRequired,
  },
  render() {
    console.log(this.name + '::render()');
    return React.DOM.span(null, this.props.count);
  }
});
```

Faça o mesmo em TextAreaCounter:

```
var TextAreaCounter = React.createClass({
  name: 'TextAreaCounter',
  // mixins: [logMixin],

  // Todos os demais métodos...

  render: function() {
    console.log(this.name + '::render()');
    // ... e o restante da renderização
  }
});
```

Agora, quando você carregar a página e *colar* a string “LOL”

substituindo “Bob”, você poderá ver o resultado mostrado na Figura 2.15.

Você verá que atualizar o texto resulta na chamada do método render() de TextAreaCounter que, por sua vez, faz o método render() de Counter ser chamado. Ao substituir “Bob” por “LOL”, o número de caracteres antes e depois da atualização é o mesmo, portanto não há mudanças na UI do contador, e não é necessário chamar o render() de Counter. Você pode ajudar a React a otimizar esse caso implementando shouldComponentUpdate() e devolvendo false quando não houver necessidade de novas renderizações. O método recebe os futuros valores de props e de state (state não é necessário nesse componente) e dentro do método, você deve comparar o valor atual com o próximo valor:

```
shouldComponentUpdate(nextProps, nextState_ignore) {
  return nextProps.count !== this.props.count;
},
```

Fazer a mesma atualização de “Bob” para “LOL” não faz Counter ser renderizado novamente (Figura 2.16).

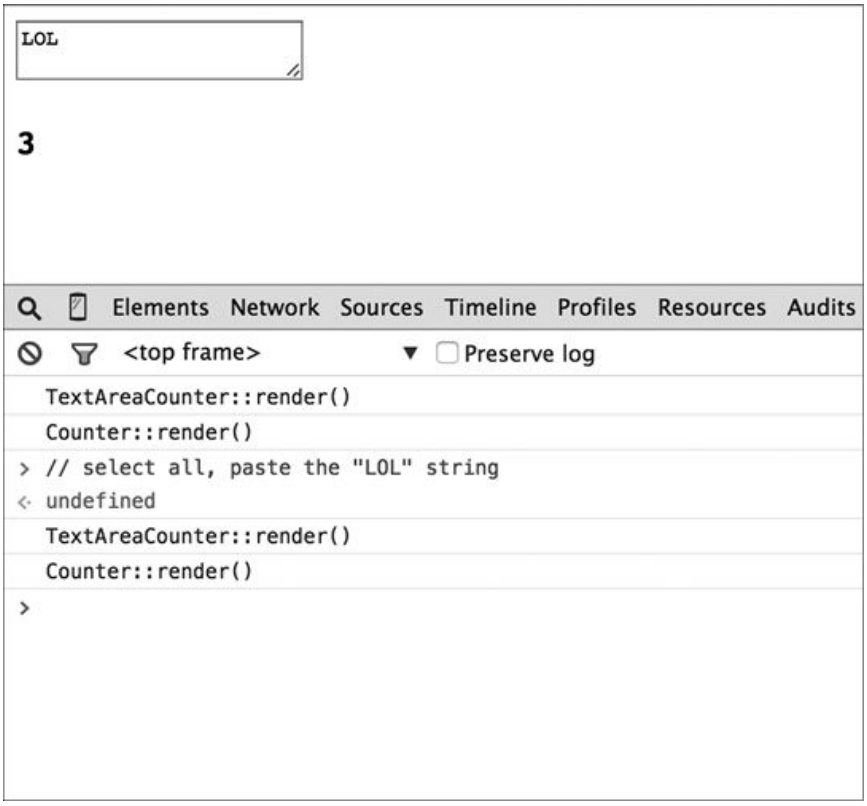


Figura 2.15 – Renderizando os dois componentes novamente.

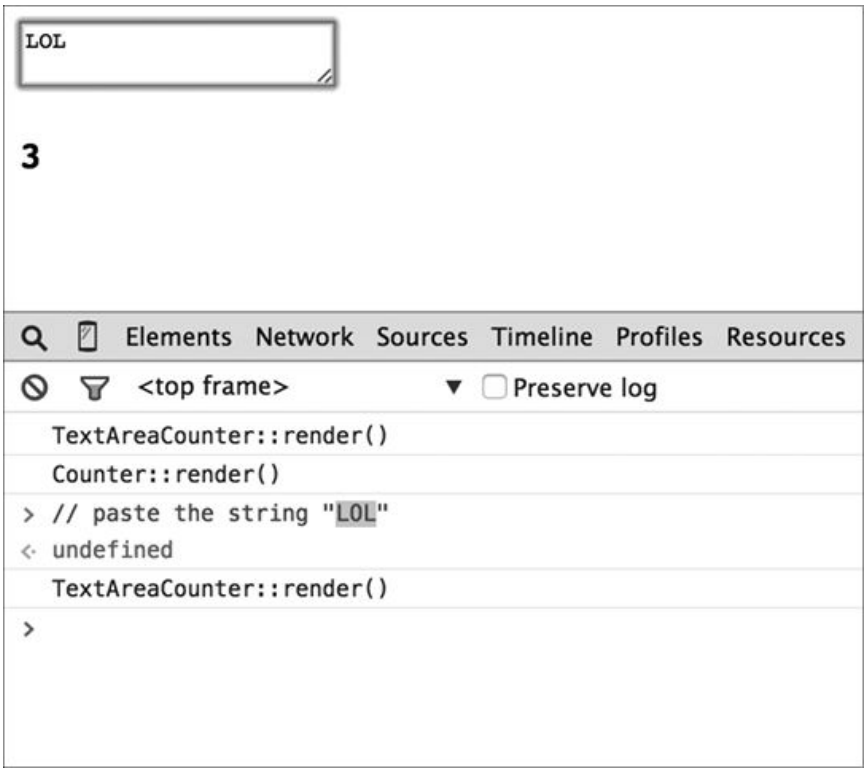


Figura 2.16 – Ganho de desempenho: economizando um ciclo de

nova renderização.

PureRenderMixin

A implementação de shouldComponentUpdate() é bem simples. Além disso, não é preciso um esforço muito maior para deixar essa implementação genérica, pois você sempre compara this.props com nextProps e this.state com nextState. A React oferece uma implementação genérica desse tipo na forma de uma mixin que você pode simplesmente incluir em qualquer componente.

Eis o modo de fazer isso:

```
<script src="react/build/react-with-addons.js"></script>
<script src="react/build/react-dom.js"></script>
<script>
  var Counter = React.createClass({
    name: 'Counter',
    mixins: [React.addons.PureRenderMixin],
    propTypes: {
      count: React.PropTypes.number.isRequired,
    },
    render: function() {
      console.log(this.name + '::render()');
      return React.DOM.span(null, this.props.count);
    }
  });
  // ....
</script>
```

O resultado (Figura 2.17) é o mesmo – o método render() de Counter não é chamado quando não há mudança no número de caracteres.

Observe que PureRenderMixin não faz parte do núcleo da React, mas de uma versão estendida de add-ons da biblioteca. Portanto, para ter acesso a ela, você deve incluir *react/build/react-with-addons.js* em vez de *react/build/react.js*. Isso oferece um novo namespace (espaço de nomes) React.addons e é aí que você pode encontrar PureRenderMixin, assim como outros add-ons interessantes.

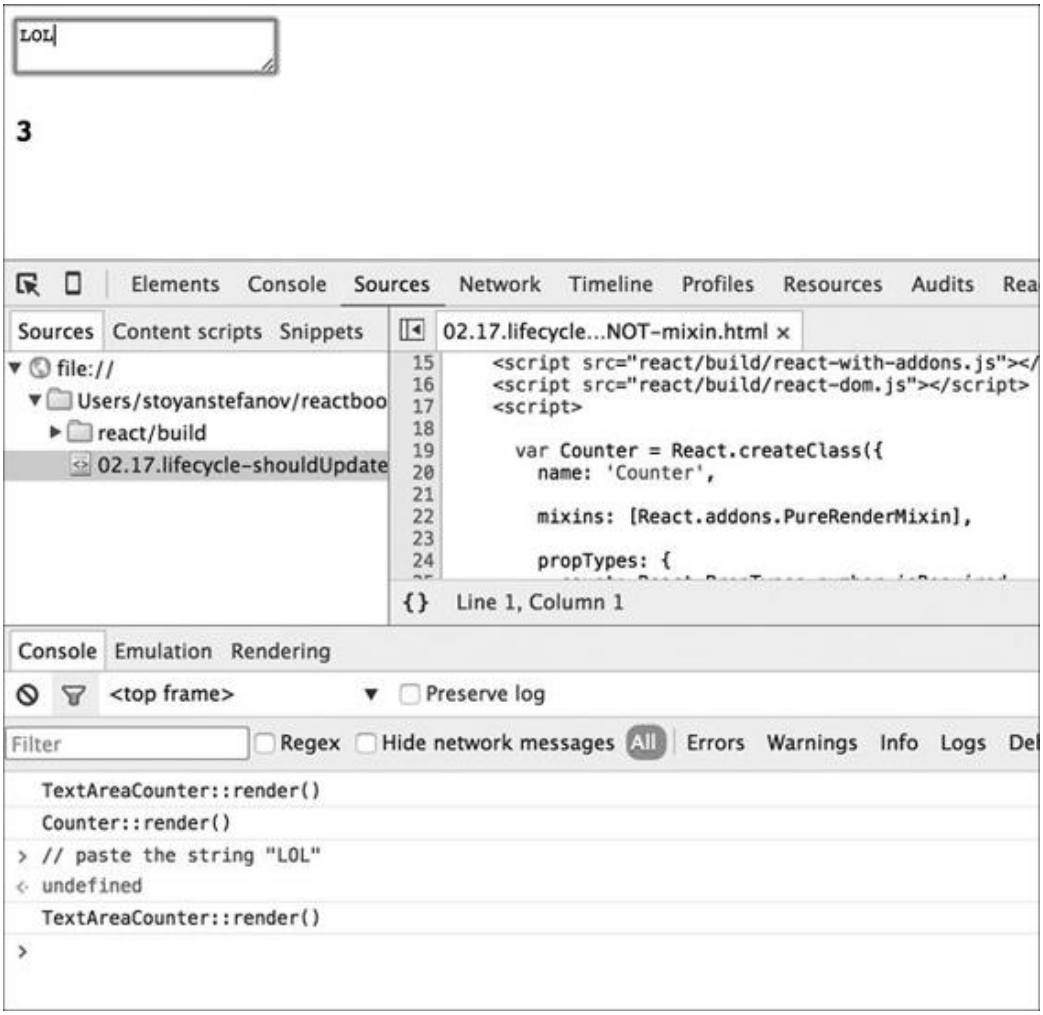


Figura 2.17 – Ganho fácil de desempenho: : combinando PureRenderMixin.

Se você não quiser incluir todos os add-ons ou quiser implementar sua própria versão da mixin, sinta-se à vontade para dar uma espiada na implementação. É bem simples e direta – contém apenas uma verificação simples (não recursiva) de igualdade, que é algo como:

```
var ReactComponentWithPureRenderMixin = {
  shouldComponentUpdate: function(nextProps, nextState) {
    return !shallowEqual(this.props, nextProps) ||
        !shallowEqual(this.state, nextState);
  }
};
```

CAPÍTULO 3

Excel: um componente de tabela elegante

Agora você já sabe como criar componentes React personalizados, compor (renderizar) UI usando componentes genéricos do DOM bem como seus próprios componentes personalizados, definir propriedades, manter um estado, associar-se ao ciclo de vida de um componente e ter um desempenho otimizado ao deixar de fazer novas renderizações quando essas não forem necessárias.

Vamos juntar tudo isso (e conhecer melhor a React nesse processo) criando um componente mais interessante – uma tabela de dados. Será algo como um protótipo inicial do Microsoft Excel v.0.1.beta, que permitirá editar o conteúdo de uma tabela de dados, além de ordenar, pesquisar (filtrar) e exportar dados na forma de arquivos que poderão ser baixados.

Primeiro os dados

As tabelas têm tudo a ver com dados, portanto, o componente de tabela elegante (por que não chamá-lo de Excel?) deve aceitar um array de dados e um array de cabeçalhos. Para testar, vamos obter uma lista de livros mais vendidos da Wikipedia (*http://en.wikipedia.org/wiki/List_of_best-selling_books*):

```
var headers = [
  "Book", "Author", "Language", "Published", "Sales"
];
var data = [
  ["The Lord of the Rings", "J. R. R. Tolkien",
```

```
    "English", "1954–1955", "150 million"],
  ["Le Petit Prince (The Little Prince)", "Antoine de Saint-Exupéry",
    "French", "1943", "140 million"],
  ["Harry Potter and the Philosopher's Stone", "J. K. Rowling",
    "English", "1997", "107 million"],
  ["And Then There Were None", "Agatha Christie",
    "English", "1939", "100 million"],
  ["Dream of the Red Chamber", "Cao Xueqin",
    "Chinese", "1754–1791", "100 million"],
  ["The Hobbit", "J. R. R. Tolkien",
    "English", "1937", "100 million"],
  ["She: A History of Adventure", "H. Rider Haggard",
    "English", "1887", "100 million"],
];
```

Laço para cabeçalhos da tabela

O primeiro passo, somente para começar, é exibir apenas os cabeçalhos. A seguir, apresentamos o que seria uma implementação bem simples:

```
var Excel = React.createClass({
  render: function() {
    return (
      React.DOM.table(null,
        React.DOM.thead(null,
          React.DOM.tr(null,
            this.props.headers.map(function(title) {
              return React.DOM.th(null, title);
            })
          )
        )
      )
    );
  }
});
```

Agora que temos um componente funcional, eis o modo de usá-lo:

```
ReactDOM.render(
  React.createElement(Excel, {
    headers: headers,
```

```
      initialData: data,
    }),
    document.getElementById("app")
  );
```

O resultado desse exemplo inicial está na Figura 3.1.

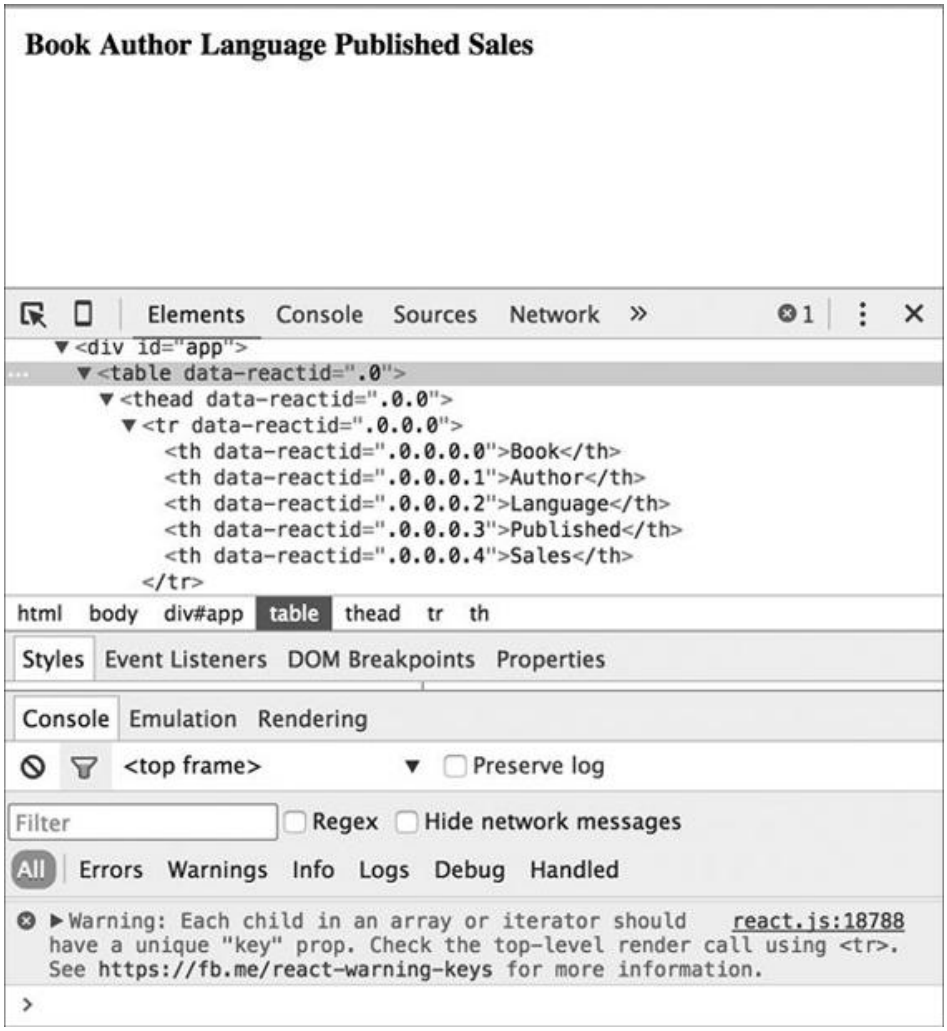


Figura 3.1 – Renderizando os cabeçalhos da tabela.

Há uma novidade aqui – o método map() de array, que é usado para devolver um array de componentes-filhos. O método map() de array toma cada elemento (do array headers nesse caso) e passa-o para uma função de callback. Nesse caso, a função de callback cria um novo componente <th> e o devolve.

Essa é parte da beleza da React – você utiliza JavaScript para criar a sua UI e toda a eficácia dessa linguagem está disponível a você.

Laços e condições funcionam normalmente, e não é necessário aprender outra linguagem de “templating” nem outra sintaxe para construir a UI.

Você pode passar filhos a um componente na forma de um argumento simples de array em vez de usar o que vimos até agora, que foi passar cada filho como um argumento separado. Portanto, essas duas soluções funcionam:

```
// argumentos separados
React.DOM.ul(
  null,
  React.DOM.li(null, 'one'),
  React.DOM.li(null, 'two')
);
// array
React.DOM.ul(
  null,
  [
    React.DOM.li(null, 'one'),
    React.DOM.li(null, 'two')
  ]
);
```

Depurando o aviso do console

A imagem de tela na Figura 3.1 mostra um aviso no console. De que se trata esse aviso e como podemos corrigi-lo? O aviso contém o seguinte: “Warning: Each child in an array or iterator should have a unique ‘key’ prop. Check the top-level render call using <tr>.” (Aviso: Cada filho em um array ou iterador deve ter uma prop “key” única. Verifique a chamada de renderização do nível superior que usa <tr>.).

A “chamada de renderização que usa <tr>”? Como há apenas um componente nessa aplicação, não precisamos de muito para concluir que o problema está aí, mas na vida real, você pode ter muitos componentes que criam elementos <tr>. Excel é apenas uma variável que recebe um componente React fora do mundo da React, portanto ela não é capaz de determinar um nome para esse

componente. Você pode ajudar declarando uma propriedade displayName:

```
var Excel = React.createClass({
  displayName: 'Excel',
  render: function() {
    // ...
  }
};
```

Agora a React é capaz de identificar onde está o problema e avisar você dizendo que “Each child in an array should have a unique “key” prop. Check the render method of `Excel\`.” (Cada filho em um array ou iterador deve ter uma prop “key” única. Verifique o método de renderização de `Excel\`.” Muito melhor. Entretanto, ainda temos um aviso. Para corrigi-lo, basta fazer o que ele diz, agora que você já sabe qual é o render() culpado:

```
this.props.headers.map(function(title, idx) {
  return React.DOM.th({key: idx}, title);
})
```

O que aconteceu aqui? As funções de callback passadas para o método Array.prototype.map() recebem três argumentos: o valor do array, seu índice (0, 1, 2 etc.) e o array todo também. Para fornecer uma propriedade key à React, você pode simplesmente usar o índice (idx) do elemento do array e pronto. As chaves só precisam ser únicas dentro desse array, e não em toda a aplicação React.

Com as chaves corrigidas e a ajuda de um pouco de CSS, você pode agora desfrutar da versão 0.0.1 de seu novo componente – bonito e sem avisos (Figura 3.2).

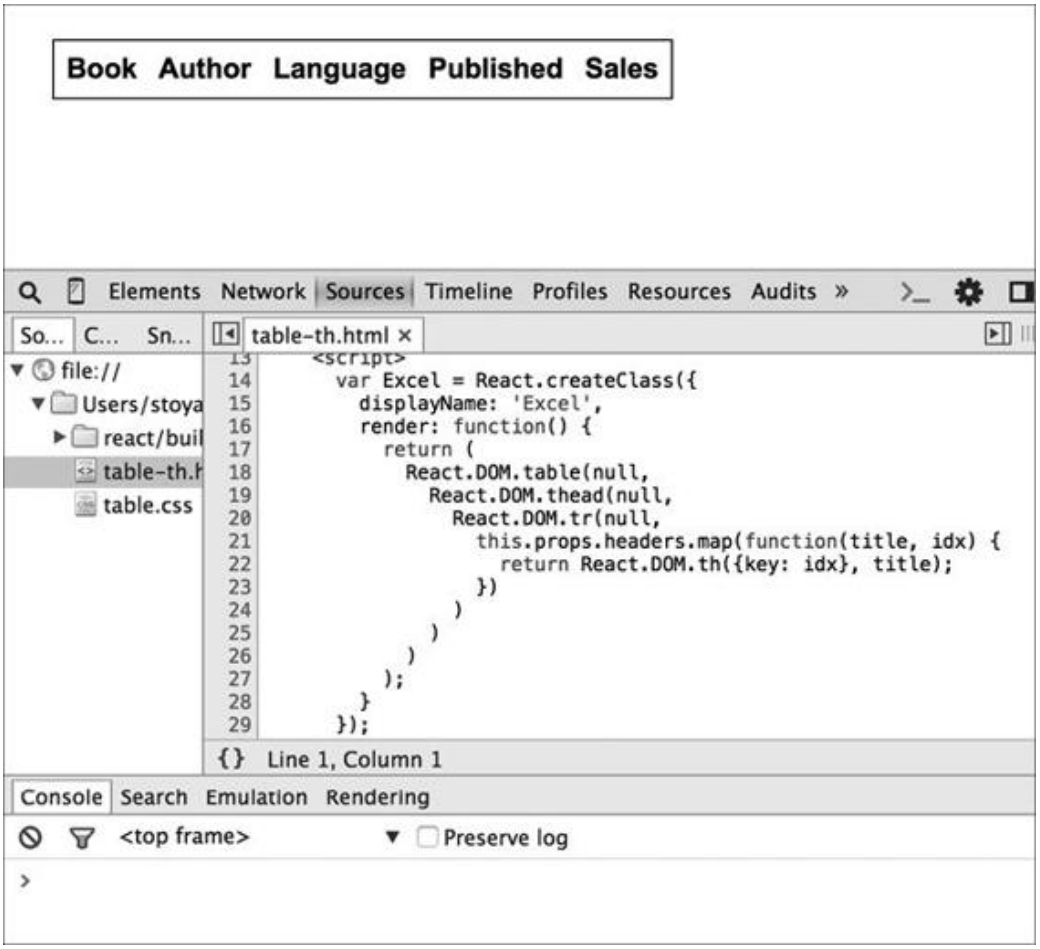


Figura 3.2 – Renderização dos cabeçalhos da tabela, sem avisos.

Adicionar displayName somente para depuração pode parecer uma inconveniência, mas há uma solução: se você usar JSX (discutido no Capítulo 4), não será necessário definir essa propriedade, pois o nome será determinado automaticamente.

Adicionando o conteúdo de <td>

Agora que você tem um cabeçalho de tabela bonito, é hora de adicionar o corpo. O conteúdo do cabeçalho é um array unidimensional (uma única linha), mas `data` é bidimensional. Portanto, você precisa de dois laços: um que percorra as linhas e outro que percorra os dados (células) de cada linha. Isso pode ser feito com os mesmos laços `.map()` que você já sabe usar:

```
data.map(function(row) {
  return (
```

```
      React.DOM.tr(null,
        row.map(function(cell) {
          return React.DOM.td(null, cell);
        })
      )
    );
  })
```

Outro aspecto adicional a ser considerado é o conteúdo da variável `data`: De onde ela vem e como ela muda? Quem chama seu componente `Excel` deve ser capaz de passar dados para inicializar a tabela. Mais tarde, porém, à medida que a tabela continuar ativa, os dados mudarão, pois o usuário deve poder ordenar, editar e assim por diante. Em outras palavras, o *estado do componente mudará.* Portanto, vamos usar `this.state.data` para monitorar as mudanças e utilizar `this.props.initialData` para permitir que quem chama o componente possa inicializá-lo. Uma implementação completa pode ter o seguinte aspecto (resultado mostrado na Figura 3.3):

```
  getInitialState: function() {
    return {data: this.props.initialData};
  },
  render: function() {
    return (
      React.DOM.table(null,
        React.DOM.thead(null,
          React.DOM.tr(null,
            this.props.headers.map(function(title, idx) {
              return React.DOM.th({key: idx}, title);
            })
          )
        ),
        React.DOM.tbody(null,
          this.state.data.map(function(row, idx) {
            return (
              React.DOM.tr({key: idx},
                row.map(function(cell, idx) {
                  return React.DOM.td({key: idx}, cell);
                })
              )
```

```
            );
          })
        )
      )
    );
  }
```

Você pode ver o {key: idx} repetitivo que atribui uma chave única a cada elemento em um array de componentes. Embora todos os laços .map() comecem pelo índice 0, isso não é um problema, pois as chaves só precisam ser únicas no laço atual, e não na aplicação toda.

A função render() já está ficando um pouco difícil de seguir, especialmente para manter o controle de } e) de fechamento. Não tenha medo – o JSX está prestes a aliviar o sofrimento!

O trecho de código anterior não tem a propriedade propTypes (é opcional, mas, com frequência, é uma boa ideia usá-la). Ela serve tanto para validação de dados quanto para documentação do componente. Vamos ser realmente específicos e tentar o máximo possível reduzir a probabilidade de alguém fornecer dados indevidos ao belo componente Excel. React.PropTypes oferece um validador de array para garantir que a propriedade seja sempre um array.

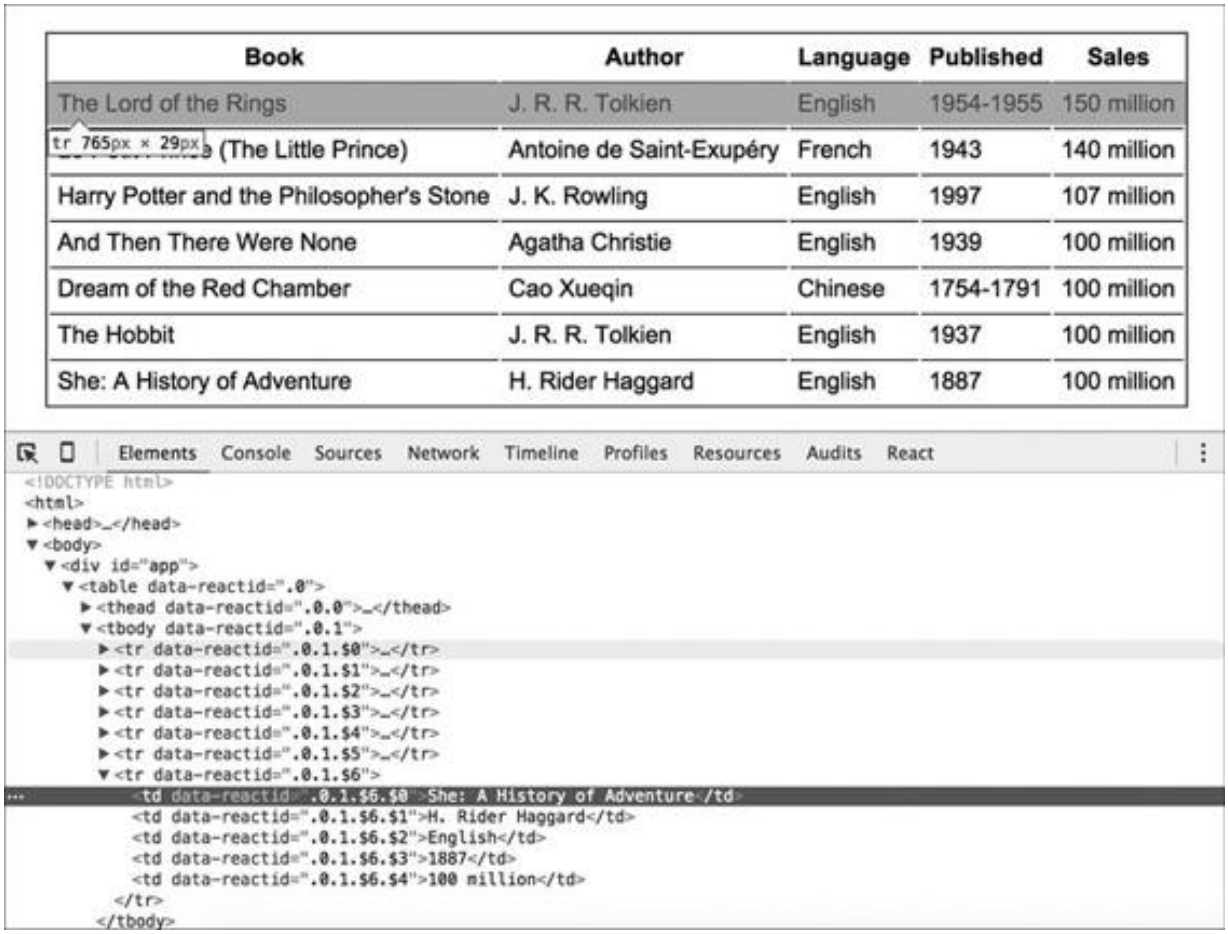


Figura 3.3 – Renderizando a tabela completa.

E ele vai além com arrayOf, com o qual você pode especificar o tipo dos elementos do array. Nesse caso, vamos aceitar apenas strings para os títulos dos cabeçalhos e para os dados:

```
  propTypes: {
    headers: React.PropTypes.arrayOf(
      React.PropTypes.string
    ),
    initialData: React.PropTypes.arrayOf(
      React.PropTypes.arrayOf(
        React.PropTypes.string
      )
    ),
  },
```

Agora isso está rigoroso!

Como podemos melhorar o componente?

Permitir dados somente do tipo string é um pouco restritivo demais em uma planilha Excel genérica. Como exercício, você pode aceitar mais tipos de dados (React.PropTypes.any) e renderizar de modo diferente de acordo com o tipo (por exemplo, alinhar os números à direita).

Ordenação

Quantas vezes você já viu uma tabela em uma página web que você desejou que estivesse ordenada de modo diferente? Felizmente, fazer isso com a React é trivial. Na verdade, esse é um exemplo em que a React se destaca, pois tudo que é necessário fazer é ordenar o array de dados, e todas as atualizações de UI serão tratadas para você.

Em primeiro lugar, acrescente um handler de clique na linha de cabeçalho:

```
React.DOM.table(null,
  React.DOM.thead({onClick: this._sort},
    React.DOM.tr(null,
    // ...
```

Agora vamos implementar a função _sort. Você deve saber qual é a coluna de acordo com a qual a ordenação será feita; ela pode ser convenientemente obtida usando a propriedade cellIndex do alvo do evento (o alvo do evento é um cabeçalho de tabela <th>):

```
var column = e.target.cellIndex;
```

Talvez você não tenha visto cellIndex ser usado com muita frequência em desenvolvimento de aplicações. É uma propriedade definida há muito tempo, desde o DOM Nível 1, como “o índice desta célula na linha” e, mais tarde, passou a ser somente para leitura no DOM Nível 2.

Também precisamos de uma cópia dos dados a serem ordenados. Caso contrário, se usarmos o método sort() do array diretamente, ele o modificará, o que significa que this.state.data.sort() modificará this.state. Como já sabemos, this.state não deve ser modificado diretamente,

mas somente por meio de setState():

```
// copia os dados
var data = this.state.data.slice(); // ou `Array.from(this.state.data)` no ES6
```

A ordenação propriamente dita é feita agora por meio de uma callback para o método sort():

```
data.sort(function(a, b) {
  return a[column] > b[column] ? 1 : -1;
});
```

Por fim, a linha a seguir define o estado com os novos dados ordenados:

```
this.setState({
  data: data,
});
```

A partir de agora, quando você clicar em um cabeçalho, o conteúdo será reorganizado em ordem alfabética (Figura 3.4).

Book	Author	Language	Published	Sales
And Then There Were None	Agatha Christie	English	1939	100 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1754-1791	100 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	107 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	140 million
She: A History of Adventure	H. Rider Haggard	English	1887	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	150 million

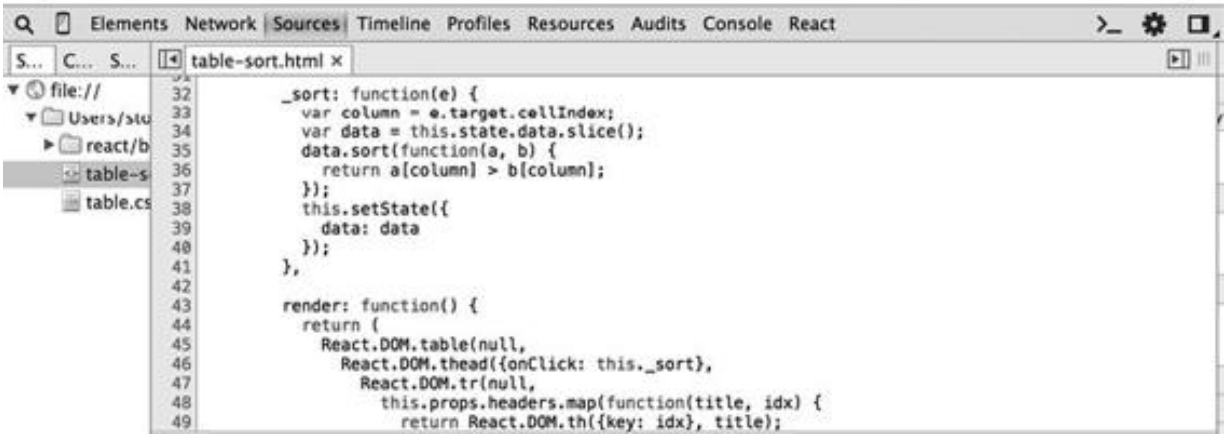


Figura 3.4 – Ordenando de acordo com o título do livro.

É isso – não é necessário mexer em nada na renderização da UI. No método render(), você já definiu de uma vez por todas qual deve ser a aparência do componente, considerando que alguns dados tenham sido fornecidos. Quando os dados mudarem, o mesmo ocorrerá com a UI; no entanto, isso não será mais de sua responsabilidade.

Como podemos melhorar o componente?

Essa é uma ordenação bem simples, apenas o suficiente para ser relevante em nossa discussão sobre a React. Você pode deixar o componente tão sofisticado quanto for necessário, fazendo parse do conteúdo para saber se os valores são numéricos, se têm ou não uma unidade de medida, e assim por diante.

Pistas sobre ordenação na UI

A tabela está bem ordenada, mas não está claro de acordo com qual coluna isso foi feito. Vamos atualizar a UI para mostrar setas com base na coluna sendo ordenada. Enquanto fazemos isso, vamos implementar também a ordem decrescente.

Para manter o controle do novo estado, você precisa de duas novas propriedades:

this.state.sortby
: É o índice da coluna sendo ordenada no momento.

this.state.descending
: É um booleano para determinar se a ordem é crescente ou decrescente.

