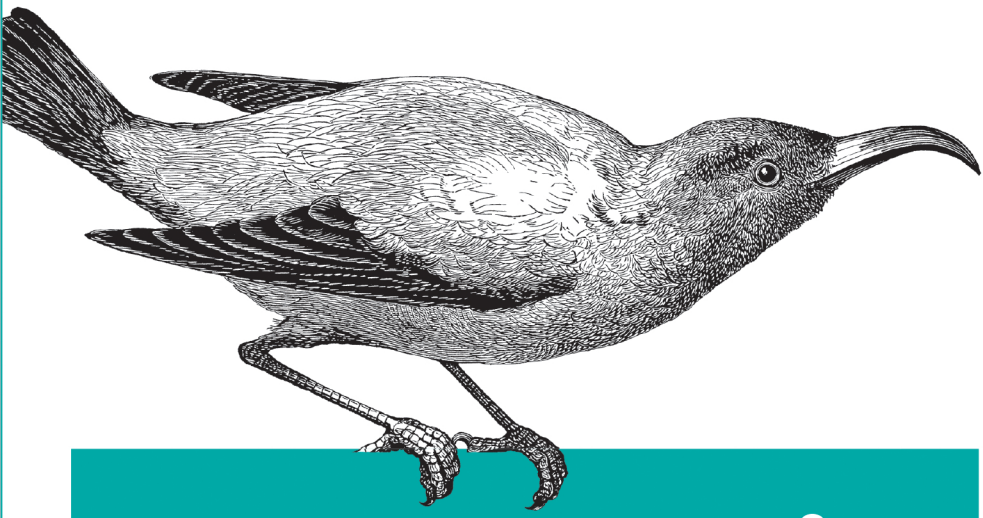
O'REILLY®



React.js

БЫСТРЫЙ СТАРТ

ПИТЕР®

Стоян Стефанов

React: Up & Running

Building Web Applications

Stoyan Stefanov

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Стоян Стефанов

React.js

БЫСТРЫЙ СТАРТ

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2017

ББК 32.988.02-018
УДК 004.738.5
С79

Стефанов Стоян

С79 React.js. Быстрый старт. — СПб.: Питер, 2017. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-03003-8

Незаменимая вводная книга по технологии React для взыскательных JavaScript-разработчиков. Все самое интересное о сверхпопулярном инструменте от компании Facebook. В книге рассмотрены основные концепции высокопроизводительного программирования при помощи React, реальные примеры кода и доступные блок-схемы. Прочитав ее, вы научитесь:

- Создавать и использовать собственные компоненты React наряду с универсальными компонентами DOM.
- Писать компоненты для таблиц данных, позволяющие редактировать, сортировать таблицу, выполнять в ней поиск и экспортировать ее содержимое.
- Использовать дополнительный синтаксис JSX в качестве альтернативы для вызовов функций.
- Запускать низкоуровневый гибкий процесс сборки, который освободит вам время и поможет сосредоточиться на работе с React.
- Работать с инструментами ESLint, Flow и Jest, позволяющими проверять и тестировать код по мере разработки приложения.
- Обеспечивать коммуникацию между компонентами при помощи Flux.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1491931820 англ.
ISBN 978-5-496-03003-8

Authorized Russian translation of the English edition of React: Up & Running,
ISBN 9781491931820 © 2016 Stoyan Stefanov
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same

Оглавление

Посвящается Еве, Златине
и Натали

Предисловие

Вот и еще одна по-волшебному теплая калифорнийская ночь. Легкий океанский бриз только способствует ощущению абсолютного блаженства. Место: Лос-Анджелес; время: двухтысячные годы. Я готовлюсь подключить по протоколу FTP свое небольшое веб-приложение под названием CSSsprites.com к своему серверу и выпустить его в окружающий мир. Последние несколько вечеров, работая над приложением, я задавался вопросом: почему на отладку основной логики работы приложения тратилось всего 20 % усилий, а 80 % уходило на доведение до ума пользовательского интерфейса? Сколько нужно создать дополнительного инструментария, чтобы не приходилось всякий раз пользоваться методом `getElementById()` и переживать за состояние приложения? (Подгрузил ли пользователь что-то на страницу? Что, произошла ошибка? А это диалоговое окно все еще на экране?) Почему на разработку пользовательского интерфейса уходит так много времени? И что происходит со всеми такими разными браузерами? И постепенно мое блаженство таяло, превращаясь в раздражение.

Перенесемся в март 2015 года, на Facebook-конференцию F8. Моя команда готовится сообщить о двух полностью переписанных веб-приложениях: предлагаемом нами средстве работы со сторонними комментариями и сопутствующем инструменте их модерации. По сравнению с моим небольшим приложением CSSsprites.com, здесь речь уже шла о полнофункциональных веб-приложениях с куда более существенными возможностями, намного большей эффективностью и безумными объемами трафика. Тем не менее разработка велась с превеликим удовольствием. Участники, не знакомые с приложением (некоторые даже не знакомые с JavaScript и CSS), могли высказать свои предложения насчет возможностей и каких-либо улучшений его различных частей, и мы тут же, без промедления и особых усилий их внедряли. Глядя на это, один из участников сказал: «Вот теперь я понимаю, что значит любить свою работу!»

Так что же произошло за это время? Появилась технология React.

React представляет собой библиотеку для создания пользовательских интерфейсов, которая помогает вам определить пользовательский интерфейс раз и навсегда. Затем при изменении состояния приложения пользовательский интерфейс перестраивается для реагирования на изменения — и вам не нужно выполнять какие-либо доработки. Ведь вы уже определили пользовательский интерфейс. Определили? Больше похоже на то, что вы его *объявили*. А для создания большого полнофункционального приложения вы воспользовались небольшими управляемыми компонентами. Теперь уже половина внутреннего кода ваших функций не занимается охотой на DOM-узлы, и вам остается лишь отслеживать состояние `state` вашего приложения (с помощью обычных объектов JavaScript), а все остальное происходит в соответствии с этим состоянием.

Изучение React воздастся сторицей — вы освоите всего одну библиотеку, а использовать ее будете для создания всего нижеперечисленного:

- веб-приложений;
- приложений для работы под управлением iOS и Android;
- canvas-приложений;
- TV-приложений;
- обычных приложений для настольных машин.

Используя ту же самую идею создания компонентов и пользовательских интерфейсов, вы сможете конструировать приложения, приспособленные под исходную систему с присущими ей производительностью и элементами управления (реальными, а не их копиями). Речь не идет о «единожды созданном и повсеместно запускаемом коде» (мы уже наступали на эти грабли), мы говорим о «единожды освоенном и повсеместно используемом средстве».

Короче говоря, изучение React позволяет сэкономить 80 % вашего времени и сконцентрироваться на истинной сути приложения (то есть на реальных задачах, для решения которых оно предназначено).

О чем эта книга

Книга посвящена изучению React с позиции веб-разработчика. В первых трех главах изучение начинается с использования пустого HTML-файла, на основе которого выстраивается весь остальной код. Это позволяет сосредоточиться на изучении React, не отвлекаясь на новый синтаксис или на применение дополнительных инструментальных средств.

В главе 4 дается введение в JSX — отдельную дополнительную технологию, обычно используемую в связке с React.

Далее перейдем к изучению приемов разработки реального приложения и освоению дополнительных инструментальных средств, которые могут оказать помощь в этом деле. В их числе средства для создания пакетов JavaScript (Browserify), для блочного тестирования (Jest), для проверки кода (ESLint), проверки соответствия типов (Flow), для организации потока данных в приложении (Flux) и для использования неизменяемых данных (Immutable.js). Рассмотрение дополнительных технологических средств сведено к минимуму, чтобы основное внимание уделялось React, а знакомство с этими средствами поможет осознанно выбирать их для решения ваших задач.

Успехов вам в изучении React, пусть оно будет приятным и плодотворным!

Соглашения, используемые в книге

В книге используются следующие условные обозначения.

Курсив

Применяется для обозначения новых понятий, а также в примерах кода для выделения элементов, которые должны быть заменены значениями, предоставляемыми пользователем или определяемыми контекстом.

`Моноширинный шрифт`

Используется для текста листингов.

Применяется внутри абзацев для выделения таких элементов программ, как имена переменных или функций, названия баз данных, типов данных, имена переменных среды, инструкции и ключевые слова.

Моноширинный курсивный шрифт

Показывает команды или другой текст, который должен быть заменен значением, предоставляемым пользователем или определяемым контекстом.

Шрифт без засечек

Используется для обозначения имен файлов и каталогов, а также путей и URL-адресов.

Этот элемент содержит совет или предложение.

Такой элемент указывает на примечание общего характера.

Использование примеров кода

Весь дополнительный материал (примеры кода, упражнения и т. д.) можно загрузить по адресу https://github.com/stoyan/reactbook.

Эта книга призвана помочь вам в вашей работе. Примеры кода из данной книги вы можете использовать в своих программах и документации. Если объем кода несущественный, связываться с нами для получения разрешения не нужно. Например, для написания программы, использующей несколько фрагментов кода из этой книги, разрешения не требуется. А вот для продажи или распространения компакт-диска с примерами из книг издательства O'Reilly нужно получить разрешение. Ответы на вопросы путем цитирования этой книги и ссылок на примеры кода не требуют разрешения. Но для включения объемных примеров кода из этой книги в документацию по вашему программному продукту разрешение понадобится.