```
getInitialState: function() {
  return {
    data: this.props.initialData,
    sortby: null,
    descending: false,
  };
```

```
},
```

Na função `_sort()`, você deve determinar a direção da ordenação. O default é a ordem crescente, a menos que o índice da nova coluna seja igual à coluna com base na qual a ordenação está efetuada no momento e a ordem ainda não seja decrescente:

```
var descending = this.state.sortby === column && !this.state.descending;
```

Você precisa também de um pequeno ajuste na callback de ordenação:

```
data.sort(function(a, b) {
  return descending
    ? (a[column] < b[column] ? 1 : -1)
    : (a[column] > b[column] ? 1 : -1);
});
```

Por fim, você deve definir o novo estado:

```
this.setState({
  data: data,
  sortby: column,
  descending: descending,
});
```

A única tarefa restante é atualizar a função `render()` para indicar a direção da ordenação. Para a coluna ordenada no momento, vamos simplesmente acrescentar um símbolo de seta no título:

```
this.props.headers.map(function(title, idx) {
  if (this.state.sortby === idx) {
    title += this.state.descending ? ' \u2191' : ' \u2193'
  }
  return React.DOM.th({key: idx}, title);
}, this)
```

Agora a funcionalidade de ordenação está completa – as pessoas podem ordenar de acordo com qualquer coluna, podem clicar uma vez para colocá-la em ordem crescente e mais uma vez para deixá-la em ordem decrescente, e a UI será atualizada com a indicação visual (Figura 3.5).

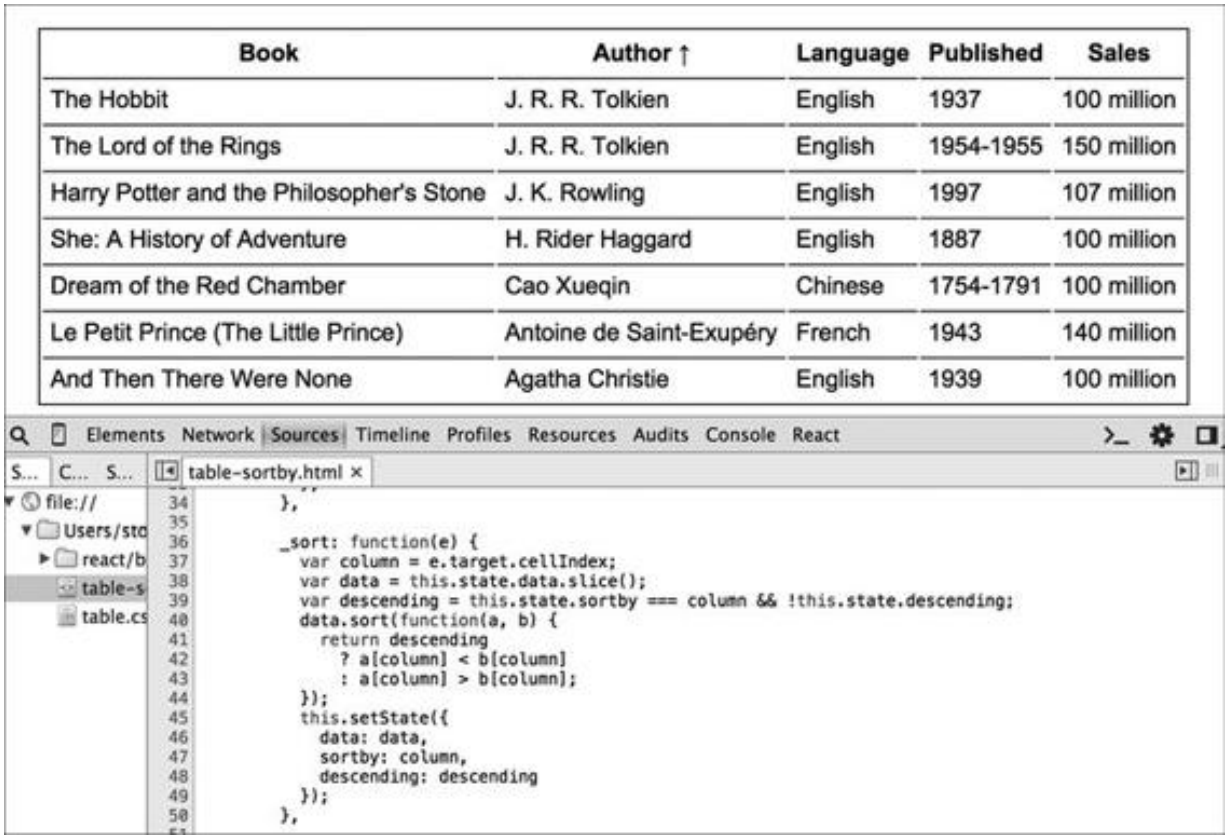


Figura 3.5 – Ordem crescente/decrescente.

Editando dados

O próximo passo no componente Excel é dar às pessoas a opção de editar os dados da tabela. Uma solução poderia funcionar assim:

1. Dê um clique duplo em uma célula. O Excel descobre qual é a célula e transforma o seu conteúdo, passando-o de texto simples para um campo de entrada previamente preenchido com o conteúdo (Figura 3.6).
2. Edite o conteúdo (Figura 3.7).
3. Tecle Enter. A campo de entrada desaparece e a tabela é atualizada com o novo texto (Figura 3.8).

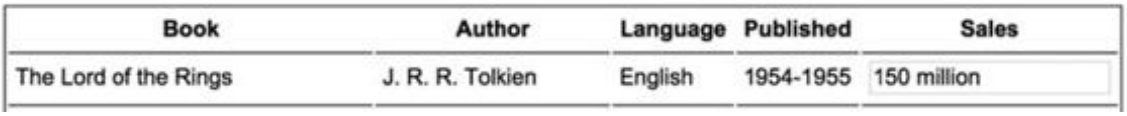


Figura 3.6 – Uma célula da tabela se transforma em um campo de

entrada após um clique duplo.

Book	Author	Language	Published	Sales
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	140 million

Figura 3.7 – Edição do conteúdo.

Book	Author	Language	Published	Sales
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	140 million

Figura 3.8 – Conteúdo atualizado ao teclar Enter.

Célula em que é possível editar

A primeira tarefa é configurar um event handler simples. Quando houver um clique duplo, o componente "se lembrará" da célula selecionada:

```
React.DOM.tbody({onDoubleClick: this._showEditor}, ....)
```

Observe o onDoubleClick mais amigável e mais fácil de ler, em oposição a ondblclick do W3C.

Vamos ver a aparência de `_showEditor`:

```
_showEditor: function(e) {
  this.setState({edit: {
    row: parseInt(e.target.dataset.row, 10),
    cell: e.target.cellIndex,
  }});
},
```

O que está acontecendo aqui?

- A função define a propriedade `edit` de `this.state`. Essa propriedade é `null` quando não há nenhuma edição em andamento, mas passa a ser um objeto com propriedades `row` e `cell`, contendo o índice da linha e o índice da célula sendo editada. Portanto, se você der um clique duplo na primeira célula, `this.state.edit` terá o valor `{row: 0, cell: 0}`.

- Para determinar o índice da célula, utilize o mesmo e.target.cellIndex que usamos antes, em que e.target é o <td> que recebeu o clique duplo.
- Não há nenhum rowIndex vindo de graça do DOM, portanto, você precisa obtê-lo por conta própria usando um atributo data-. Cada célula deve ter um atributo data-row com o índice da linha, no qual você pode aplicar parseInt() para obter o índice.

Por fim, há mais alguns esclarecimentos e pré-requisitos. Em primeiro lugar, a propriedade edit não existia antes e deve ser inicializada também no método getInitialState() que, a partir de agora, deve ter o seguinte aspecto:

```
getInitialState: function() {
  return {
    data: this.props.initialData,
    sortby: null,
    descending: false,
    edit: null, // {row: index, cell: index}
  };
},
```

A propriedade data-row é um dado de que você precisa para poder manter o controle dos índices das linhas. Vamos ver a implementação completa de tbody():

```
React.DOM.tbody({onDoubleClick: this._showEditor},
  this.state.data.map(function(row, rowidx) {
    return (
      React.DOM.tr({key: rowidx},
        row.map(function(cell, idx) {
          var content = cell;

          // TODO - transformar `content` em uma entrada se `idx`
          // e `rowidx` forem iguais aos da célula sendo editada;
          // caso contrário, simplesmente mostra o conteúdo textual

          return React.DOM.td({
            key: idx,
            'data-row': rowidx
          }, content);
        }, this)
```

```
          )
        );
      }, this)
    )
```

PPor fim, precisamos fazer o que está especificado em TODO. Vamos criar um campo de entrada quando for necessário. A função render() completa é chamada novamente, apenas por causa da chamada a setState(), que configura a propriedade edit. A React renderiza novamente a tabela, o que dá a você a oportunidade de atualizar a célula da tabela que recebeu o clique duplo.

Célula com campo de entrada

Vamos dar uma olhada no código que substituirá o comentário TODO. Inicialmente, guarde o estado da edição:

```
var edit = this.state.edit;
```

Verifique se edit está ligado e, em caso afirmativo, se essa é exatamente a célula sendo editada:

```
if (edit && edit.row === rowidx && edit.cell === idx) {
  // ...
}
```

Se essa for a célula-alvo, vamos criar um formulário e um campo de entrada com o conteúdo da célula:

```
content = React.DOM.form({onSubmit: this._save},
  React.DOM.input({
    type: 'text',
    defaultValue: content,
  })
);
```

Como podemos ver, é um formulário com uma única entrada, e ela está previamente preenchida com o texto da célula. Quando for submetido, o formulário será capturado no método _save() privado.

Salvando

A última parte do quebra-cabeça de edição é salvar as mudanças de conteúdo depois que o usuário terminar de digitar e tiver submetido o formulário (por meio da tecla Enter):

```
_save: function(e) {
  e.preventDefault();
  // ... salva
},
```

Depois de evitar o comportamento default (para que a página não seja recarregada), você precisa obter uma referência para o campo de entrada:

```
var input = e.target.firstChild;
```

Clone os dados para não manipular `this.state` diretamente:

```
var data = this.state.data.slice();
```

Atualize os dados, considerando o novo valor e os índices da célula e da linha armazenados na propriedade `edit` de `state`:

```
data[this.state.edit.row][this.state.edit.cell] = input.value;
```

Por fim, defina o estado, o que provoca uma nova renderização da UI:

```
this.setState({
  edit: null, // edição finalizada
  data: data,
});
```

Conclusão e diferenças com o DOM virtual

A essa altura, a funcionalidade de edição está completa. Ela não exigiu muito código. Tudo que você precisou fazer foi:

- controlar qual célula seria editada por meio de `this.state.edit`;
- renderizar um campo de entrada ao exibir a tabela se os índices da linha e da célula coincidirem com a célula em que o usuário deu um clique duplo;
- atualizar o array de dados com o novo valor proveniente do campo de entrada.

Assim que você executou setState() com o novo dado, a React chamou o método render() do componente e a UI foi atualizada num passe de mágica. Pode parecer que não seria particularmente eficiente renderizar a tabela toda por causa da mudança no conteúdo de uma única célula. Na verdade, a React atualiza apenas uma única célula.

Se você abrir as ferramentas de desenvolvedor de seu navegador, poderá ver quais partes da árvore do DOM são atualizadas à medida que interagir com a sua aplicação. Na Figura 3.9, podemos ver as ferramentas de desenvolvedor destacando a mudança no DOM após a mudança do idioma de “The Lord of the Rings” de English para Engrish.

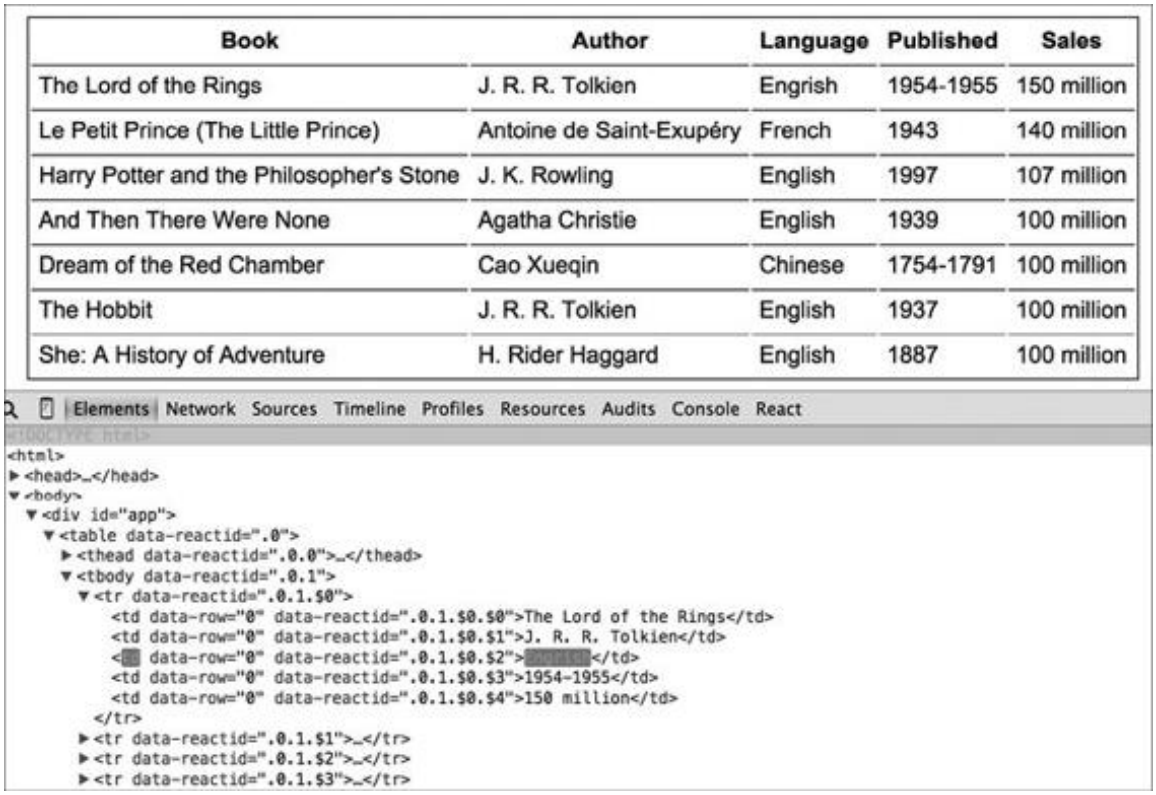


Figura 3.9 – Destacando as mudanças no DOM.

Internamente, a React chama o seu método render() e cria uma representação de árvore leve do DOM resultante desejado. Isso é conhecido como uma *árvore virtual de DOM.* Quando o método render() é chamado novamente (depois de uma chamada a setState(),

por exemplo), a React calcula a diferença entre a árvore virtual antes e depois. Com base nessa diferença, a React determina as operações mínimas de DOM necessárias (por exemplo, `appendChild()`, `textContent` etc.) para tratar essa mudança no DOM do navegador.

Na Figura 3.9, há apenas uma alteração necessária na célula, e não é preciso renderizar novamente toda a tabela. Ao determinar o conjunto mínimo de alterações e reunir as operações de DOM em lote, a React "toca" de leve o DOM, pois o fato de as operações de DOM serem lentas (em comparação com operações JavaScript puras, chamadas de função etc.) é um problema conhecido e, com frequência, são o gargalo no desempenho da renderização de aplicações web sofisticadas.

Resumindo uma longa história, a React protege você quando se trata de desempenho e de atualização de UI porque:

- toca o DOM de leve;
- usa delegação de eventos para interações com o usuário.

Pesquisa

Em seguida, vamos adicionar uma funcionalidade de pesquisa no componente `Excel`, que permita aos usuários filtrar o conteúdo da tabela. Eis o plano:

- adicionar um botão para ativar e desativar a nova funcionalidade de forma alternada (toggle) – veja a Figura 3.10;

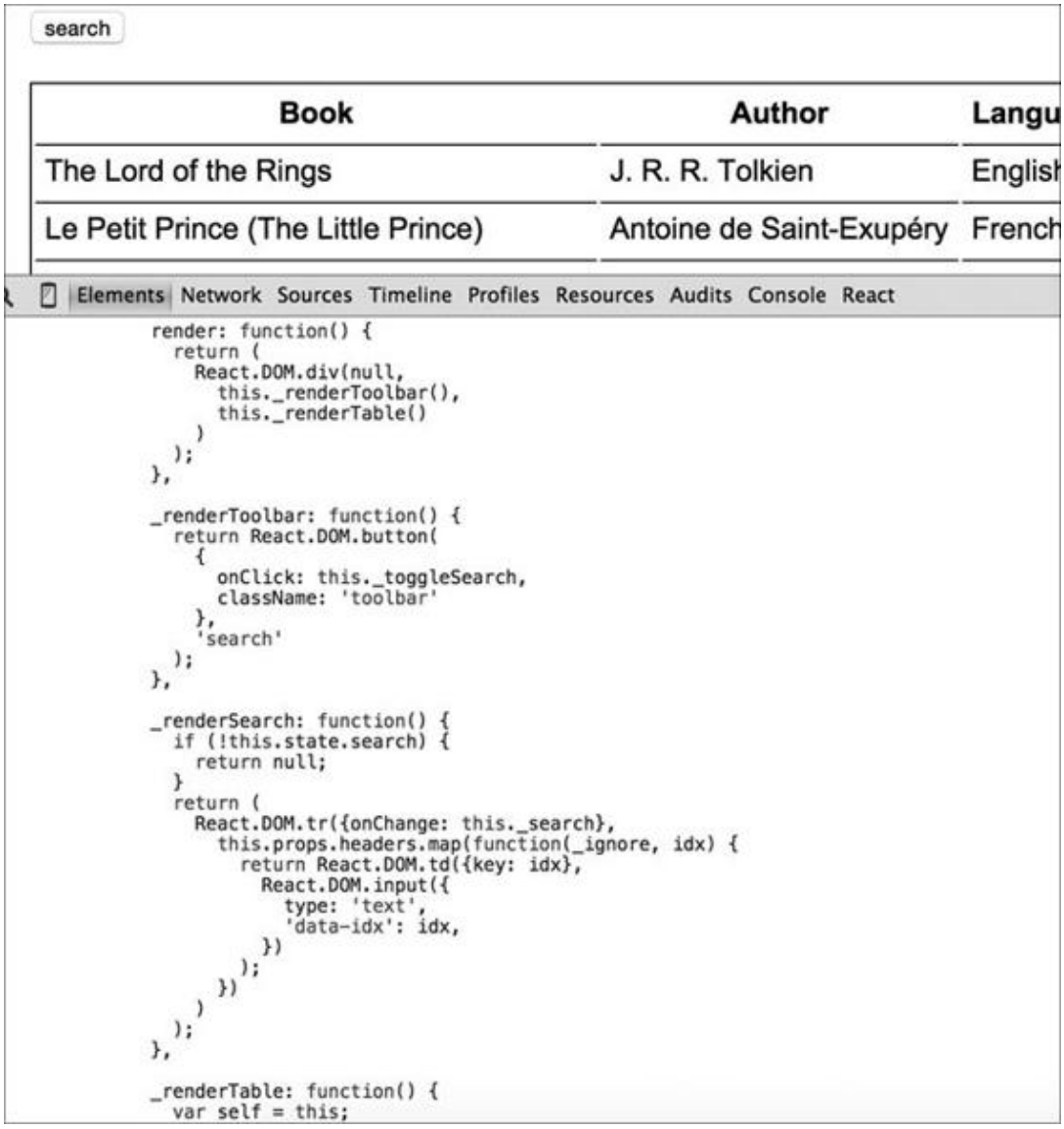


Figura 3.10 – Botão de pesquisa.

- se a pesquisa estiver ativa, acrescentar uma linha de entradas em que cada uma pesquise na coluna correspondente (Figura 3.11);
- à medida que o usuário digitar em uma caixa de entrada, filtrar o array de state.data para que mostre somente o conteúdo correspondente (Figura 3.12).

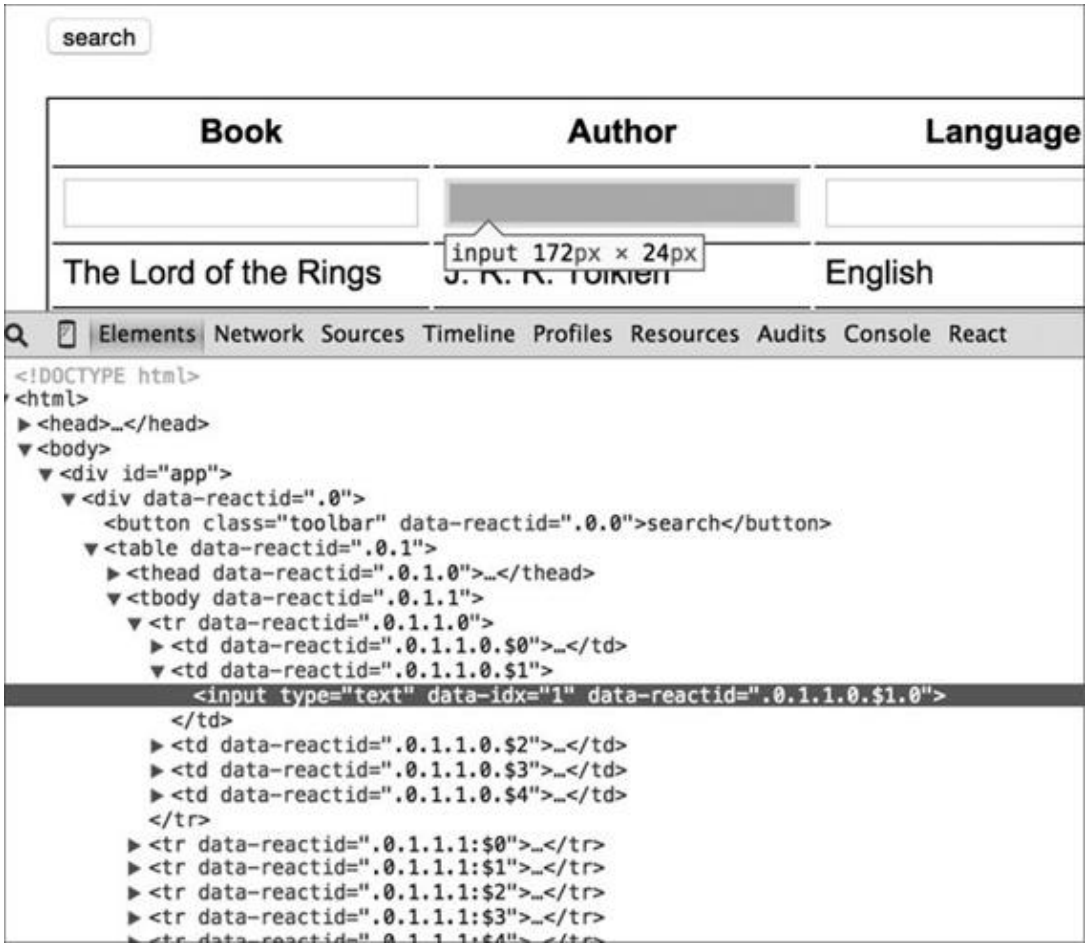


Figura 3.11 – Linha de entradas para pesquisa/filtro.

search

Book	Author	Language	Published	Sales
	J.			
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	150 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	107 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million

Figura 3.12 – Resultados da pesquisa.

Estado e UI

A primeira tarefa a ser feita é acrescentar uma propriedade search ao objeto this.state para monitorar se a funcionalidade de pesquisa está ativa:

```
  getInitialState: function() {
    return {
      data: this.props.initialData,
      sortby: null,
      descending: false,
      edit: null, // [row index, cell index],
      search: false,
    };
  },
```

Em seguida, faremos a atualização da UI. Para manter tudo mais administrável, vamos dividir a função render() em porções menores e dedicadas. Até agora, a função render() estava apenas renderizando uma tabela. Vamos renomeá-la para _renderTable(). Em seguida, o botão de pesquisa fará parte de uma barra de ferramentas completa (você adicionará uma funcionalidade de "exportação" em breve), portanto vamos renderizá-la como parte de uma função _renderToolbar().

O resultado tem o seguinte aspecto:

```
  render: function() {
    return (
      React.DOM.div(null,
        this._renderToolbar(),
        this._renderTable()
      )
    );
  },
  _renderToolbar: function() {
    // TODO
  },
  _renderTable: function() {
    // igual à função anteriormente conhecida como `render()`
  },
```

Como podemos ver, a nova função render() devolve um contêiner div com dois filhos: a barra de ferramentas e a tabela. Você já conhece a aparência da renderização da tabela, e a barra de ferramentas contém apenas um botão por enquanto:

```
  _renderToolbar: function() {
    return React.DOM.button(
      {
        onClick: this._toggleSearch,
        className: 'toolbar',
      },
      'search'
    );
  },
```

Se a pesquisa estiver ativa (o que significa que this.state.search é true), você precisará de uma nova linha de tabela cheia de entradas. Vamos deixar que uma função _renderSearch() cuide disso:

```
  _renderSearch: function() {
    if (!this.state.search) {
      return null;
    }
    return (
      React.DOM.tr({onChange: this._search},
        this.props.headers.map(function(_ignore, idx) {
          return React.DOM.td({key: idx},
            React.DOM.input({
              type: 'text',
              'data-idx': idx,
            })
          );
        })
      )
    );
  },
```

Como podemos ver, se a funcionalidade de pesquisa não estiver ativa, a função não precisará renderizar nada, portanto, ela devolverá null. Outra opção, é claro, é deixar quem chama essa função tomar a decisão de não a chamar se a pesquisa não estiver ativa. Porém, o exemplo anterior ajuda a função _renderTable() já muito atarefada a ficar um pouco mais simples.

Eis o que _renderTable() precisa fazer:

- **Antes**:

```
React.DOM.tbody({onDoubleClick: this._showEditor},
  this.state.data.map(function(row, rowidx) { // ...
```

- **Depois**:

```
React.DOM.tbody({onDoubleClick: this._showEditor},
  this._renderSearch(),
  this.state.data.map(function(row, rowidx) { // ...
```

As entradas da pesquisa são apenas outro nó-filho antes do laço principal de `data` (aquele que cria todas as linhas e células da tabela). Quando `_renderSearch()` devolve `null`, a React simplesmente não renderiza o filho adicional e passa para a tabela.

A essa altura, é tudo de que precisamos para atualizações de UI. Vamos dar uma olhada no cerne da funcionalidade, isto é, na "lógica de negócios", se você preferir: a pesquisa propriamente dita.

Filtrando o conteúdo

A funcionalidade de pesquisa será razoavelmente simples: tome o array de dados, chame o método `Array.prototype.filter()` nesse array e devolva um array filtrado com os elementos que correspondam à string de pesquisa.

A UI continua usando `this.state.data` para a renderização, mas `this.state.data` é uma versão reduzida de si mesma.

Você precisa de uma cópia dos dados antes da pesquisa, para que não perca esses dados para sempre. Isso permite que o usuário retorne à tabela completa ou mude a string de pesquisa para obter resultados diferentes. Vamos chamar essa cópia (na verdade, é uma referência) de `_preSearchData`:

```
var Excel = React.createClass({
  // código..
  _preSearchData: null,
  // mais código...
});
```

Quando o usuário clicar no botão "search", a função `_toggleSearch()` será chamada. A tarefa dessa função é alterar a funcionalidade de

pesquisa entre ativa e não ativa. Ela faz o seu trabalho:

- definindo this.state.search para true ou false conforme a funcionalidade esteja ativa ou não;
- quando a pesquisa for ativada, os dados antigos devem ser "lembrados";
- quando a pesquisa for desativada, os dados antigos devem ser restaurados.

Eis uma possível implementação para essa função:

```
_toggleSearch: function() {
  if (this.state.search) {
    this.setState({
      data: this._preSearchData,
      search: false,
    });
    this._preSearchData = null;
  } else {
    this._preSearchData = this.state.data;
    this.setState({
      search: true,
    });
  }
},
```

A última tarefa a ser feita é implementar a função _search(), que é chamada sempre que algo na linha de pesquisa mudar, o que significa que o usuário está digitando em uma das entradas. Eis a implementação completa, seguida de mais alguns detalhes:

```
_search: function(e) {
  var needle = e.target.value.toLowerCase();
  if (!needle) { // a string de pesquisa foi apagada
    this.setState({data: this._preSearchData});
    return;
  }
  var idx = e.target.dataset.idx; // a coluna a ser pesquisada
  var searchdata = this._preSearchData.filter(function(row) {
    return row[idx].toString().toLowerCase().indexOf(needle) > -1;
  });
```

```
    this.setState({data: searchdata});
  },
```

Obtenha a string de pesquisa a partir do alvo do evento de mudança (que é a caixa de entrada):

```
  var needle = e.target.value.toLowerCase();
```

Se não houver nenhuma string de pesquisa (o usuário apagou o que foi digitado), a função tomará os dados originais em cache, e esses dados passarão a constituir o novo estado:

```
  if (!needle) {
    this.setState({data: this._preSearchData});
    return;
  }
```

Se houver uma string de pesquisa, filtre os dados originais e defina os resultados filtrados como o novo estado dos dados:

```
  var idx = e.target.dataset.idx;
  var searchdata = this._preSearchData.filter(function(row) {
    return row[idx].toString().toLowerCase().indexOf(needle) > -1;
  });
  this.setState({data: searchdata});
```

Com isso, a funcionalidade de pesquisa está completa. Para implementar a funcionalidade, tudo que você precisou fazer foi:

- adicionar a UI de pesquisa;
- mostrar/ocultar a nova UI por demanda;
- implementar a “lógica de negócios” propriamente dita – uma simples chamada a `filter()` no array.

Nada na renderização original da tabela teve que ser realmente alterado. Como sempre, você só precisa se preocupar com o estado de seus dados e deixar a React cuidar da renderização (e de todo o trabalho pesado associado ao DOM) sempre que o estado dos dados mudar.

Como podemos melhorar a pesquisa?

Esse foi um exemplo funcional simples para ilustração. Podemos

melhorar a funcionalidade?

Algo simples que podemos fazer é alternar o rótulo do botão de pesquisa. Por exemplo, quando a pesquisa estiver ativa (`this.state.search === true`), o botão informará “Done searching” (Pesquisa feita).

Outra tentativa é implementar uma *pesquisa aditiva* em várias caixas, o que significa filtrar os dados já filtrados. Se o usuário digitar “Eng” na linha de idiomas e então pesquisar usando uma caixa de pesquisa diferente, por que não fazer a pesquisa somente nos resultados da pesquisa anterior? Como você implementaria essa funcionalidade?

Reprodução instantânea

Como você já sabe agora, os seus componentes se preocupam com seus estados e deixam a React renderizar e renderizar novamente sempre que for apropriado. Isso quer dizer que, considerando os mesmos dados (estado e propriedades), a aplicação terá exatamente a mesma aparência, independentemente do que mudou antes ou depois desse estado de dados em particular. Isso dá a você uma ótima oportunidade de depuração no mundo real.

Suponha que alguém encontre um bug enquanto estava usando a sua aplicação – essa pessoa pode clicar em um botão para informar o bug sem precisar explicar o que aconteceu. O relatório de bug pode simplesmente enviar uma cópia de `this.state` e de `this.props`, e você deverá ser capaz de recriar o estado exato da aplicação e ver o resultado visualmente.

Um “undo” (desfazer) poderia ser outra funcionalidade baseada no fato de a React renderizar sua aplicação da mesma maneira quando tiver as mesmas propriedades e o mesmo estado. De fato, a implementação de “undo” é trivial: basta retornar ao estado anterior.

Vamos levar essa ideia um pouco mais além, somente por diversão. Vamos registrar cada mudança de estado no componente `Excel` e

então reproduzi-las. É fascinante observar todas as suas ações serem reproduzidas diante de você.

No que diz respeito à implementação, não vamos nos preocupar com a questão de *quando* a alteração ocorreu e simplesmente "reproduzir" as mudanças no estado da aplicação a intervalos de um segundo. Para implementar essa funcionalidade, tudo que você precisa fazer é acrescentar um método `_logSetState()` e procurar/substituir todas as chamadas a `setState()` por chamadas à nova função.

Assim, todas as chamadas a…

```
this.setState(newSate);
```

…passam a ser

```
this._logSetState(newState);
```

`_logSetState` deve executar duas tarefas: fazer log do novo estado e então passá-lo para `setState()`. Eis uma implementação de exemplo, em que você faz uma cópia profunda (deep copy) do estado e a concatena em `this._log`:

```
var Excel = React.createClass({
  _log: [],
  _logSetState: function(newState) {
    // guarda o estado antigo em um clone
    this._log.push(JSON.parse(JSON.stringify(
      this._log.length === 0 ? this.state : newState
    )));
    this.setState(newState);
  },
  // ....
});
```

Agora que todas as mudanças de estado foram registradas em log, vamos reproduzi-las. Para disparar a reprodução, vamos acrescentar um listener de eventos simples que capture ações de teclado e chame a função `_replay()`:

```
componentDidMount: function() {
  document.onkeydown = function(e) {
```

```
      if (e.altKey && e.shiftKey && e.keyCode === 82) { // ALT+SHIFT+R(eplay)
        this._replay();
      }
    }.bind(this);
  },
```

Por fim, vamos adicionar o método `_replay()`. Ele utiliza `setInterval()` e, uma vez por segundo, lê o próximo objeto do log e passa-o para `setState()`:

```
  _replay: function() {
    if (this._log.length === 0) {
      console.warn('No state to replay yet');
      return;
    }
    var idx = -1;
    var interval = setInterval(function() {
      idx++;
      if (idx === this._log.length - 1) { // fim
        clearInterval(interval);
      }
      this.setState(this._log[idx]);
    }.bind(this), 1000);
  },
```

Como podemos melhorar a reprodução?

Que tal implementar uma funcionalidade Undo/Redo (Desfazer/Refazer)? Por exemplo, quando a pessoa usar a combinação de teclas ALT+Z, você retorna um passo no log de estados e se teclar ALT+SHIFT+Z, você avança.

Uma implementação alternativa?

Há outra maneira de implementar o tipo de funcionalidade replay/undo (reproduzir/desfazer) sem alterar todas as suas chamadas a `setState()`? Talvez usando um método apropriado de ciclo de vida (Capítulo 2)?

Download dos dados da tabela

Depois de toda ordenação, edição e pesquisa, o usuário finalmente está satisfeito com o estado dos dados da tabela. Seria interessante se o usuário pudesse fazer download do resultado de todo o trabalho para usá-lo mais tarde.

Felizmente, não há nada mais fácil com a React. Tudo que você precisa fazer é obter o `this.state.data` atual e devolvê-lo – em formato JSON ou CSV.

A Figura 3.13 mostra o resultado final quando um usuário clicar em "Export CSV" (Exportar CSV), fazer download do arquivo chamado *data.csv* (veja a parte inferior à esquerda da janela do navegador) e abrir esse arquivo no Microsoft Excel.

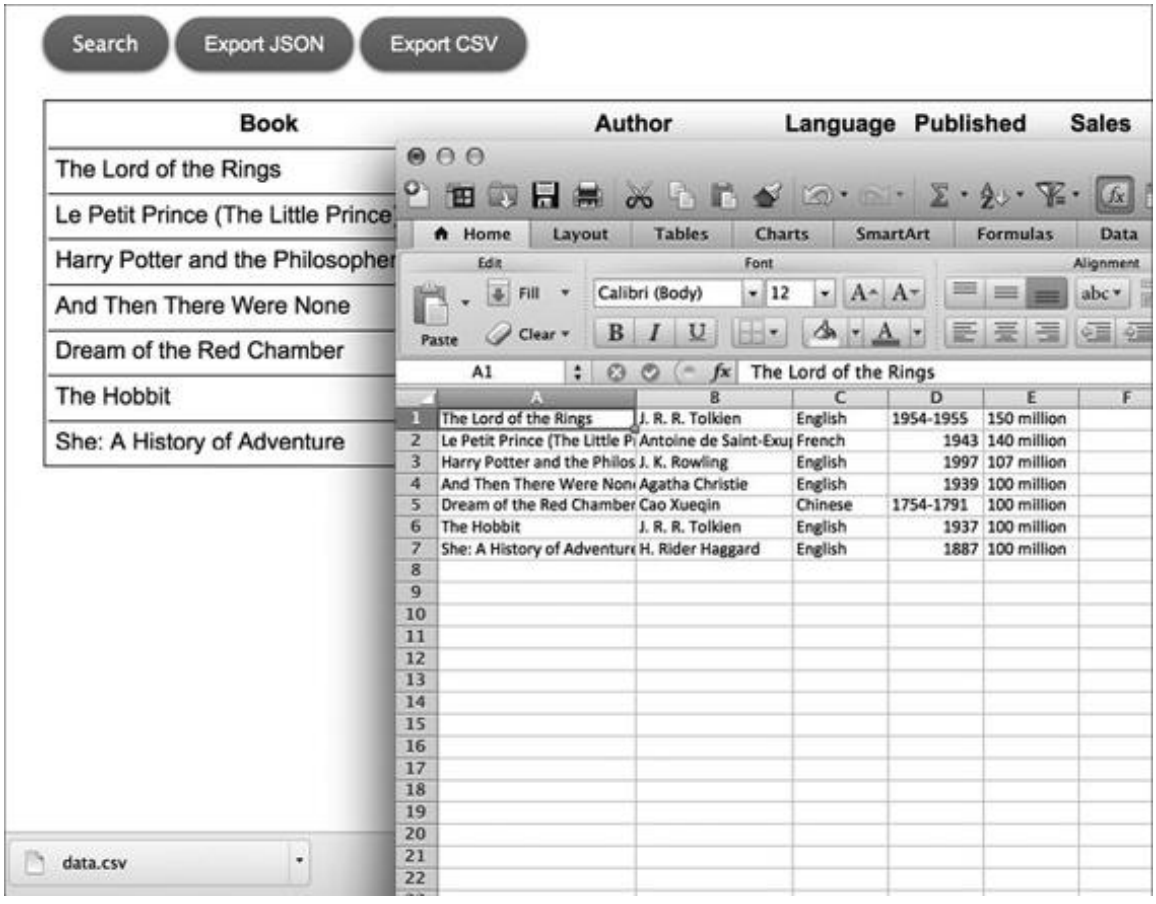


Figura 3.13 – Exportando dados da tabela para o Microsoft Excel usando CSV.

A primeira tarefa é adicionar novas opções na barra de ferramentas. Vamos usar um pouco da mágica de HTML5 que force links <a> a disparar downloads de arquivos, portanto os novos “botões” devem ser links disfarçados de botões com um pouco de CSS:

```
  _renderToolbar: function() {
    return React.DOM.div({className: 'toolbar'},
      React.DOM.button({
        onClick: this._toggleSearch
      }, 'Search'),
      React.DOM.a({
        onClick: this._download.bind(this, 'json'),
        href: 'data.json'
      }, 'Export JSON'),
      React.DOM.a({
        onClick: this._download.bind(this, 'csv'),
        href: 'data.csv'
      }, 'Export CSV')
    );
  },
```

Agora vamos ver a função _download(). Embora exportar para JSON seja trivial, o CSV exige um pouco mais de trabalho. Essencialmente, é apenas um laço que percorre todas as linhas e todas as células de uma linha, gerando uma string longa. Feito isso, a função inicia os downloads por meio do atributo download e o blob href criado por window.URL:

```
  _download: function(format, ev) {
    var contents = format === 'json'
      ? JSON.stringify(this.state.data)
      : this.state.data.reduce(function(result, row) {
        return result
          + row.reduce(function(rowresult, cell, idx) {
            return rowresult
              + '"'
              + cell.replace(/"/g, '""')
              + '"'
              + (idx < row.length - 1 ? ',' : '');
            }, '')
          + "\n";
```

```
        }, '');
    var URL = window.URL || window.webkitURL;
    var blob = new Blob([contents], {type: 'text/' + format});
    ev.target.href = URL.createObjectURL(blob);
    ev.target.download = 'data.' + format;
},
```

CAPÍTULO 4

JSX

Até agora no livro vimos como suas interfaces de usuário são definidas nas funções render() usando chamadas a React.createElement() e à família React.DOM.* (por exemplo, React.DOM.span()). Uma inconveniência com essas muitas chamadas de função é que passa a ser um pouco difícil manter o controle de todos os parênteses e chaves que devem ser fechados. Há uma maneira mais simples: usar o JSX.

O JSX é uma tecnologia distinta da React e é totalmente opcional. Como podemos ver, os três primeiros capítulos nem sequer usaram o JSX. Você pode optar por não o usar de forma alguma. Porém, é muito provável que depois de experimentá-lo, você não voltará para as chamadas de função.

O significado do acrônimo JSX não é muito claro, mas provavelmente é JavaScriptXML ou JavaScript Syntax eXtension (Extensão da Sintaxe JavaScript). O repositório oficial do projeto de código aberto está em *http://facebook.github.io/jsx/*.

Hello JSX

Vamos rever o último exemplo “Hello World” do Capítulo 1:

```
<script src="react/build/react.js"></script>
<script src="react/build/react-dom.js"></script>
<script>
  ReactDOM.render(
    React.DOM.h1(
      {id: "my-heading"},
      React.DOM.span(null,
        React.DOM.em(null, "Hell"),
```

```
            "o"
          ),
          " world!"
        ),
        document.getElementById('app')
      );
    </script>
```

Há muitas chamadas de função na função `render()`. O uso de JSX a simplifica:

```
ReactDOM.render(
  <h1 id="my-heading">
    <span><em>Hell</em>o</span> world!
  </h1>,
  document.getElementById('app')
);
```

Essa sintaxe parece exatamente com HTML, que você já conhece. A única questão é que, como não é uma sintaxe JavaScript válida, ela não pode ser executada no navegador da forma como está. Você precisa transformar (*transpile*, ou transpilar) esse código em JavaScript puro, possível de ser executado pelo navegador.

Transpilação do JSX

A transpilação é o processo de reescrever um código-fonte para obter o mesmo resultado, porém usando uma sintaxe que seja compreendida por navegadores mais antigos. É diferente de usar *polyfills*.

Um exemplo de um polyfill é a adição de um método em `Array.prototype`, por exemplo, `map()`, que foi introduzido no ECMAScript5, e fazê-lo funcionar em navegadores que aceitem ECMAScript3, assim:

```
if (!Array.prototype.map) {
  Array.prototype.map = function() {
    // implementa o método
  };
}
```

```
// uso
typeof [].map === 'function'; // true, `map()` agora pode ser usado
```

Um polyfill é uma solução no domínio de JavaScript puro. É uma boa solução quando adicionamos novos métodos a objetos existentes ou implementamos novos objetos (como JSON). Porém, não é suficiente quando uma nova sintaxe é introduzida na linguagem. Uma nova sintaxe, por exemplo, fazer a palavra reservada `class` funcionar, é apenas uma sintaxe inválida que gera um erro de parse em navegadores sem suporte a `class`, e não há uma maneira de resolver isso com polyfill. Desse modo, uma sintaxe nova exige um passo de compilação (transpilação) para que ela seja transformada *antes* de ser servida ao navegador.

Fazer a transpilação de JavaScript está se tornando cada vez mais comum, pois os programadores desejam usar recursos do ECMAScript6 (também conhecido como ECMAScript2015), além de outros, mas não querem esperar que os navegadores ofereçam suporte a eles. Se você já tem um processo de construção (build) configurado (que faça, por exemplo, minificação ou transpilação de ECMAScript6 para ECMAScript5), você pode simplesmente acrescentar o passo de transformação de JSX nesse processo. Entretanto, vamos supor que você não tenha um processo de construção e vá executar os passos para configurar um processo leve do lado cliente.

Babel

O Babel (anteriormente conhecido como *6to5*) é um transpilador de código aberto que aceita os recursos mais recentes de JavaScript, além de incluir suporte para JSX. É um pré-requisito para usar o JSX. No próximo capítulo, você verá como configurar um processo de construção que permita lançar aplicações React para usuários do mundo real. Porém, para esta discussão sobre JSX, vamos simplificar e fazer a transpilação no lado cliente.

> Aviso obrigatório: transformações do lado cliente são apenas para prototipagem, aprendizado e exploração. Por questões de desempenho, elas não devem ser

usadas em aplicações do mundo real.

Para transformações no navegador (lado cliente), você precisa de um arquivo chamado *browser.js*. O Babel não o disponibiliza mais desde a versão 6, mas você sempre pode obter a cópia funcional mais recente:

```
$ mkdir ~/reactbook/babel
$ cd ~/reactbook/babel
$ curl https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/browser.js >
browser.js
```

Antes da v0.14, a React incluía um script JSXTransformer do lado cliente. Além disso, o pacote NPM react-tools instalava um utilitário jsx de linha de comando nas versões anteriores. Eles se tornaram obsoletos em favor do Babel.

Lado do cliente

Há duas tarefas que você deve executar em sua página para fazer o JSX funcionar:

- incluir o *browser.js*, que é o script capaz de fazer a transpilação de JSX;
- marcar as tags de script que usam JSX para que o Babel saiba que tem trabalho a fazer.

Todos os exemplos do livro até agora incluíram a biblioteca da React desta maneira:

```
<script src="react/build/react.js"></script>
<script src="react/build/react-dom.js"></script>
```

Além delas, você deve incluir agora o transformador:

```
<script src="react/build/react.js"></script>
<script src="react/build/react-dom.js"></script>
<script src="babel/browser.js"></script>
```

O segundo passo é adicionar `text/babel` (que não é aceito pelos navegadores) como um atributo `type` nas tags `<script>` que exijam transformação.

- **Antes**:

```
<script>
  ReactDOM.render(/*...*/);
</script>
```

- **Depois**:

```
<script type="text/babel">
  ReactDOM.render(/*...*/);
</script>
```

Ao carregar a página, *browser.js* é disparado, encontra todos os scripts `text/jsx` e transforma seu conteúdo em algo que os navegadores *possam usar.* A Figura 4.1 mostra o que acontece no Chrome quando você tenta executar um script com a sintaxe JSX diretamente. Você obterá um erro de sintaxe, exatamente conforme esperado. Na Figura 4.2, podemos ver que a página funciona bem depois que *browser.js* transpila os blocos de script com `type="text/babel"`.

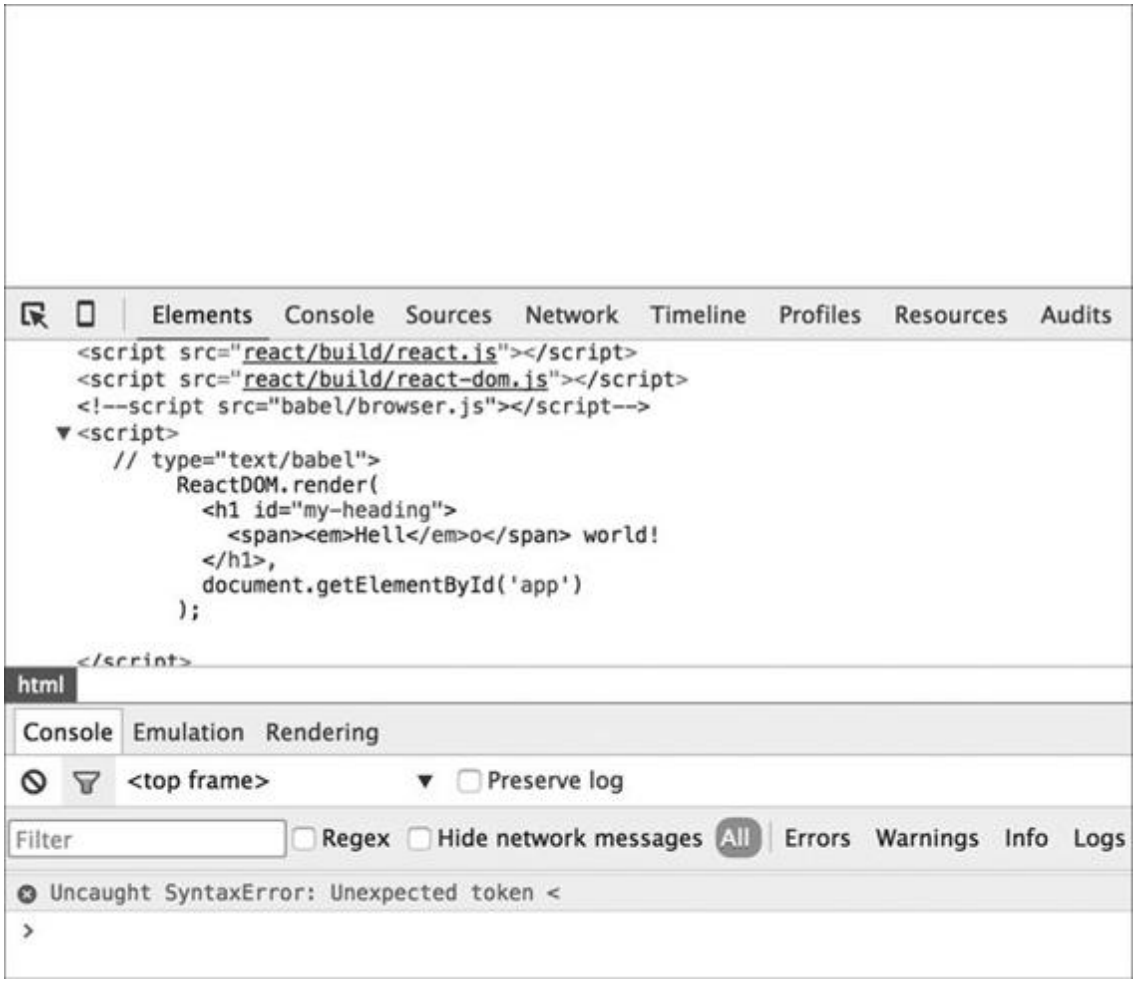


Figura 4.1 – Os navegadores não compreendem a sintaxe JSX.

Figura 4.2 – Script browser do Babel e tipo de conteúdo text/babel.

Sobre a transformação de JSX

Para testar e ter familiaridade com as transformações de JSX, você pode interagir com o editor online em *https://babel.js.io/repl/* (Figura 4.3).

Como você pode ver na Figura 4.4, a transformação de JSX é leve e simples: o código-fonte de JSX para “Hello World” se transforma em uma série de chamadas para React.createElement(), usando a mesma sintaxe de funções que você já conhece. É apenas JavaScript, portanto é fácil de ler e de entender.

Figura 4.3 – Ferramenta online de transformação de JSX.

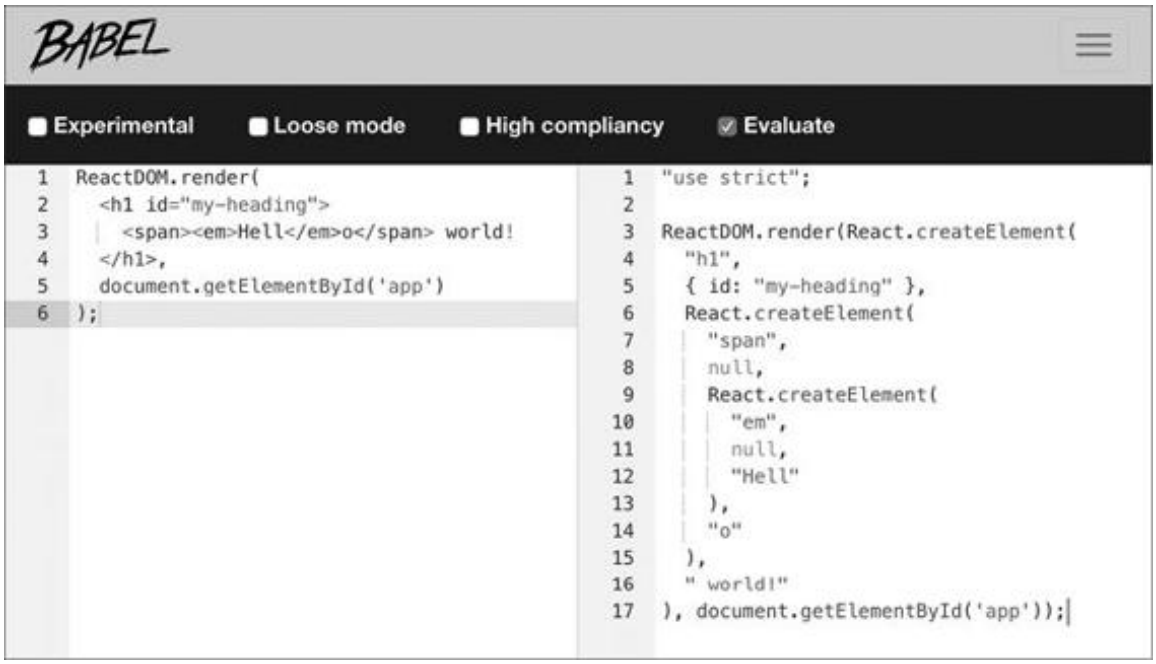


Figura 4.4 – "Hello World" transformado.

Há outra ferramenta online que talvez você ache útil quando estiver aprendendo JSX ou fazendo a transição da marcação de uma aplicação existente a partir de HTML: um transformador de HTML para JSX (*https://facebook.github.io/react/html-jsx.html*) – veja a Figura 4.5.

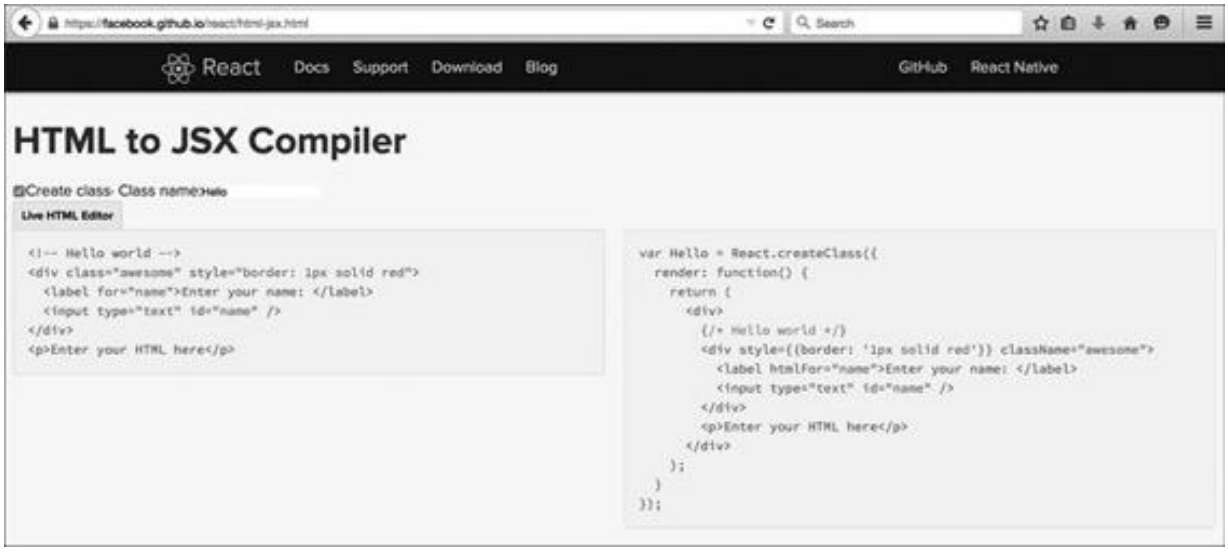


Figura 4.5 – Ferramenta de HTML para JSX.

JavaScript em JSX

Quando construir uma UI, com frequência você precisará usar variáveis, condições e laços. Em vez de criar outra sintaxe de templating, o JSX permite escrever JavaScript dentro da marcação. Tudo que você deve fazer é colocar o seu código JavaScript entre chaves.

Considere, por exemplo, um dos exemplos com Excel do capítulo anterior. Para substituir a sintaxe de funções por JSX, você acabará com algo semelhante a:

```
var Excel = React.createClass({
  /* trecho omitido... */
  render: function() {
    var state = this.state;
    return (
      <table>
        <thead onClick={this._sort}>
          <tr>
            {this.props.headers.map(function(title, idx) {
              if (state.sortby === idx) {
                title += state.descending ? ' \u2191' : ' \u2193';
              }
```

```
            return <th key={idx}>{title}</th>;
          })}
        </tr>
      </thead>
      <tbody>
        {state.data.map(function(row, idx) {
          return (
            <tr key={idx}>
              {row.map(function(cell, idx) {
                return <td key={idx}>{cell}</td>;
              })}
            </tr>
          );
        })}
      </tbody>
    </table>
  );
 }
});
```

Como podemos ver, para usar variáveis, devemos colocá-las entre chaves:

```
<th key={idx}>{title}</th>
```

Para laços, podemos colocar as chamadas a `map()` entre chaves também:

```
<tr key={idx}>
  {row.map(function(cell, idx) {
    return <td key={idx}>{cell}</td>;
  })}
</tr>
```

Você pode ter JSX em JavaScript em JSX aninhados tão profundamente quanto for necessário. Podemos pensar no JSX como JavaScript (após uma transformação leve), porém com uma sintaxe HTML familiar. Até mesmo os membros de sua equipe que não sejam tão versados em JavaScript quanto você, mas que conheçam HTML, poderão escrever JSX. Além disso, eles poderão aprender JavaScript apenas o suficiente para usar variáveis e laços e construir a UI com dados ativos.

No exemplo com Excel que acabamos de mostrar, há uma condição if em uma callback map(). Embora seja uma condição aninhada, com um pouco de ajuda da formatação, você pode transformá-la em um único ternário legível:

```
return (
  <th key={idx}>{
    state.sortby === idx
      ? state.descending
        ? title + ' \u2191'
        : title + ' \u2193'
      : title
  }</th>
);
```

Você percebeu o title repetido nesse último exemplo? Podemos nos livrar dele assim:

```
return (
  <th key={idx}>{title}{
    state.sortby === idx
      ? state.descending
        ? ' \u2191'
        : ' \u2193'
      : null
  }</th>
);
```

Contudo, nesse caso, você deve modificar a função de ordenação do exemplo. A função de ordenação supõe que uma pessoa clica em um <th> e utiliza cellIndex para descobrir qual é o <th>. Porém, quando há blocos {} adjacentes no JSX, teremos tags para diferenciá-los. Em outras palavras, <th>{1}{2}</th> aparece no DOM como <th>12</th>.

Espaço em branco no JSX

Espaço em branco no JSX é semelhante ao espaço no HTML, mas nem tanto:

```
<h1>
  {1} plus {2} is {3}
</h1>
```

…resulta em

```
<h1>
  <span>1</span><span> plus </span><span>2</span><span> is </span><span>3</
span>
</h1>
```

…que é renderizado como “1 plus 2 is 3”, exatamente como você esperaria em HTML: vários espaços se transformam em um.

Todavia, neste exemplo:

```
<h1>
  {1}
  plus
  {2}
  is
  {3}
</h1>
```

…você acabará com

```
<h1>
  <span>1</span><span>plus</span><span>2</span><span>is</span>
<span>3</span>
</h1>
```

Como podemos ver, todos os espaços em branco são removidos e o resultado final é “1plus2is3”.

Você sempre pode acrescentar espaços nos locais em que julgar necessário com `{' '}` (que gera mais tags `<span>`) ou transformar as strings literais em expressões e adicionar aí os espaços. Em outras palavras, qualquer uma das soluções a seguir funcionará:

```
<h1>
  {/* expressões de espaço */}
  {1}
  {' '}plus{' '}
  {2}
  {' '}is{' '}
  {3}
</h1>
```

```
<h1>
{/* espaços ligados a expressões de string */}
  {1}
```

```
    {' plus '}
    {2}
    {' is '}
    {3}
  </h1>
```

Comentários no JSX

Nos exemplos anteriores, podemos ver como um novo conceito se infiltrou – a adição de comentários na marcação JSX.

Como as expressões entre {} são apenas JavaScript, você pode facilmente acrescentar comentários de várias linhas usando /* comentário */. Você também pode acrescentar comentários de uma só linha usando // comentário, mas deve garantir que a chave de fechamento } da expressão esteja em uma linha separada para que ela não seja considerada parte do comentário:

```
  <h1>
    {/* comentário multilinha */}
    {/*
      comentário
      multi
      linha
      */}
    {
      // única linha
    }
    Hello
  </h1>
```

Como {// comentário} não funciona (} agora está comentado), há pouca vantagem em usar comentários de uma só linha, e você pode manter seus comentários consistentes e ater-se aos comentários multilinhas em todos os casos.

Entidades HTML

Você pode usar entidades HTML no JSX assim:

```
  <h2>
```

```
    More info &raquo;
  </h2>
```

Esse exemplo gera “aspas angulares voltadas para a direita” como mostra a Figura 4.6.

Figura 4.6 – Entidade HTML no JSX.

No entanto, se você usar a entidade como parte de uma expressão, terá problemas de codificação dupla. Neste exemplo…

```
  <h2>
    {"More info &raquo;"}
  </h2>
```

…o HTML será codificado e você verá o resultado mostrado na Figura 4.7.

Figura 4.7 – Entidade HTML com codificação dupla.

Para evitar a codificação dupla, podemos usar a versão Unicode da entidade HTML, que, nesse caso, é \u00bb (veja *http://dev.w3.org/html5/html-author/charref*):

```
  <h2>
    {"More info \u00bb"}
  </h2>
```

Por questões de conveniência, você pode definir uma constante em algum lugar no início de seu módulo, juntamente com qualquer espaçamento usual. Por exemplo:

```
  const RAQUO = ' \u00bb';
```

Então utilize a constante conveniente em qualquer lugar em que ela for necessária, desta maneira:

```
<h2>
  {"More info" + RAQUO}
</h2>
<h2>
  {"More info"}{RAQUO}
</h2>
```

Você percebeu o uso de const no lugar de var? Bem-vindo ao admirável mundo novo de Babel, em que você pode dispor de tudo que o JavaScript moderno tem para oferecer. Veja os detalhes no Capítulo 5.

AntiXSS

Talvez você esteja se perguntando por que precisa fazer um bocado de ginástica para usar entidades HTML. Há um bom motivo que justifica as desvantagens: você precisa lutar contra o *XSS*.

A React escapa todas as strings a fim de evitar uma classe de ataques XSS. Desse modo, quando você pedir aos usuários que forneçam uma entrada e eles oferecerem uma string maliciosa, a React protegerá você. Considere esta entrada de usuário como exemplo:

```
var firstname = 'John<scr'+'ipt src="http://evil/co.js"></scr'+'ipt>';
```

Em algumas circunstâncias, você poderia acabar escrevendo isso no DOM. Por exemplo:

```
document.write(firstname);
```

Isso é um desastre, pois a página diz “John”, mas a tag `<script>` carrega um JavaScript malicioso e compromete a sua aplicação e os usuários que confiam em você.

A React protege você de casos como esse automaticamente, se usar:

```
React.render(
  <h2>
    Hello {firstname}!
  </h2>,
  document.getElementById('app')
);
```

…a React escapará o conteúdo de firstname (Figura 4.8).

Figura 4.8 – Escapando strings.

Propagação de atributos

O JSX empresta uma funcionalidade útil do ECMAScript6 chamada *operador de propagação* (spread operator), e o adota como uma conveniência na definição de propriedades.

Suponha que você tenha uma coleção de atributos que queira passar para um componente <a>:

```
var attr = {
  href: 'http://example.org',
  target: '_blank',
};
```

Você sempre pode fazer o seguinte:

```
return (
  <a
    href={attr.href}
    target={attr.target}>
    Hello
  </a>
);
```

Contudo, parece haver muito código boilerplate aqui. Ao usar propagação de atributos, você pode fazer isso usando apenas uma linha:

```
return <a {...attr}>Hello</a>;
```

No exemplo, você tem um objeto de atributos que deseja definir (talvez condicionalmente) com antecedência. Isso, por si só, é útil, porém um uso mais comum ocorre quando você obtém esse objeto de atributos de fora – geralmente, de um componente-pai. Vamos ver como esse caso se desenrola.

Propagação de atributos de pai para filho

Suponha que você esteja construindo um componente FancyLink que utilize um <a> comum internamente. Você quer que seu componente aceite todos os atributos que <a> aceita (href, style, target etc.), além de outros (por exemplo, size). Desse modo, as pessoas podem usar o seu componente assim:

```
<FancyLink
  href="http://example.org"
  style={ {color: "red"} }
  target="_blank"
  size="medium">
  Hello
</FancyLink>
```

Como a sua função render() pode tirar proveito da propagação de atributos e evitar a necessidade de redefinir todas as propriedades de <a>?

```
var FancyLink = React.createClass({
  render: function() {
    switch(this.props.size) {
      // faz algo com base na propriedade `size`
    }
    return <a {...this.props}>{this.props.children}</a>;
  }
});
```

Você percebeu o uso de this.props.children? Esse é um método simples e conveniente para permitir que qualquer número de filhos seja passado para o seu componente e acessá-los quando estiver compondo a sua interface.

No trecho de código anterior, você executa a sua tarefa personalizada com base no valor da propriedade size e, em seguida, passa todas as propriedades para <a>. Elas incluem a propriedade size. React.DOM.a não tem nenhum conceito de size, portanto ele silenciosamente ignora essa propridade, porém usa todas as demais.

Podemos fazer algo um pouco melhor e deixar de passar as propriedades desnecessárias implementando algo como:

```
var FancyLink = React.createClass({
  render: function() {
    switch(this.props.size) {
      // faz algo com base na propriedade `size`
    }
    var attribs = Object.assign({}, this.props); // clone raso
    delete attribs.size;
    return <a {...attribs}>{this.props.children}</a>;
  }
});
```

Se usarmos a sintaxe proposta pelo ECMAScript7 (disponibilizada a você gratuitamente pelo Babel!), isso se tornará mais fácil ainda, sem qualquer clonagem:

```
var FancyLink = React.createClass({
  render: function() {
    var {size, ...attribs} = this.props;
    switch (size) {
      // faz algo com base na propriedade `size`
    }
    return <a {...attribs}>{this.props.children}</a>;
  }
});
```

Devolvendo vários nós no JSX

Sua função render() sempre deve devolver um único nó. Devolver dois nós não é permitido. Em outras palavras, isto é um erro:

```
// Erro de sintaxe:
// Elementos JSX adjacentes devem estar encapsulados por uma tag
var Example = React.createClass({
  render: function() {
    return (
      <span>
        Hello
      </span>
      <span>
        World
```

```
        </span>
      );
    }
  });
```

A correção é simples – basta colocar todos os nós em outro componente, por exemplo, em uma :

```
  var Example = React.createClass({
    render: function() {
      return (
        <div>
          <span>
            Hello
          </span>
          <span>
            World
          </span>
        </div>
      );
    }
  });
```

Embora sua função render() não possa devolver um array de nós, você pode usar arrays durante a composição, desde que os nós do array tenham atributos key apropriados:

```
  var Example = React.createClass({
    render: function() {
      var greeting = [
        <span key="greet">Hello</span>,
        ' ',
        <span key="world">World</span>,
        '!'
      ];
      return (
        <div>
          {greeting}
        </div>
      );
    }
  });
```

Observe como você também pode inserir espaços em branco e outras strings no array, e esses não precisam de `key`.

De certa forma, isso é semelhante a aceitar qualquer quantidade de filhos passados pelo pai e propagá-los em sua função `render()`:

```
var Example = React.createClass({
  render: function() {
    console.log(this.props.children.length); // 4
    return (
      <div>
        {this.props.children}
      </div>
    );
  }
});
React.render(
  <Example>
    <span key="greet">Hello</span>
    {' '}
    <span key="world">World</span>
    !
  </Example>,
  document.getElementById('app')
);
```

Diferenças entre JSX e HTML

O JSX deve parecer bastante familiar – é exatamente como HTML, mas com a vantagem de ter uma maneira fácil de adicionar valores dinâmicos, laços e condições (basta colocá-los entre `{}`). Para começar a trabalhar com o JSX, você sempre pode usar a ferramenta de HTML para JSX (*https://facebook.github.io/react/html-jsx.html*), mas quanto antes começar a digitar o seu próprio JSX, melhor. Vamos considerar as poucas diferenças entre HTML e JSX que podem deixar você surpreso no início, quando estiver começando a conhecê-lo.

Algumas dessas diferenças foram descritas no Capítulo 1, mas

vamos revê-las rapidamente.

Sem class nem for

Em vez dos atributos `class` e `for` (ambas são palavras reservadas no ECMAScript), você deve usar `className` e `htmlFor`:

```
// Não, não!
var em = <em class="important" />;
var label = <label for="thatInput" />;

// OK
var em = <em className="important" />;
var label = <label htmlFor="thatInput" />;
```

style é um objeto

O atributo `style` aceita um objeto, e não uma string separada por ponto e vírgula. Além disso, os nomes das propriedades CSS são `camelCase`, e não delimitadas-por-traço:

```
// NÃO, NÃO!
var em = <em style="font-size: 2em; line-height: 1.6" />;

// OK
var styles = {
  fontSize: '2em',
  lineHeight: '1.6'
};
var em = <em style={styles} />;

// inline também está OK
// observe as chaves duplas { {} } – uma para o valor dinâmico no JSX e
// outra para o objeto JS
var em = <em style={ {fontSize: '2em', lineHeight: '1.6'} } />;
```

Tags de fechamento

Em HTML, algumas tags não precisam ser fechadas; em JSX (XML), isso é necessário:

```
// NÃO, NÃO
// não deixe tags sem fechamento, mesmo que isso seja permitido em HTML
```

```
var gimmeabreak = <br>;
var list = <ul><li>item</ul>;
var meta = <meta charset="utf-8">;

// OK
var gimmeabreak = <br />;
var list = <ul><li>item</li></ul>;
var meta = <meta charSet="utf-8" />;

// ou
var meta = <meta charSet="utf-8"></meta>;
```

Atributos camelCase

Você percebeu charset *versus* charSet no trecho de código anterior? Todos os atributos no JSX devem ser camelCase. Essa é uma fonte comum de confusão no início – pode acontecer de você digitar onclick e perceber que nada acontece até retornar e alterá-lo para onClick:

```
// Não, não!
var a = <a onclick="reticulateSplines()" />;

// OK
var a = <a onClick={reticulateSplines} />;
```

Exceções a essa regra são todos os atributos prefixados com data- e aria-; eles são exatamente como no HTML.

JSX e formulários

Há algumas diferenças entre JSX e HTML quando trabalhamos com formulários. Vamos dar uma olhada.

Handler onChange

Ao usar elementos de formulário, os usuários mudam seus valores quando interagem com eles. Na React, você pode se registrar para ser informado dessas mudanças usando o atributo onChange. Isso é muito mais consistente que usar o valor checked de botões de rádio e caixas de seleção, e selected em opções <select>. Além disso, quando houver digitação em textareas e em campos <input type="text">, onChange

dispara à medida que o usuário digitar, o que é muito mais útil que disparar quando o elemento perde o foco. Isso significa que não é mais necessário se registrar para receber todo tipo de eventos de mouse e de teclado apenas para monitor mudanças na digitação.

value *versus* defaultValue

Em HTML, se você tiver <input id="i" value="hello" /> e então alterar o valor digitando “bye”, então…

```
i.value; // "bye"
i.getAttribute('value'); // "hello"
```

Na React, a propriedade value sempre tem o conteúdo atualizado da entrada de texto. Se quiser especificar um default, você pode usar defaultValue.

No trecho de código a seguir, temos um componente <input> com um conteúdo “hello” previamente preenchido e o handler onChange. Apagar o último “o” em “hello” resulta em value igual a “hell”, mas defaultValue continua sendo “hello”:

```
function log(event) {
  console.log("value: ", event.target.value);
  console.log("defaultValue: ", event.target.defaultValue);
}
React.render(
  <input defaultValue="hello" onChange={log} />,
  document.getElementById('app')
);
```

Esse é um padrão que você deve usar em seus próprios componentes: se você aceitar uma propriedade que dê indícios de que deva estar atualizado (por exemplo, value, data), então mantenha-o assim. Caso contrário, chame-o de initialData (como vimos no Capítulo 3), defaultValue ou algo semelhante para atender devidamente às expectativas.

Valor de <textarea>

Por questões de consistência com as entradas de texto, a versão da React de <textarea> aceita as propriedades value e defaultValue. Ela

mantém value atualizado, enquanto defaultValue permanece com o valor original. Se você usar o estilo HTML e utilizar um filho de textarea para definir um valor (não recomendado), ele será tratado como se fosse um defaultValue.

O motivo pelo qual a <textarea> de HTML (conforme definida pelo W3C) aceita um filho como valor é para que os desenvolvedores possam usar quebras de linha na entrada. No entanto, a React, pelo fato de ser totalmente JavaScript, não sofre dessa limitação. Quando precisar de uma quebra de linha, basta usar \n.

Considere os exemplos a seguir e seus resultados mostrados na Figura 4.9:

```
function log(event) {
  console.log(event.target.value);
  console.log(event.target.defaultValue);
}
React.render(
  <textarea defaultValue="hello\nworld" onChange={log} />,
  document.getElementById('app1')
);
React.render(
  <textarea defaultValue={"hello\nworld"} onChange={log} />,
  document.getElementById('app2')
);
React.render(
  <textarea onChange={log}>hello
world
  </textarea>,
  document.getElementById('app3')
);
React.render(
  <textarea onChange={log}>{"hello\n\
world"}
  </textarea>,
  document.getElementById('app4')
);
```

Figura 4.9 – Quebras de linha em textareas.

Observe as diferenças entre usar uma string literal "hello\nworld" como um valor de propriedade *versus* usar a string JavaScript {"hello\nworld"}.

Além disso, observe como uma string multilinha em JavaScript precisa ser escapada com um \ (quarto exemplo).

Por fim, veja como a React avisa você a respeito do uso do filho de <textarea> no modo tradicional para definir o valor.

Valor de <select>

Quando usar uma entrada <select> em HTML, você deve especificar entradas previamente selecionadas usando <option selected>, assim:

```
<!-- HTML tradicional -->
<select>
  <option value="stay">Should I stay</option>
  <option value="move" selected>or should I go</option>
</select>
```

Na React, você deve especificar value ou, melhor ainda, defaultValue, no elemento <select>:

```
// React/JSX
<select defaultValue="move">
  <option value="stay">Should I stay</option>
  <option value="move">or should I go</option>
```

```
</select>
```

O mesmo se aplica quando você tiver uma seleção múltipla; nesse caso, você deve fornecer um array de valores previamente selecionados:

```
<select defaultValue={["stay", "move"]} multiple={true}>
  <option value="stay">Should I stay</option>
  <option value="move">or should I go</option>
  <option value="trouble">If I stay it will be trouble</option>
</select>
```

A React avisará se você se confundir e definir o atributo selected de uma <option>.

Usar <select value> em vez de <select defaultValue> também é permitido, embora não seja recomendado, pois isso exigirá que você cuide de atualizar o valor que o usuário vê. Caso contrário, quando o usuário selecionar uma opção diferente, <select> permanecerá inalterado. Em outras palavras, você precisa de um código como:

```
var MySelect = React.createClass({
  getInitialState: function() {
    return {value: 'move'};
  },
  _onChange: function(event) {
    this.setState({value: event.target.value});
  },
  render: function() {
    return (
      <select value={this.state.value} onChange={this._onChange}>
        <option value="stay">Should I stay</option>
        <option value="move">or should I go</option>
        <option value="trouble">If I stay it will be trouble</option>
      </select>
    );
  }
});
```

O componente Excel em JSX

Para concluir, vamos usar o JSX e reescrever todos os métodos

`render*()` da versão final do componente `Excel` do capítulo anterior. Deixarei esse exercício para você se divertir, mas você poderá comparar a sua solução com o exemplo que está no repositório de código que acompanha este livro (*https://github.com/stoyan/reactbook/*).

CAPÍTULO 5

Preparando-se para o desenvolvimento de aplicações

Para qualquer desenvolvimento e implantação sérios, exceto prototipagem ou testes de JSX, você deve configurar um processo de construção (build). Se você já tem um processo definido, será necessário apenas acrescentar-lhe a transformação com Babel. Porém, vamos supor que você não tenha nenhum tipo de configuração de construção e que está começando do zero.

O objetivo é usar o JSX e qualquer outro JavaScript moderno sem esperar que os navegadores implementem esses recursos. Você deve configurar uma transformação que execute em segundo plano, enquanto você está desenvolvendo. O processo de transformação deve gerar um código que seja o máximo possível semelhante ao código que seus usuários executarão no site ao vivo (o que significa que não haverá mais transformações do lado cliente). O processo também deve ser o mais suave possível para que você não precise alternar entre contextos de desenvolvimento e de construção.

A comunidade e o ecossistema de JavaScript oferecem muitas opções quando se trata de processos de desenvolvimento e de construção. Vamos manter a construção enxuta e em um nível baixo, sem usar qualquer ferramenta, criando uma abordagem do tipo faça você mesmo, de modo que você possa:

- entender o que está acontecendo;
- fazer uma opção bem fundamentada no futuro, quando escolher suas ferramentas de construção;

• concentrar-se no lado da React, sem se distrair demais.

Uma aplicação boilerplate

Vamos começar definindo um “template” genérico para uma nova aplicação. É uma aplicação do lado cliente, no estilo SPA (Single-Page Application, ou Aplicação de Página Única). A aplicação utiliza JSX assim como muitas inovações que a linguagem JavaScript tem a oferecer: ES5, ES6 (também conhecido como ES2015) e propostas do futuro ES7.

Arquivos e pastas

Conforme a prática comum, você precisa de pastas */css*, */js* e */images* e um *index.html* para ligar tudo. Vamos separar também */js* em */js/source* (scripts com sintaxe JSX) e em */js/build* (scripts amigáveis ao navegador). Além disso, há uma categoria */scripts* que hospeda scripts de linha de comando para fazer a construção.

A estrutura de diretórios de sua aplicação template (boilerplate) pode ter um aspecto semelhante ao da Figura 5.1.

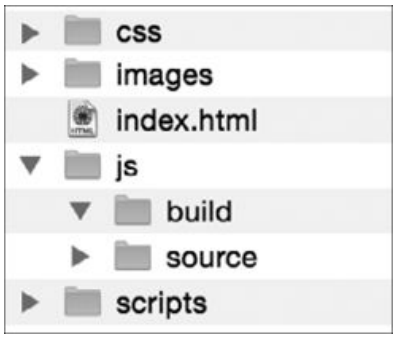


Figura 5.1 – Aplicação boilerplate.

Vamos separar mais ainda os diretórios */css* e */js* (Figura 5.2) de modo a incluir:

• arquivos genéricos usados por todo o aplicativo;

• arquivos associados a componentes específicos.

Isso ajuda a deixar os componentes independentes, com propósito

único e reutilizáveis o máximo possível. Afinal de contas, você quer construir sua aplicação grande usando vários componentes menores com seus propósitos específicos. Dividir e conquistar.

Por fim, vamos criar um componente simples de exemplo chamado <Logo> (as aplicações tendem a ter logos). Uma convenção comum consiste em usar nomes começados com letra maiúscula para componentes – daí “Logo”, e não “logo”. Agora que todos os arquivos relacionados a componentes estão consistentes, vamos convencionar que usaremos */js/source/components/Component.js* para implementar o componente e */css/components/Component.css* para a estilização associada. A Figura 5.2 mostra a estrutura de diretórios completa, com o componente simples <Logo>.

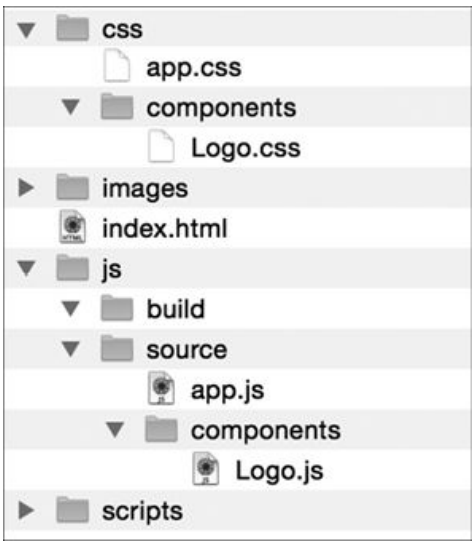


Figura 5.2 – Componentes separados.

index.html

Com a estrutura de diretórios definida, vamos ver como podemos fazer tudo funcionar em um estilo “Hello World”. O arquivo *index.html* deve incluir:

- todo o CSS em um único arquivo *bundle.css*;
- todo o JavaScript em um único *bundle.js* (inclui sua aplicação, assim como todos os seus componentes e suas dependências

de biblioteca, incluindo React);

- como sempre, um lugar `<div id="app">` para sua aplicação ser inserida:

```
<!DOCTYPE html>
<html>
  <head>
    <title>App</title>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css" href="bundle.css">
  </head>
  <body>
    <div id="app"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

.css e *.js* únicos são surpreendentemente eficazes para uma grande variedade de aplicações. À medida que sua aplicação crescer até atingir proporções como do Facebook ou do Twitter, esses scripts podem ficar grandes demais para uma carga prévia inicial, e, de qualquer maneira, o usuário não precisará de todas as funcionalidades de imediato. Você criará um carregador de scripts/estilos para que possa carregar mais código à medida que o usuário exigir (isso ficará como exercício para você e, novamente, você tem inúmeras opções de código aberto). Nesse cenário, *.css* e *.js* únicos passam a ser uma espécie de arquivos de *bootstrapping* que, no mínimo, podem apresentar algo ao usuário assim que for possível. Portanto, o padrão de único arquivo ainda tem o seu lugar, mesmo que a aplicação cresça.

Você verá como gerar *bundle.js* e *bundle.css* a partir de arquivos individuais logo a seguir. Antes disso, vamos considerar quais códigos CSS/JS devem ser colocados em quais lugares.

CSS

O */css/app.css* global deve conter estilos comuns, usados em toda a aplicação, portanto terá a seguinte aparência:

```
html {
  background: white;
  font: 16px Arial;
}
```

Além dos estilos usados em toda a aplicação, você precisa de estilos específicos para cada componente. De acordo com a convenção de ter um arquivo CSS (e um arquivo JS) por componente React e tê-los em */css/components* (e em */js/source/components*), vamos implementar */css/components/Logo.css* assim:

```
.Logo {
  background-image: url('../../images/react-logo.svg');
  background-size: cover;
  display: inline-block;
  height: 50px;
  vertical-align: middle;
  width: 50px;
}
```

Outra convenção simples, que pode se mostrar útil, é manter os nomes das classes CSS dos componentes iniciados com letra maiúscula e fazer o elemento-raiz do componente ter um nome de classe que coincida com o nome do componente – daí `className="Logo"`.

JavaScript

O script de ponto de entrada da aplicação, */js/source/app.js*, é o local em que tudo começa, portanto vamos usar:

```
React.render(
  <h1>
    <Logo /> Welcome to The App!
  </h1>,
  document.getElementById('app')
);
```

Por fim, implemente o componente React `<Logo>` de exemplo em */js/source/components/Logo.js*:

```
var Logo = React.createClass({
  render: function() {
    return <div className="Logo" />;
  }
```

```
});
```

JavaScript: modernizado

Os exemplos do livro até agora trabalharam apenas com componentes simples e garantiram que `React` e `ReactDOM` estivessem disponíveis como variáveis globais. À medida que você prosseguir em direção a aplicações mais complicadas, com vários componentes, será necessário ter melhor organização. Espalhar variáveis globais por aí é perigoso (elas tendem a causar colisões de nomes) e contar que elas sempre estejam presentes também é perigoso (e se você mudar para um empacotamento diferente de JS, em que *nem tudo* esteja em um único *bundle.js*?).

Você precisa de *módulos.*

Módulos

A comunidade JavaScript criou várias ideias para módulos JavaScript, e uma delas, amplamente adotada, é o *CommonJS*. No CommonJS, o código é colocado em um arquivo que, ao ser executado, exporta um ou mais símbolos (mais comumente um objeto, mas poderia ser uma função ou até mesmo uma única variável):

```
var Logo = React.createClass({/* ... */});
module.exports = Logo;
```

Uma convenção que pode se mostrar útil é: um módulo exporta um item (por exemplo, um componente React).

Esse módulo precisa de `React` para executar `React.createClass()`. Não há mais variáveis globais, portanto `React` não está disponível como um símbolo global. Você deve incluí-la (ou executar `require`) assim:

```
var React = require('react');
var Logo = React.createClass({/* ... */});
module.exports = Logo;
```

Vamos definir esse código como um template para cada

componente: declare as exigências no início, exporte no final e implemente o “cerne” no meio.

Módulos ECMAScript

As especificações ECMAScript propõem levar essa ideia mais adiante e introduzem uma nova sintaxe (em oposição a se resolver com require() e module.exports). Você pode tirar proveito da nova sintaxe, pois o Babel oferece suporte quando se trata da transpilação da nova sintaxe para algo que os navegadores possam aceitar.

Ao declarar dependências em outros módulos, em vez de:

```
var React = require('react');
```

…utilize:

```
import React from 'react';
```

Além disso, quando exportar a partir de seu módulo, em vez de:

```
module.exports = Logo;
```

…utilize:

```
export default Logo
```

A ausência de ponto e vírgula no final de export não é um erro deste livro, mas uma característica do ECMAScript.

Classes

O ECMAScript atualmente tem classes, portanto vamos tirar proveito da nova sintaxe.

- **Antes**:

```
var Logo = React.createClass({/* ... */});
```

- **Depois**:

```
class Logo extends React.Component {/* ... */}
```

Nos locais em que você anteriormente declarava “classes” React com um objeto, agora, com classes reais, há algumas diferenças:

- Não há mais propriedades aleatórias no objeto, mas apenas funções (métodos). Se você precisar de uma propriedade, faça a

sua atribuição em `this` no construtor (mais exemplos e opções a seguir).

- A sintaxe de método é `render() {};` não há mais necessidade da palavra reservada `function`.
- Os métodos não são mais separados com `,` como em `var obj = {a: 1, b: 2};`.

```
class Logo extends React.Component {
  someMethod() {
  } // sem vírgulas
  another() { // sem `function`
  }
  render() {
    return <div className="Logo" />;
  }
}
```

Reunindo tudo

Há mais recursos do ECMAScript a serem apresentados à medida que você avançar no livro, mas isso é suficiente para um boilerplate, apenas para dar início a uma nova aplicação rapidamente e com o mínimo possível.

Você agora tem o *index.html*, o CSS usado por toda a aplicação (*app.css*), um CSS por componente (*/css/components/Logo.css*) e, por fim, o ponto de entrada JavaScript da aplicação (*app.js*), além de cada componente React implementado em um módulo específico (por exemplo, */js/source/components/Logo.js*).

Eis a versão final de *app.js*:

```
'use strict'; // sempre uma boa ideia
import React from 'react';
import ReactDOM from 'react-dom';
import Logo from './components/Logo';
ReactDOM.render(
  <h1>
    <Logo /> Welcome to The App!
```

```
    </h1>,
    document.getElementById('app')
  );
```

…e *Logo.js*:

```
  import React from 'react';

  class Logo extends React.Component {
    render() {
      return <div className="Logo" />;
    }
  }

  export default Logo
```

Você percebeu a diferença entre importar a React e o componente Logo: `from'react'` e `from './components/Logo'`? A última importação parece um path de diretório, e é – você está dizendo ao módulo para obter a dependência especificando a localização do arquivo em relação ao módulo, enquanto a primeira extrai uma dependência de um local compartilhado, instalado por meio de `npm`. Vamos ver como fazer tudo isso funcionar, incluindo toda a nova sintaxe e os módulos, como num passe de mágica, sem falhas, em um navegador (até mesmo nos IEs mais antigos!).

Você pode encontrar essa configuração boilerplate no repositório de código que acompanha este livro (*https://github.com/stoyan/reactbook/*) e usá-la para dar início rapidamente às suas aplicações.

Instalando os pré-requisitos

Antes de poder carregar *index.html* e vê-lo funcionando, você deve:

- criar *bundle.css*. Essa é uma concatenação simples, sem pré-requisitos exigidos;
- deixar seu código compreensível pelos navegadores. Você precisa do Babel para a transpilação;
- criar *bundle.js*. Vamos usar o *Browserify* para isso.

Você precisa do Browserify não só para concatenar scripts, mas também para:

- resolver e incluir todas as dependências. Basta passar-lhe o path para *app.js*, e ele identificará todas as dependências (React, *Logo.js* etc.);
- incluir uma implementação de *CommonJS* para fazer as chamadas a require() funcionarem. O Babel transforma todas as instruções import em chamadas da função require().

Em suma, você deve instalar o Babel e o Browserify. Instale-os usando npm (Node Package Manager, ou Gerenciador de Pacotes do Node) que acompanha o Node.js.

Node.js

Para instalar o Node.js, acesse *http://nodejs.org* e obtenha o instalador para o seu sistema operacional. Siga as instruções do instalador e pronto. Agora você pode dispor dos serviços oferecidos pelo npm.

Para conferir, digite o seguinte em seu terminal:

```
$ npm --version
```

Se você não tiver experiência em usar um terminal (um prompt de comandos), agora é uma boa hora para começar! No Mac OS X, clique na ferramenta de pesquisa Spotlight (o ícone de lente de aumento no canto superior direito) e digite Terminal. No Windows, localize o menu Start (Iniciar) (clique no ícone do Windows no canto inferior esquerdo da tela), selecione Run (Executar) e digite power shell.

Neste livro, todos os comandos que você digitar em seu terminal estão prefixados com $ somente como um indício para diferenciá-los do código comum. Você deve omitir o $ quando digitar em seu terminal.

Browserify

Instale o Browserify usando npm, digitando o seguinte em seu terminal:

```
$ npm install --global browserify
```

Para conferir se ele está funcionando, digite:

```
$ browserify --version
```

Babel

Para instalar a CLI (Command-Line Interface, ou Interface de Linha de Comando) do Babel, digite:

```
$ npm install --global babel-cli
```

Para conferir se ela está funcionando, digite:

```
$ babel --version
```

Você já percebeu um padrão?

Geralmente, é melhor instalar pacotes Node localmente, sem a flag --global que vimos nos exemplos. (Veja outro padrão: global === ruim?) Ao instalar localmente, você terá diferentes versões dos mesmos pacotes conforme forem necessárias a cada aplicação com a qual você estiver trabalhando ou de que precise. Porém, no caso do Browserify e do Babel, instalá-los globalmente possibilita um acesso global (a partir de qualquer diretório) à CLI (Command-Line Interface, ou Interface de Linha de Comando).

React e outros pacotes

Você só precisa de mais alguns pacotes e estará pronto:

- react, é claro;
- react-dom, que é distribuído separadamente;
- babel-preset-react, que oferece ao Babel suporte ao JSX e a outras funcionalidades interessantes da React;
- babel-preset-es2015, que oferece suporte às funcionalidades mais recentes do JavaScript.

Inicialmente, acesse o diretório de sua aplicação (isto é, usando cd ~/reactbook/reactbook-boiler) para que você possa instalar esses pacotes localmente:

```
$ npm install --save-dev react
$ npm install --save-dev react-dom
$ npm install --save-dev babel-preset-react
$ npm install --save-dev babel-preset-es2015
```

Você perceberá que a sua aplicação agora tem um diretório *node_modules* com os pacotes locais e suas dependências. Os dois módulos globais (Babel, Browserify) estão em *node_modules*, que pode estar em um lugar diferente, conforme o seu sistema operacional (por exemplo, */usr/local/lib/node_modules* ou *C:\Users{seunome}\AppData\Roaming\npm*).

Vamos fazer a construção

O processo de construção deve executar três tarefas: concatenação de CSS, transpilação de JS e empacotamento de JS. Isso é simples, e basta executar três comandos.

Transpilar o JavaScript

Inicialmente, faça a transpilação do JavaScript com o Babel:

```
$ babel --presets react,es2015 js/source -d js/build
```

Isso significa tomar todos os arquivos de *js/source*, transpilar usando a React e os recursos de ES2015 e copiá-los para *js/build*. Você verá uma saída como esta:

```
js/source/app.js -> js/build/app.js
js/source/components/Logo.js -> js/build/components/Logo.js
```

Essa lista aumentará à medida que você adicionar novos componentes.

Empacotar o JavaScript

Em seguida, temos o empacotamento:

```
$ browserify js/build/app.js -o bundle.js
```

Diga ao Browserify para começar com *app.js*, seguir todas as dependências e escrever o resultado em *bundle.js*, que é o arquivo que, no final das contas, você incluirá em seu *index.html*. Para verificar se o arquivo foi realmente criado, digite less bundle.js.

Empacotar o CSS

O empacotamento de CSS é tão simples (nessa fase) que você nem sequer precisa de uma ferramenta especial; basta concatenar todos os arquivos CSS em um só (usando cat). No entanto, como estamos mudando a localização do arquivo, as referências às imagens falharão, portanto vamos reescrevê-las com uma chamada simples a sed:

```
cat css/*/* css/*.css | sed 's/..\/..\/images/images/g' > bundle.css
```

Há pacotes NPM que fazem um trabalho muito melhor, mas, por enquanto, isso será suficiente.

Resultados!

A essa altura, você terminou e está pronto para ver os resultados de seu trabalho árduo. Carregue *index.html* em seu navegador e veja a tela de boas-vindas (Figura 5.3).

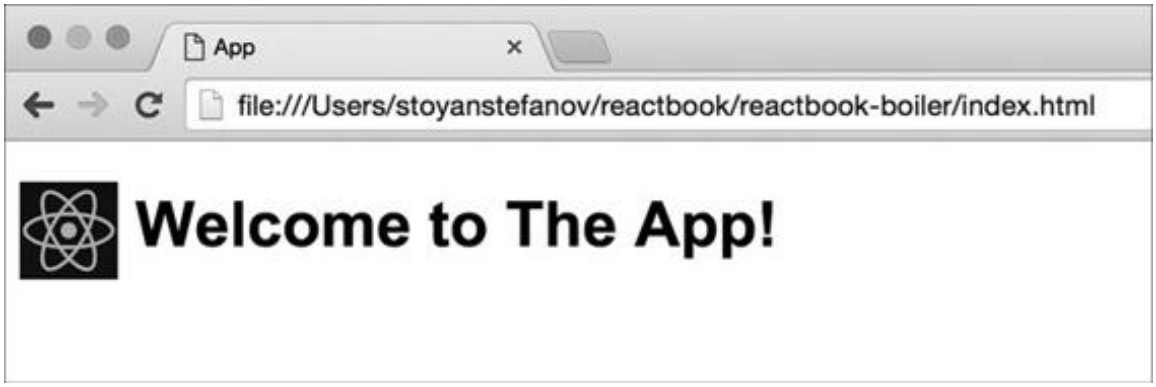


Figura 5.3 – Bem-vindo à aplicação.

Versão para Windows

Os comandos anteriores fazem sentido no Linux ou no Mac OS X. No Windows, eles não são muito diferentes. Os dois primeiros comandos são idênticos, exceto pelo separador de diretório. Portanto:

```
$ babel --presets react,es2015 js\source -d js\build
```

```
$ browserify js\build\app.js -o bundle.js
```

Não há `cat` no Windows, mas você pode concatenar assim:

```
$ type css\components\* css\* > bundle.css
```

Para substituir strings em um arquivo (fazer o CSS encontrar as imagens em *images*, e não em *../../images*), você precisa dos recursos um pouco mais sofisticados do “powershell”:

```
$ (Get-Content bundle.css).replace('../../images', 'images') | Set-Content bundle.css
```

Construindo durante o desenvolvimento

É inconveniente ter que executar um processo de construção sempre que você mudar um arquivo. Felizmente, você pode *observar* se há mudanças em um diretório a partir de um script e executar o script de construção automaticamente.

Em primeiro lugar, vamos colocar os três comandos que compõem o processo de construção em um arquivo chamado *scripts/build.sh*:

```
# transformação de js
babel --presets react,es2015 js/source -d js/build
# empacotamento de js
browserify js/build/app.js -o bundle.js
# empacotamento de css
cat css/*/* css/*.css | sed 's/..\/..\/images/images/g' > bundle.css
# pronto
date; echo;
```

Em seguida, instale o pacote NPM `watch`:

```
$ npm install --save-dev watch
```

Executar `watch` é como dizer para monitorar os diretórios *js/source/* e */css* para ver se houve alguma mudança e, se houver, execute o shell script que se encontra em *scripts/build.sh*:

```
$ watch "sh scripts/build.sh" js/source css

> Watching js/source/
> Watching css/
js/source/app.js -> js/build/app.js
js/source/components/Logo.js -> js/build/components/Logo.js
Sat Jan 23 19:41:38 PST 2016
```

É claro que você também pode colocar esse comando em *scripts/watch.sh*, de modo que, sempre que começar a trabalhar na aplicação, basta executar:

```
$ sh scripts/watch.sh
```

…e pronto. Você poderá fazer alterações nos arquivos-fontes e, em seguida, atualizar o navegador e ver a nova versão construída.

Implantação

Implantar a sua aplicação não é um grande problema agora, pois você já está fazendo a construção à medida que desenvolve, portanto não deve haver nenhuma surpresa. Antes de sua aplicação atingir usuários ao vivo, talvez você precise de algum processamento extra, como minificação e otimização de imagens.

Vamos usar a ferramenta popular de minificação de JS uglify e a ferramenta de minificação de CSS cssshrink como exemplos. Você pode prosseguir com a minificação de HTML, otimização de imagens, cópia de arquivos para um CDN (Content Delivery Network, ou Rede de Fornecimento de Conteúdo) e tudo o mais que você quiser.

scripts/deploy.sh pode ter o seguinte aspecto:

```
# limpa a última versão
rm -rf __deployme
mkdir __deployme

# construção
sh scripts/build.sh

# minificação de JS
uglify -s bundle.js -o __deployme/bundle.js
# minificação de CSS
cssshrink bundle.css > __deployme/bundle.css
# cópia de HTML e imagens
cp index.html __deployme/index.html
cp -r images/ __deployme/images/

# pronto
date; echo;
```

Após executar o script, você terá um novo diretório *__deployme* contendo:

- *index.html*;
- *bundle.css* minificado;
- *bundle.js*, também minificado;
- *pasta images/.*

Tudo que você precisa fazer então é copiar esse diretório para um servidor próximo a você e começar a servir seus usuários com sua aplicação atualizada.

Prosseguindo

Agora você já tem um exemplo de um pipeline de construção e de implantação simples, baseado em shell, em que você faz a construção enquanto desenvolve. Você pode expandi-lo conforme precisar ou pode experimentar uma ferramenta de construção mais especializada (por exemplo, Grunt ou Gulp) que seja mais apropriada às suas necessidades.

Com toda a questão da construção e da transpilação resolvida, é hora de passar para assuntos mais interessantes: construir e testar uma aplicação real, ao mesmo tempo que tiramos proveito dos muitos recursos que o JavaScript moderno tem a oferecer.

CAPÍTULO 6

Construindo uma aplicação

Agora que você já conhece todo o básico para criar componentes React personalizados (e usar os componentes embutidos), utilizar (ou não) o JSX para definir as interfaces de usuário e construir e implantar o resultado, é hora de criar uma aplicação mais completa.

A aplicação se chama “Whinepad” e permite que os usuários façam anotações e classifiquem todos os vinhos que experimentarem. Na verdade, não é necessário que sejam vinhos; pode ser qualquer outro produto sobre os quais os usuários queiram comentar. A aplicação deve fazer tudo que você esperaria de uma aplicação CRUD (Create, Read, Update and Delete, ou Criar, Ler, Atualizar e Apagar). Ela também deve ser uma aplicação do lado cliente, armazenando os dados no cliente. O objetivo é conhecer a React, portanto as partes da narrativa não relacionadas a ela (por exemplo, armazenagem, apresentação) serão mantidas em um nível mínimo.

Durante o processo, você aprenderá a:

- construir a aplicação a partir de componentes pequenos e reutilizáveis;
- efetuar a comunicação entre os componentes e fazê-los trabalhar em conjunto.

Whinepad v.0.0.1

Agora que você já tem o boilerplate do capítulo anterior, vamos dar início ao Whinepad. É uma aplicação de classificação, na qual você faz anotações sobre novos itens que experimenta. Que tal a tela de boas-vindas ser a lista de itens que você já classificou em uma bela

tabela? Isso significa simplesmente reutilizar o componente <Excel> do Capítulo 3.

Configuração

Inicialmente, copie a aplicação boilerplate reactbook-boiler para o lugar em que você vai trabalhar (obtenha a cópia de *https://github.com/stoyan/reactbook/*) e renomeie para whinepad v0.0.1.

Em seguida, inicie o script de observação (watch) para que tudo possa ser construído à medida que você fizer alterações:

```
$ cd ~/reactbook/whinepad\ v0.0.1/
$ sh scripts/watch.sh
```

Comece a programar

Atualize o título em *index.html* e use id="pad" para que corresponda à nova aplicação:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Whinepad v.0.0.1</title>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css" href="bundle.css">
  </head>
  <body>
    <div id="pad"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

Vamos obter a versão de Excel com JSX (conforme foi apresentada no final do Capítulo 4) e copiá-la para *js/source/components/Excel.js*:

```
import React from 'react';

var Excel = React.createClass({

  // implementação...

  render: function() {
```

```
    return (
      <div className="Excel">
        {this._renderToolbar()}
        {this._renderTable()}
      </div>
    );
  },
  // outras implementações ...
});
export default Excel
```

Podemos ver algumas diferenças em relação ao Excel anterior:

- instruções import/export;
- a raiz do componente agora tem um className="Excel" para estar de acordo com a convenção recém-estabelecida.

De modo análogo, todo o CSS está prefixado da seguinte maneira:

```
.Excel table {
  border: 1px solid black;
  margin: 20px;
}
.Excel th {
  /* e assim por diante */
}
  /* e assim por diante */
```

O que falta agora é apenas atualizar o *app.js* principal incluindo o <Excel>. Conforme mencionamos, vamos usar uma área de armazenagem do lado cliente (localStorage) para simplificar e deixar tudo do lado cliente. Somente para começar, vamos definir alguns defaults:

```
var headers = localStorage.getItem('headers');
var data = localStorage.getItem('data');
if (!headers) {
  headers = ['Title', 'Year', 'Rating', 'Comments'];
  data = [['Test', '2015', '3', 'meh']];
}
```

Agora passe os dados para o <Excel>:

```
ReactDOM.render(
  <div>
    <h1>
      <Logo /> Welcome to Whinepad!
    </h1>
    <Excel headers={headers} initialData={data} />
  </div>,
  document.getElementById('pad')
);
```

Com alguns ajustes em *Logo.css*, você terá terminado a versão 0.0.1 (veja a Figura 6.1)!

Figura 6.1 – Whinepad v.0.0.1.

Os componentes

Reutilizar o componente <Excel> existente é uma maneira simples de começar; no entanto, esse componente tem recursos demais. É melhor dividir e conquistar, separando-o em componentes pequenos e reutilizáveis. Por exemplo, os botões devem ser componentes separados para que possam ser reutilizados fora do contexto da tabela Excel.

Além do mais, a aplicação precisa de outros componentes especializados, como um widget de classificação que mostre estrelas em vez de apresentar apenas um número.

Vamos criar a nova aplicação e uma ferramenta auxiliar adicional

para descoberta de componentes. Essa ferramenta:

- permite que você desenvolva e teste componentes isoladamente. Com frequência, usar um componente em uma aplicação leva você a “casar” o componente com a aplicação e reduz a possibilidade de reutilizá-lo. Ter o componente isolado força tomar decisões melhores sobre o desacoplamento do componente;
- permite que outros membros da equipe descubram e reutilizem componentes existentes. À medida que sua aplicação crescer, o mesmo ocorrerá com a equipe. Para minimizar o risco de duas pessoas trabalharem em um componente notadamente semelhante e promover a reutilização de componentes (o que resulta em um desenvolvimento mais rápido de aplicações), é uma boa ideia deixar todos os componentes em um só lugar, juntamente com exemplos de como eles devem ser usados.

Configuração

Use CTRL+C para interromper o antigo script de observação (watch) para que você possa iniciar um novo. Copie o MVP (Minimum Viable Product, ou Produto Mínimo Viável) inicial whinepad v.0.0.1 para uma nova pasta chamada *whinepad*:

```
$ cp -r ~/reactbook/whinepad\ v0.0.1/ ~/reactbook/whinepad
$ cd ~/reactbook/whinepad
$ sh scripts/watch.sh

> Watching js/source/
> Watching css/
js/source/app.js -> js/build/app.js
js/source/components/Excel.js -> js/build/components/Excel.js
js/source/components/Logo.js -> js/build/components/Logo.js
Sun Jan 24 11:10:17 PST 2016
```

Descoberta

Vamos chamar a ferramenta de descoberta de componentes de *discovery.html* e colocá-la na raiz:

```
$ cp index.html discovery.html
```

Ela não precisa carregar toda a aplicação, portanto, em vez de usar *app.js*, usaremos um *discover.js*, que contém todos os exemplos de componentes. Isso implica que você também não incluirá o *bundle.js* da aplicação, mas um bundle separado que você pode chamar de *discover-bundle.js*:

```
<!DOCTYPE html>
<html>
  <!-- igual a index.html -->
  <body>
    <div id="pad"></div>
    <script src="discover-bundle.js"></script>
  </body>
</html>
```

Construir o novo bundle à medida que você implementar também é trivial – basta acrescentar mais uma linha no script *build.sh*:

```
# empacotamento de js
browserify js/build/app.js -o bundle.js
browserify js/build/discover.js -o discover-bundle.js
```

Por fim, acrescente o <Logo> de exemplo à ferramenta de descoberta (*js/build/discover.js*):

```
'use strict';

import Logo from './components/Logo';
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <div style={ {padding: '20px'} }>
    <h1>Component discoverer</h1>

    <h2>Logo</h2>
    <div style={ {display: 'inline-block', background: 'purple'} }>
      <Logo />
    </div>

    {/* outros componentes devem ser inseridos aqui... */}

  </div>,
  document.getElementById('pad')
);
```

Sua nova ferramenta de descoberta de componentes (Figura 6.2) é o local para começar a interagir com seus novos componentes à medida que eles ganharem vida. Vamos começar a trabalhar e construí-los – um de cada vez.

Figura 6.2 – A ferramenta de descoberta de componentes do Whinepad.

Componente <Button>

Não é um exagero generalizar dizendo que toda aplicação precisa de um botão. Com frequência, é um <button> comum, elegantemente estilizado, mas, ocasionalmente, talvez deva ser um <a>, como foi necessário no Capítulo 3 para os botões de download. Que tal fazer o novíssimo <Button> aceitar uma propriedade href opcional? Se ela estiver presente, um <a> será renderizado sob tudo.

No espírito de TDD (Test-Driven Development, ou Desenvolvimento Orientado a Testes), você pode começar de trás para a frente definindo exemplos de uso do componente na ferramenta *discovery.js*.

Antes:

```
import Logo from './components/Logo';

{/* ... */}

{/* outros componentes devem ser inseridos aqui... */}
```

Depois:

```
import Button from './components/Button';
import Logo from './components/Logo';

{/* ... */}

<h2>Buttons</h2>
<div>Button with onClick: <Button onClick={() => alert('ouch')}>Click me</Button></div>
<div>A link: <Button href="http://reactjs.com">Follow me</Button></div>
<div>Custom class name: <Button className="custom">I do nothing</Button></div>

{/* outros componentes devem ser inseridos aqui... */}
```

(Devemos então chamar isso de DDD [Discovery-Driven Development, ou Desenvolvimento Orientado a Descobertas?])

Você notou o padrão () => alert('ouch')? Esse é um exemplo de uma *função de seta (arrow function) do ES2015.*

Outros usos:

- () => {} é uma função vazia, como function() {};
- (what, not) => console.log(what, not) é uma função com parâmetros;
- (a, b) => { var c = a + b; return c;}: quando você tiver mais de uma linha no corpo da função, é preciso usar {};
- let fn = arg => {}: quando apenas um argumento é aceito, () é opcional.

Button.css

O estilo do componente <Button> deve estar em */css/components/Button.css*, conforme exigido pela convenção. Não há nada de especial em relação a esse arquivo: ele contém apenas um pouco da mágica de CSS na esperança de deixar o botão mais atraente. Vamos listá-lo a seguir como um exemplo e combinar que não vamos nos dar o trabalho de discutir o CSS dos demais componentes:

```
.Button {
  background-color: #6f001b;
  border-radius: 28px;
  border: 0;
  box-shadow: 0px 1px 1px #d9d9d9;
  color: #fff;
  cursor: pointer;
```

```
  display: inline-block;
  font-size: 18px;
  font-weight: bold;
  padding: 5px 15px;
  text-decoration: none;
  transition-duration: 0.1s;
  transition-property: transform;
}
.Button:hover {
  transform: scale(1.1);
}
```

Button.js

Vamos analisar o */js/source/components/Button.js* completo:

```
import classNames from 'classnames';
import React, {PropTypes} from 'react';
function Button(props) {
  const cssclasses = classNames('Button', props.className);
  return props.href
    ? <a {...props} className={cssclasses} />
    : <button {...props} className={cssclasses} />;
}
Button.propTypes = {
  href: PropTypes.string,
};
export default Button
```

Esse componente é pequeno, mas está cheio de novos conceitos e sintaxe. Vamos explorá-lo desde o começo!

Pacote classnames

```
import classNames from 'classnames';
```

O pacote `classnames` (obtenha-o com `npm i --save-dev classnames`) oferece uma função útil para lidar com nomes de classe CSS. Fazer seu componente poder usar suas próprias classes, mas também ser flexível o suficiente para permitir uma personalização por meio de nomes de classe passados pelo componente-pai é uma tarefa comum. Anteriormente, havia um utilitário no pacote de add-ons da

React que fazia exatamente isso, mas ele se tornou obsoleto em favor desse pacote separado de terceiros. Utilize a única função do pacote assim:

```
const cssclasses = classNames('Button', props.className);
```

Isso significa mesclar o nome da classe `Button` com quaisquer nomes de classe (se houver) passados como propriedades quando o componente for criado (Figura 6.3).

Você sempre pode fazer isso por conta própria e concatenar os nomes das classes, mas classnames é um pacote minúsculo que deixa essa tarefa comum ser mais conveniente. Ela também permite que você defina nomes de classe de forma condicional, o que também é conveniente, como em:

```
<div className={classNames({
  'mine': true, // não condicional
  'highlighted': this.state.active, // depende do
                                    // estado do componente...
  'hidden': this.props.hide, // ... ou de propriedades
})} />
```

Figura 6.3 – <Button> com um nome de classe personalizado.

Destructuring assignment (atribuição desestruturante ou paralela)

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

Essa é apenas uma maneira concisa de declarar:

```
import React from 'react';
const PropTypes = React.PropTypes;
```

Componente funcional sem estado

Quando um componente é muito simples (nada de errado com isso!) e não precisa manter um estado, você pode usar uma função para defini-lo. O corpo da função é o substituto para o seu método render().

A função aceita todas as propriedades como o seu primeiro argumento – é por isso que, no corpo, você utiliza `props.href` em vez da versão de classe/objeto `this.props.href`.

Se usar uma função de seta, você pode reescrever essa função assim:

```
const Button = props => {
  // ...
};
```

Se realmente quiser deixar o corpo com uma só linha, você pode até mesmo omitir `{}`, `;` e `return`:

```
const Button = props =>
  props.href
    ? <a {...props} className={classNames('Button', props.className)} />
    : <button {...props} className={classNames('Button', props.className)} />
```

propTypes

Se você usa a sintaxe das classes ES2015 ou componentes funcionais, você deve definir qualquer propriedade, por exemplo, `propTypes`, como propriedades estáticas, depois da definição do componente. Em outras palavras…

Antes (ES3, ES5):

```
var PropTypes = React.PropTypes;

var Button = React.createClass({
  propTypes: {
    href: PropTypes.string
  },
  render: function() {
    /* renderiza */
  }
});
```

Depois (classe ES2015):

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

class Button extends Component {
  render() {
    /* renderiza */
```

```
    }
  }
  Button.propTypes = {
    href: PropTypes.string,
  };
```

O mesmo vale se você usar um componente funcional sem estado:

```
  import React, {Component, PropTypes} from 'react';
  const Button = props => {
    /* renderiza */
  };
  Button.propTypes = {
    href: PropTypes.string,
  };
```

Formulários

A essa altura, tudo está certo com o componente <Button>. Vamos passar para a próxima tarefa, que é essencial para qualquer aplicação com entrada de dados: lidar com formulários. Como desenvolvedores de aplicações, raramente estamos satisfeitos com a aparência das entradas de formulário embutidas no navegador, e temos a tendência de criar nossas próprias versões. A aplicação Whinepad não poderia ser uma exceção.

Vamos criar um componente <FormInput> genérico com um método getValue() que dê acesso ao dado de entrada a quem o chamar. Conforme a propriedade type, esse componente deve delegar a criação da entrada a componentes mais especializados – por exemplo, entrada <Suggest>, entrada <Rating>, e assim por diante.

Vamos começar com os componentes de nível mais baixo; tudo de que eles precisam são os métodos render() e getValue().

<Suggest>

Entradas elegantes com sugestão automática (também conhecida como auto-suggest ou typeahead) são comuns na web, mas vamos

manter a simplicidade (Figura 6.4) e pegar carona no que o navegador já oferece – isto é, um elemento HTML <datalist> *(https://developer.mozilla.org/en/docs/Web/HTML/Element/datalist).*

Antes de mais nada, atualize a aplicação de descoberta:

```
<h2>Suggest</h2>
<div><Suggest options={['eenie', 'meenie', 'miney', 'mo']} /></div>
```

Vamos agora implementar o componente em */js/source/components/Suggest.js*:

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

class Suggest extends Component {

  getValue() {
    return this.refs.lowlevelinput.value;
  }

  render() {
    const randomid = Math.random().toString(16).substring(2);
    return (
      <div>
        <input
          list={randomid}
          defaultValue={this.props.defaultValue}
          ref="lowlevelinput"
          id={this.props.id} />
        <datalist id={randomid}>{
          this.props.options.map((item, idx) =>
            <option value={item} key={idx} />
          )
        }</datalist>
      </div>
    );
  }
}

Suggest.propTypes = {
  options: PropTypes.arrayOf(PropTypes.string),
};

export default Suggest
```

Como mostra o código anterior, não há nada realmente especial

nesse componente; ele é apenas um wrapper em torno de um <input>, com um <datalist> associado (por meio de randomid).