Мы приветствуем указание ссылки на источник, но не делаем это обязательным требованием. Такая ссылка обычно включает название книги, имя автора, название издательства и ISBN. Например, «React: Up & Running, автор Стоян Стефанов (Stoyan Stefanov) (O'Reilly). Copyright 2016 Stoyan Stefanov, 978-1-491-93182-0».

Если вам покажется, что использование кода примеров выходит за рамки оговоренных выше условий и разрешений, непременно свяжитесь с нами по адресу permissions@oreilly.com.

Благодарности

Хочу поблагодарить всех, кто вычитывал черновые варианты этой книги и поделился своими предложениями по внесению исправлений: Андреа Маноле (Andrea Manole), Ильяна Пейчева (Iliyan Peychev), Костадина Илова (Kostadin Ilov), Марка Дуппенталера (Mark Duppenthaler), Стефана Албера (Stephan Alber), Асена Божилова (Asen Bozhilov).

Спасибо всем сотрудникам Facebook, работавшим над React (или применявшим эту библиотеку) и каждый день отвечавшим на мои вопросы. Я также благодарен всему React-сообществу, которое создает великолепные инструментальные средства, библиотеки, публикует статьи и создает шаблоны применения этой технологии.

Премного благодарен Джордану Уалке (Jordan Walke).

Спасибо всем сотрудникам издательства O'Reilly, содействовавшим выходу этой книги: Мегу Фоли (Meg Foley), Киму Коферу (Kim Cofer), Николь Шелби (Nicole Shelby) и многим другим.

Спасибо Явору Ватчкову (Yavor Vatchkov), дизайнеру пользовательского интерфейса, примененного в примере приложения, разрабатываемого в данной книге (работу приложения можно оценить на сайте whinepad.com).

Об авторе

Стоян Стефанов (Stoyan Stefanov) работает инженером в компании Facebook. Ранее в компании Yahoo! он был создателем интерактивного средства оптимизации изображений smush.it, а также средства выявления проблем производительности YSlow 2.0. Стоян — автор книг *JavaScript Patterns* (O'Reilly, 2010) и *Object-Oriented JavaScript* (Packt Publishing, 2008), соавтор публикаций *Even Faster Web Sites* и *High-Performance JavaScript*, блогер и частый докладчик на многих конференциях, в числе которых Velocity, JSConf и Fronteers.

1 Hello World

Итак, приступим к освоению процесса разработки приложения с использованием React. В этой главе мы рассмотрим установку React и вы напишете свое первое веб-приложение Hello World.

Установка

Для начала нужно получить копию библиотеки React. К счастью, сделать это проще простого.

Перейдите по адресу http://reactjs.com (оттуда вас перенаправят на официальную страницу GitHub), нажмите кнопку Download (Загрузить), потом кнопку Download Starter Kit (Загрузить начальный инструментарий) — и получите копию ZIP-файла. Распакуйте и скопируйте каталог, содержащийся в загрузке, в то место, где вы его легко сможете найти.

Например:

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

Теперь ваш рабочий каталог (reactbook) приобретет вид, показанный на рис. 1.1[1].

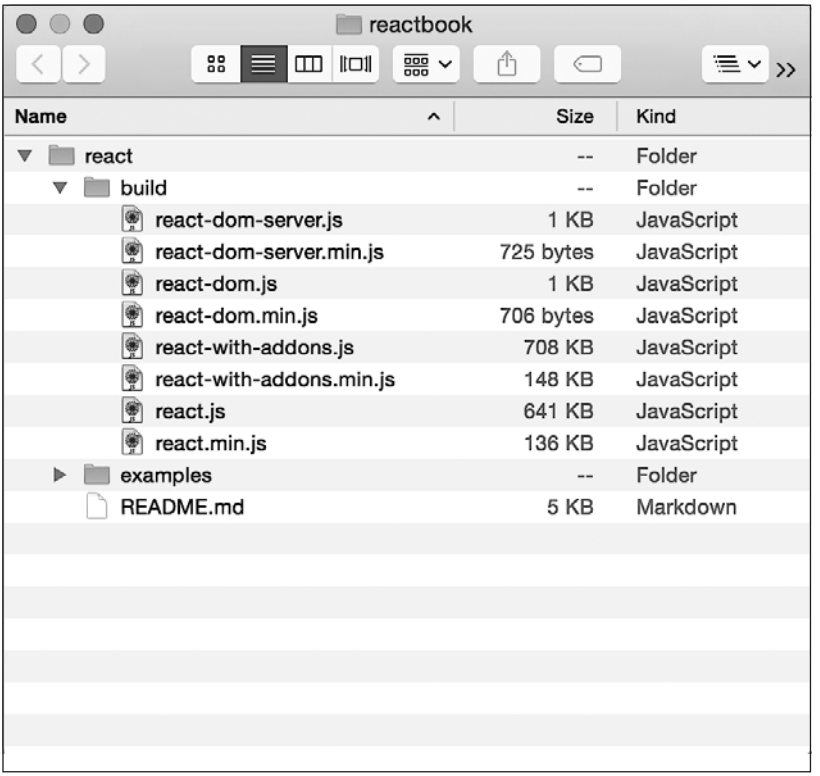


Рис. 1.1. Внешний вид вашего каталога React

[1] Вероятно, описанный автором процесс получения исходных файлов для формирования показанной на рисунке структуры рабочего каталога устарел и не может быть повторен. Реальнее всего получить все указанные файлы, загрузив сопровождающий книгу полный пакет инструментальных средств и примеров исходного кода с адреса https://github.com/stoyan/reactbook/, нажав кнопку Clone or download. — *Примеч. пер.*

Единственный файл, который вам понадобится для начала, — это ~/reactbook/react/build/react.js. Остальные файлы будут рассмотрены чуть позже.

Следует отметить, что React не навязывает какую-либо структуру каталогов, вы можете свободно переместить файлы в другой каталог или переименовать react.js, как вам захочется.

Привет, мир React

Начнем с простой страницы в вашем рабочем каталоге (~/reactbook/01.01.hello.html):