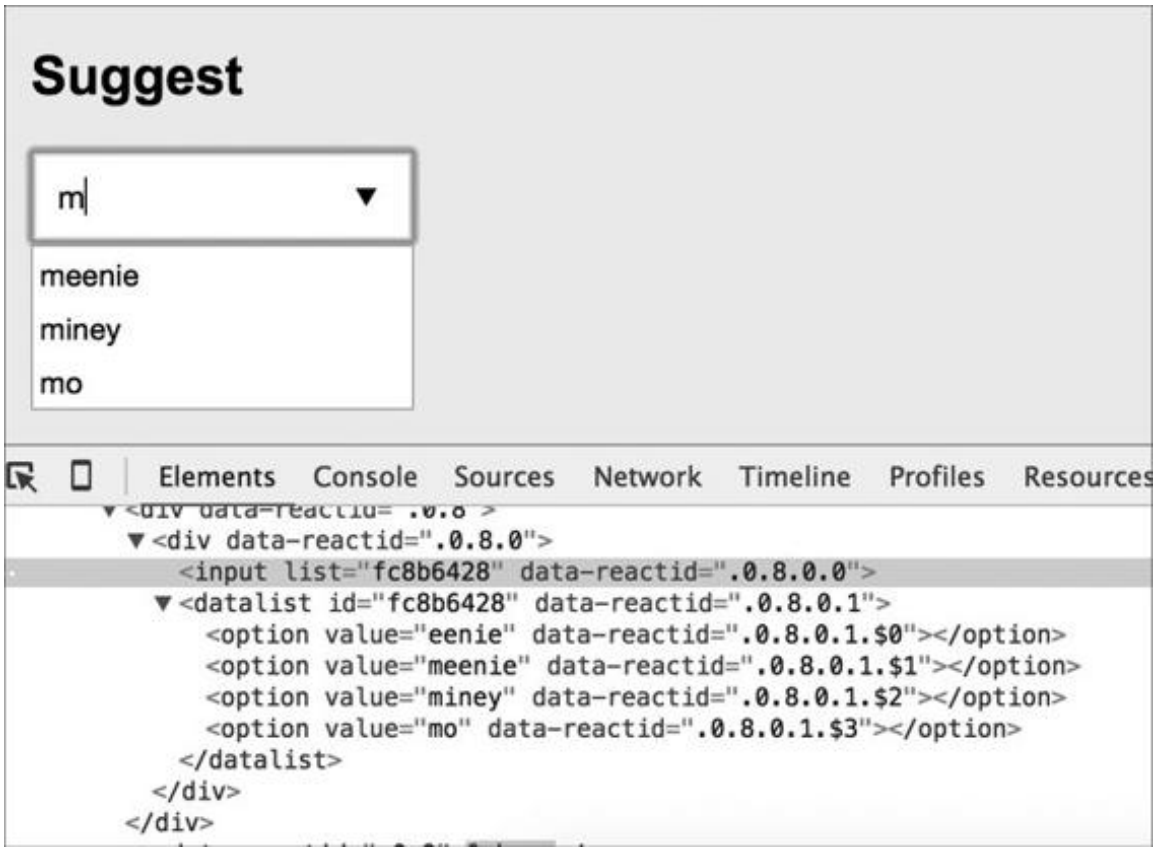


Figura 6.4 – A entrada <Suggest> em ação.

No que concerne à nova sintaxe de ES, podemos ver como é possível usar a *destructuring assignment* (atribuição desestruturante ou paralela) para atribuir mais de uma propriedade a uma variável:

```
// antes
import React from 'react';
const Component = React.Component;
const PropTypes = React.PropTypes;

// depois
import React, {Component, PropTypes} from 'react';
```

Quanto aos novos conceitos da React, podemos ver o uso de um atributo ref.

ref

Considere este código:

```
<input ref="domelement" id="hello">
/* mais tarde ... */
console.log(this.refs.domelement.id === 'hello'); // true
```

O atributo ref permite nomear uma instância específica de um componente React e *referenciá-la* posteriormente. Você pode adicionar um ref a qualquer componente, mas, em geral, ele é usado para referenciar elementos do DOM quando realmente precisamos ter acesso ao DOM subjacente. Com frequência, usar ref é uma solução de contorno, e pode haver outras maneiras de fazer o mesmo.

No caso anterior, queremos obter o valor de <input> quando for necessário. Considerando que mudanças na entrada podem ser pensadas como mudanças no estado do componente, podemos passar a usar this.state a fim de manter o controle:

```
class Suggest extends Component {
  constructor(props) {
    super(props);
    this.state = {value: props.defaultValue};
  }
  getValue() {
    return this.state.value; // não usa mais `ref`
  }
  render() {}
}
```

Então <input> não tem mais necessidade de um ref, mas precisa de um handler onChange para atualizar o status:

```
<input
  list={randomid}
  defaultValue={this.props.defaultValue}
  onChange={e => this.setState({value: e.target.value})}
  id={this.props.id} />
```

Observe o uso de this.state = {}; no constructor(): ele é um substituto para getInitialState() que usamos no mundo pré-ES6.

Componente <Rating>

Essa aplicação diz respeito a guardar anotações relacionadas a itens que você experimenta. A maneira mais simples de fazer anotações é usar classificações com estrelas, por exemplo, de 1 a 5.

Esse componente altamente reutilizável pode ser configurado para:

- usar qualquer quantidade de estrelas. O default é 5, mas por que não 11, por exemplo?
- ser somente de leitura, pois, às vezes, você não vai querer que cliques acidentais nas estrelas alterem os dados importantes de classificação.

Teste o componente na ferramenta de descoberta (Figura 6.5):

```
<h2>Rating</h2>
<div>No initial value: <Rating /></div>
<div>Initial value 4: <Rating defaultValue={4} /></div>
<div>This one goes to 11: <Rating max={11} /></div>
<div>Read-only: <Rating readonly={true} defaultValue={3} /></div>
```

Figura 6.5 – Um widget para classificação.

As necessidades básicas para a implementação incluem configurar os tipos das propriedades e o estado a ser mantido:

```
import classNames from 'classnames';
import React, {Component, PropTypes} from 'react';

class Rating extends Component {

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

```
    this.state = {
      rating: props.defaultValue,
      tmpRating: props.defaultValue,
    };
  }
  /* mais métodos... */
}
Rating.propTypes = {
  defaultValue: PropTypes.number,
  readonly: PropTypes.bool,
  max: PropTypes.number,
};
Rating.defaultProps = {
  defaultValue: 0,
  max: 5,
};
export default Rating
```

As propriedades são autoexplicativas: `max` é o número de estrelas e `readonly` deixa o widget somente para leitura. O estado contém `rating`, que é o valor atual de estrelas atribuídas, e `tmpRating`, que deve ser usado quando o usuário passar o mouse no componente, mas não está preparado ainda para clicar e confirmar uma classificação.

A seguir, apresentamos alguns métodos auxiliares para manter o estado atualizado à medida que o usuário interagir com o componente:

```
getValue() { // todas as nossas entradas disponibilizam isso
  return this.state.rating;
}
setTemp(rating) { // no mouse over
  this.setState({tmpRating: rating});
}
setRating(rating) { // no clique
  this.setState({
    tmpRating: rating,
    rating: rating,
});
}
```

```
reset() { // no mouse out, retorna à verdadeira classificação
  this.setTemp(this.state.rating);
}
componentWillReceiveProps(nextProps) { // reage a mudanças externas
  this.setRating(nextProps.defaultValue);
}
```

Por fim, temos o método render(). Ele contém:

- um laço para que haja estrelas de 1 a this.props.max. As estrelas são o símbolo ☆. Quando o estilo RatingOn é aplicado, as estrelas se tornam amarelas;
- uma entrada oculta que age como uma verdadeira entrada de formulário e permite que o valor possa ser obtido de forma genérica (assim como qualquer outro velho <input>):

```
render() {
  const stars = [];
  for (let i = 1; i <= this.props.max; i++) {
    stars.push(
      <span
        className={i <= this.state.tmpRating ? 'RatingOn' : null}
        key={i}
        onClick={!this.props.readonly && this.setRating.bind(this, i)}
        onMouseOver={!this.props.readonly && this.setTemp.bind(this, i)}
      >
        &#9734;
      </span>);
  }
  return (
    <div
      className={classNames({
        'Rating': true,
        'RatingReadonly': this.props.readonly,
      })}
      onMouseOut={this.reset.bind(this)}
    >
      {stars}
      {this.props.readonly || !this.props.id
        ? null
        : <input
```

```
            type="hidden"
            id={this.props.id}
            value={this.state.rating} />
        }
      </div>
    );
  }
```

Um aspecto que você pode observar aqui é o uso de `bind`. No laço de estrelas, faz sentido lembrar o valor atual de `i`, mas por que `this.reset.bind(this)`? Bem, isso é algo que você deve fazer quando usar a sintaxe de classe do ES. Você tem três opções para fazer a vinculação (bind):

- `this.method.bind(this)`, como vimos no exemplo anterior;
- uma função de seta (arrow function) faz uma autovinculação, como em `(_unused_event_) => this.method()`;
- fazer a vinculação uma vez no construtor.

Para explicar melhor a terceira opção, você deve fazer o seguinte:

```
class Comp extents Component {
  constructor(props) {
    this.method = this.method.bind(this);
  }
  render() {
    return <button onClick={this.method}>
  }
}
```

Uma vantagem está no fato de você usar a referência `this.method` do mesmo modo que antes (como nos componentes criados com `React.createClass({})`). Outra vantagem é que vinculamos o método de uma vez por todas, em oposição a fazê-lo sempre que `render()` for chamado. A desvantagem é ter mais boilerplate no controlador.

Uma “factory” <FormInput>

Em seguida, temos um `<FormInput>` genérico capaz de gerar entradas diferentes de acordo com as propriedades especificadas. Todas as

entradas geradas se comportam de modo consistente (oferecem `getValue()` quando for necessário).

Testando na aplicação de descoberta (Figura 6.6):

```
<h2>Form inputs</h2>
<table><tbody>
  <tr>
    <td>Vanilla input</td>
    <td><FormInput /></td>
  </tr>
  <tr>
    <td>Prefilled</td>
    <td><FormInput defaultValue="it's like a default" /></td>
  </tr>
  <tr>
    <td>Year</td>
    <td><FormInput type="year" /></td>
  </tr>
  <tr>
    <td>Rating</td>
    <td><FormInput type="rating" defaultValue={4} /></td>
  </tr>
  <tr>
    <td>Suggest</td>
    <td><FormInput
      type="suggest"
      options={['red', 'green', 'blue']}
      defaultValue="green" />
    </td>
  </tr>
  <tr>
    <td>Vanilla textarea</td>
    <td><FormInput type="text" /></td>
  </tr>
</tbody></table>
```

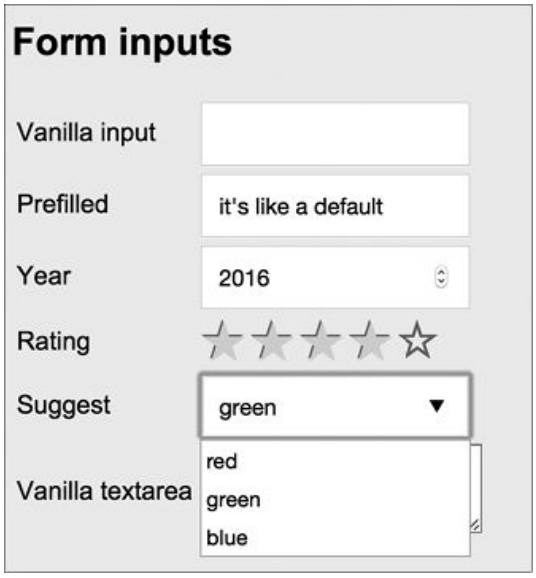


Figura 6.6 – Entradas de formulário.

A implementação de <FormInput> (*js/source/components/FormInput.js*) exige o boilerplate usual para importação, exportação e propTypes para validação:

```
import Rating from './Rating';
import React, {Component, PropTypes} from 'react';
import Suggest from './Suggest';

class FormInput extends Component {
  getValue() {}
  render() {}
}

FormInput.propTypes = {
  type: PropTypes.oneOf(['year', 'suggest', 'rating', 'text', 'input']),
  id: PropTypes.string,
  options: PropTypes.array, // como em <option>s de preenchimento automático
  defaultValue: PropTypes.any,
};

export default FormInput
```

O método render() é uma grande instrução switch, que delega a criação individual da entrada a um componente mais específico ou usa os elementos embutidos <input> e <textarea> do DOM como alternativa:

```
render() {
  const common = { // propriedades aplicáveis a todos
    id: this.props.id,
```

```
      ref: 'input',
      defaultValue: this.props.defaultValue,
    };
    switch (this.props.type) {
      case 'year':
        return (
          <input
            {...common}
            type="number"
            defaultValue={this.props.defaultValue || new Date().getFullYear()} />
        );
      case 'suggest':
        return <Suggest {...common} options={this.props.options} />;
      case 'rating':
        return (
          <Rating
            {...common}
            defaultValue={parseInt(this.props.defaultValue, 10)} />
        );
      case 'text':
        return <textarea {...common} />;
      default:
        return <input {...common} type="text" />;
    }
  }
```

Você notou a propriedade `ref`? Ela pode se mostrar útil no momento de obter o valor da entrada:

```
  getValue() {
    return 'value' in this.refs.input
      ? this.refs.input.value
      : this.refs.input.getValue();
  }
```

Nesse caso, `this.refs.input` é uma referência ao elemento do DOM subjacente. Para elementos comuns do DOM, como `<input>` e `<textarea>`, você obtém `value` do DOM com `this.refs.input.value` (como faria no modo tradicional do DOM com `document.getElementById('some-input').value`). Caso contrário, para as entradas elegantes e personalizadas como `<Suggest>` e `<Rating>`, você deverá lançar mão de

seus métodos getValue() individuais.

<Form>

Agora você tem:

- entradas personalizadas (por exemplo, <Rating>);
- entradas embutidas (por exemplo, <textarea>);
- <FormInput> – uma factory (fábrica) que cria entradas de acordo com a propriedade type.

É hora de fazer tudo isso funcionar em conjunto em um <Form> (Figura 6.7).

O componente de formulário deve ser reutilizável, e não deve haver nada fixo no código na aplicação de classificação de vinhos. (Para ir um passo além, nada sobre vinhos deve estar fixo no código, de modo que a aplicação possa ser readaptada para qualquer produto.) O componente <Form> pode ser configurado com um array de fields, em que cada campo é definido por:

- type da entrada; o default é “input”;
- id, para que a entrada possa ser encontrada mais tarde;
- label para ser colocado ao lado da entrada;
- options opcional para passar para a entrada com sugestão automática.

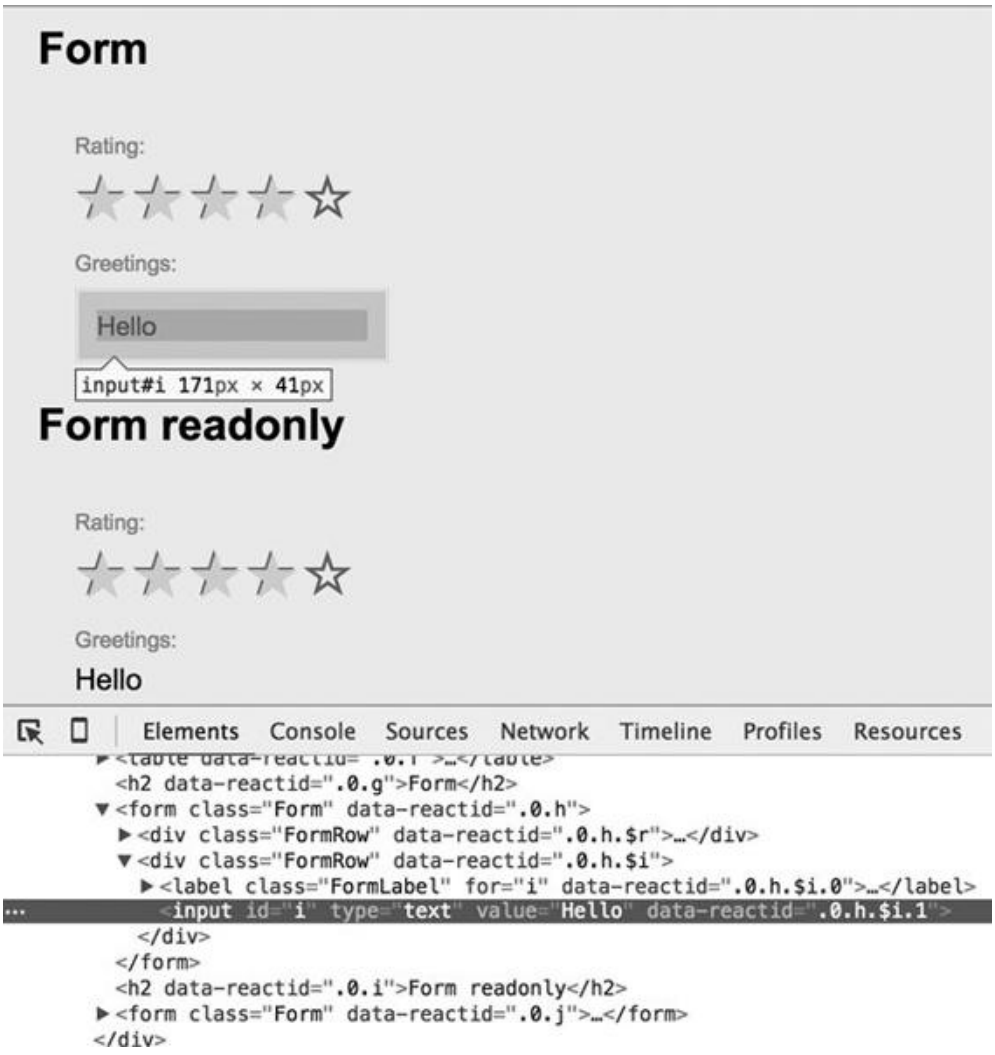


Figura 6.7 – Formulários.

<Form> também aceita um mapa de valores default e é capaz de renderização somente para leitura, de modo que o usuário não possa editá-lo:

```
import FormInput from './FormInput';
import Rating from './Rating';
import React, {Component, PropTypes} from 'react';

class Form extends Component {
  getData() {}
  render() {}
}

Form.propTypes = {
  fields: PropTypes.arrayOf(PropTypes.shape({
    id: PropTypes.string.isRequired,
```

```
    label: PropTypes.string.isRequired,
    type: PropTypes.string,
    options: PropTypes.arrayOf(PropTypes.string),
  })).isRequired,
  initialData: PropTypes.object,
  readonly: PropTypes.bool,
};
export default Form
```

Observe o uso de PropTypes.shape. Ele permite que você seja específico quanto ao que esperar em um mapa. É mais estrito que usar generalizações como fields: PropTypes.arrayOf(PropTypes.object) ou fields: PropTypes.array e, certamente, capturará mais erros antes que eles ocorram à medida que outros desenvolvedores começarem a usar seus componentes.

initialData é um mapa de {fieldname: value}, e esse também é o formato dos dados devolvidos pelo getData() do componente.

Eis um exemplo de uso de <Form> na ferramenta de descoberta:

```
<Form
  fields={[
    {label: 'Rating', type: 'rating', id: 'rateme'},
    {label: 'Greetings', id: 'freetext'},
  ]}
  initialData={ {rateme: 4, freetext: 'Hello'} } />
```

Vamos agora voltar para a implementação. O componente precisa de getData() e de render():

```
getData() {
  let data = {};
  this.props.fields.forEach(field =>
    data[field.id] = this.refs[field.id].getValue()
  );
  return data;
}
```

Como podemos ver, tudo que precisamos é de apenas um laço pelo getValue() de todas as entradas usando as propriedades ref definidas no método render().

O próprio método render() é simples e não usa nenhuma sintaxe ou padrão que não tenhamos visto antes:

```
  render() {
    return (
      <form className="Form">{this.props.fields.map(field => {
        const prefilled = this.props.initialData
          && this.props.initialData[field.id];
        if (!this.props.readonly) {
          return (
            <div className="FormRow" key={field.id}>
              <label className="FormLabel"
                htmlFor={field.id}>{field.label}:</label>
              <FormInput {...field} ref={field.id} defaultValue={prefilled} />
            </div>
          );
        }
        if (!prefilled) {
          return null;
        }
        return (
          <div className="FormRow" key={field.id}>
            <span className="FormLabel">{field.label}:</span>
            {
              field.type === 'rating'
                ? <Rating readonly={true}
                    defaultValue={parseInt(prefilled, 10)} />
                : <div>{prefilled}</div>
            }
          </div>
        );
      }, this)}</form>
    );
  }
```

<Actions>

Ao lado de cada linha da tabela de dados deve haver ações (Figura 6.8) que você pode executar em cada linha: apagar, editar, visualizar (quando nem todas as informações cabem em uma linha).

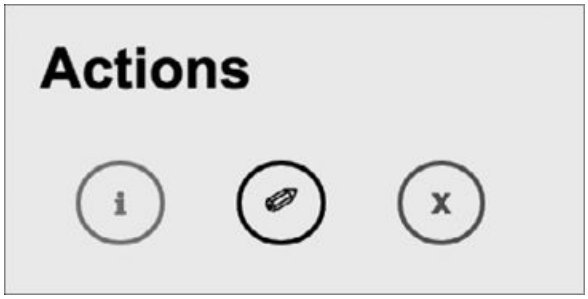


Figura 6.8 – Ações.

Eis o componente Actions sendo testado na ferramenta Discovery:

```
<h2>Actions</h2>
<div><Actions onAction={type => alert(type)} /></div>
```

E a implementação, que é razoavelmente simples:

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

const Actions = props =>
  <div className="Actions">
    <span
      tabIndex="0"
      className="ActionsInfo"
      title="More info"
      onClick={props.onAction.bind(null, 'info')}>&#8505;</span>
    <span
      tabIndex="0"
      className="ActionsEdit"
      title="Edit"
      onClick={props.onAction.bind(null, 'edit')}>&#10000;</span>
    <span
      tabIndex="0"
      className="ActionsDelete"
      title="Delete"
      onClick={props.onAction.bind(null, 'delete')}>x</span>
  </div>

Actions.propTypes = {
  onAction: PropTypes.func,
};

Actions.defaultProps = {
  onAction: () => {},
};

export default Actions
```

Actions é um componente simples que só precisa renderizar e não mantém nenhum estado. Portanto, ele pode ser definido como um componente funcional sem estado usando uma função de seta com a sintaxe mais sucinta possível: sem retorno, sem {}, sem instrução function (dificilmente reconhecível como uma função dos velhos tempos!).

Quem chama o componente pode se inscrever para receber o evento “action” usando a propriedade onAction. Esse é um padrão simples para um filho informar seu pai sobre uma mudança no componente. Como podemos ver, eventos personalizados (como onAction, onAlienAttack etc.) são simples assim.

Diálogos

A seguir, apresentaremos um componente de diálogo genérico a ser usado para qualquer tipo de mensagem (em vez de alert()) ou popups (Figura 6.9). Por exemplo, todos os formulários de adição/edição poderiam ser apresentados em um diálogo modal, sobre a tabela de dados.

Figura 6.9 – Diálogos.

Uso:

```
<Dialog
  header="Out-of-the-box example"
  onAction={type => alert(type)}>
    Hello, dialog!
</Dialog>

<Dialog
  header="No cancel, custom button"
  hasCancel={false}
  confirmLabel="Whatever"
  onAction={type => alert(type)}>
    Anything goes here, see:
    <Button>A button</Button>
</Dialog>
```

A implementação é semelhante a `<Actions>` – sem estado (tudo que você precisa é de um `render()`) e uma callback para `onAction` quando o usuário clicar em um botão no rodapé do diálogo:

```
import Button from './Button';
import React, {Component, PropTypes} from 'react';

class Dialog extends Component {

}

Dialog.propTypes = {
  header: PropTypes.string.isRequired,
  confirmLabel: PropTypes.string,
  modal: PropTypes.bool,
  onAction: PropTypes.func,
  hasCancel: PropTypes.bool,
};

Dialog.defaultProps = {
  confirmLabel: 'ok',
  modal: false,
  onAction: () => {},
  hasCancel: true,
};

export default Dialog
```

No entanto, esse componente é definido como uma classe e não como uma função de seta, pois deve ter dois métodos extras de

ciclo de vida:

```
componentWillUnmount() {
  document.body.classList.remove('DialogModalOpen');
}
componentDidMount() {
  if (this.props.modal) {
    document.body.classList.add('DialogModalOpen');
  }
}
```

Isso é necessário quando criamos um diálogo modal: o componente adiciona um nome de classe ao corpo do documento para que esse possa ser estilizado (desabilitado com cinza).

Por fim, temos o método `render()` para compor o wrapper modal, o cabeçalho, o corpo e o rodapé. O corpo acomoda qualquer outro componente (ou texto simples); o diálogo não é realmente complicado no que diz respeito ao seu conteúdo:

```
render() {
  return (
    <div className={this.props.modal ? 'Dialog DialogModal' : 'Dialog'}>
      <div className={this.props.modal ? 'DialogModalWrap' : null}>
        <div className="DialogHeader">{this.props.header}</div>
        <div className="DialogBody">{this.props.children}</div>
        <div className="DialogFooter">
          {this.props.hasCancel
            ? <span
              className="DialogDismiss"
              onClick={this.props.onAction.bind(this, 'dismiss')}>
              Cancel
            </span>
            : null
          }
          <Button onClick={this.props.onAction.bind(this,
            this.props.hasCancel ? 'confirm' : 'dismiss')}>
            {this.props.confirmLabel}
          </Button>
        </div>
      </div>
    </div>
```

```
    );
  }
```

Ideias alternativas:

- em vez de um único onAction, outra opção é oferecer onConfirm (o usuário clica em OK) e onDismiss;
- uma área para melhoria seria fechar o diálogo quando o usuário pressionar a tecla Esc. Como você implementaria isso?
- o wrapper div tem um nome de classe condicional e um nome não condicional. O componente certamente poderia se beneficiar do módulo classnames conforme apresentado a seguir.

Antes:

```
<div className={this.props.modal ? 'Dialog DialogModal' : 'Dialog'}>
```

Depois:

```
<div className={classNames({
  'Dialog': true,
  'DialogModal': this.props.modal,
})}>
```

Configuração da aplicação

A essa altura, todos os componentes de baixo nível estão prontos; os únicos dois restantes são a nova tabela de dados Excel melhorada e o componente-pai Whinepad de nível mais alto. Ambos são configuráveis por meio de um objeto de “esquema” – uma descrição do tipo de dado com o qual você vai querer lidar na aplicação. Eis um exemplo (*js/source/schema.js*) para que você possa dar início à aplicação voltada a vinhos:

```
import classification from './classification';

export default [
  {
    id: 'name',
    label: 'Name',
    show: true, // mostra a tabela `Excel`
    sample: '$2 chuck',
    align: 'left', // alinhamento no `Excel`
```

```
    },
    {
      id: 'year',
      label: 'Year',
      type: 'year',
      show: true,
      sample: 2015,
    },
    {
      id: 'grape',
      label: 'Grape',
      type: 'suggest',
      options: classification.grapes,
      show: true,
      sample: 'Merlot',
      align: 'left',
    },
    {
      id: 'rating',
      label: 'Rating',
      type: 'rating',
      show: true,
      sample: 3,
    },
    {
      id: 'comments',
      label: 'Comments',
      type: 'text',
      sample: 'Nice for the price',
    },
  ]
```

Esse é um exemplo de um dos módulos ECMAScript mais simples que você possa imaginar – um módulo que exporta uma única variável. Ela também importa outro módulo simples que contém algumas opções extensas para preencher previamente os formulários (*js/source/classification.js*):

```
export default {
  grapes: [
    'Baco Noir',
```

```
      'Barbera',
      'Cabernet Franc',
      'Cabernet Sauvignon',
      // ....
    ],
  }
```

Com a ajuda do módulo schema, você pode agora configurar o tipo de dado com o qual poderá lidar na aplicação.

<Excel>: novo e melhorado

O componente Excel do Capítulo 3 era um pouco eficiente demais. A versão nova e melhorada deve ser mais reutilizável. Assim, vamos nos livrar dos recursos de pesquisa (transfira para <Whinepad> no nível mais alto) e de download (sinta-se à vontade para adicioná-lo em Whinepad por conta própria). O componente deve estar totalmente voltado para a parte “RUD” da funcionalidade CRUD (Figura 6.10). É apenas uma tabela que pode ser editada. Ela deve oferecer ao pai Whinepad a capacidade de receber notificações quando os dados da tabela mudarem, por meio da propriedade onDataChange.

Figura 6.10 – Excel.

O Whinepad deve cuidar da pesquisa – o “C” de CRUD (criar uma nova entrada) – e da armazenagem permanente dos dados, usando localStorage. (É provável que você também vá armazenar os dados no servidor em uma aplicação no mundo real.)

Os dois componentes usam o mapa schema para configurar os tipos de dados.

Prepare-se para a implementação completa de Excel (é parecida com o que você já viu no Capítulo 3, exceto por alguns recursos):

```
import Actions from './Actions';
import Dialog from './Dialog';
import Form from './Form';
import FormInput from './FormInput';
import Rating from './Rating';
import React, {Component, PropTypes} from 'react';
import classNames from 'classnames';

class Excel extends Component {
  constructor(props) {
    super(props);
    this.state = {
      data: this.props.initialData,
      sortby: null, // schema.id
      descending: false,
      edit: null, // [row index, schema.id],
      dialog: null, // {type, idx}
    };
  }
  componentWillReceiveProps(nextProps) {
    this.setState({data: nextProps.initialData});
  }
  _fireDataChange(data) {
    this.props.onDataChange(data);
  }
  _sort(key) {
    let data = Array.from(this.state.data);
    const descending = this.state.sortby === key && !this.state.descending;
    data.sort(function(a, b) {
      return descending
        ? (a[column] < b[column] ? 1 : -1)
        : (a[column] > b[column] ? 1 : -1);
    });
    this.setState({
      data: data,
      sortby: key,
      descending: descending,
    });
    this._fireDataChange(data);
  }
```

```
_showEditor(e) {
  this.setState({edit: {
    row: parseInt(e.target.dataset.row, 10),
    key: e.target.dataset.key,
  }});
}
_save(e) {
  e.preventDefault();
  const value = this.refs.input.getValue();
  let data = Array.from(this.state.data);
  data[this.state.edit.row][this.state.edit.key] = value;
  this.setState({
    edit: null,
    data: data,
  });
  this._fireDataChange(data);
}
_actionClick(rowidx, action) {
  this.setState({dialog: {type: action, idx: rowidx}});
}
_deleteConfirmationClick(action) {
  if (action === 'dismiss') {
    this._closeDialog();
    return;
  }
  let data = Array.from(this.state.data);
  data.splice(this.state.dialog.idx, 1);
  this.setState({
    dialog: null,
    data: data,
  });
  this._fireDataChange(data);
}
_closeDialog() {
  this.setState({dialog: null});
}
_saveDataDialog(action) {
  if (action === 'dismiss') {
    this._closeDialog();
```

```
      return;
    }
    let data = Array.from(this.state.data);
    data[this.state.dialog.idx] = this.refs.form.getData();
    this.setState({
      dialog: null,
      data: data,
    });
    this._fireDataChange(data);
  }
  render() {
    return (
      <div className="Excel">
        {this._renderTable()}
        {this._renderDialog()}
    </div>
    );
  }
  _renderDialog() {
    if (!this.state.dialog) {
      return null;
    }
    switch (this.state.dialog.type) {
      case 'delete':
        return this._renderDeleteDialog();
      case 'info':
        return this._renderFormDialog(true);
      case 'edit':
        return this._renderFormDialog();
      default:
        throw Error(`Unexpected dialog type ${this.state.dialog.type}`);
    }
  }
  _renderDeleteDialog() {
    const first = this.state.data[this.state.dialog.idx];
    const nameguess = first[Object.keys(first)[0]];
    return (
      <Dialog
        modal={true}
        header="Confirm deletion"
```

```
        confirmLabel="Delete"
        onAction={this._deleteConfirmationClick.bind(this)}
      >
        {`Are you sure you want to delete "${nameguess}"?`}
      </Dialog>
    );
  }
  _renderFormDialog(readonly) {
    return (
      <Dialog
        modal={true}
        header={readonly ? 'Item info' : 'Edit item'}
        confirmLabel={readonly ? 'ok' : 'Save'}
        hasCancel={!readonly}
        onAction={this._saveDataDialog.bind(this)}
      >
        <Form
          ref="form"
          fields={this.props.schema}
          initialData={this.state.data[this.state.dialog.idx]}
          readonly={readonly} />
      </Dialog>
    );
  }
  _renderTable() {
    return (
      <table>
        <thead>
          <tr>{
            this.props.schema.map(item => {
              if (!item.show) {
                return null;
              }
              let title = item.label;
              if (this.state.sortby === item.id) {
                title += this.state.descending ? ' \u2191' : ' \u2193';
              }
              return (
                <th
                  className={`schema-${item.id}`}
```

```
            key={item.id}
            onClick={this._sort.bind(this, item.id)}
          >
            {title}
          </th>
        );
      }, this)
    }
    <th className="ExcelNotSortable">Actions</th>
    </tr>
</thead>
<tbody onDoubleClick={this._showEditor.bind(this)}>
  {this.state.data.map((row, rowidx) => {
    return (
      <tr key={rowidx}>{
        Object.keys(row).map((cell, idx) => {
          const schema = this.props.schema[idx];
          if (!schema || !schema.show) {
            return null;
          }
          const isRating = schema.type === 'rating';
          const edit = this.state.edit;
          let content = row[cell];
          if (!isRating && edit && edit.row === rowidx
              && edit.key === schema.id) {
            content = (
              <form onSubmit={this._save.bind(this)}>
                <FormInput ref="input"
                  {...schema} defaultValue={content} />
              </form>
            );
          } else if (isRating) {
            content = <Rating readonly={true}
              defaultValue={Number(content)} />;
          }
          return (
            <td
              className={classNames({
                [`schema-${schema.id}`]: true,
                'ExcelEditable': !isRating,
                'ExcelDataLeft': schema.align === 'left',
```

```
                    'ExcelDataRight': schema.align === 'right',
                    'ExcelDataCenter': schema.align
                    !== 'left' &&schema.align !== 'right',
                  })}
                  key={idx}
                  data-row={rowidx}
                  data-key={schema.id}>
                  {content}
                </td>
              );
            }, this)}
            <td className="ExcelDataCenter">
              <Actions onAction=
                {this._actionClick.bind(this, rowidx)} />
            </td>
          </tr>
        );
      }, this)}
    </tbody>
  </table>
);
}
}

Excel.propTypes = {
  schema: PropTypes.arrayOf(
    PropTypes.object
  ),
  initialData: PropTypes.arrayOf(
    PropTypes.object
  ),
  onDataChange: PropTypes.func,
};

export default Excel
```

Há alguns pontos que devem ser discutidos com um pouco mais de detalhes…

```
render() {
  return (
    <div className="Excel">
      {this._renderTable()}
```

```
        {this._renderDialog()}
      </div>
    );
  }
```

O componente renderiza a tabela e (talvez) um diálogo. O diálogo pode ser uma confirmação: “Sure you want to delete?” (Você tem certeza que quer apagar?), um formulário de edição ou um formulário somente de leitura, apenas para ler as informações sobre um item. Também pode não haver nenhum diálogo, que é o estado-padrão. Então você define a propriedade dialog de this.state; isso provoca uma nova renderização, a qual, por sua vez, renderizará um diálogo, caso seja necessário.

dialog é definido no estado quando o usuário clicar em um dos botões <Action>:

```
  _actionClick(rowidx, action) {
    this.setState({dialog: {type: action, idx: rowidx}});
  }
```

Quando os dados da tabela mudarem (com this.setState({data: /**/})), você disparará um evento de mudança que permita que o componente-pai saiba e possa atualizar a área de armazenagem permanente:

```
  _fireDataChange(data) {
    this.props.onDataChange(data);
  }
```

AA comunicação na outra direção – do componente-pai Whinepad para o filho Excel – ocorre quando o pai altera a propriedade initialData. O Excel está pronto para reagir a essas mudanças com:

```
  componentWillReceiveProps(nextProps) {
    this.setState({data: nextProps.initialData});
  }
```

Como criamos um formulário para entrada de dados (Figura 6.11)? Ou uma visualização dos dados (Figura 6.12)? Abra um Dialog contendo um Form. A configuração dos dados do formulário é proveniente de schema e a entrada de dados, de this.state.data:

```
_renderFormDialog(readonly) {
  return (
    <Dialog
      modal={true}
      header={readonly ? 'Item info' : 'Edit item'}
      confirmLabel={readonly ? 'ok' : 'Save'}
      hasCancel={!readonly}
      onAction={this._saveDataDialog.bind(this)}
    >
    <Form
      ref="form"
      fields={this.props.schema}
      initialData={this.state.data[this.state.dialog.idx]}
      readonly={readonly} />
    </Dialog>
  );
}
```

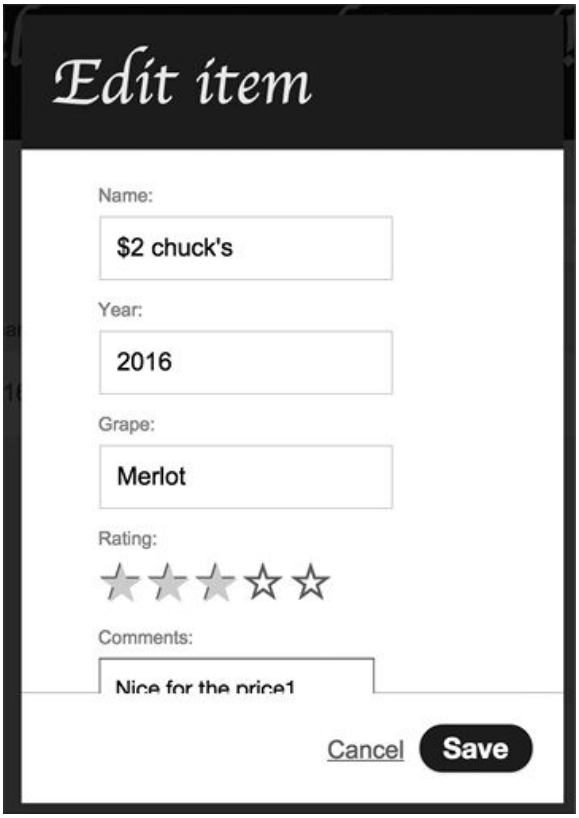


Figura 6.11 – Diálogo para editar dados (U em CRUD).

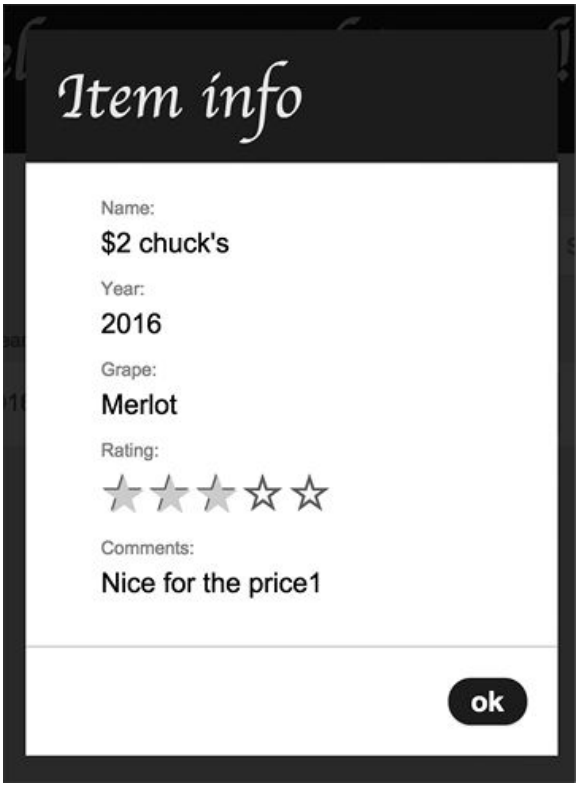


Figura 6.12 – Diálogo para visualizar dados (R em CRUD).

Quando o usuário terminar de editar os dados, tudo que você deve fazer é atualizar o estado e deixar que qualquer um que tenha se registrado para receber essa informação saiba:

```
_saveDataDialog(action) {
  if (action === 'dismiss') {
    this._closeDialog(); // apenas define this.state.dialog com null
    return;
  }
  let data = Array.from(this.state.data);
  data[this.state.dialog.idx] = this.refs.form.getData();
  this.setState({
    dialog: null,
    data: data,
  });
  this._fireDataChange(data);
}
```

No que diz respeito à nova sintaxe do ES, não há muita novidade além de um uso mais intenso de strings de template:

```
// Antes
```

```
"Are you sure you want to delete " + nameguess + "?"
// Depois
{`Are you sure you want to delete "${nameguess}"?`}
```

Observe também o uso de templates em nomes de classe, pois a aplicação permite personalizar a tabela de dados adicionando os IDs de schema. Assim:

```
// Antes
<th className={"schema-" + item.id}}>
// Depois
<th className={`schema-${item.id}`}>
```

A sintaxe mais inusitada provavelmente é usar uma string de template como o nome de uma propriedade em um objeto com a ajuda de colchetes []. Isso não está relacionado com a React, mas você pode achar curioso o fato de o código a seguir também ser possível com strings de template:

```
{
  [`schema-${schema.id}`]: true,
  'ExcelEditable': !isRating,
  'ExcelDataLeft': schema.align === 'left',
  'ExcelDataRight': schema.align === 'right',
  'ExcelDataCenter': schema.align !== 'left' && schema.align !== 'right',
}
```

<Whinepad>

É hora de implementar o último componente, que é o pai de todos os componentes (Figura 6.13). Ele é mais simples que o componente de tabela Excel e tem menos dependências.

```
import Button from './Button'; // <- para "add new item" (adicionar novo item)
import Dialog from './Dialog'; // <- para exibir o formulário "add new item"
// (adicionar novo item)
import Excel from './Excel'; // <- a tabela com todos os itens
import Form from './Form'; // <- o formulário "add new item" (adicionar
// novo item)
import React, {Component, PropTypes} from 'react';
```

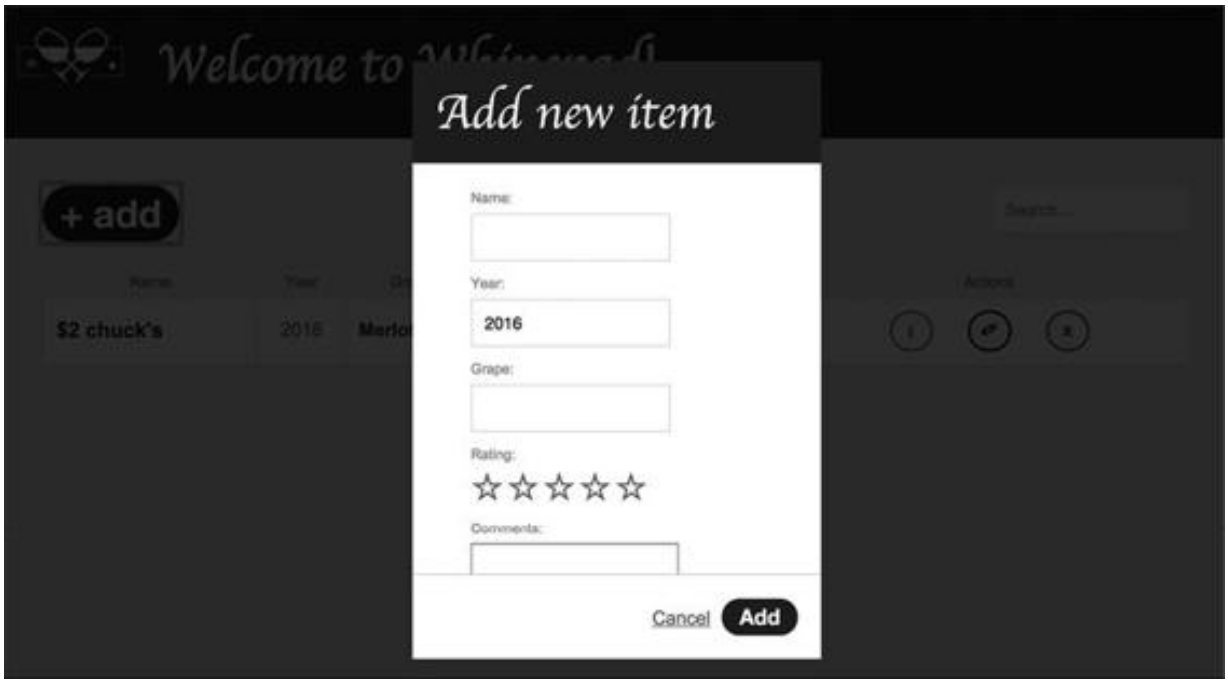


Figura 6.13 – Whinepad em ação com o "C" de CRUD.

O componente tem apenas duas propriedades – o schema dos dados e os itens existentes:

```
Whinepad.propTypes = {
  schema: PropTypes.arrayOf(
    PropTypes.object
  ),
  initialData: PropTypes.arrayOf(
    PropTypes.object
  ),
};
export default Whinepad;
```

Depois de ter lido a implementação de Excel, o código a seguir não deve ser muito difícil:

```
class Whinepad extends Component {

  constructor(props) {
    super(props);
    this.state = {
      data: props.initialData,
      addnew: false,
    };
    this._preSearchData = null;
```

```
}
_addNewDialog() {
  this.setState({addnew: true});
}
_addNew(action) {
  if (action === 'dismiss') {
    this.setState({addnew: false});
    return;
  }
  let data = Array.from(this.state.data);
  data.unshift(this.refs.form.getData());
  this.setState({
    addnew: false,
    data: data,
  });
  this._commitToStorage(data);
}
_onExcelDataChange(data) {
  this.setState({data: data});
  this._commitToStorage(data);
}
_commitToStorage(data) {
  localStorage.setItem('data', JSON.stringify(data));
}
_startSearching() {
  this._preSearchData = this.state.data;
}
_doneSearching() {
  this.setState({
    data: this._preSearchData,
  });
}
_search(e) {
  const needle = e.target.value.toLowerCase();
  if (!needle) {
    this.setState({data: this._preSearchData});
    return;
  }
  const fields = this.props.schema.map(item => item.id);
```

```
    const searchdata = this._preSearchData.filter(row => {
      for (let f = 0; f < fields.length; f++) {
        if (row[fields[f]].toString().toLowerCase().indexOf(needle) > -1) {
          return true;
        }
      }
      return false;
    });
    this.setState({data: searchdata});
  }
  render() {
    return (
      <div className="Whinepad">
        <div className="WhinepadToolbar">
          <div className="WhinepadToolbarAdd">
            <Button
              onClick={this._addNewDialog.bind(this)}
              className="WhinepadToolbarAddButton">
              + add
            </Button>
          </div>
          <div className="WhinepadToolbarSearch">
            <input
              placeholder="Search..."
              onChange={this._search.bind(this)}
              onFocus={this._startSearching.bind(this)}
              onBlur={this._doneSearching.bind(this)} />
          </div>
        </div>
        <div className="WhinepadDatagrid">
          <Excel
            schema={this.props.schema}
            initialData={this.state.data}
            onDataChange={this._onExcelDataChange.bind(this)} />
        </div>
        {this.state.addnew
          ? <Dialog
            modal={true}
            header="Add new item"
            confirmLabel="Add"
            onAction={this._addNew.bind(this)}
```

```
            >
              <Form
                ref="form"
                fields={this.props.schema} />
              </Dialog>
            : null}
        </div>
      );
    }
  }
```

Observe como o componente se inscreve para ser informado de mudanças nos dados de Excel com onDataChange. Além disso, note que todos os dados são simplesmente armazenados em localStorage:

```
  _commitToStorage(data) {
    localStorage.setItem('data', JSON.stringify(data));
  }
```

Esse seria o local para fazer qualquer requisição assíncrona (também conhecida como XHR, XMLHttpRequest, Ajax) para armazenar os dados no servidor, e não apenas no cliente.

Concluindo

A entrada principal para a aplicação é *app.js*, como vimos no início deste capítulo. O script *app.js* não é um componente nem um módulo; ele não exporta nada. Apenas contém tarefas de inicialização – ler os dados existentes de localStorage e configurar o componente <Whinepad>:

```
  'use strict';

  import Logo from './components/Logo';
  import React from 'react';
  import ReactDOM from 'react-dom';
  import Whinepad from './components/Whinepad';
  import schema from './schema';

  let data = JSON.parse(localStorage.getItem('data'));

  // dados default de exemplo, lê de schema
  if (!data) {
    data = {};
```

```
    schema.forEach(item => data[item.id] = item.sample);
    data = [data];
  }
  ReactDOM.render(
    <div>
      <div className="app-header">
        <Logo /> Welcome to Whinepad!
      </div>
      <Whinepad schema={schema} initialData={data} />
    </div>,
    document.getElementById('pad')
  );
```

Com isso, a aplicação está completa. Você pode interagir com ela em *http://whinepad.com* e navegar pelo código em *https://github.com/stoyan/reactbook/*.

CAPÍTULO 7

Lint, Flow, testar, repetir

O Capítulo 8 apresenta o Flux, que é uma alternativa para administrar a comunicação entre os componentes (ele assume o lugar de algo como onDataChange). Portanto, haverá um pouco de refatoração. Não seria bom ter menos erros ao refatorar? Vamos considerar algumas ferramentas que ajudarão a manter a sua sanidade à medida que sua aplicação inevitavelmente crescer e evoluir. As ferramentas são: ESLint, Flow e Jest.

Antes, porém, vamos dar uma olhada no pré-requisito comum chamado *package.json*.

package.json

Você já sabe usar o npm (Node Package Manager, ou Gerenciador de Pacotes do Node) para instalar bibliotecas e ferramentas de terceiros. Além disso, o npm também permite empacotar e compartilhar o seu projeto em *http://npmjs.com* para que outras pessoas possam instalá-lo. No entanto, você não *precisa* colocar o seu código em *npmjs.com* para tirar proveito do que o npm tem a oferecer.

O empacotamento está relacionado ao uso de um arquivo *package.json* que você pode colocar na raiz de sua aplicação para configurar as dependências e outras ferramentas adicionais. Há inúmeras configurações que podem ser colocadas nesse arquivo (acesse *https://docs.npmjs.com/files/package.json* para ver a história completa), mas vamos ver como utilizá-lo, ao mesmo tempo que manteremos o seu uso em um nível mínimo.

Crie um novo arquivo chamado *package.json* no diretório de sua aplicação:

```
$ cd ~/reactbook/whinepad2
$ touch package.json
```

Acrescente o seguinte no arquivo:

```
{
  "name": "whinepad",
  "version": "2.0.0",
}
```

É tudo de que você precisa. Em seguida, basta continuar adicionando mais configurações nesse arquivo.

Configure o Babel

O script *build.sh* que vimos no Capítulo 5 executa o Babel assim:

```
$ babel --presets react,es2015 js/source -d js/build
```

Podemos simplificar o comando passando a configuração de presets para *package.json*:

```
{
  "name": "whinepad",
  "version": "2.0.0",
  "babel": {
    "presets": [
      "es2015",
      "react"
    ]
  },
}
```

O comando agora passa a ser:

```
$ babel js/source -d js/build
```

O Babel (assim como muitas outras ferramentas do ecossistema de JavaScript) verifica se *package.json* está presente e obtém daí as opções de configuração.

scripts

O NPM permite configurar scripts e executá-los com npm run scriptname. Como exemplo, vamos passar *./scripts/watch.sh* de uma só linha do Capítulo 3 para *package.json*:

```
{
  "name": "whinepad",
  "version": "2.0.0",
  "babel": {/* ... */},
  "scripts": {
    "watch": "watch \"sh scripts/build.sh\" js/source css/"
  }
}
```

A partir de agora, para construir o seu código enquanto desenvolve, você poderá executar:

```
# antes
$ sh ./scripts/watch.sh

# depois
$ npm run watch
```

Se quiser continuar, você poderá também substituir *build.sh*, passando-o para *package.json* da mesma maneira. Ou usar uma ferramenta especializada de construção (Grunt, Gulp etc.), que também pode ser configurada no *package.json*. Para esta discussão sobre a React, porém, esse é todo o conhecimento sobre *package.json* de que você precisará.

ESLint

O ESLint (*http://eslint.org/*) é uma ferramenta que verifica o seu código em busca de padrões possivelmente perigosos. Ele também ajuda a garantir a consistência de sua base de código verificando, por exemplo, o uso de indentação e de outros espaçamentos. A ferramenta ajuda a identificar erros tolos de digitação ou variáveis desnecessárias à medida que você desenvolve. Além de executá-lo como parte de seu processo de construção, o ideal é que você também tenha essa ferramenta integrada ao seu sistema de controle de códigos-fontes bem como ao editor de texto de sua

preferência para que ele possa monitorar enquanto você estiver mais próximo de seu código.

Configuração

Além do ESLint propriamente dito, você precisa de plugins para React e Babel a fim de ajudar o ESLint a entender a sintaxe mais recente do ECMAScript, assim como para se beneficiar das “regras” mais específicas do JSX e da React:

```
$ npm i -g eslint eslint-plugin-react eslint-plugin-babel
```

Acrescente eslintConfig em *package.json*:

```
{
  "name": "whinepad",
  "version": "2.0.0",
  "babel": {},
  "scripts": {},
  "eslintConfig": {
    "parser": "babel-eslint",
    "plugins": [
      "babel",
      "react"
    ],
  }
}
```

Execução

Para executar o lint em um arquivo, execute:

```
$ eslint js/source/app.js
```

Esse comando deve executar sem erros, o que é bom – significa que o ESLint compreende o JSX e outras sintaxes inusitadas. Entretanto, também não é tão bom assim porque o lint não fez a validação em relação a nenhuma *regra*. O ESLint utiliza regras para cada verificação. Para começar, parta de (*estenda*) uma coleção de regras que o ESLint recomenda:

```
"eslintConfig": {
```

```
  "parser": "babel-eslint",
  "plugins": [],
  "extends": "eslint:recommended"
}
```

Uma nova execução apresenta alguns erros:

```
$ eslint js/source/app.js

/Users/stoyanstefanov/reactbook/whinepad2/js/source/app.js
   4:8 error "React" is defined but never used no-unused-vars
   9:23 error "localStorage" is not defined no-undef
  25:3 error "document" is not defined no-undef

✖ 3 problems (3 errors, 0 warnings)
```

A segunda e a terceira mensagens referem-se a variáveis não definidas (consequências da regra chamada no-undef), mas elas estão globalmente disponíveis no navegador, portanto a correção está mais relacionada à configuração:

```
"env": {
  "browser": true
}
```

O primeiro erro é específico da React. Por um lado, você deve incluir React, mas do ponto de vista do ESLint, ela parece ser uma variável não utilizada, que não precisaria estar presente. Acrescentar uma das regras de eslint-plugin-react ajuda nesse caso:

```
"rules": {
  "react/jsx-uses-react": 1
}
```

A execução no script *schema.js* resulta em outro tipo de erro:

```
$ eslint js/source/schema.js

/Users/stoyanstefanov/reactbook/whinepad2/js/source/schema.js
   9:18 error Unexpected trailing comma comma-dangle
  16:17 error Unexpected trailing comma comma-dangle
  25:18 error Unexpected trailing comma comma-dangle
  32:14 error Unexpected trailing comma comma-dangle
  38:33 error Unexpected trailing comma comma-dangle
  39:4 error Unexpected trailing comma comma-dangle

✖ 6 problems (6 errors, 0 warnings)
```

Vírgulas pendentes (como em `let a = [1,]` em oposição a `let a = [1]`) podem ser consideradas ruins (pois constituíam um erro de sintaxe no passado em determinados navegadores), mas também são convenientes, pois ajudam a manter o “blame” (anotações) no sistema de controle de versões e facilitam as atualizações. Uma configuração rápida transforma a prática de sempre usar vírgulas em algo bom:

```
"rules": {
  "comma-dangle": [2, "always-multiline"],
  "react/jsx-uses-react": 1
}
```

Todas as regras

Para ver a lista completa de regras, verifique o repositório de código do livro (*https://github.com/stoyan/reactbook/*), que (como uma expressão de lealdade ao projeto) é uma cópia da própria lista de regras da biblioteca React.

Por fim, acrescente o lint como parte de *build.sh* para que o ESLint faça verificações enquanto você desenvolve, a fim de garantir que o seu código tenha uma boa qualidade de forma consistente:

```
# QA
eslint js/source
```

Flow

O Flow (*http://flowtype.org*) é um verificador de tipos estáticos para JavaScript. Há duas opiniões sobre tipos em geral, especialmente em JavaScript.

Algumas pessoas gostam do fato de haver alguém observando por sobre os seus ombros para garantir que o programa está lidando com dados saudáveis. Assim como o linting e os testes de unidade, isso dá a você certo grau de confiança de que não causará falhas no código em um lugar que você não tenha conferido (ou que achou que não fosse importante). A tipagem torna-se mais valiosa à

medida que a aplicação crescer e, inevitavelmente, o número de pessoas em contato com o código aumentar.

Outras pessoas gostam da natureza dinâmica e sem tipos de JavaScript, e acham que os tipos são muito irritantes por causa dos ocasionais castings de tipo necessários.

É claro que cabe a você e à sua equipe decidir se querem tirar proveito dessa ferramenta, mas ela existe e está disponível para ser explorada.

Configuração

```
$ npm install -g flow-bin
$ cd ~/reactbook/whinepad2
$ flow init
```

O comando init cria um arquivo *.flowconfig* vazio em seu diretório. Faça os seguintes acréscimos nas seções ignore e include:

```
[ignore]
.*/react/node_modules/.*

[include]
node_modules/react
node_modules/react-dom
node_modules/classnames

[libs]
[options]
```

Execução

Tudo que você precisa fazer é digitar:

```
$ flow
```

Para verificar apenas um arquivo ou diretório, use:

```
$ flow js/source/app.js
```

Por fim, acrescente-o ao script de construção como parte do processo de QA (Quality Assurance, ou Garantia de Qualidade):

```
# QA
eslint js/source
```

```
flow
```

Inscrevendo-se para verificação de tipos

Você deve ter o texto `@flow` no primeiro comentário dos arquivos em que deseja fazer a verificação de tipos. Do contrário, o Flow deixará você livre. Portanto, ele é totalmente opcional.

Vamos começar inscrevendo o componente mais simples do último capítulo – `<Button>`:

```
/* @flow */

import classNames from 'classnames';
import React, {PropTypes} from 'react';

const Button = props =>
  props.href
    ? <a {...props} className={classNames('Button', props.className)} />
    : <button {...props} className={classNames('Button', props.className)} />

Button.propTypes = {
  href: PropTypes.string,
};

export default Button
```

Execute o Flow:

```
$ flow js/source/components/Button.js
js/source/components/Button.js:6
  6: const Button = props =>
^^^^^ parameter `props`. Missing annotation

Found 1 error
```

Há um erro, mas isso é bom – temos uma oportunidade para melhorar o código! O Flow está reclamando que não sabe de que tipo deve ser o argumento `props`.

O Flow espera que uma função como esta:

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

…tenha anotações como:

```
function sum(a: number, b: number): number {
```

```
  return a + b;
}
```

…para que você não acabe com resultados inesperados, por exemplo:

```
sum('1' + 2); // "12"
```

Corrigindo <Button>

O argumento `props` que a função aceita é um objeto. Portanto, você pode fazer o seguinte:

```
const Button = (props: Object) =>
```

…e deixar o Flow satisfeito:

```
$ flow js/source/components/Button.js
No errors!
```

A anotação `Object` funciona, mas você pode ser mais específico sobre o que deve ser usado e criar um tipo personalizado:

```
type Props = {
  href: ?string,
};

const Button = (props: Props) =>
  props.href
    ? <a {...props} className={classNames('Button', props.className)} />
    : <button {...props} className={classNames('Button', props.className)} />

export default Button
```

Como podemos ver, passar a usar um tipo personalizado permite substituir a definição de `propTypes` da React.

Isso significa:

- livrar-se de uma verificação de tipo em tempo de execução. O resultado, inevitavelmente, é um código um pouco mais rápido em tempo de execução;
- enviar menos código (menos bytes) ao cliente.

Também é interessante que os tipos das propriedades estejam no início do componente e sirvam como uma documentação local mais conveniente para o componente.

O ponto de interrogação em `href: ?string` significa que essa propriedade pode ser null.

Agora que propTypes não está mais presente, o ESLint reclama da variável PropTypes não ser usada. Assim,

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

…passa a ser:

```
import React from 'react';
```

Não é bom ter ferramentas como o ESLint observando pequenas faltas tolas como essa?

A execução do Flow apresenta outro erro:

```
$ flow js/source/components/Button.js
js/source/components/Button.js:12
  12: ? <a {...props} className={classNames('Button', props.className)} />
^^^^^^^^
property `className`.
Property not
found in
  12: ? <a {...props} className={classNames('Button', props.className)} />
^^^^^ object type
```

O problema é que o Flow não esperava encontrar um `className` no objeto `props`, que agora é do tipo `Prop`. Para resolver esse problema, acrescente `className` ao novo tipo:

```
type Props = {
  href: ?string,
  className: ?string,
};
```

app.js

A execução de Flow no *app.js* principal mostra que há problemas:

```
$ flow js/source/app.js
js/source/app.js:11
  11: let data = JSON.parse(localStorage.getItem('data'));
^^^^^^^^^^^^^^^^^^^^^^^^^^^^ call of method `getItem`
  11: let data = JSON.parse(localStorage.getItem('data'));
^^^^^^^^^^^^^^^^^^^^^^^^^^^^ null. This type is
incompatible with
```

```
383: static parse(text: string, reviver?: (key: any, value: any) => any): any;
^^^^^^ string. See lib: /private/tmp/flow/flow
lib_28f8ac7e/core.js:383
```

O Flow espera que você passe apenas strings para `JSON.parse()` e, solicitamente, apresenta a assinatura de `parse()`. Como você pode obter `null` de `localStorage`, isso é inaceitável. Uma correção simples consiste em simplesmente acrescentar um default:

```
let data = JSON.parse(localStorage.getItem('data') || '');
```

No entanto, `JSON.parse('')` é um erro no navegador (apesar de não ser um problema em se tratando de verificações de tipo) porque uma string vazia não é aceitável como um dado codificado em JSON. Uma pequena reescrita é necessária nesse caso para satisfazer o Flow, ao mesmo tempo que não provocamos erros no navegador.

Você pode perceber que lidar com tipos pode se tornar irritante, mas a vantagem é que o Flow faz você pensar duas vezes sobre os valores que são passados por aí.

A parte relevante do código de *app.js* é:

```
let data = JSON.parse(localStorage.getItem('data'));

// dados default de exemplo, lê de schema
if (!data) {
  data = {};
  schema.forEach((item) => data[item.id] = item.sample);
  data = [data];
}
```

Outro problema com esse código é que `data` já foi um array e, então, passou a ser um objeto e depois voltou a ser um array. JavaScript não tem problemas com isso, mas não parece ser uma boa prática – um valor é de um tipo agora e passa a ser de outro tipo depois. As engines JavaScript dos navegadores, na verdade, atribuem tipos internamente em um esforço para otimizar o código. Portanto, quando mudamos os tipos durante a execução, o navegador poderá deixar de lado o modo “otimizado”, e isso não é bom.

Vamos corrigir todos esses problemas.

Você pode ser extremamente rigoroso e definir `data` como um array

de objetos:

```
let data: Array<Object>;
```

Então você tentará ler qualquer item armazenado (ou null, por causa do ?) na string chamada storage:

```
const storage: ?string = localStorage.getItem('data');
```

Se encontrar uma string em storage, faça parse dela e pronto. Caso contrário, mantenha data como um array e preencha seu primeiro elemento com os valores da amostra:

```
if (!storage) {
  data = [{}];
  schema.forEach(item => data[0][item.id] = item.sample);
} else {
  data = JSON.parse(storage);
}
```

Agora dois dos arquivos estão em conformidade com o Flow. Vamos economizar um pouco de papel: não listaremos todo o código com tipos neste capítulo, mas vamos nos concentrar em mais alguns recursos interessantes do Flow. O repositório do livro (*https://github.com/stoyan/reactbook/*) tem o código completo.

Mais sobre verificação de tipos em propriedades e no estado

Ao criar o seu componente React com uma função sem estado, você pode usar anotações em props, conforme vimos antes:

```
type Props = {/* ... */};
const Button = (props: Props) => {/* ... */};
```

De modo semelhante, podemos usar isso no construtor de uma classe:

```
type Props = {/* ... */};
class Rating extends Component {
  constructor(props: Props) {/* ... */}
}
```

Porém, o que acontecerá se você não precisar de um construtor?

Como neste caso:

```
class Form extends Component {
  getData(): Object {}
  render() {}
}
```

Para isso, temos outro recurso do ECMAScript para ajudá-lo – uma propriedade de classe:

```
type Props = {/* ... */};
class Form extends Component {
  props: Props;
  getData(): Object {}
  render() {}
}
```

Na época em que escrevi este livro, as propriedades de classe ainda não eram aceitas no padrão ECMAScript, mas você pode tirar proveito delas graças ao mais novo preset stage-0 do Babel. Instale o pacote NPM babel-preset-stage-0 e atualize a seção Babel de seu *package.json* assim:

```
{
  "babel": {
    "presets": [
      "es2015",
      "react",
      "stage-0"
    ]
  }
}
```

De modo semelhante, você pode usar anotações no *estado* de seu componente. Além de ser conveniente para verificar os tipos, a definição de estado no nível mais alto serve como documentação para as pessoas que estão procurando bugs em seu componente. Eis um exemplo:

```
type Props = {
  defaultValue: number,
  readonly: boolean,
  max: number,
};
type State = {
```

```
  rating: number,
  tmpRating: number,
};
class Rating extends Component {
  props: Props;
  state: State;
  constructor(props: Props) {
    super(props);
    this.state = {
      rating: props.defaultValue,
      tmpRating: props.defaultValue,
    };
  }
}
```

E, é claro, você deve usar seus tipos personalizados sempre que puder:

```
componentWillReceiveProps(nextProps: Props) {
  this.setRating(nextProps.defaultValue);
}
```

Exportação/importação de tipos

Dê uma olhada no componente <FormInput>:

```
type FormInputFieldType = 'year' | 'suggest' | 'rating' | 'text' | 'input';
export type FormInputFieldValue = string | number;
export type FormInputField = {
  type: FormInputFieldType,
  defaultValue?: FormInputFieldValue,
  id?: string,
  options?: Array<string>,
  label?: string,
};
class FormInput extends Component {
  props: FormInputField;
  getValue(): FormInputFieldValue {}
  render() {}
}
```

Nesse caso, vemos como é possível fazer anotações usando uma

lista de valores permitidos, semelhante ao tipo de propriedade oneOf() da React.

Você também pode ver como podemos usar um tipo personalizado (FormInputFieldType) como parte de outro tipo personalizado (FormInputField).

Por fim, vamos ver a exportação de tipos. Quando outro componente utiliza o mesmo tipo, ele não precisa redefini-lo. Ele pode *importá-lo*, desde que o seu componente seja suficientemente gentil para exportá-lo. Eis o modo como o componente <Form> utiliza um tipo de <FormInput>:

```
import type FormInputField from './FormInput';

type Props = {
  fields: Array<FormInputField>,
  initialData?: Object,
  readonly?: boolean,
};
```

Na verdade, o formulário precisa dos dois tipos de FormInput, e a sintaxe é:

```
import type {FormInputField, FormInputFieldValue} from './FormInput';
```

Casting de tipo

O Flow permite especificar que um determinado valor seja de um tipo diferente daquele que ele suspeita ser.

Um exemplo está nos event handlers, para os quais passamos um objeto de evento; o Flow não considera o target do evento como aquilo que você acha que ele é. Considere o trecho de código a seguir do componente Excel:

```
_showEditor(e: Event) {
  const target = e.target;
  this.setState({edit: {
    row: parseInt(target.dataset.row, 10),
    key: target.dataset.key,
  }});
}
```

O Flow não gosta disso:

```
js/source/components/Excel.js:87
87: row: parseInt(target.dataset.row, 10),
^^^^^^^ property `dataset`. Property not found in
87: row: parseInt(target.dataset.row, 10),
^^^^^^ EventTarget
js/source/components/Excel.js:88
88: key: target.dataset.key,
^^^^^^^ property `dataset`. Property not found in
88: key: target.dataset.key,
^^^^^^ EventTarget
Found 2 errors
```

Se observar as definições em *https://github.com/facebook/flow/blob/master/lib/dom.js*, verá que EventTarget não tem uma propriedade dataset. Porém, HTMLElement tem. Desse modo, o casting de tipo vem para nos salvar:

```
const target = ((e.target: any): HTMLElement);
```

A sintaxe pode ser um pouco estranha à primeira vista, mas faz sentido se você a separar: valor, dois-pontos, tipo. Além disso, temos os parênteses para englobar os três itens. Um valor do tipo A passa a ser do tipo B.

Nesse caso, um objeto do tipo any tem o mesmo valor, mas passa a ser do tipo HTMLElement.

Invariantes

No componente Excel, existem duas propriedades no estado para controlar se o usuário está editando um campo e se há um diálogo ativo:

```
this.state = {
  // ...
  edit: null, // {row index, schema.id},
  dialog: null, // {type, idx}
};
```

Essas duas propriedades são null (não há edição, não há diálogo) ou objetos que contêm algumas informações sobre a edição ou o

diálogo. Os tipos dessas duas propriedades podem ter o seguinte aspecto:

```
type EditState = {
  row: number,
  key: string,
};
type DialogState = {
  idx: number,
  type: string,
};
type State = {
  data: Data,
  sortby: ?string,
  descending: boolean,
  edit: ?EditState,
  dialog: ?DialogState,
};
```

O problema em geral é que os valores, às vezes, são iguais a `null`, enquanto outras vezes não são. Isso é suspeito para o Flow, e ele tem razão. Quando tentamos usar `this.state.edit.row` ou `this.state.edit.key`, o Flow gera um erro:

```
Property cannot be accessed on possibly null value
```

Você utiliza as propriedades dessa forma somente quando sabe que elas estão disponíveis. No entanto, o Flow não sabe disso. Além do mais, não há nenhuma garantia de que, futuramente, à medida que sua aplicação crescer, você não vá acabar em um estado imprevisto. Quando isso ocorrer, você vai querer saber. Para satisfazer o Flow e, ao mesmo tempo, ser notificado caso a aplicação se comporte mal, você pode verificar se o valor com o qual está trabalhando não é null:

Antes:

```
data[this.state.edit.row][this.state.edit.key] = value;
```

Depois:

```
if (!this.state.edit) {
  throw new Error('Messed up edit state');
```

```
}
data[this.state.edit.row][this.state.edit.key] = value;
```

Agora tudo está no lugar certo. Se os trechos de código com throw condicional se tornarem muito repetitivos, você poderá usar uma função invariant(). Crie uma função desse tipo por conta própria ou obtenha uma função de código aberto já existente.

O NPM oferece suporte a você:

```
$ npm install --save-dev invariant
```

Acrescente o seguinte em *.flowconfig*:

```
[include]
node_modules/react
node_modules/react-dom
node_modules/classnames
node_modules/invariant
```

E agora faça a chamada da função:

```
invariant(this.state.edit, 'Messed up edit state');
data[this.state.edit.row][this.state.edit.key] = value;
```

Testes

A próxima parada no caminho para uma aplicação crescer sem problemas são os testes automatizados. Novamente, há muitas opções entre as quais você pode escolher quando se trata de testes. A React utiliza a ferramenta Jest (*http://facebook.github.io/jest/*) para executar testes, portanto vamos experimentá-la e ver como ela pode ajudar. A React também oferece um pacote chamado react-addons-test-utils para ajudar você em seu caminho.

É hora de mais configuração.

Configuração

Instale a interface de linha de comando do Jest:

```
$ npm i -g jest-cli
```

Você também precisa do babel-jest (para poder escrever seus testes em estilo ES06) e o pacote de utilitários para testes da React:

```
$ npm i --save-dev babel-jest react-addons-test-utils
```

Em seguida, atualize *package.json*:

```
{
  /* ... */
  "eslintConfig": {
    /* ... */
    "env": {
      "browser": true,
      "jest": true
    },
  /* ... */
  "scripts": {
    "watch": "watch \"sh scripts/build.sh\" js/source js/__tests__ css/",
    "test": "jest"
  },
  "jest": {
    "scriptPreprocessor": "node_modules/babel-jest",
    "unmockedModulePathPatterns": [
      "node_modules/react",
      "node_modules/react-dom",
      "node_modules/react-addons-test-utils",
      "node_modules/fbjs"
    ]
  }
}
```

Agora você pode executar o Jest com:

```
$ jest testname.js
```

Ou com npm:

```
$ npm test testname.js
```

O Jest procura testes em um diretório __tests__, portanto vamos colocar todos eles em js/__tests__.