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello React</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="app">
        <!-- здесь будет отображено мое приложение -->
    </div>
    <script src="react/build/react.js"></script>
    <script src="react/build/react-dom.js"></script>
    <script>
        // код моего приложения
    </script>
  </body>
</html>
```

Весь представленный в книге код можно найти в сопровождающем ее хранилище по адресу https://github.com/stoyan/reactbook/.

В этом файле заслуживают внимание только две особенности:

- включение библиотеки React и ее DOM-дополнения (с помощью тегов `<script src>`);
- определение места, где должно находиться ваше приложение на странице (`<div id="app">`).

С React-приложением всегда можно смешивать обычное HTML-наполнение, а также другие библиотеки JavaScript. На одной и той же странице также могут находиться сразу несколько React-приложений. Вам понадобится лишь место в объектной модели документов (DOM), на которое можно нацелить библиотеку React и сказать ей: «Твори здесь свое волшебство».

Теперь добавим код, который говорит hello. Для этого обновим содержимое файла 01.01.hello.html, заменив строку `// код моего приложения` следующими строками:

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

Загрузите 01.01.hello.html в браузер — и увидите свое новое приложение в действии (рис. 1.2).

Поздравляю, вы только что создали свое первое React-приложение!

На рис. 1.2 также показано, в каком виде *сгенерированный* код демонстрируется в области инструментов разработчика браузера Chrome: можно увидеть, что содержимое контейнера `<div id="app">` заменено содержимым, созданным вашим React-приложением.

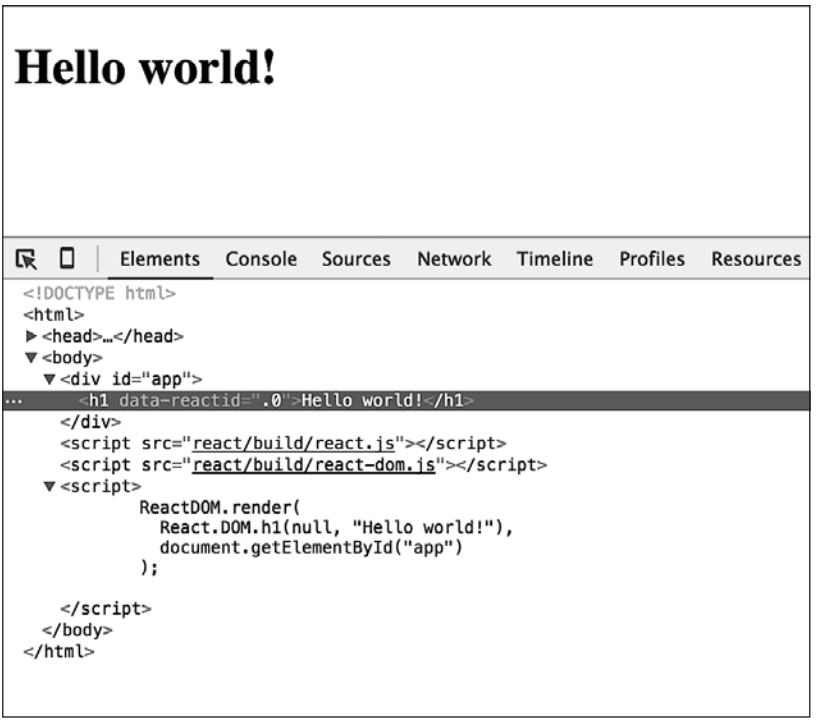


Рис. 1.2. Hello World в действии

Так что же сейчас произошло?

В коде, заставившем работать ваше первое приложение, есть несколько любопытных мест.

В первую очередь в нем видно использование React-объекта. Через него происходит обращение ко всем доступным вам API-интерфейсам. API-интерфейс намеренно сделан минимальным, чтобы не пришлось запоминать слишком много имен методов.

Можно также посмотреть на `ReactDOM`-объект. В нем всего лишь горстка методов, наиболее полезным из которых является `render()`. Ранее эти методы были составной частью React-объекта, но начиная с версии 0.14 они выделены в самостоятельный объект, чтобы подчеркнуть тот факт, что отображение приложения является отдельной задачей. React-приложение можно создать для отображения в различных средах, например в HTML (в DOM-модели браузера), на холсте (canvas) или прямо в Android либо iOS.

Затем следует обратить внимание на концепцию *компонентов*. Ваш пользовательский интерфейс создается с использованием компонентов, которые сочетаются в любом (подходящем для вас) виде. В своем приложении вы, конечно же, займетесь созданием собственных привычных компонентов, но для начала React предоставляет вам надстройки над HTML DOM-элементами. Эти надстройки используются посредством `React.DOM`-объекта. В нашем первом примере показано использование компонента `h1`. Он соответствует HTML-элементу `<h1>` и доступен по вызову `React.DOM.h1()`.

И наконец, вы можете заметить хорошо известный DOM-доступ к контейнеру, осуществляемый с помощью метода `document.getElementById("app")`. Он используется, чтобы подсказать React, где именно на странице должно размещаться приложение. Это своеобразный мост, переброшенный между манипуляцией с известными вам DOM-объектами и React-территорией.

После переброски моста от DOM к React можно больше не беспокоиться о работе с DOM, поскольку React выполняет трансформацию своих компонентов в базовую платформу (DOM-модель браузера, холст, исходное приложение). Вам не нужно заниматься DOM-моделью, но это не означает, что вы не можете этого делать. React предоставляет вам «резервные переходы», если возникнет необходимость вернуться к истокам DOM-модели.

После выяснения роли каждой строки посмотрим на всю картину в целом. Получается следующее: вы отображаете один React-компонент в выбранном вами месте DOM-модели. Всегда отображается только один компонент верхнего уровня, и он может иметь любое необходимое вам количество дочерних (а также внучатых и т. д.) компонентов. Фактически даже в этом простом примере у компонента `h1` имеется дочерний компонент — текст «Hello World!».

React.DOM.*

Как уже известно, React.DOM-объект предоставляет возможность использования нескольких элементов HTML (на рис. 1.3 показано, как получить полный список, используя браузерную консоль). Изучим этот API-интерфейс.

Обратите внимание на разницу между React.DOM и ReactDOM. Первое выражение относится к числу готовых к использованию HTML-элементов, а второе является способом отображения приложения в браузере (подумайте о вызове метода ReactDOM.render()).

Посмотрим на параметры, получаемые всеми методами `React.DOM.*`. Вспомним, что приложение Hello World выглядело следующим образом:

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

Первый параметр для `h1()` (который в данном случае имеет значение `null`) является объектом, указывающим на любые свойства (подумайте об атрибутах DOM-модели), которые

желательно было бы передать вашему компоненту. Например, можно сделать следующее:

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

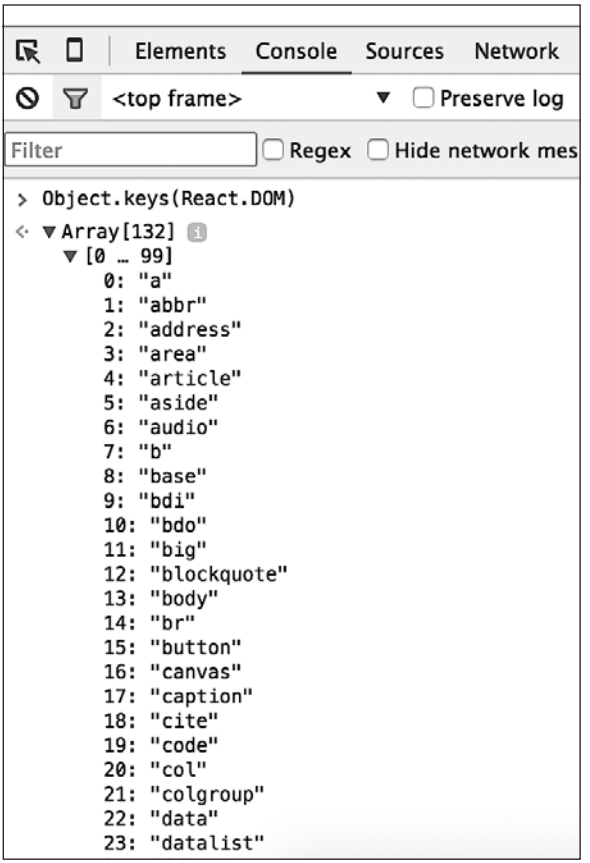


Рис. 1.3. Список свойств React.DOM

Код HTML, созданный в этом примере, показан на рис. 1.4.

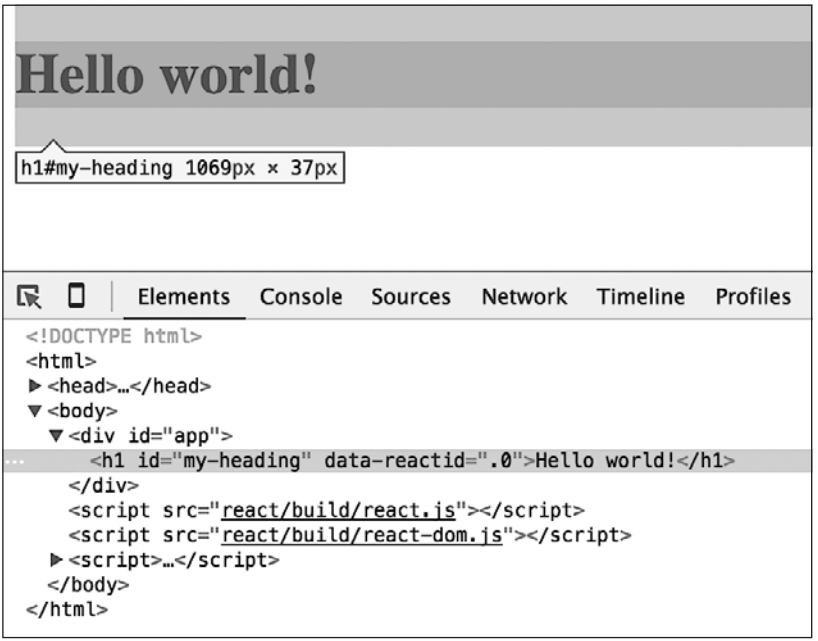


Рис. 1.4. Код HTML, созданный вызовом React.DOM

Второй параметр (в данном примере `"Hello World!"`) определяет дочерний элемент компонента. Как видно из предыдущего кода, в самом простом варианте это всего лишь текстовый дочерний элемент (в DOM-терминологии — text-узел). Но у вас может быть сколько угодно вложенных дочерних элементов, передаваемых в качестве дополнительных параметров функции. Например:

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

А вот еще один пример, на этот раз со следующими вложенными компонентами (результат показан на рис. 1.5):

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

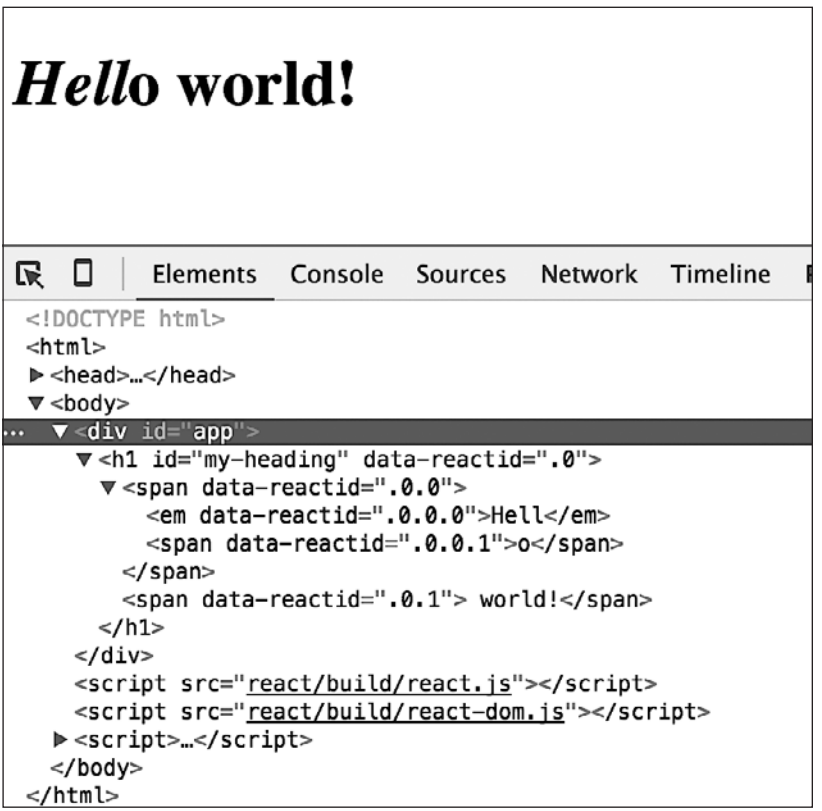


Рис. 1.5. Код HTML, созданный вложенными вызовами React.DOM

Заметьте, стоит только начать использовать вложенные компоненты, как сразу же появится множество вызовов функций и масса круглых скобок, за правильной расстановкой которых приходится следить. Чтобы упростить положение дел, можно воспользоваться JSX-синтаксисом. Тема JSX рассматривается отдельно (в главе 4), а пока помучаемся с чистым синтаксисом JavaScript. Дело в том, что синтаксис JSX поначалу у многих вызывает отторжение (как это так — XML в моем JavaScript!), но затем переходит в разряд незаменимых. Чтобы дать вам прочувствовать, что это такое, покажем прежний фрагмент кода, но уже с использованием синтаксиса JSX:

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

Специальные DOM-атрибуты

Вам должны быть известны специальные DOM-атрибуты, например `class`, `for` и `style`.

Вы не можете использовать `class` и `for`, поскольку в JavaScript они являются зарезервированными словами. Вместо них нужно выбрать названия `className` и `htmlFor`:

```
// Пример неработоспособного кода
React.DOM.h1(
  {
    class: "pretty",
    for: "me",
  },
  "Hello World!"
);

// Пример работоспособного кода
```

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

Что касается атрибута `style`, то вы не можете использовать строку, как это обычно делается в HTML, вместо этого следует задействовать объект JavaScript. Отказ от использования строк всегда приветствуется (сокращается риск быть атакованным с применением межсайтовых сценариев (XSS)), поэтому такие изменения будут только поощряться.

```
// Пример неработоспособного кода
React.DOM.h1(
  {
    style: "background: black; color: white; font-family:
      Verdana",
  },
  "Hello World!"
);

// Пример работоспособного кода
React.DOM.h1(
  {
    style: {
      background: "black",
      color: "white",
      fontFamily: "Verdana",
    }
  },
  "Hello World!"
);
```

Следует также заметить, что при работе со свойствами CSS необходимо использовать имена API-интерфейсов JavaScript, например `fontFamily`, а не `font-family`.

Расширение браузера React DevTools

Если открыть консоль браузера при попытке запуска некоторых примеров из этой главы, вы увидите следующее сообщение: Download the React DevTools for a better development experience: https://fb.me/react-devtools, повествующее о том, что для повышения эффективности разработки рекомендуется загрузить инструментарий React DevTools с сайта https://fb.me/react-devtools. Если перейти по указанному URL-адресу, можно найти ссылки на установку браузерного расширения, помогающего вести отладку React-приложений (рис. 1.6).

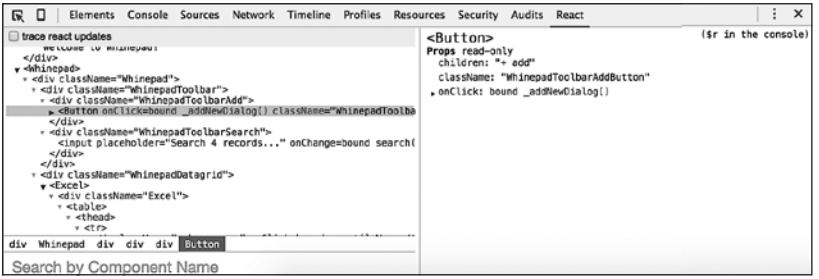


Рис. 1.6. Расширение React DevTools

Поначалу это может показаться трудным в освоении, но, когда вы доберетесь до главы 4, все станет на свои места.

Далее: настраиваемые компоненты

Итак, вы справились с простейшим приложением Hello World. Теперь вы обладаете знаниями, позволяющими:

- установить библиотеку React, произвести ее настройку и воспользоваться ею (фактически все сводится к использованию всего лишь двух тегов `<script>`);

- вывести React-компонент в выбранное вами место в DOM-модели (например, `ReactDOM.render(reactWhat, domWhere)`, где *reactWhat* — выводимый React-компонент, а *domWhere* — узел DOM-модели, в который он выводится);
- использовать встроенные компоненты, являющиеся оболочками для обычных DOM-элементов (например, `React.DOM.div(attributes, children)`, где *attributes* — атрибуты, а *children* — дочерние элементы).

Но реальная сила React проявится, как только вы для создания (и обновления!) пользовательского интерфейса вашего приложения начнете использовать настраиваемые компоненты. Рассмотрим это в следующей главе.

2 Жизнь компонента

После того как вы узнали о порядке использования готовых DOM-компонентов, пора приступить к изучению способов создания собственных компонентов.

Самый минимум

API-интерфейс для создания нового компонента имеет следующий вид:

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

В качестве «спецификаций» используется объект JavaScript, имеющий один обязательный метод `render()` и несколько необязательных методов и свойств. Самый простой пример может выглядеть следующим образом:

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

Как видите, единственное, что нужно сделать, — это реализовать метод `render()`. Он должен возвратить React-компонент, именно поэтому вы видите в фрагменте кода компонент `span`; просто текст возвратить нельзя.

Использование вашего компонента в приложении похоже на использование DOM-компонентов:

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

Результат отображения вашего пользовательского компонента показан на рис. 2.1.

Один из способов создания экземпляра вашего компонента — применение метода `React.createElement()`. При создании сразу нескольких экземпляров применяется фабрика:

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

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

Учтите, что уже известные вам методы семейства `React.DOM.*` фактически являются всего лишь удобными оболочками вокруг `React.createElement()`. Иными словами, этот код также работает с DOM-компонентами:

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

Рис. 2.1. Ваш первый пользовательский компонент

Как видите, DOM-элементы, в отличие от функций JavaScript, определяются в виде строк, как и в случае с пользовательскими компонентами.

Свойства

Ваши компоненты получают свойства и в зависимости от их значений по-разному выводятся на экран или ведут себя в приложении. Все свойства доступны через объект `this.props`. Рассмотрим пример:

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

Передача свойства при отображении компонента выглядит следующим образом:

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

Результат показан на рис. 2.2.

Свойство this.props следует считать пригодным только для чтения. Свойства можно с успехом применять для переноса настроек из родительских компонентов в дочерние (и, как будет показано далее, из дочерних компонентов — в родительские). Если вас все же что-то подстегивает назначить свойство с применением this.props, просто воспользуйтесь вместо этого дополнительными переменными или свойствами спецификации объекта вашего компонента (как в образце this.thing в противоположность образцу this.props.thing). На деле в браузерах, отвечающих спецификации ECMAScript5, вам не захочется видоизменять свойство this.props, поскольку:

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

Рис. 2.2. Использование свойств компонента

propTypes

Для объявления списка свойств, принимаемых вашим компонентом, и их типов вы можете добавить в свои компоненты свойство `propTypes` (типы свойств). Рассмотрим пример:

```
var Component = React.createClass({
  propTypes: {
    name: React.PropTypes.string.isRequired,
```

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

Свойство `propTypes` не обязательно использовать, но оно предоставляет два преимущества:

- вы заранее объявляете, какие свойства ожидает ваш компонент. Пользователям вашего компонента не придется тщательно изучать исходный (потенциально весьма длинный) код функции `render()`, чтобы определить, какими свойствами можно воспользоваться для настройки компонента;
- React проводит проверку значений свойств в ходе выполнения программы, поэтому свою функцию `render()` можно создавать без лишних опасений (или чрезмерной подозрительности) насчет данных, получаемых вашими компонентами.

Рассмотрим проверку в действии. Выражение `name: React.PropTypes.string.isRequired` явно просит для свойства `name` обязательное строковое значение. Если вы забудете передать значение, в консоли появится предупреждение (рис. 2.3):

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

Предупреждение также будет получено при предоставлении значения неверного типа, скажем целого числа (рис. 2.4):

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

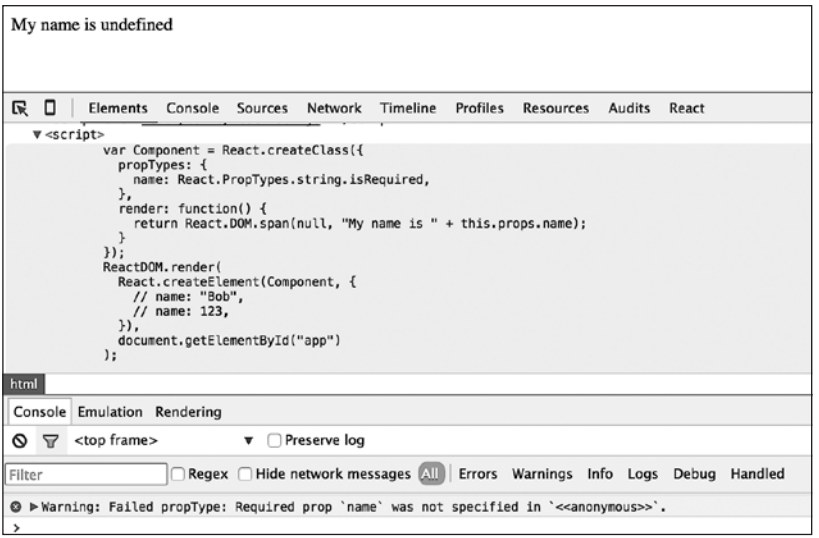


Рис. 2.3. Предупреждение, появляющееся в случае, когда обязательное свойство не предоставлено

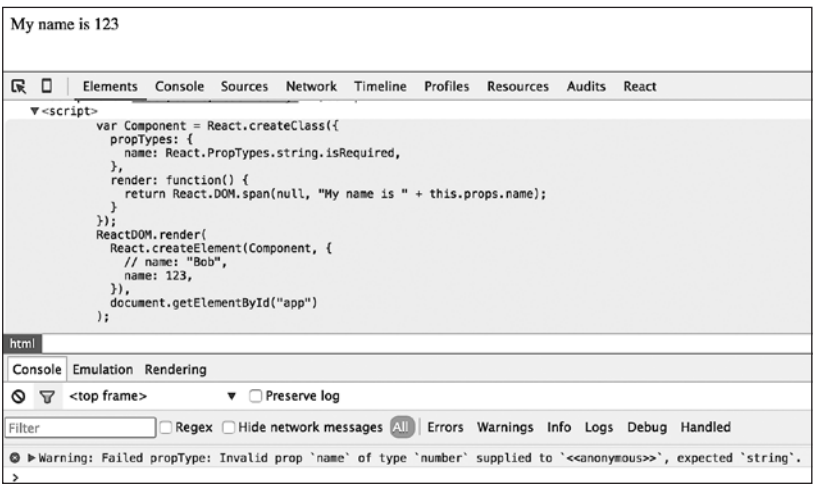


Рис. 2.4. Предупреждение, которое появляется при предоставлении значения неверного типа

Рисунок 2.5 даст представление обо всех доступных свойствах `PropTypes`, которыми можно воспользоваться для объявления ваших ожиданий.

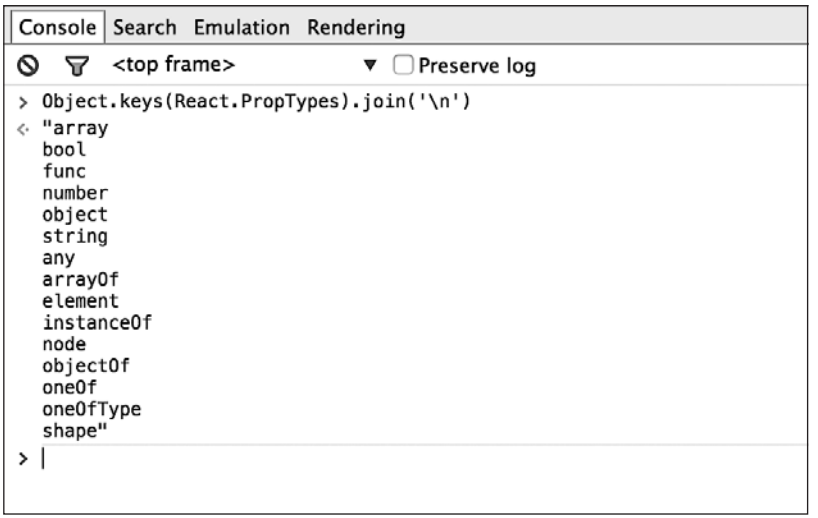


Рис. 2.5. Список всех типов, используемых в React.PropTypes

Объявление propTypes в ваших компонентах не является обязательным (это означает, что здесь можно перечислить лишь некоторые, а не абсолютно все свойства). Можно назвать идею объявления не всех свойств порочной, но имейте в виду, что можете столкнуться с этим при отладке чужого кода.

Значения свойств, используемые по умолчанию. Когда ваши компоненты получают необязательные свойства, следует уделить особое внимание их работоспособности в том случае, когда свойства не объявляются. Это неизбежно приводит к применению защитного шаблона, например:

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

Избавиться от необходимости написания такого кода (сконцентрировавшись на более важных аспектах программы) можно, реализовав метод `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() {/* ... */}
});
```

Как видите, `getDefaultProps()` возвращает объект, предоставляя допустимые значения для каждого необязательного свойства (из числа тех, для которых не указывается `.isRequired`).

Состояние

До сих пор примеры были довольно статичными (или «не имеющими состояния»). Они преследовали простую цель: дать вам представление о создании блоков при составлении вашего пользовательского интерфейса. Но истинный блеск React (на фоне усложнения ситуации при работе с традиционной DOM-моделью браузера) проявляется, когда изменяются данные в вашем приложении. В React используется понятие состояния, которое

представляет собой данные, используемые вашими компонентами для их самостоятельного отображения на экране. При изменении состояния React перестраивает пользовательский интерфейс без какого-либо вашего участия. Таким образом, после начального создания пользовательского интерфейса (в функции `render()`) вам останется лишь обновлять данные, совершенно не заботясь насчет изменений пользовательского интерфейса. По сути, ваш метод `render()` уже предоставил план внешнего вида компонента.

Обновление пользовательского интерфейса после вызова метода setState() выполняется с использованием механизма выстраивания очереди, который рационально группирует изменения, поэтому непосредственное обновление this.state (чего не следует делать) может привести к непредсказуемому поведению. Считайте, что объект this.state, так же как и this.props, предназначен лишь для чтения — не только потому, что его применение в другом качестве считается неприемлемым, но и потому, что такое применение может привести к неожиданным результатам. Аналогично никогда не вызывайте this.render() самостоятельно, лучше предоставьте такое право библиотеке React, чтобы дать ей возможность сгруппировать изменения, выявить их наименьшее количество и вызвать render() в самый подходящий момент.

Точно так же, как свойства доступны путем обращения к объекту `this.props`, к состоянию можно обратиться через объект `this.state`. Для обновления состояния предназначен метод `this.setState()`. При вызове `this.setState()` React вызывает ваш метод `render()` и обновляет пользовательский интерфейс.

React обновляет пользовательский интерфейс при вызове setState(). Это наиболее распространенный вариант, но чуть позже вы увидите, что есть еще и запасной вариант. Предотвратить обновление пользовательского интерфейса можно, вернув false в специальном методе жизненного цикла — shouldComponentUpdate().

Компонент textarea, отслеживающий свое состояние

Создадим новый компонент — `textarea` (текстовая область), сохраняющий количество набранных в нем символов (рис. 2.6).

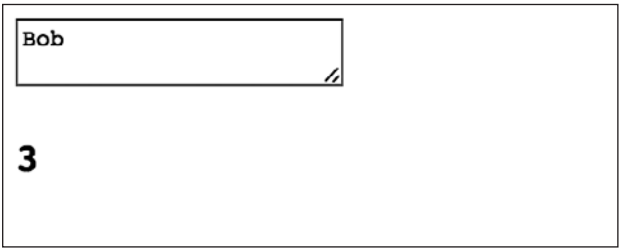


Рис. 2.6. Конечный результат работы пользовательского компонента textarea

Вы (как и все другие потребители этого многократно используемого компонента) можете воспользоваться новым компонентом следующим образом:

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

Теперь реализуем компонент. Начнем с создания версии «без состояния», не обрабатывающей обновления, поскольку она не слишком отличается от всех предыдущих примеров:

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

Возможно, вы заметили, что textarea в предыдущем фрагменте кода получает свойство defaultValue, в отличие от привычного вам дочернего элемента text в HTML. Все дело в небольшой разнице между React и традиционным HTML, которая проявляется при работе с формами. Этот вопрос рассматривается в главе 4, и спешу заверить, что различий не так уж и много. Кроме того, вы поймете, что эти различия вполне резонны и призваны упростить вашу жизнь разработчика.

Как видите, компонент получает необязательное строковое свойство `text` и выводит текстовую область с заданным значением, а также получает элемент `<h3>`, который показывает длину строки (рис. 2.7).

Следующий шаг заключается в превращении этого лишенного состояния компонента в компонент с поддержкой состояния. Иными словами, получим компонент, обрабатывающий некие данные (состояние) и использующий их для исходного вывода самого себя на экран с последующим самостоятельным обновлением (повторным выводом) при их изменении.

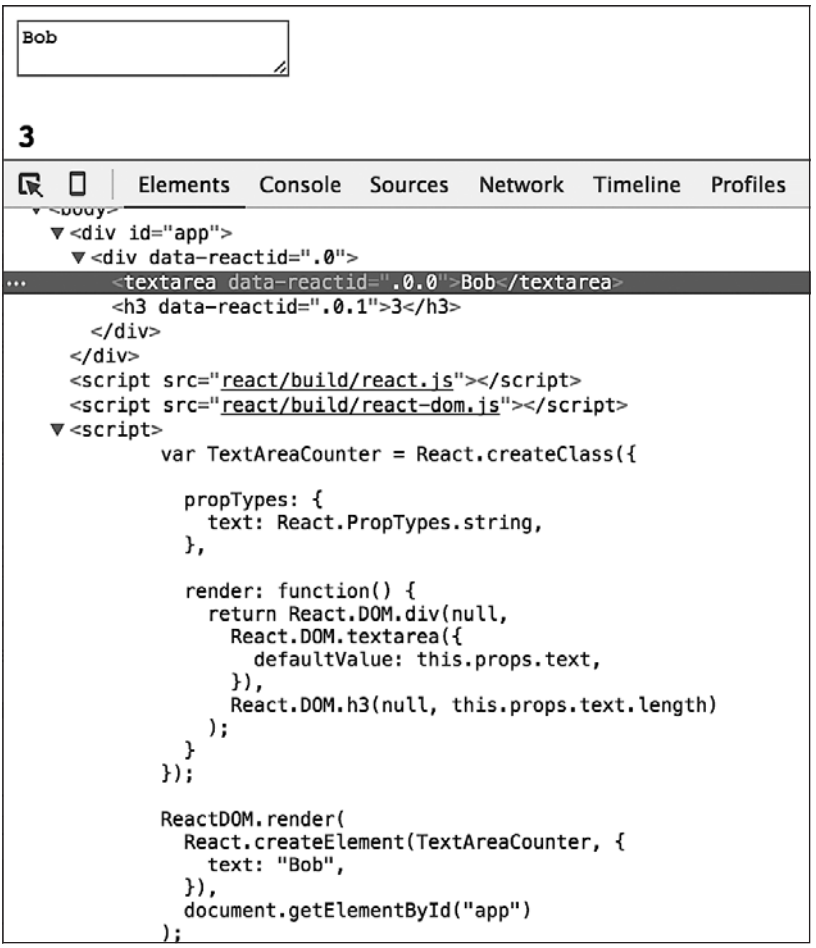


Рис. 2.7. Компонент TextAreaCounter в работе

Реализуйте в вашем компоненте метод `getInitialState()`, чтобы быть уверенными, что работа всегда будет вестись с допустимыми данными:

```
getInitialState: function() {
  return {
```

```
    text: this.props.text,
  };
},
```

Данные, обрабатываемые этим компонентом, являются простым текстом в текстовой области, поэтому у состояния есть только одно свойство `text`, доступ к которому можно получить через выражение `this.state.text`. Изначально (в `getInitialState()`) вы просто копируете свойство `text`. Позже при изменении данных (в ходе набора пользователем текста в текстовой области) компонент обновляет свое состояние, используя вспомогательный метод:

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

Состояние всегда обновляется с помощью метода `this.setState()`, который получает объект и объединяет его с уже существующими в `this.state` данными. Нетрудно догадаться, что `_textChange()` является отслеживателем событий, который получает объект события `ev` и извлекает из него текст, введенный в текстовую область.

Остается лишь обновить метод `render()` для использования вместо `this.props` свойства `this.state` и установить отслеживатель событий:

```
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)
  );
}
```

Теперь, как только пользователь что-нибудь набирает в текстовой области, значение счетчика обновляется, чтобы соответствовать содержимому этой области (рис. 2.8).

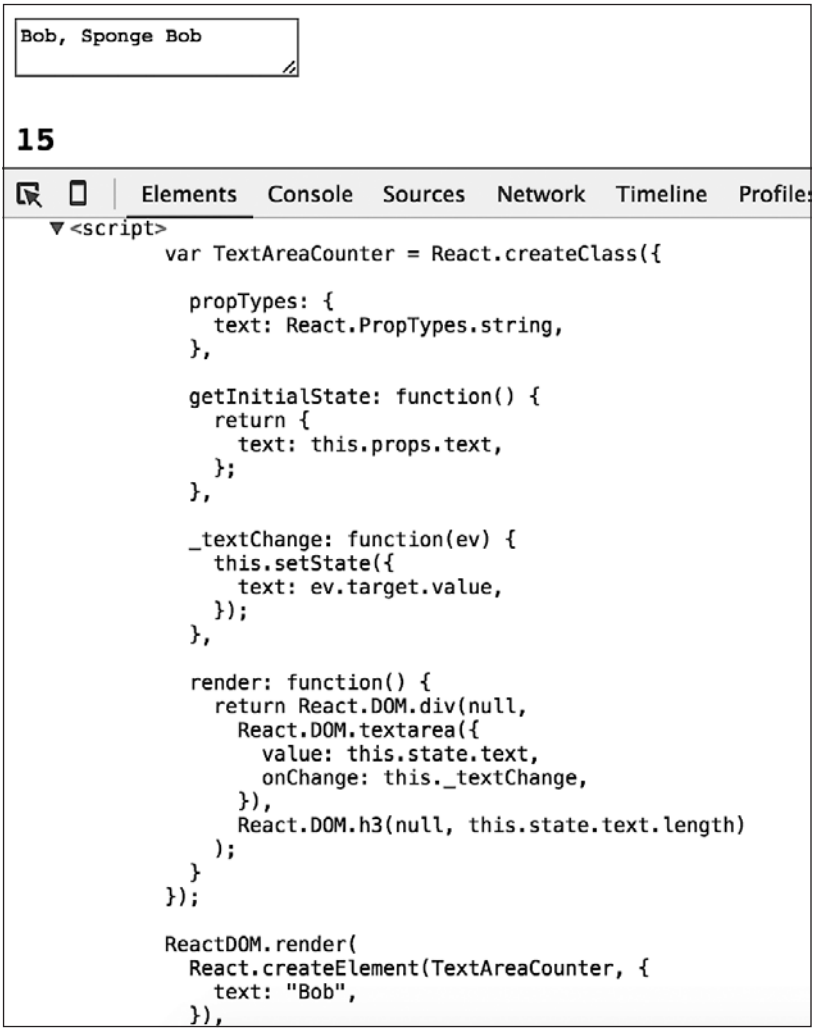


Рис. 2.8. Набор текста в текстовой области

Немного о DOM-событиях

Во избежание путаницы насчет следующей строки необходимо кое-что пояснить:

```
onChange: this._textChange
```

В целях повышения производительности, а также для удобства работы React использует собственную систему *искусственно* создаваемых событий. Чтобы легче было разобраться в причинах этого, следует рассмотреть, как все происходит в подлинном мире DOM-модели.

Обработка событий в прежние времена

Чтобы выполнять какие-либо действия, очень удобно применять *встроенные* обработчики событий:

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

При всем удобстве и легкой узнаваемости (отслеживатель событий находится там же, где и пользовательский интерфейс) пользоваться слишком большим количеством разбросанных подобным образом отслеживателей событий крайне неэффективно. Также трудно пользоваться на одной и той же кнопке более чем одним отслеживателем, особенно если эта кнопка является не вашим, а чьим-то «компонентом» или входит в другую библиотеку и вам не хочется туда внедряться и «править» или разветвлять код. Именно поэтому в мире DOM-модели для установки отслеживателей используются метод `element.addEventListener` (что приводит к наличию кода в двух и более местах) и *делегирование событий* (для решения проблем производительности). Делегирование событий означает, что отслеживание событий

осуществляется в родительском узле, скажем в , содержащем множество кнопок, и для всех кнопок устанавливается один отслеживатель.

С использованием делегирования событий выполняется следующее:

```
<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;

  // Выполнение разных действий на основе того,
  // какая из кнопок была нажата
  switch (button.id) {
    case 'ok':
      console.log('OK!');
      break;
    case 'cancel':
      console.log('Cancel');
      break;
    default:
      new Error('Непредвиденный идентификатор кнопки');
  };
});
</script>
```

Со своей работой этот код справляется, но у него имеются недостатки:

- объявление отслеживателя находится далеко от компонента пользовательского интерфейса, что затрудняет поиск и отладку кода;

- делегирование с неизменным использованием инструкции `switch` создает ненужный шаблонный код непосредственно перед переходом к реальным действиям (в данном случае к реакции на нажатие кнопки);
- браузерная несовместимость (которая здесь не рассматривается) требует от этого кода более пространного решения.

К сожалению, как только дело доходит до практического применения данного кода реальными пользователями, для его поддержки всеми браузерами требуется предпринять ряд дополнительных мер:

- в дополнение к методу `addEventListener` требуется применение метода `attachEvent`;
- в самом начале кода отслеживателя требуется применение выражения `var event = event || window.event;`;
- требуется применение выражения `var button = event.target || event.srcElement;`.

Все эти необходимые и весьма неприятные нюансы в конечном итоге наводят на мысль о применении какой-нибудь библиотеки, связанной с обработкой событий. Но зачем добавлять еще одну библиотеку (и изучать дополнительные API-интерфейсы), когда React поставляется в комплекте с решениями, избавляющими от всех неприятностей, связанных с обработкой событий?

Обработка событий в React

Чтобы охватить и привести к единому формату все браузерные события, в React используются *искусственные события*, ликвидирующие проблему несовместимости браузеров. Это позволяет вам быть уверенными, что свойство `event.target` доступно во всех браузерах. Именно поэтому в фрагменте кода `TextAreaCounter`

нужно лишь воспользоваться свойством `ev.target.value` — и все заработает. Это также означает, что API-интерфейс для отмены событий един для всех браузеров; иными словами, методы `event.stopPropagation()` и `event.preventDefault()` работают даже на старых версиях IE.

Синтаксис упрощает совместную поддержку элементов пользовательского интерфейса и отслеживателей событий. Он похож на традиционные встроенные обработчики событий, но за кулисами все обстоит иначе. Фактически React в целях повышения производительности использует делегирование событий.

Для обработчиков событий в React применяется синтаксис с использованием «верблюжьего» регистра букв (camelCase), поэтому вместо `onclick` используется форма записи `onClick`.

Если по каким-то причинам вам понадобится исходное браузерное событие, оно доступно в виде свойства `event.nativeEvent`, но вряд ли вам это когда-либо пригодится.