Por fim, atualize o script de construção para também chamar lint e executar os testes como parte de cada construção:

```
# QA
```

```
eslint js/source js/__tests__
flow
npm test
```

…e o *watch.sh* para observar se houve alterações nos testes (lembre-se de que essa funcionalidade está duplicada em *package.json*):

```
watch "sh scripts/build.sh" js/source js/__tests__ css/
```

Primeiro teste

O Jest foi desenvolvido com base no framework popular Jasmine, cuja API lembra um inglês falado. Comece definindo uma *suíte de testes* com `describe('suite', callback)`, uma ou mais *especificações de teste* (test specs) com `it('test name', callback)` e em cada especificação, faça *asserções com a função* `expect()`.

Um exemplo simples e completo seria:

```
describe('A suite', () => {
  it('is a spec', () => {
    expect(1).toBe(1);
  });
});
```

Executando o teste:

```
$ npm test js/__tests__/dummy-test.js

> whinepad@2.0.0 test /Users/stoyanstefanov/reactbook/whinepad2
> jest "js/__tests__/dummy-test.js"

Using Jest CLI v0.8.2, jasmine1
PASS js/__tests__/dummy-test.js (0.206s)
1 test passed (1 total in 1 test suite, run time 0.602s)
```

Se houver uma asserção incorreta em seu teste, por exemplo…

```
expect(1).toBeFalsy();
```

…a execução do teste falhará mostrando uma mensagem, conforme vemos na Figura 7.1.

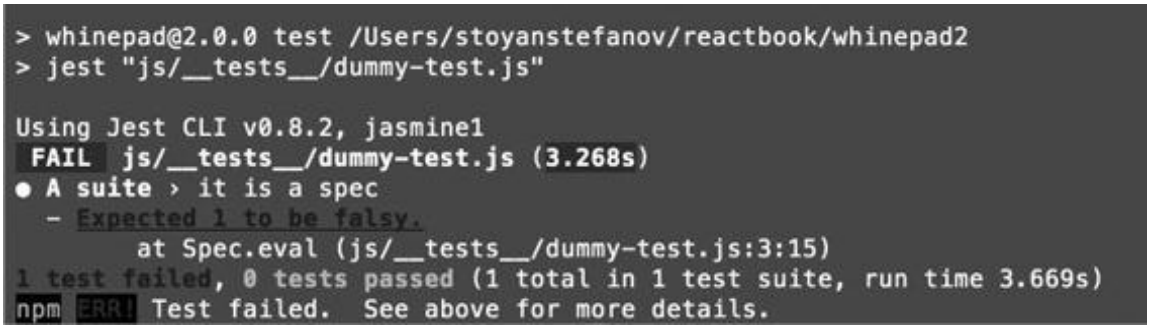


Figura 7.1 – Uma execução de teste com falha.

Primeiro teste com a React

Trazendo o que você já sabe sobre o Jest para o mundo da React, podemos começar testando um botão simples do DOM. Faça inicialmente as importações:

```
import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';
```

Configure a suíte de testes:

```
describe('We can render a button', () => {
  it('changes the text after click', () => {
    // ...
  });
});
```

Agora que o boilerplate está resolvido, é hora de começar a renderizar e a testar. Vamos renderizar um JSX simples:

```
const button = TestUtils.renderIntoDocument(
  <button
    onClick={ev => ev.target.innerHTML = 'Bye'}>
    Hello
  </button>
);
```

Nesse caso, usamos a biblioteca de utilitários para testes da React para renderizar o JSX – no exemplo, temos um botão cujo texto é alterado quando clicado.

Depois que renderizar algo, é hora de inspecionar se o que foi renderizado é o que você esperava:

```
expect(ReactDOM.findDOMNode(button).textContent).toEqual('Hello');
```

Como podemos ver, `ReactDOM.findDOMNode()` é usado para acessar um nó do DOM. A partir daí, podemos usar a API muito conhecida de DOM para inspecionar o nó.

Com frequência, você vai querer testar interações do usuário com a sua UI. De modo conveniente, a React oferece `TestUtils.simulate` para fazer exatamente isso:

```
TestUtils.Simulate.click(button);
```

A última tarefa é verificar se a UI respondeu à interação:

```
expect(ReactDOM.findDOMNode(button).textContent).toEqual('Bye');
```

No restante do capítulo, você verá mais exemplos e APIs que podem ser usados, mas suas ferramentas principais são estas:

- `TestUtils.renderIntoDocument(arbitraryJSX)`;
- `TestUtils.Simulate.*` para interagir com a interface;
- `ReactDOM.findDOMNode()` (ou mais alguns métodos de `TestUtils`) para obter uma referência a um nó do DOM e verificar se ele tem exatamente a aparência que deveria.

Testando o componente <Button>

O componente `<Button>` tem o seguinte aspecto:

```
/* @flow */

import React from 'react';
import classNames from 'classnames';

type Props = {
  href: ?string,
  className: ?string,
};

const Button = (props: Props) =>
  props.href
    ? <a {...props} className={classNames('Button', props.className)} />
    : <button {...props} className={classNames('Button', props.className)} />

export default Button
```

Vamos testar se:

- ele renderiza <a> ou <button> conforme haja uma propriedade href (primeira *especificação);*
- aceita nomes de classe personalizados (segunda *especificação).*

Iniciando o novo teste:

```
jest
  .dontMock('../source/components/Button')
  .dontMock('classnames')
;

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';
```

As instruções import são iguais às anteriores, mas agora há novas chamadas para jest.dontMock().

Uma *simulação* (mock) consiste em substituir uma funcionalidade por um código falso que finge estar fazendo a tarefa. Isso é comum em testes de unidade, pois você quer testar uma "unidade" – uma pequena parte *isolada* – e deseja reduzir os efeitos colaterais de tudo o mais no sistema. As pessoas investem um volume considerável de esforços para escrever simulações, e é por isso que o Jest adota a abordagem inversa: tudo é simulado, por padrão. Você tem a opção de desativar a simulação com dontMock(), pois quer testar o código real, e não uma simulação.

No exemplo anterior, você declarou que não deseja simular <Button> nem a biblioteca classnames que ele utiliza.

A seguir, vamos incluir <Button>:

```
const Button = require('../source/components/Button');
```

> Na época em que escrevi este livro, embora estivesse descrita na documentação do Jest, essa chamada a require() não funcionava. Em seu lugar, precisávamos de:
>
> ```
> const Button = require('../source/components/Button').default;
> ```
>
> UUm import também não funcionava:
>
> ```
> import Button from '../source/components/Button';
> ```
>
> Por outro lado, algo como o código a seguir não apresentava problemas:
>
> ```
> import _Button from '../source/components/Button';
> ```

```
const Button = _Button.default;
```

OOutra opção é não usar export default Button, mas export {Button} no componente <Button>. Em seguida, importe com import {Button} from '../source/component/Button'.

Espero que quando você estiver lendo isto, as importações de default estejam funcionando exatamente como esperado.

Primeira especificação

Vamos configurar a suíte (com describe()) e a primeira *especificação (com* it()*):*

```
describe('Render Button components', () => {
  it('renders <a> vs <button>', () => {
    /* ... renderizando e definindo expectativas (com expect()) ... */
  });
});
```

Vamos renderizar um botão simples – ele não tem um href, portanto um <button> deve ser renderizado:

```
const button = TestUtils.renderIntoDocument(
  <div>
    <Button>
      Hello
    </Button>
  </div>
);
```

Observe que você deve colocar componentes funcionais sem estado como <Button> em outro nó do DOM a fim de encontrá-los com ReactDOM mais tarde.

Agora ReactDOM.findDOMNode(button) devolve o wrapper , portanto,

para obter o <button>, acesse o primeiro filho e inspecione-o para garantir que ele seja realmente um botão:

```
expect(ReactDOM.findDOMNode(button).children[0].nodeName).toEqual('BUTTON');
```

De modo semelhante, como parte da mesma *especificação de teste*, verifique se um nó de âncora é usado quando houver um href:

```
const a = TestUtils.renderIntoDocument(
  <div>
    <Button href="#">
      Hello
    </Button>
  </div>
);
expect(ReactDOM.findDOMNode(a).children[0].nodeName).toEqual('A');
```

Segunda especificação

Na segunda especificação, adicione nomes de classe personalizados e então verifique se estão nos lugares em que deveriam estar:

```
it('allows custom CSS classes', () => {
  const button = TestUtils.renderIntoDocument(
    <div><Button className="good bye">Hello</Button></div>
  );
  const buttonNode = ReactDOM.findDOMNode(button).children[0];
  expect(buttonNode.getAttribute('class')).toEqual('Button good bye');
});
```

É importante destacar aqui um fato sobre a simulação do Jest. Às vezes, você escreverá um teste como esse e ele não funcionará conforme esperado. Isso poderia acontecer se você se esquecer de desativar uma simulação do Jest. Portanto, se você tiver o seguinte no início do teste:

```
jest
  .dontMock('../source/components/Button')
  // .dontMock('classnames')
;
```

…o Jest simulará o módulo classnames e ele não fará nada. Você pode observar esse comportamento escrevendo:

```
const button = TestUtils.renderIntoDocument(
  <div><Button className="good bye">Hello</Button></div>
);
console.log(ReactDOM.findDOMNode(button).outerHTML);
```

Esse código escreve o HTML gerado no console:

```
<div data-reactid=".2">
  <button data-reactid=".2.0">Hello</button>
</div>
```

Como podemos ver, nenhum nome de classe é necessário, pois `classNames()` não faz nada quando está simulado.

Traga `dontMock()` de volta:

```
jest
  .dontMock('../source/components/Button')
  .dontMock('classnames')
;
```

…e você verá que `outerHTML` será:

```
<div data-reactid=".2">
  <button class="Button good bye" data-reactid=".2.0">Hello</button>
</div>
```

…e seus testes passarão com sucesso.

Quando um teste se comportar mal e você estiver se perguntando qual é a aparência da marcação gerada, uma solução rápida e prática é usar console.log(node.outerHTML) – o HTML então se apresentará.

Testando <Actions>

`<Actions>` é outro componente sem estado, o que significa que você deve encapsulá-lo para poder inspecioná-lo mais tarde. Uma opção, como vimos com `<Button>`, é colocá-lo em uma `div` e acessá-lo assim:

```
const actions = TestUtils.renderIntoDocument(
  <div><Actions /></div>
);
ReactDOM.findDOMNode(actions).children[0]; // O nó-raiz de <Actions>
```

Um wrapper de componente

Outra opção é usar um elemento da React que seja um wrapper, o

que permite usar uma série de métodos de TestUtils para procurar nós a serem inspecionados.

O wrapper é simples e você pode defini-lo em um módulo próprio para que seja reutilizável:

```
import React from 'react';
class Wrap extends React.Component {
  render() {
    return <div>{this.props.children}</div>;
  }
}
export default Wrap
```

Agora a parte boilerplate do teste passa a ser:

```
jest
  .dontMock('../source/components/Actions')
  .dontMock('./Wrap')
;
import React from 'react';
import TestUtils from 'react-addons-test-utils';

const Actions = require('../source/components/Actions');
const Wrap = require('./Wrap');

describe('Click some actions', () => {
  it('calls you back', () => {
    /* renderiza */
    const actions = TestUtils.renderIntoDocument(
      <Wrap><Actions /></Wrap>
    );
    /* ... procura e inspeciona */
  });
});
```

Funções simuladas

Não há nada de especial a respeito do componente <Actions>. Ele tem o seguinte aspecto:

```
const Actions = (props: Props) =>
  <div className="Actions">
    <span
      tabIndex="0"
```

```
        className="ActionsInfo"
        title="More info"
        onClick={props.onAction.bind(null, 'info')}>&#8505;</span>
      {/* ... mais dois spans */}
    </div>
```

A única funcionalidade que você deve testar é se, quando clicadas, essas ações chamam a callback de `onAction` de forma apropriada. O Jest permite definir funções simuladas e verificar como elas são chamadas. Isso é perfeito para usar funções de callback.

No corpo do teste, crie uma nova função simulada e passe-a para `Actions` como uma callback:

```
const callback = jest.genMockFunction();
const actions = TestUtils.renderIntoDocument(
  <Wrap><Actions onAction={callback} /></Wrap>
);
```

A seguir, temos os cliques nas ações:

```
TestUtils
  .scryRenderedDOMComponentsWithTag(actions, 'span')
  .forEach(span => TestUtils.Simulate.click(span));
```

Observe o uso de um dos métodos de `TestUtils` para encontrar nós do DOM. Ele devolve um array de três nós `<span>` e você simula um clique em cada um.

Agora a sua função simulada de callback deve ter sido chamada três vezes. Confira se isso é realmente o que você esperava usando `expect()`:

```
const calls = callback.mock.calls;
expect(calls.length).toEqual(3);
```

Como podemos ver, `callback.mock.calls` é um array. Cada chamada também tem um array dos argumentos que foram passados para ela.

A primeira ação é “info” e chama `onAction` passando o tipo de ação “info” como `props.onAction.bind(null, 'info')`. Portanto, o primeiro argumento (0) da primeira callback simulada (0) deve ter sido “info”:

```
expect(calls[0][0]).toEqual('info');
```

O código é semelhante para as outras duas ações:

```
expect(calls[1][0]).toEqual('edit');
expect(calls[2][0]).toEqual('delete');
```

find e scry

TestUtils (*https://facebook.github.io/react/docs/test-utils.html*) oferece uma série de funções para encontrar nós do DOM em uma árvore renderizada pela React. Por exemplo, podemos procurar um nó de acordo com um nome de tag ou de classe. Um exemplo que vimos antes é:

```
TestUtils.scryRenderedDOMComponentsWithTag(actions, 'span')
```

Outro exemplo é:

```
TestUtils.scryRenderedDOMComponentsWithClass(actions, 'ActionsInfo')
```

Correspondendo aos métodos scry* , temos find*. Por exemplo:

```
TestUtils.findRenderedDOMComponentWithClass(actions, 'ActionsInfo')
```

Observe o uso de Component *versus* Components. Enquanto scry* fornece um array de correspondências (mesmo que haja apenas uma, ou até mesmo nenhuma), find* devolve uma única correspondência. Se não houver correspondências ou houver várias, isso será um erro. Portanto, encontrar correspondências com find* já é uma asserção segundo a qual um e somente um nó do DOM deve estar presente na árvore.

Mais interações simuladas

Vamos testar o widget Rating. Ele muda de estado quando ocorrer mouseover, mouseout e clique. Eis o boilerplate:

```
jest
  .dontMock('../source/components/Rating')
  .dontMock('classnames')
;

import React from 'react';
import TestUtils from 'react-addons-test-utils';

const Rating = require('../source/components/Rating');
```

```
describe('works', () => {
  it('handles user actions', () => {
    const input = TestUtils.renderIntoDocument(<Rating />);

    /* defina suas expectativas aqui (com `expect()`) */

  });
});
```

Observe que você não precisa de um wrapper para `<Rating>` ao renderizá-lo. Ele não é um componente funcional sem estado, portanto não há problemas.

O widget tem um número de estrelas (5 por padrão), cada uma em um span. Vamos encontrá-las:

```
const stars = TestUtils.scryRenderedDOMComponentsWithTag(input, 'span');
```

O teste simula mouseover, depois mouseout e então um clique na quarta estrela (`span[3]`). Quando isso ocorrer, as estrelas de 1 a 4 devem ser “ativadas”; em outras palavras, devem ter o nome de classe `RatingOn`, enquanto a quinta estrela deve permanecer “desativada”:

```
TestUtils.Simulate.mouseOver(stars[3]);
expect(stars[0].className).toBe('RatingOn');
expect(stars[3].className).toBe('RatingOn');
expect(stars[4].className).toBeFalsy();
expect(input.state.rating).toBe(0);
expect(input.state.tmpRating).toBe(4);

TestUtils.Simulate.mouseOut(stars[3]);
expect(stars[0].className).toBeFalsy();
expect(stars[3].className).toBeFalsy();
expect(stars[4].className).toBeFalsy();
expect(input.state.rating).toBe(0);
expect(input.state.tmpRating).toBe(0);

TestUtils.Simulate.click(stars[3]);
expect(input.getValue()).toBe(4);
expect(stars[0].className).toBe('RatingOn');
expect(stars[3].className).toBe('RatingOn');
expect(stars[4].className).toBeFalsy();
expect(input.state.rating).toBe(4);
expect(input.state.tmpRating).toBe(4);
```

Observe também como o teste acessa o estado do componente para verificar os valores corretos de state.rating e de state.tmpRating. Isso pode ser um pouco invasivo demais para um teste – afinal de contas, se os resultados "públicos" estão conforme o esperado, por que você se importaria com o estado interno que o componente optou por administrar? No entanto, isso certamente é possível.

Testando interações completas

Vamos escrever alguns testes para o Excel. Afinal de contas, ele é poderoso o suficiente para causar danos sérios no comportamento da aplicação, caso algo dê errado. Para começar:

```
jest.autoMockOff();

import React from 'react';
import TestUtils from 'react-addons-test-utils';

const Excel = require('../source/components/Excel');
const schema = require('../source/schema');

let data = [{}];
schema.forEach(item => data[0][item.id] = item.sample);

describe('Editing data', () => {
  it('saves new data', () => {
    /* ... renderize, interaja, inspecione */
  });
});
```

Inicialmente, observe o jest.autoMockOff(); no começo. Em vez de listar todos os componentes usados pelo Excel (e os componentes que, por sua vez, eles usam), você pode desabilitar toda a simulação de uma só vez.

Em seguida, observe que precisamos de um schema e um data de exemplo para inicializar o componente (exatamente como *app.js*).

Vamos agora passar para a renderização:

```
const callback = jest.genMockFunction();
const table = TestUtils.renderIntoDocument(
  <Excel
    schema={schema}
```

```
    initialData={data}
    onDataChange={callback} />
);
```

Até agora, tudo está bem; vamos então alterar o valor da primeira célula da primeira linha. O novo valor é:

```
const newname = '$2.99 chuck';
```

A célula em questão é:

```
const cell = TestUtils.scryRenderedDOMComponentsWithTag(table, 'td')[0];
```

Na época em que escrevi este livro, um pequeno hack era necessário para oferecer suporte a dataset, que não estava presente na implementação de DOM usada pelo Jest:

```
cell.dataset = { // hack em torno do suporte ao DOM no Jest
  row: cell.getAttribute('data-row'),
  key: cell.getAttribute('data-key'),
};
```

Dar um clique duplo na célula transforma o seu conteúdo em um formulário com uma entrada textual:

```
TestUtils.Simulate.doubleClick(cell);
```

Altere o valor da entrada e submeta o formulário:

```
cell.getElementsByTagName('input')[0].value = newname;
TestUtils.Simulate.submit(cell.getElementsByTagName('form')[0]);
```

Agora o conteúdo da célula não é mais um formulário, mas um texto simples:

```
expect(cell.textContent).toBe(newname);
```

A callback de `onDataChange` foi chamada com um array contendo objetos de pares chave-valor com os dados da tabela. Você pode conferir se a callback simulada recebeu os novos dados de forma apropriada:

```
expect(callback.mock.calls[0][0][0].name).toBe(newname);
```

Nesse caso, `[0][0][0]` significa que a primeira chamada para a função simulada tem como primeiro argumento um array em que o primeiro elemento é um objeto (correspondendo a um registro da tabela) com uma propriedade `name` igual a “$2.99 chuck”.

Em vez de usar TestUtils.Simulate.submit, você poderia optar por usar

TestUtils.Simulate.keyDown e enviar um evento informando que o botão Enter foi pressionado, o que também faria o formulário ser submetido.

Como uma segunda especificação de teste, vamos apagar a única linha de dados da amostra:

```
it('deletes data', () => {
  // Mesmo que antes
  const callback = jest.genMockFunction();
  const table = TestUtils.renderIntoDocument(
    <Excel
      schema={schema}
      initialData={data}
      onDataChange={callback} />
  );
  TestUtils.Simulate.click( // ícone x
    TestUtils.findRenderedDOMComponentWithClass(table, 'ActionsDelete')
  );
  TestUtils.Simulate.click( // diálogo de confirmação
    TestUtils.findRenderedDOMComponentWithClass(table, 'Button')
  );
  expect(callback.mock.calls[0][0].length).toBe(0);
});
```

Como no exemplo anterior, `callback.mock.calls[0][0]` é o novo array de dados após a interação. Desta vez, porém, não há nenhum dado restante, pois o teste apagou o único registro.

Cobertura

Depois de ter dominado esses assuntos, o quadro geral deve ter se tornado mais simples e talvez um pouco repetitivo. Cabe a você garantir que o maior número possível de cenários completos que você possa imaginar seja testado. Por exemplo, clicar na ação “info”, desistir, clicar em “delete”, desistir, clicar novamente e só então apagar.

Usar testes é uma ótima ideia, pois eles ajudam você a desenvolver mais rapidamente, com mais confiança e a refatorar sem medo. Os

testes ajudam a chamar a atenção de seus colegas de trabalho para os casos em que eles achem que uma mudança é isolada, mas, no final das contas, ela apresenta um alcance maior que o esperado. Uma maneira de “gamificar” o processo de escrita de testes é usar o recurso de *cobertura de código* (code coverage).

Você pode executar:

```
$ jest --coverage
```

…que executará todos os testes que a ferramenta puder encontrar e então fornecerá um relatório de quantas linhas, funções etc. você testou (ou *cobriu*). Para um exemplo, veja a Figura 7.2.

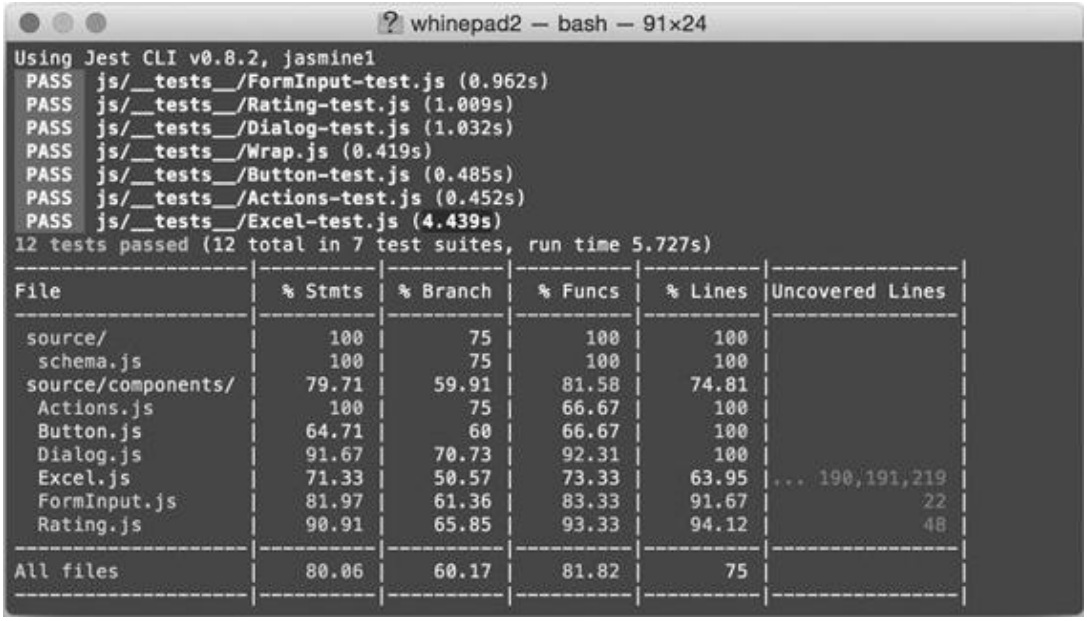


Figura 7.2 – Relatório de cobertura de código.

Podemos ver que nem tudo é perfeito; definitivamente, há uma oportunidade para escrever mais testes. Um recurso interessante do relatório de cobertura é que ele aponta as linhas não cobertas. Desse modo, apesar de ter testado FormInput, a linha 22 não foi coberta.

A linha em questão é o return em:

```
getValue(): FormInputFieldValue {
  return 'value' in this.refs.input
    ? this.refs.input.value
```

```
      : this.refs.input.getValue();
  }
```

Parece que os testes jamais testaram essa função. É hora de remediar isso com uma especificação de teste rápida:

```
  it('returns input value', () => {
    let input = TestUtils.renderIntoDocument(<FormInput type="year" />);
    expect(input.getValue()).toBe(String(new Date().getFullYear()));
    input = TestUtils.renderIntoDocument(
      <FormInput type="rating" defaultValue="3" />
    );
    expect(input.getValue()).toBe(3);
  });
```

O primeiro `expect()` testa uma entrada embutida do DOM, e o segundo testa uma entrada personalizada. Ambas as saídas do ternário em `getValue()` agora devem ser executadas.

O relatório de cobertura de código recompensa você com um resultado que mostra que agora a linha 22 está coberta (Figura 7.3).

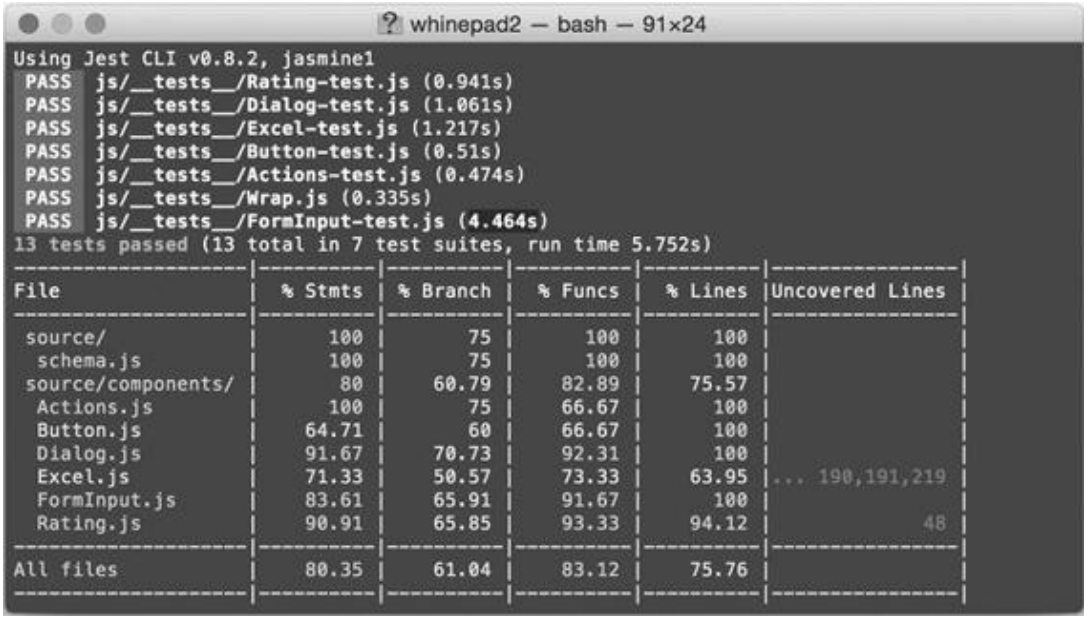


Figura 7.3 – Relatório de cobertura de código atualizado.

CAPÍTULO 8

Flux

O último capítulo apresenta o Flux (*https://facebook.github.io/flux/*), que é uma forma alternativa de administrar a comunicação entre os componentes e uma maneira de gerenciar o fluxo de dados geral de sua aplicação. Até agora, vimos como a comunicação pode ocorrer por meio da passagem de propriedades de um componente-pai para um filho e, então, ouvindo as mudanças no filho (por exemplo, com `onDataChange`). No entanto, ao passar propriedades dessa maneira, você poderá, às vezes, acabar com um componente que tenha muitas propriedades. Isso dificulta testar esse componente e conferir se todas essas combinações e trocas de propriedades funcionam conforme esperado.

Além disso, ocasionalmente, você acabará em cenários nos quais precisará passar propriedades por meio de um "pipe" do componente-pai para um filho, depois para um neto, para um bisneto, e assim sucessivamente. Isso tende a ser repetitivo (o que, por si só, é ruim), mas também confuso, além de exigir mais esforço mental da pessoa que ler o código (há muitos detalhes para monitorar ao mesmo tempo).

O Flux é uma maneira de ajudar você a superar esses obstáculos e preservar a sua sanidade, ao mesmo tempo que o fluxo de dados de sua aplicação permanece administrável. O Flux não é uma biblioteca de código: é mais uma ideia sobre como organizar os dados de sua aplicação. Afinal de contas, na maioria dos casos, os dados é que são importantes. Os usuários lançam mão de sua aplicação para lidar com dinheiro, emails, fotos ou qualquer que seja o dado. Mesmo que a UI seja um pouco desajeitada, eles poderão

conviver com ela. Contudo, não deve haver confusão com o estado dos dados em nenhum momento (“Eu acabei de enviar 30 dólares ou não?”).

Há muitas implementações de código aberto das ideias do Flux. Em vez de discutir essas opções, este capítulo discute uma abordagem mais DYI (Do It Yourself, ou Faça Você Mesmo). Depois que compreender a ideia (e se convencer de suas vantagens), você poderá continuar a explorar as opções disponíveis ou continuar trabalhando em uma solução própria.

A grande ideia

A ideia é que sua aplicação seja totalmente voltada a dados. Os dados estão contidos em uma *Store* (Área de Armazenagem). Seus componentes React (a *View*, ou Visão) leem os dados da Store e os renderiza. Então o usuário da aplicação entra em cena e executa uma *Action* (Ação, por exemplo, clicar em um botão). A Action faz com que os dados da Store sejam atualizados, o que afeta a View. O ciclo prossegue indefinidamente (Figura 8.1). Os dados fluem em uma única direção (de modo *unidirecional*), o que facilita seguir, compreender e depurá-los.

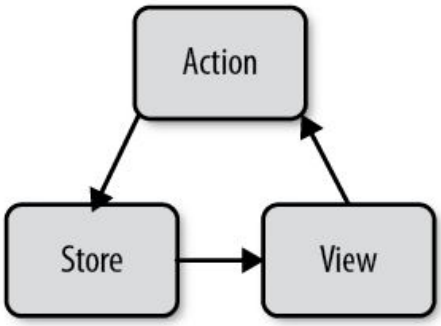


Figura 8.1 – Fluxo de dados unidirecional.

Há outras variantes e expansões dessa ideia geral, incluindo mais Actions, várias Stores e um *Dispatcher*, mas vamos dar uma olhada em um pouco de código antes de nos perdermos em explicações improdutivas.

Analisando o Whinepad novamente

A aplicação Whinepad tem um componente React principal chamado <Whinepad>, criado da seguinte maneira:

```
<Whinepad
  schema={schema}
  initialData={data} />
```

O <Whinepad>, por sua vez, cria um componente <Excel>:

```
<Excel
  schema={this.props.schema}
  initialData={this.state.data}
  onDataChange={this._onExcelDataChange.bind(this)} />
```

Inicialmente, o schema, que é uma descrição dos dados com que a aplicação trabalha, é passado (por meio de pipe) da forma como está, de <Whinepad> para <Excel>. (E então para <Form> também.) Isso é um pouco repetitivo, com aspecto de boilerplate. E se você tivesse que fazer um pipe de várias propriedades como essa? Em breve, a *superfície* de seus componentes ficaria extensa demais para poucas vantagens.

A palavra “superfície” nesse contexto são as propriedades que um componente recebe. É usada como sinônimo de “API” ou “assinatura de função”. Como sempre, em programação, é melhor manter a superfície em um nível mínimo. Uma função que aceite dez argumentos é muito mais difícil de usar, depurar e testar que uma que aceite dois. Ou que não aceite nenhum.

O schema é simplesmente passado da forma como está, enquanto, aparentemente, os dados não o são. <Whinepad> aceita uma propriedade initialData, mas então passa alguma versão dela para <Excel> (this.state.data em oposição a this.props.initialData). Nesse ponto, surge uma questão: De que modo o dado novo é diferente do original? E quem tem a *única fonte da verdade* quando se trata dos dados mais recentes?

Na implementação que vimos no capítulo anterior, <Whinepad> realmente contém os dados mais atualizados, o que é apropriado. Porém, o motivo pelo qual um componente de UI (a React tem tudo a ver com UI) deva guardar a fonte da verdade não é muito óbvio.

Vamos introduzir uma Store para cuidar dessa tarefa.

A Store

Começaremos com uma cópia do código até agora:

```
$ cd ~/reactbook
$ cp -r whinepad2 whinepad3
$ cd whinepad3
$ npm run watch
```

Em seguida, crie um novo diretório para armazenar os módulos do Flux (para diferenciá-los dos componentes de UI da React), dos quais haverá apenas dois – Store e Actions:

```
$ mkdir js/source/flux
$ touch js/source/flux/CRUDStore.js
$ touch js/source/flux/CRUDActions.js
```

Na arquitetura do Flux, pode haver muitas Stores (por exemplo, uma para dados do usuário, outra para configurações da aplicação etc.), mas vamos nos concentrar em apenas uma: a Store CRUD. A Store CRUD é uma lista de registros; nesse exemplo, são registros de tipos de vinho e o que você acha deles.

A CRUDStore não tem nada a ver com a React; com efeito, ela pode ser implementada como um objeto JavaScript simples:

```
/* @flow */

let data;
let schema;

const CRUDStore = {
  getData(): Array<Object> {
    return data;
  },

  getSchema(): Array<Object> {
    return schema;
  },
};

export default CRUDStore
```

Como podemos ver, a Store mantém a única fonte da verdade na

forma das variáveis de módulo locais `data` e `schema`, e as devolve com prazer a quem estiver interessado. A Store também permite que os dados sejam atualizados (não o esquema, que é constante durante a vida da aplicação):

```
  setData(newData: Array<Object>, commit: boolean = true) {
    data = newData;
    if (commit && 'localStorage' in window) {
      localStorage.setItem('data', JSON.stringify(newData));
    }
    emitter.emit('change');
  },
```

Nesse caso, além de atualizar o `data` local, a Store atualiza a área de armazenagem permanente que, nesse exemplo, é `localStorage`, mas também poderia ser uma requisição XHR para o servidor. Isso ocorre apenas no "commit" porque você nem sempre vai querer atualizar a área de armazenagem permanente. Por exemplo, quando fizer uma pesquisa, você vai querer que os dados mais recentes sejam o resultado da pesquisa, mas não quer armazenar esses resultados de modo permanente. O que acontecerá se houver uma queda de energia após uma chamada a `setData()` e você perder todos os dados, exceto os resultados da pesquisa?

Por fim, podemos ver nesse código que o evento "change" está sendo emitido. (Você verá mais informações sobre essa parte em breve.)

Outros métodos úteis que a Store pode oferecer são devolver o contador total de linhas de dados e os dados de uma única linha:

```
  getCount(): number {
    return data.length;
  },

  getRecord(recordId: number): ?Object {
    return recordId in data ? data[recordId] : null;
  },
```

Para iniciar a aplicação, é preciso inicializar a Store. Essa tarefa era tratada anteriormente em *app.jss*, mas, na verdade, é de

responsabilidade da Store agora; desse modo, você terá apenas um lugar que lida com os dados:

```
init(initialSchema: Array<Object>) {
  schema = initialSchema;
  const storage = 'localStorage' in window
    ? localStorage.getItem('data')
    : null;

  if (!storage) {
    data = [{}];
    schema.forEach(item => data[0][item.id] = item.sample);
  } else {
    data = JSON.parse(storage);
  }
},
```

A partir da agora, *app.js* inicia a aplicação a*ssim:*

```
// ...
import CRUDStore from './flux/CRUDStore';
import Whinepad from './components/Whinepad';
import schema from './schema';

CRUDStore.init(schema);

ReactDOM.render(
  <div>
    {/* mais JSX */}
    <Whinepad />
  {/* ... */}
);
```

Como podemos ver, depois que a Store é inicializada, <Whinepad> não precisa receber nenhuma propriedade. Os dados de que ele precisa estão disponíveis por meio de CRUDStore.getData() e a descrição dos dados é obtida com uma chamada a CRUDStore.getSchema().

Você pode estar se perguntando por que a própria Store lê os dados, mas depende de o esquema ser passado para ela externamente. É claro que você pode fazer a Store importar o módulo schema. Talvez, porém, faça sentido deixar que a aplicação trate o lugar de onde vem o esquema. É proveniente de um módulo, está fixo no código, é definido pelo usuário?

Eventos da Store

Você se lembra da parte `emitter.emit('change');`, quando a Store atualiza seus dados? Essa é uma maneira de a Store informar a qualquer módulo de UI interessado que os dados mudaram e que eles podem prosseguir e atualizar a si mesmos lendo os novos dados da Store. No entanto, como essa emissão de evento é implementada?

Há vários modos de implementar um padrão de inscrição para recepção de eventos – em sua essência, trata-se de reunir uma lista de partes interessadas (os inscritos) e, caso um evento de "publicação" ocorra, chamar a callback de cada inscrito (uma função fornecida no ato da inscrição por quem se inscreveu).

Em vez de fazer uma implementação por conta própria, vamos usar uma pequena biblioteca de código aberto chamada `fbemitter` para as partes referentes à inscrição para recepção de eventos:

```
$ npm i --save-dev fbemitter
```

Atualize o *.flowconfig*:

```
[ignore]
.*/fbemitter/node_modules/.*
# e assim por diante...

[include]
node_modules/classnames
node_modules/fbemitter
# e assim por diante...
```

A importação e a inicialização do emissor de eventos ocorrem no início do módulo Store:

```
/* @flow */

import {EventEmitter} from 'fbemitter';

let data;
let schema;
const emitter = new EventEmitter();

const CRUDStore = {
  // ...
};

export default CRUDStore
```

As duas tarefas do emissor são:

- reunir inscrições;
- notificar os inscritos (como vimos com emitter.emit('change') em setData()).