И еще: событие `onChange` (в том же виде, в каком оно использовалось в примере с текстовой областью) ведет себя согласно вашим ожиданиям: оно инициируется, когда пользователь выполняет набор текста, а не после того, как набор будет завершен и произойдет переход за пределы данного поля, как это происходит в обычной DOM-модели.

Сравнение свойств и состояния

Вы уже знаете, что при решении задачи отображения вашего компонента в методе `render()` у вас есть доступ к свойствам через выражение `this.props` и к состоянию — через `this.state`. Может возникнуть вопрос, когда следует использовать одно из этих выражений, а когда — другое.

Свойства — это механизм, предназначенный для внешнего мира (для пользователей компонента) и служащий для настройки вашего компонента. А состояние относится к работе с внутренними данными. Поэтому, если искать аналогии в объектно-ориентированном программировании, `this.props` является подобием *аргументов, переданных конструктору класса*, а `this.state` можно представить в виде набора ваших *закрытых свойств*.

Свойства в исходном состоянии: антишаблон

Ранее уже был показан пример использования выражения `this.props` внутри метода `getInitialState()`:

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

Фактически этот пример относится к антишаблонам. В идеале используется любая комбинация выражений `this.state` и `this.props`, подходящая для создания пользовательского интерфейса в методе `render()`.

Однако иногда приходится брать значение, переданное вашему компоненту, и использовать его для построения исходного состояния. Здесь нет ничего крамольного, за исключением того, что код, вызывающий ваш компонент, может предполагать, что свойство (`text` в предыдущем примере) всегда будет иметь самое последнее значение, а пример противоречит этому предположению. Чтобы предположения не вызывали никаких сомнений, достаточно просто воспользоваться другим именем, например вместо `text` назвать свойство как-нибудь вроде `defaultText` или `initialValue`:

```
propTypes: {
  defaultValue: React.PropTypes.string
},

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

В главе 4 показано, как в библиотеке React решается эта проблема для ее собственной реализации полей ввода и текстовых областей, где у кого-то могут возникать предположения, основанные на имеющемся опыте работы с HTML.

Доступ к компоненту извне

Позволить себе роскошь запускать совершенно новое React-приложение можно не всегда. Порой приходится внедряться в существующее приложение или в сайт и постепенно переходить к React. К счастью, библиотека React была сконструирована для работы с любым ранее созданным и имеющимся в вашем распоряжении базовым кодом. Ведь те, кто начинал создавать React, не могли переписать целиком огромное приложение (Facebook).

Один из способов, позволяющих вашему React-приложению общаться с внешним миром, заключается в получении ссылки на выводимый вами компонент с помощью метода `ReactDOM.render()` и ее использовании за пределами компонента:

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

Теперь можно воспользоваться компонентом `myTextAreaCounter` для доступа к тем же методам и свойствам, к которым обычно обращаются с помощью выражения `this` внутри компонента. Можно даже манипулировать компонентами, используя вашу консоль JavaScript (рис. 2.9).

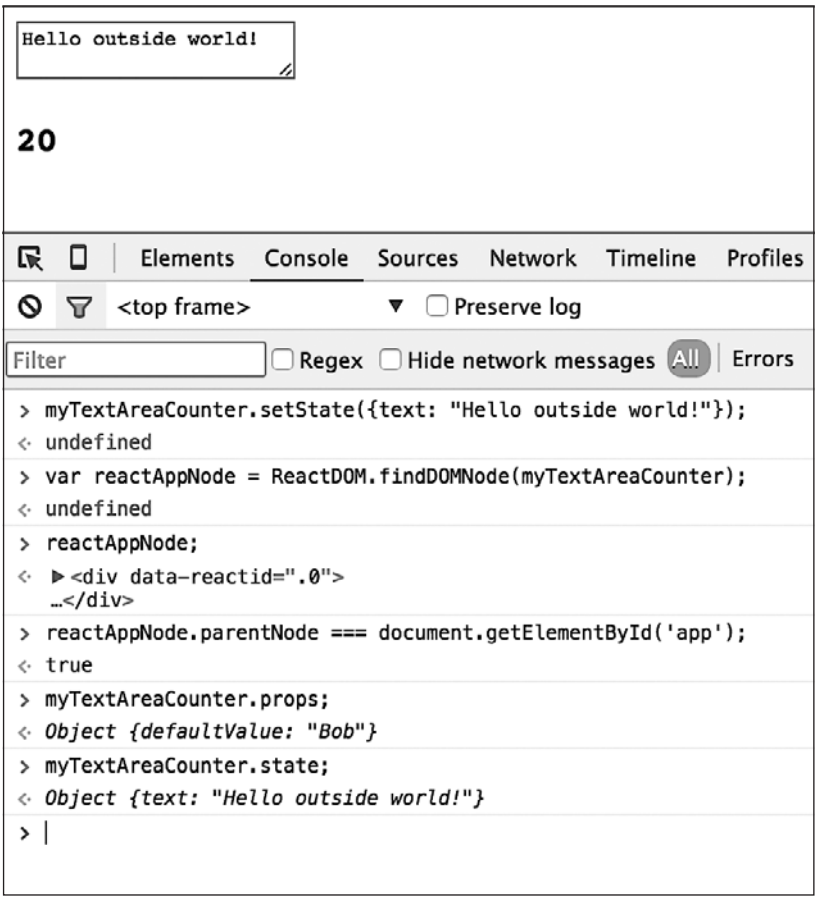


Рис. 2.9. Обращение к выведенному на экран компоненту с помощью ссылки

В этой строке кода устанавливается новое состояние:

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

В этой строке кода берется ссылка на основной родительский DOM-узел, создаваемый React:

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

Это первый дочерний элемент родительского `<div id="app">`, где библиотеке React предписывается творить чудеса:

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

А вот как можно получить доступ к свойствам и состоянию:

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

Вы получили доступ ко всему API-интерфейсу компонента за пределами вашего компонента. Но пользоваться этими сверхвозможностями следует весьма осмотрительно и только при крайней необходимости. Возможно, ReactDOM.findDOMNode() придется воспользоваться, если нужно получить габаритные размеры узла, чтобы убедиться, что он помещается на всю вашу страницу, но не более того. Может быть, манипулирование состоянием компонентов, владельцем которых вы не являетесь, и их подгонка покажутся вам весьма заманчивой затеей, но не стоит плодить ошибки, поскольку компонент не предполагает подобных вторжений. Например, следующий код вполне работоспособен, но использовать его не рекомендуется:

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

Изменение свойств на лету

Как известно, настройка компонента осуществляется с помощью его свойств.

Соответственно, изменение свойств извне после того, как компонент был создан, должно быть обоснованным. Но ваш компонент должен быть готов к обработке подобного развития событий.

Если посмотреть на метод `render()` из предыдущих примеров, в нем используется лишь выражение `this.state`:

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

Если свойства изменятся за пределами компонента, это не возымеет никакого эффекта на экране. Иными словами, содержимое текстовой области после того, как вы сделаете следующее, не изменится:

```
myTextAreaCounter = ReactDOM.render(
  React.createElement(TextAreaCounter, {
    defaultValue: "Hello", // ранее известное как "Bob"
  }),
  document.getElementById("app")
);
```

Даже притом что компонент myTextAreaCounter переписан с использованием нового вызова ReactDOM.render(), состояние приложения останется прежним. React ничего не стирает, а сопоставляет состояние приложения до и после. При этом он вносит минимальные изменения.

Теперь содержимое `this.props` изменилось (но содержимое пользовательского интерфейса осталось прежним):

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

Обновление пользовательского интерфейса выполняется путем установки состояния:

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

Но это порочная практика, поскольку она может привести к несогласованному состоянию в более сложных компонентах, например внести путаницу во внутренние счетчики, булевы флаги, отслеживатели событий и т. д.

Если нужно корректно справиться с внешним вторжением (изменением свойств), к нему можно приготовиться, реализовав метод `componentWillReceiveProps()`:

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

Как видите, этот метод получает новый объект свойств, и вы можете соответствующим образом установить состояние, а также выполнить любые другие действия, требующиеся для поддержания компонента в рабочем состоянии.

Методы управления жизненным циклом

Метод `componentWillReceiveProps()` из предыдущего фрагмента кода относится к так называемым методам *управления жизненным циклом*, предлагаемым React. Эти методы можно использовать для отслеживания изменений в ваших компонентах. К числу других методов управления жизненным циклом, которые могут быть вами реализованы, относятся следующие.

- `componentWillUpdate()`. Выполняется до того, как метод `render()` вашего компонента будет вызван еще раз (в результате изменений свойств или состояния).
- `componentDidUpdate()`. Выполняется после завершения работы метода `render()` и применения новых изменений в отношении исходной DOM-модели.
- `componentWillMount()`. Выполняется перед вставкой узла в DOM-модель.
- `componentDidMount()`. Выполняется после вставки узла в DOM-модель.
- `componentWillUnmount()`. Выполняется непосредственно перед тем, как компонент удаляется из DOM-модели.
- `shouldComponentUpdate(`*`newProps, newState`*`)`. Вызывается перед вызовом метода `componentWillUpdate()` и дает возможность возвратить `false` и отменить обновление, что означает отказ от вызова метода `render()`. Пригодится в тех местах приложения, для которых критична производительность, когда вы полагаете, что ничего важного не изменилось и повторный вывод на экран необязателен. Это решение принимается на основе сравнения аргумента *`newState`* с существующим значе-

нием выражения `this.state` и сравнения *`newProps`* с `this.props` или просто на основании сведений о том, что компонент статичен и не подвергается изменениям. (Соответствующий пример будет вскоре рассмотрен.)

Примеры управления жизненным циклом

Тотальная регистрация

Чтобы лучше разобраться в жизни компонента, добавим к компоненту `TextAreaCounter` регистрацию. Просто реализуем все методы управления жизненным циклом для регистрации их вызова наряду с показом всех их аргументов в консоли:

```
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);
  },

  // ...
  // продолжение реализации, render() и т. д.