Você pode expor o acréscimo de inscritos como um método da Store, de modo que quem fizer a chamada não precise conhecer nenhum dos detalhes:

```
const CRUDStore = {
  // ...
  addListener(eventType: string, fn: Function) {
    emitter.addListener(eventType, fn);
  },
  // ...
};
```

Com isso, as funcionalidades de CRUDStore estão completas.

Usando a Store em <Whinepad>

O componente <Whinepad> é bem mais simples no mundo do Flux. A maior parte das simplificações é resultado da passagem de funcionalidades para CRUDActions (que veremos em breve), mas CRUDStore também ajuda. Não é mais necessário manter this.state.data. O único motivo pelo qual ele era necessário era permitir que fosse passado para <Excel>. Agora, porém, o <Excel> pode acessar a Store para obter os dados. De fato, <Whinepad> não precisa nem mesmo interagir com a Store. No entanto, vamos adicionar mais uma funcionalidade que exige a Store. A funcionalidade consiste em mostrar o total de registros no campo de pesquisa (Figura 8.2).

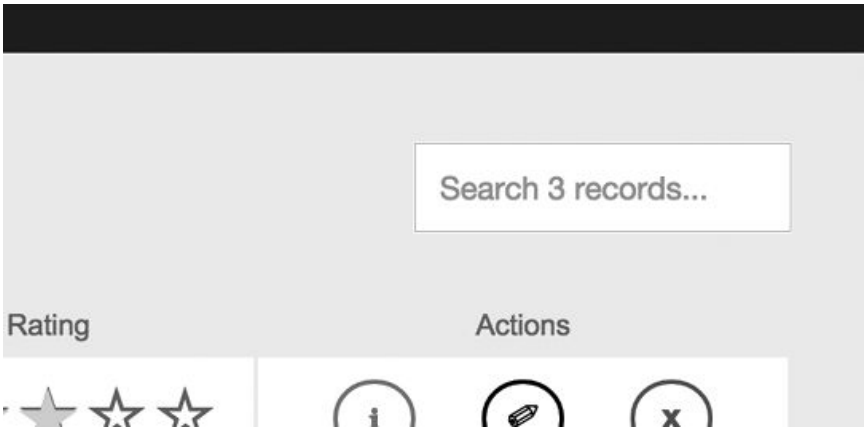


Figura 8.2 – Contador de registros no campo de pesquisa.

Anteriormente, o método constructor() de <Whinepad> definia o estado assim:

```
this.state = {
  data: props.initialData,
  addnew: false,
};
```

Agora não precisamos mais de data, mas devemos ter o contador, portanto faça a inicialização lendo da Store:

```
/* @flow */

// ...
import CRUDStore from '../flux/CRUDStore';
// ...

class Whinepad extends Component {
  constructor() {
    super();
    this.state = {
      addnew: false,
      count: CRUDStore.getCount(),
    };
  }
  /* ... */
}

export default Whinepad
```

Outra tarefa a ser feita no construtor é se inscrever para ser notificado de mudanças na Store, a fim de ter uma oportunidade para atualizar o contador total em this.state:

```
  constructor() {
    super();
    this.state = {
      addnew: false,
      count: CRUDStore.getCount(),
    };
    CRUDStore.addListener('change', () => {
      this.setState({
        count: CRUDStore.getCount(),
      })
    });
  }
```

Essa é toda a interação necessária com a Store. Sempre que os dados da Store, de algum modo, forem alterados (e `setData()` na `CRUDStore` for chamado), a Store emitirá um evento “change”. O `<Whinepad>` ouve o evento esse evento e atualiza o seu estado. Como já sabemos, definir o estado provoca uma nova renderização, portanto o método `render()` será chamado novamente. Esse método faz o seu trabalho usual, simplesmente compondo a UI com base em `state` e em `props`

```
  render() {
    return (
      {/* ... */}
        <input
          placeholder={this.state.count === 1
            ? 'Search 1 record...'
            : `Search ${this.state.count} records...`
          }
        />
      {/* ... */}
    );
  }
```

Outra tarefa inteligente que `<Whinepad>` pode fazer é implementar o método `shouldComponentUpdate()`. É possível que haja mudanças nos dados que não afetem o contador total (por exemplo, editar um registro ou um único campo de um registro). Nesse caso, o componente não precisa se renderizar novamente:

```
shouldComponentUpdate(newProps: Object, newState: State): boolean {
  return (
    newState.addnew !== this.state.addnew ||
    newState.count !== this.state.count
  );
}
```

Por fim, <Whinepad> não precisa mais passar as propriedades com os dados e o esquema para <Excel>.

Também não precisa se inscrever para onDataChange, pois todas as mudanças agora serão notificadas por meio de um evento “change” da Store. Assim, a parte relevante de render() em <Whinepad> é somente esta:

```
render() {
  return (
    {/* ... */}
    <div className="WhinepadDatagrid">
      <Excel />
    </div>
    {/* ... */}
  );
}
```

Usando a Store em <Excel>

De modo semelhante ao <Whinepad>, o <Excel> não precisa mais de propriedades. O construtor pode ler o schema da Store e armazená-lo como this.schema. Não há realmente nenhuma diferença entre armazenar em this.state.schema e em this.schema, exceto que state supõe certo grau de mudança, enquanto schema é constante.

Quanto aos dados, o this.state.data inicial é simplesmente lido da Store, não sendo mais recebido como uma propriedade.

Por fim, o construtor se inscreve para receber o evento “change” da Store para que o estado possa ser atualizado com os dados mais recentes (e disparar uma nova renderização):

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

```
    this.state = {
      data: CRUDStore.getData(),
      sortby: null, // schema.id
      descending: false,
      edit: null, // {row index, schema.id},
      dialog: null, // {type, idx}
    };
    this.schema = CRUDStore.getSchema();
    CRUDStore.addListener('change', () => {
      this.setState({
      data: CRUDStore.getData(),
      })
    });
  }
```

Isso é tudo que <Excel> precisa fazer para tirar proveito da Store. O método render() continua lendo de this.state para apresentar os dados, como antes.

Talvez você esteja se perguntando por que deve copiar os dados da Store para this.state. É possível que o método render() acesse a Store e leia diretamente daí? Certamente, isso é possível. No entanto, o componente perde a sua "pureza". Lembre-se de que um *componente puramente de renderização* é aquele que renderiza somente com base em props e state. Qualquer chamada de função em render() passa a se tornar suspeito – você nunca sabe que tipo de valores obterá de uma chamada externa. Torna-se mais difícil de depurar, e a aplicação passa a ser menos previsível: "Por que ela mostra 2 quando o estado contém 1? Ah, há uma chamada de função em render()."

Usando a Store em <Form>

O componente de formulário também aceita o esquema (como uma propriedade fields) e uma propriedade defaultValues para preencher previamente o formulário ou mostrar uma versão somente de leitura. Ambos os dados agora estão na Store. O formulário pode aceitar uma propriedade recordId e procurar o dado propriamente dito na

Store:

```
/* @flow */
import CRUDStore from '../flux/CRUDStore';
// ...
type Props = {
  readonly?: boolean,
  recordId: ?number,
};
class Form extends Component {
  fields: Array<Object>;
  initialData: ?Object;
  constructor(props: Props) {
    super(props);
    this.fields = CRUDStore.getSchema();
    if ('recordId' in this.props) {
      this.initialData = CRUDStore.getRecord(this.props.recordId);
    }
  }
  // ...
}
export default Form
```

O formulário não se inscreve para receber o evento “change” da Store porque não prevê uma mudança de dados enquanto esses estiverem sendo editados no formulário. Entretanto, isso poderia ser um cenário válido: suponha que outro usuário edite os dados ao mesmo tempo, ou o mesmo usuário abra a mesma aplicação em duas abas e edite o mesmo registro em ambas. Nesse caso, você poderá ouvir as mudanças de dados e alertar o usuário de que os dados estão sendo alterados em outro lugar.

Definindo o limite

Em que ponto você define o limite entre usar a Store do Flux *versus* usar propriedades como na implementação anterior ao Flux? A Store é uma implementação centralizada e conveniente para todas as necessidades relacionadas aos dados. Ela evita que você precise

passar propriedades por aí. No entanto, ela deixa os componentes menos reutilizáveis. Agora você não pode reutilizar Excel em um contexto totalmente diferente, pois o código está fixo para procurar dados em CRUDStore. Entretanto, desde que o novo contexto seja do tipo CRUD (o que é provável, pois, do contrário, por que você precisaria de uma tabela de dados possível de editar?), você poderá incluir a Store também. Lembre-se de que uma aplicação pode usar tantas Stores quantas forem necessárias.

É melhor que componentes de baixo nível como botões e entradas de formulário não tenham conhecimento da Store. Eles podem facilmente se resolver usando apenas propriedades. Qualquer tipo de componente que esteja entre os extremos – botões simples (como <Button>) e componentes-pais em geral (como <Whinepad>) – fazem parte de uma área cinzenta e cabe a você decidir. <Form> deve estar associado à Store CRUD como mostramos antes ou deve ser independente da Store e ser reutilizável em qualquer lugar? Faça uso de seu melhor julgamento, considerando a tarefa em mãos, e as possibilidades de reutilizar o que você está construindo no momento.

Actions

As *actions* são o modo como os dados na Store são alterados. Quando os usuários interagem com a View, eles executam uma ação que atualiza a Store; por sua vez, ela envia um evento às views interessadas nessa mudança.

Para implementar a CRUDActions que atualize a CRUDStore, você pode simplificar – basta usar outro objeto JavaScript comum:

```
/* @flow */
import CRUDStore from './CRUDStore';
const CRUDActions = {
  /* métodos */
};
export default CRUDActions
```

CRUDActions

Que tipo de métodos devem ser implementados no módulo CRUDActions? Bem, os suspeitos usuais – create(), delete(), update..., exceto que nessa aplicação, é possível atualizar um registro todo ou apenas um único campo, portanto vamos implementar updateRecord() e updateField():

```
/* @flow */
/* ... */
const CRUDActions = {
  create(newRecord: Object) {
    let data = CRUDStore.getData();
    data.unshift(newRecord);
    CRUDStore.setData(data);
  },
  delete(recordId: number) {
    let data = CRUDStore.getData();
    data.splice(recordId, 1);
    CRUDStore.setData(data);
  },
  updateRecord(recordId: number, newRecord: Object) {
    let data = CRUDStore.getData();
    data[recordId] = newRecord;
    CRUDStore.setData(data);
  },
  updateField(recordId: number, key: string, value: string|number) {
    let data = CRUDStore.getData();
    data[recordId][key] = value;
    CRUDStore.setData(data);
  },
  /* ... */
};
```

Tudo parece ser razoavelmente trivial: lemos os dados atuais da Store, manipulamos esses dados de alguma forma (atualizar, apagar, adicionar/criar) e então os escrevemos de volta.

Você não precisa do R de CRUD, pois ele é disponibilizado pela Store.

Pesquisando e ordenando

Na implementação anterior, o componente <Whinepad> era responsável pela pesquisa nos dados. Isso ocorria simplesmente porque o campo de pesquisa, por acaso, estava no render() do componente. Contudo, ele deveria estar realmente em outro lugar mais próximo aos dados.

De modo semelhante, a funcionalidade de ordenação fazia parte do componente <Excel> porque era aí que estavam os cabeçalhos da tabela, e os handlers onclick dos cabeçalhos faziam a ordenação. Mais uma vez, porém, a ordenação será melhor implementada se estiver mais próxima dos dados.

Você pode discutir se a pesquisa e a ordenação dos dados pertencem a Actions ou a Store. Os dois lugares parecem ser apropriados. Nessa implementação, porém, vamos deixar a Store razoavelmente simplória. Ela pode executar apenas get e set, e será responsável também por enviar eventos. O módulo Actions é o lugar em que a manipulação de dados ocorre, portanto vamos migrar a ordenação e a pesquisa dos componentes da UI para o módulo CRUDActions:

```
/* @flow */
/* ... */
const CRUDActions = {
  /* ... métodos CRUD ... */
  _preSearchData: null,
  startSearching() {
    this._preSearchData = CRUDStore.getData();
  },
  search(e: Event) {
    const target = ((e.target: any): HTMLInputElement);
    const needle: string = target.value.toLowerCase();
```

```
    if (!needle) {
      CRUDStore.setData(this._preSearchData);
      return;
    }
    const fields = CRUDStore.getSchema().map(item => item.id);
    if (!this._preSearchData) {
      return;
    }
    const searchdata = this._preSearchData.filter(row => {
      for (let f = 0; f < fields.length; f++) {
        if (row[fields[f]].toString().toLowerCase().indexOf(needle) > -1) {
          return true;
        }
      }
      return false;
    });
    CRUDStore.setData(searchdata, /* commit */ false);
  },
  _sortCallback(
    a: (string|number), b: (string|number), descending: boolean
  ): number {
    let res: number = 0;
    if (typeof a === 'number' && typeof b === 'number') {
      res = a - b;
    } else {
      res = String(a).localeCompare(String(b));
    }
    return descending ? -1 * res : res;
  },
  sort(key: string, descending: boolean) {
    CRUDStore.setData(CRUDStore.getData().sort(
      (a, b) => this._sortCallback(a[key], b[key], descending)
    ));
  },
};
```

Com isso, as funcionalidades de CRUDActions estão completas. Vamos ver como ela é usada pelos componentes <Whinepad> e <Excel>.

> Você pode argumentar que esta parte da função sort() não pertence a CRUDActions:

```
search(e: Event) {
  const target = ((e.target: any): HTMLInputElement);
  const needle: string = target.value.toLowerCase();
  /* ... */
}
```

Talvez o módulo Actions não devesse saber nada sobre a UI, de modo que a assinatura “apropriada” deveria ser algo como:

```
search(needle: string) {
  /* ... */
}
```

Esse é um argumento válido, e você pode seguir por esse caminho também. Seria apenas um pouco mais inconveniente para <Whinepad> e exigiria um pouco mais que apenas um <input onChange="CRUDActions.search">.

Usando Actions em <Whinepad>

Vamos dar uma olhada na aparência de <Whinepad> agora, depois de usarmos as Actions do Flux. Inicialmente, é óbvio que ele deve incluir o módulo Actions:

```
/* @flow */

/* ... */
import CRUDActions from '../flux/CRUDActions';
/* ... */

class Whinepad extends Component {/* ... */}

export default Whinepad
```

Você deve se lembrar que Whinepad é responsável pela adição de novos registros e por pesquisar os registros existentes (Figura 8.3).

Figura 8.3 – Área do Whinepad com responsabilidade pelos dados.

Em se tratando de adicionar novos registros, o Whinepad anteriormente era responsável por manipular o seu próprio this.state.data…

```
_addNew(action: string) {
  if (action === 'dismiss') {
    this.setState({addnew: false});
  } else {
    let data = Array.from(this.state.data);
    data.unshift(this.refs.form.getData());
    this.setState({
      addnew: false,
      data: data,
    });
    this._commitToStorage(data);
  }
}
```

…mas agora, essa responsabilidade foi passada para o módulo Actions, que deve atualizar a Store (também conhecida como a única fonte da verdade):

```
_addNew(action: string) {
  this.setState({addnew: false});
  if (action === 'confirm') {
    CRUDActions.create(this.refs.form.getData());
  }
}
```

Não há mais estados para manter nem dados para manipular. Se houver uma ação do usuário, basta delegar e deixar que ela siga o fluxo de dados unidirecional.

O mesmo vale para a pesquisa. Embora antes isso fosse feito no próprio this.state.data do componente, agora temos somente:

```
<input
  placeholder={this.state.count === 1
    ? 'Search 1 record...'
    : `Search ${this.state.count} records...`
  }
```

```
  onChange={CRUDActions.search.bind(CRUDActions)}
  onFocus={CRUDActions.startSearching.bind(CRUDActions)} />
```

Usando Actions no <Excel>

O Excel é o consumidor da ordenação, da remoção e da atualização fornecidas pelo CRUDActions. Para você se lembrar, essa era a aparência da remoção anteriormente:

```
  _deleteConfirmationClick(action: string) {
    if (action === 'dismiss') {
      this._closeDialog();
      return;
    }
    const index = this.state.dialog ? this.state.dialog.idx : null;
    invariant(typeof index === 'number', 'Unexpected dialog state');
    let data = Array.from(this.state.data);
    data.splice(index, 1);
    this.setState({
      dialog: null,
      data: data,
    });
    this._fireDataChange(data);
  }
```

Agora esse código é assim:

```
  _deleteConfirmationClick(action: string) {
    this.setState({dialog: null});
    if (action === 'dismiss') {
      return;
    }
    const index = this.state.dialog && this.state.dialog.idx;
    invariant(typeof index === 'number', 'Unexpected dialog state');
    CRUDActions.delete(index);
  }
```

Um evento de mudança de dados não é mais disparado porque ninguém está ouvindo Excel; todos os interessados estão sintonizados com a Store. Além disso, não há mais a necessidade de manipular this.state.data. Em vez disso, vamos deixar o módulo Actions fazer a manipulação e, em seguida, fazer uma atualização

quando a Store enviar um evento.

É semelhante quando se trata da ordenação e da atualização de registros. Toda a manipulação de dados se transforma em chamadas únicas aos métodos de CRUDActions:

```
/* @flow */

/* ... */
import CRUDActions from '../flux-imm/CRUDActions';
/* ... */

class Excel extends Component {
  /* ... */

  _sort(key: string) {
    const descending = this.state.sortby === key && !this.state.descending;
    CRUDActions.sort(key, descending);
    this.setState({
      sortby: key,
      descending: descending,
    });
  }

  _save(e: Event) {
    e.preventDefault();
    invariant(this.state.edit, 'Messed up edit state');
    CRUDActions.updateField(
      this.state.edit.row,
      this.state.edit.key,
      this.refs.input.getValue()
    );
    this.setState({
      edit: null,
    });
  }

  _saveDataDialog(action: string) {
    this.setState({dialog: null});
    if (action === 'dismiss') {
      return;
    }
    const index = this.state.dialog && this.state.dialog.idx;
    invariant(typeof index === 'number', 'Unexpected dialog state');
    CRUDActions.updateRecord(index, this.refs.form.getData());
```

```
  }
  /* ... */
};
export default Excel
```

A versão totalmente convertida da aplicação Whinepad que utiliza Flux está disponível no repositório de código do livro (*https://github.com/stoyan/reactbook/*).

Recapitulando o Flux

É isso. A aplicação agora foi migrada para usar a arquitetura do Flux (ou uma espécie de versão composta manualmente). Temos a View enviando Actions que atualizam a Store única, a qual envia eventos. A View então ouve esses eventos da Store e se atualiza. É um ciclo completo.

Existem outras extensões para essa ideia, que podem se mostrar úteis à medida que a aplicação crescer.

Não é somente a View que pode enviar Actions (Figura 8.4). As Actions também podem ser enviadas do servidor. Talvez alguns dados tenham se tornado obsoletos. Quem sabe outros usuários afetaram os dados e a aplicação descobriu ao se sincronizar com o servidor. Ou talvez o tempo simplesmente tenha passado e alguma ação precisou ser executada (seu prazo para comprar as entradas reservadas venceu, a sessão expirou, comece novamente!).

Quando você se vir em uma situação em que tenha Actions originadas de vários lugares, a ideia de um *único Dispatcher* torna-se conveniente (Figura 8.5). O Dispatcher é responsável pelo piping de todas essas Actions para a Store (ou as Stores).

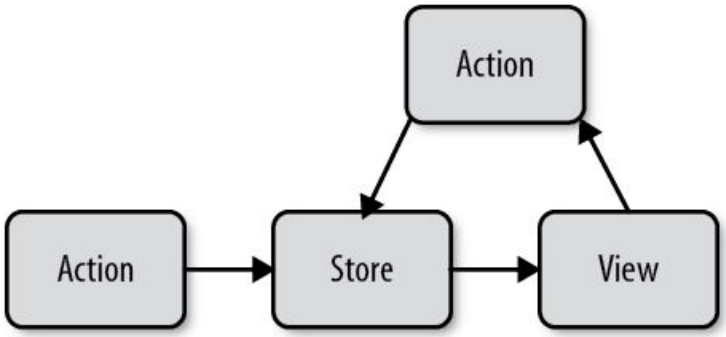


Figura 8.4 – Mais Actions.

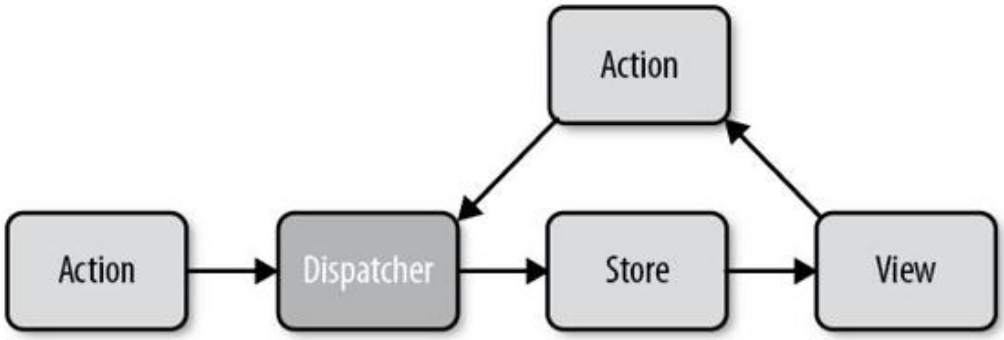


Figura 8.5 – Um Dispatcher.

Em uma aplicação mais interessante, você acabará com diferentes Actions provenientes da UI, do servidor ou de outro lugar, e várias Stores, cada uma responsável pelos seus próprios dados (Figura 8.6).

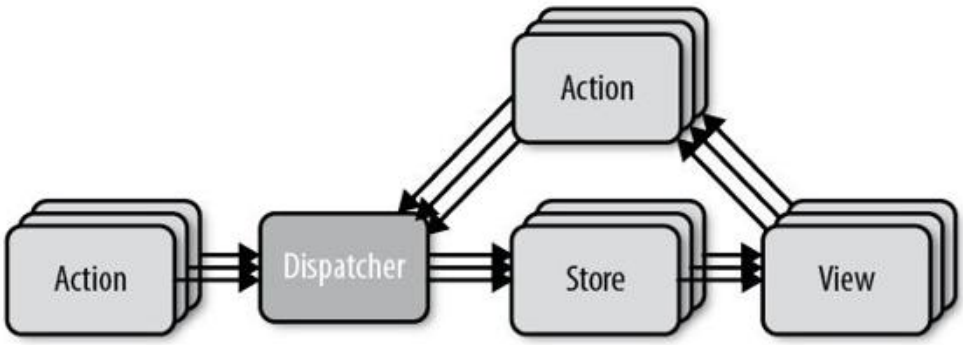


Figura 8.6 – Um fluxo complicado, porém ainda unidirecional.

Há muitas soluções de código aberto quando se trata de implementar uma arquitetura de Flux. Contudo, você sempre pode tentar começar com uma implementação pequena e crescer à

medida que avançar, seja pela evolução de uma solução doméstica ou escolhendo uma das ofertas de código aberto que agora você já sabe como avaliar.

Imutável

Vamos concluir o livro com uma pequena alteração nas partes do Flux – na Store e nas Actions. Vamos passar para uma estrutura de dados imutável para os registros de vinhos. *Imutável* é um tema comum que você verá associado a aplicações React, apesar de não ter nada a ver com a React propriamente dita.

Um objeto imutável é criado uma vez e não pode ser alterado. Objetos imutáveis geralmente são mais simples de entender e é mais fácil raciocinar com eles. Por exemplo, strings com frequência são implementadas como objetos imutáveis internamente.

Em JavaScript, podemos usar o pacote NPM immutable para tirar proveito da ideia:

```
$ npm i --save-dev immutable
```

Além disso, acrescente o seguinte em seu *.flowconfig*:

```
# ....

[include]
# ...
node_modules/immutable

# ...
```

A documentação completa da biblioteca está disponível online (*http://facebook.github.io/immutable-js/*).

Como toda manipulação de dados agora ocorre nos módulos Store e Actions, esses são realmente os únicos dois lugares que devem ser atualizados.

Dados imutáveis na Store

A biblioteca immutable oferece estruturas de dados List, Stack e Map,

entre outras.

Vamos escolher List, pois é a mais parecida com o array que a aplicação estava usando antes:

```
/* @flow */

import {EventEmitter} from 'fbemitter';
import {List} from 'immutable';

let data: List<Object>;
let schema;
const emitter = new EventEmitter();
```

Observe o novo tipo de data – é uma List imutável.

Crie novas listas usando let list = List() e passando alguns valores iniciais. Vamos ver como a Store inicializa a lista agora:

```
const CRUDStore = {

  init(initialSchema: Array<Object>) {
    schema = initialSchema;
    const storage = 'localStorage' in window
      ? localStorage.getItem('data')
      : null;
    if (!storage) {
      let initialRecord = {};
      schema.forEach(item => initialRecord[item.id] = item.sample);
      data = List([initialRecord]);
    } else {
      data = List(JSON.parse(storage));
    }
  },

  /* .. */
};
```

Como podemos ver, a lista é inicializada com um array. A partir daí você deve usar a API de lista para manipular os dados. Depois de criada, a lista será imutável e não poderá ser alterada. (Entretanto, toda a manipulação ocorre em CRUDActions, como veremos em breve.)

Além da inicialização e da anotação de tipo, não há muitas mudanças na Store – tudo que ela faz é set e get nos dados.

Uma pequena mudança em `getCount()` se faz necessária porque a lista imutável não tem uma propriedade `length`:

```
// Antes
getCount(): number {
  return data.length;
},

// Depois
getCount(): number {
  return data.count(); // `data.size` também funciona
},
```

Por fim, fazemos uma atualização em `getRecord()`, porque a biblioteca `immutable` não é capaz de oferecer um acesso com chave como o fazem os arrays embutidos:

```
// Antes
getRecord(recordId: number): ?Object {
  return recordId in data ? data[recordId] : null;
},

// Depois
getRecord(recordId: number): ?Object {
  return data.get(recordId);
},
```

Manipulação de dados imutáveis

Lembre-se de como os métodos de string funcionam em JavaScript:

```
let hi = 'Hello';
let ho = hi.toLowerCase();
hi; // "Hello"
ho; // "hello"
```

A string atribuída a `hi` não mudou. Em vez disso, uma nova string foi criada.

É semelhante ao que acontece com uma lista imutável:

```
let list = List([1, 2]);
let newlist = list.push(3, 4);
list.size; // 2
newlist.size; // 4
```

```
list.toArray(); // Array [ 1, 2 ]
newlist.toArray() // Array [ 1, 2, 3, 4 ]
```

Você notou o método push()? Listas imutáveis se comportam, em sua maior parte, como arrays, portanto map(), forEach() e outros métodos estão disponíveis. Essa é parte dos motivos pelos quais os componentes de UI não precisam realmente mudar. (Esclarecendo totalmente: apenas uma mudança foi necessária – um acesso a array com colchetes.) A outra parte do motivo é que, conforme mencionamos, os dados agora são tratados principalmente na Store e em Actions.

Então como o módulo Actions foi afetado pela mudança na estrutura de dados? Não muito, na verdade. Como a lista imutável oferece sort() e filter(), nenhuma mudança foi necessária nas partes relacionadas à ordenação e pesquisa. As únicas alterações estão nos métodos create(), delete() e nos dois update*().

Considere o método delete():

```
/* @flow */

import CRUDStore from './CRUDStore';
import {List} from 'immutable';

const CRUDActions = {

  /* ... */

  delete(recordId: number) {
    // Antes:
    // let data = CRUDStore.getData();
    // data.splice(recordId, 1);
    // CRUDStore.setData(data);

    // Depois:
    let data: List<Object> = CRUDStore.getData();
    CRUDStore.setData(data.remove(recordId));
  },

  /* ... */

};

export default CRUDActions;
```

O splice() de JavaScript talvez tenha um nome um pouco estranho e devolve uma parte extraída do array, ao mesmo tempo que modifica o original. Tudo isso faz com que seja um pouco confuso usá-lo em uma só linha. A lista imutável, por outro lado, pode estar em uma só

linha. Se não fosse pela gloriosa anotação de tipo, ela poderia ser simplesmente:

```
  delete(recordId: number) {
    CRUDStore.setData(CRUDStore.getData().remove(recordId));
  },
```

No mundo dos imutáveis, o método `remove()`, cujo nome é apropriado, não afeta a lista original. O original é imutável. O método `remove()` devolve uma nova lista com um item removido. Então você deve atribuir a nova lista como o novo dado a ser salvo na Store.

Os outros métodos de manipulação de dados são semelhantes e também mais simples que trabalhar com arrays:

```
  /* ... */
  create(newRecord: Object) { // unshift() - como arrays
    CRUDStore.setData(CRUDStore.getData().unshift(newRecord));
  },
  updateRecord(recordId: number, newRecord: Object) { // set(), pois não há []
    CRUDStore.setData(CRUDStore.getData().set(recordId, newRecord));
  },
  updateField(recordId: number, key: string, value: string|number) {
    let record = CRUDStore.getData().get(recordId);
    record[key] = value;
    CRUDStore.setData(CRUDStore.getData().set(recordId, record));
  },
  /* ... */
```

Pronto! O que você tem agora é uma aplicação que usa:

- componentes React para definir a UI;
- JSX para formar os componentes;
- Flux para organizar o fluxo de dados;
- dados imutáveis;
- Babel para utilizar os recursos mais recentes do ECMAScript;
- Flow para verificação de tipos e erros de sintaxe;
- ESLint para verificar outros erros e convenções;
- Jest para testes de unidade.

Como sempre, você pode conferir a versão funcional completa número 3 da aplicação Whinepad (“A versão imutável”) no repositório de código do livro (*https://github.com/stoyan/reactbook/*). Além disso, você pode interagir com a aplicação em *http://whinepad.com*.

Sobre o autor

Stoyan Stefanov é engenheiro do Facebook. Anteriormente na Yahoo, ele criou a smush.it – uma ferramenta de otimização de imagens online – e foi arquiteto da ferramenta de desempenho YSlow 2.0. Stoyan é autor de *Padrões JavaScript* (Novatec, 2010) e de *Object-Oriented JavaScript* (Packt Publishing, 2008), colaborador do *Even Faster Web Sites* e *High-Performance JavaScript*, blogueiro (*http://phpied.com*) e palestrante frequente em conferências, incluindo Velocity, JSConf, Fronteers e muitas outras.

Colofão

O animal na capa de *Primeiros Passos com React* é uma ave ‘i’iwi (pronuncia-se i-í-vi) ou Ívi, também conhecida como honeycreeper (trepador do mel) escarlate havaiano. A filha do autor escolheu essa ave depois de fazer um trabalho escolar sobre ela. O ‘i’iwi é a terceira ave terrestre nativa mais comum das ilhas havaianas, embora muitas espécies de sua família, a *Fringillidae*, estejam ameaçadas ou foram extintas. Essa pequena ave de cores brilhantes é um símbolo reconhecido no Havaí e as maiores colônias vivem nas ilhas de Havaí, Mauí e Kauai.

Os ‘i’iwis adultos são predominantemente escarlates, com asas e cauda negras e um bico longo e curvo. A cor vermelha brilhante contrasta bem com a folhagem verde ao redor, fazendo com que o ‘i’iwi seja facilmente identificável na natureza. Embora suas penas tenham sido intensamente utilizadas para decorar mantos e capacetes da nobreza havaiana, sua extinção foi evitada porque ele era considerado menos sagrado que seu parente, o mamo havaiano.

A dieta do ‘i’iwi’ é constituída em sua maior parte de néctar de flores e da árvore ‘ōhi‘a lehua, embora ocasionalmente ele se alimente de pequenos insetos. Essa ave também faz migração altitudinal; ela segue o desenvolvimento das flores à medida que elas desabrocham em altitudes cada vez mais elevadas ao longo do ano. Isso significa que elas podem migrar entre as ilhas, embora raramente estejam em O’ahu e em Moloka’i por causa da destruição do *habitat*, e estejam extintas em Lāna‘i desde 1929.

Muito esforço tem sido investido para preservar a população atual de ‘i’iwis; as aves são muito suscetíveis à varíola e à gripe aviária e sofrem dos efeitos do desmatamento e de espécies de plantas

invasoras. Porcos selvagens criam lamaçais que abrigam pernilongos, portanto isolar áreas florestais tem ajudado a controlar as doenças transmitidas por esses insetos; além disso, há projetos em andamento que tentam restaurar as florestas e remover espécies de plantas não nativas, dando às flores preferidas pelos ‘i’iwis uma chance de florescer.

Muitos dos animais nas capas dos livros da O’Reilly estão ameaçados; todos eles são importantes para o mundo. Para saber mais sobre como você pode ajudar, acesse *animals.oreilly.com*.

A imagem da capa foi extraída do livro *Wood’s Illustrated Natural History*.


UMA INTRODUÇÃO À PROGRAMAÇÃO DE COMPUTADORES
COM EXEMPLOS E EXERCÍCIOS PARA INICIANTES
Lógica de Programação e Algoritmos com JavaScript
novatec
Edécio Fernando Iepsen

Lógica de Programação e Algoritmos com JavaScript

Iepsen, Edécio Fernando

9788575226575

320 páginas

Os conteúdos abordados em Lógica de Programação e Algoritmos são fundamentais a todos aqueles que desejam ingressar no universo da Programação de Computadores. Esses conteúdos, no geral, impõem algumas dificuldades aos iniciantes. Neste livro, o autor utiliza sua experiência de mais de 15 anos em lecionar a disciplina de Algoritmos em cursos de graduação, para trabalhar o assunto passo a passo. Cada capítulo foi cuidadosamente planejado a fim de evitar a sobrecarga de informações ao leitor, com exemplos e exercícios de fixação para cada assunto. Os exemplos e exercícios são desenvolvidos em JavaScript, linguagem amplamente utilizada no desenvolvimento de páginas para a internet. Rodar os programas JavaScript não exige nenhum software adicional; é preciso apenas abrir a página em seu navegador favorito. Como o aprendizado de Algoritmos ocorre a partir do estudo das técnicas de programação e da prática de exercícios, este livro pretende ser uma importante fonte de conhecimentos para você ingressar nessa fascinante área da programação de computadores. Assuntos abordados no livro: Fundamentos de Lógica de

Programação Programas de entrada, processamento e saída Integração do código JavaScript com as páginas HTML Estruturas condicionais e de repetição Depuração de Programas JavaScript Manipulação de listas de dados (vetores) Operações sobre cadeias de caracteres (strings) e datas Eventos JavaScript e funções com passagem de parâmetros Persistência dos dados de um programa com localStorage Inserção de elementos HTML via JavaScript com referência a DOM No capítulo final, um jogo em JavaScript e novos exemplos que exploram os recursos discutidos ao longo do livro são apresentados com comentários e dicas, a fim de incentivar o leitor a prosseguir nos estudos sobre programação.

O'REILLY

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações nativas de nuvem

Bilgin Ibryam
Roland Huß

novatec

Padrões para Kubernetes

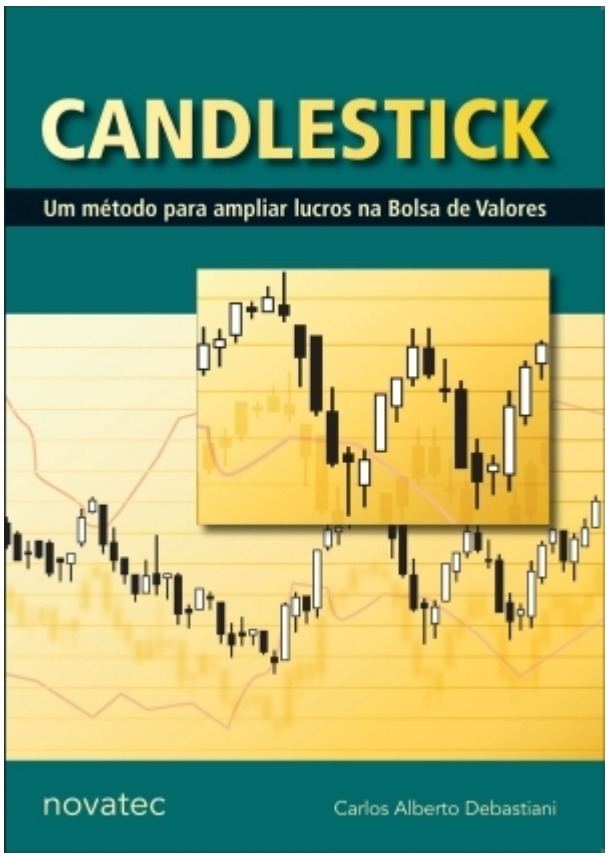
Ibryam, Bilgin

9788575228159

272 páginas

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões: • Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres. • Padrões comportamentais, que exploram conceitos mais

específicos para administrar contêineres e interações com a plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).


CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

Compre agora e leia


AVALIANDO
EMPRESAS
INVESTINDO
EM AÇÕES
A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS
novatec
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

Compre agora e leia


Marcos Abe
MANUAL DE
ANÁLISE
TÉCNICA
ESSÊNCIA E ESTRATÉGIAS AVANÇADAS
TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE
novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.