};
```

Все происходящее после загрузки страницы показано на рис. 2.10.

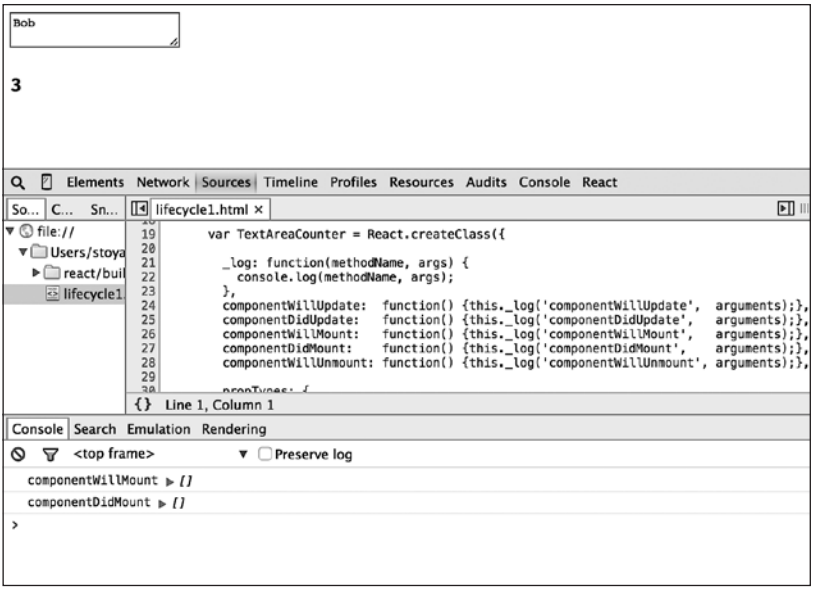


Рис. 2.10. Установка компонента

Как видите, были вызваны два метода без аргументов. Обычно из этих двух методов наиболее интересен `componentDidMount()`. Если нужно, доступ к только что установленному DOM-узлу можно получить путем вызова метода `ReactDOM.findDOMNode(this)` (например, для получения габаритных размеров компонента).

Теперь, когда ваш компонент ожил, можно выполнять с ним любые действия по инициализации.

Интересно, что произойдет, если набрать `s`, превращая текст в Bobs? (См. рис. 2.11.)

Рис. 2.11. Обновление компонента

Будет вызван метод `componentWillUpdate(`*`nextProps, nextState`*`)` с новыми данными, которые будут использованы для повторного отображения компонента. Первый аргумент представляет собой

будущее значение свойства `this.props` (которое в данном примере не изменяется), а второй — будущее значение нового свойства `this.state`. Третьим по значимости является контекст `context`, который на данном этапе особого интереса для нас не представляет. Вы можете сравнить аргументы (например, *`newProps`*) с текущим значением `this.props` и решить, нужно ли совершать с ними какие-либо действия.

Как видите, после метода `componentWillUpdate()` вызван метод `componentDidUpdate(`*`oldProps, oldState`*`)` с передачей ему значений свойств и состояния до изменения. Он позволяет работать с DOM, когда компонент уже обновлен. В нем можно указать выражение `this.setState()`, чего нельзя сделать в `componentWillUpdate()`.

Предположим, что вам требуется ограничить количество символов, набираемых в текстовой области. Это нужно делать в обработчике события `_textChange()`, вызываемом при наборе пользователем текста. Но что, если кто-нибудь (кто наивнее и моложе вас) вызовет `setState()` за пределами компонента? (Как уже ранее упоминалось, такой вызов — порочная практика.) Сможете ли вы защитить согласованность и нормальное состояние своего компонента? Разумеется. Вы можете провести проверку в методе `componentDidUpdate()` и, если количество символов превысит разрешенное, вернуть состояние к его прежнему значению. То есть нужно сделать нечто подобное:

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

Хотя подобные опасения и излишни, в случае чего всегда можно отыграть все назад.

Обратите внимание на использование метода replaceState() вместо setState(). Метод setState(obj) объединяет свойства obj с теми, которые имелись в this.state, а метод replaceState() полностью все перезаписывает.

Использование примеси

В предыдущем примере были показаны регистрируемые вызовы четырех из пяти методов управления жизненным циклом. Пятый метод, `componentWillUnmount()`, лучше всего продемонстрировать при наличии дочерних компонентов, удаляемых их родителем. В этом примере вам нужно зарегистрировать все изменения в обоих дочерних компонентах и в родительском компоненте. Поэтому введем новое понятие для многократно используемого кода: *примесь*, или *миксин* (mixin).

Примесью называют объект JavaScript, содержащий коллекцию методов и свойств. Примеси предназначены не для самостоятельного использования, а для включения (подмешивания) в свойства другого объекта. В примере регистрации примесь может иметь следующий вид:

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

В мире, где React не используется, можно применить цикл `for-in` и скопировать все свойства в новый объект, получив все функциональные возможности примеси. В мире React в вашем распоряжении имеется сокращенная форма записи: свойство `mixins`, имеющее следующий вид:

```
var MyComponent = React.createClass({

  mixins: [obj1, obj2, obj3],

  // все остальные методы ...

};
```

Вы присваиваете свойству `mixins` массив объектов JavaScript, а React берет на себя всю остальную работу. Включение `logMixin` в ваш компонент выглядит следующим образом:

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

Как видите, во фрагменте кода также добавляется удобное свойство `name`, чтобы идентифицировать вызывающий код.

Если запустить пример с примесью, вы увидите регистрацию в действии (рис. 2.12).

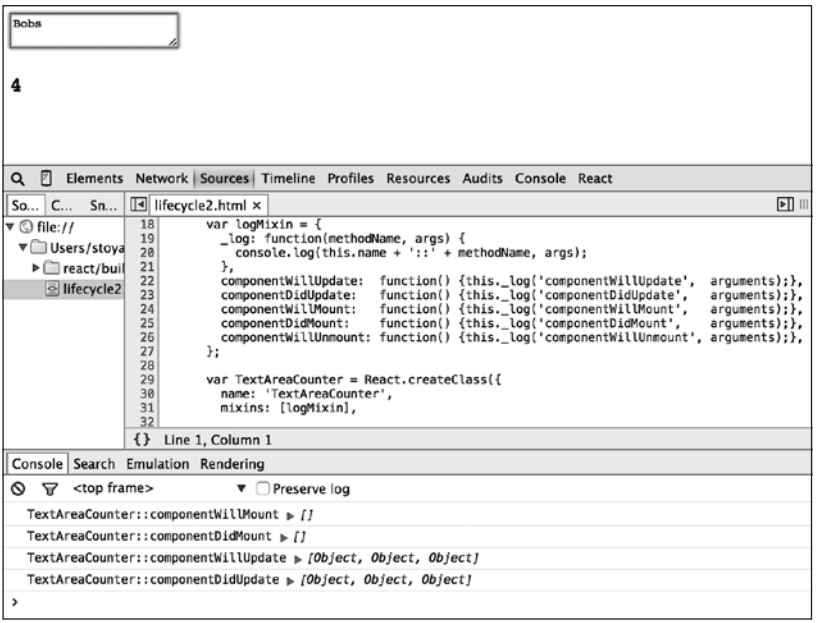


Рис. 2.12. Использование примеси и идентификация компонента

Применение дочернего компонента

Как известно, React-компоненты могут смешиваться и вкладываться друг в друга любым нужным вам образом. До сих пор в методах `render()` вы видели только компоненты `React.DOM` (и не видели пользовательских компонентов). Посмотрим на простой пользовательский компонент, применяемый в качестве дочернего.

Часть счетчика может быть выделена в свой собственный компонент:

```
var Counter = React.createClass({
  name: 'Counter',
```

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

Этот компонент составляет только часть счетчика — он выводит на экран контейнер `<span>` и не работает с состоянием, а только отображает свойство `count`, задаваемое родительским компонентом. Он также подмешивает `logMixin` для регистрации вызовов методов управления жизненным циклом.

Теперь обновим метод `render()` родительского компонента `TextAreaCounter`. При определенных условиях он должен использовать компонент `Counter`, и если `count` имеет значение 0 — число даже не показывается:

```
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
    );
}
```

Когда текстовая область пуста, переменная `counter` имеет значение `null`. Когда в ней имеется какой-нибудь текст, переменная `counter` содержит часть пользовательского интерфейса, отвечающую за отображение количества символов. Быть встроенным в качестве аргументов главного компонента `React.DOM.div` всему пользовательскому интерфейсу нет никакой необходимости. Вы можете присвоить фрагменты пользовательского интерфейса переменным и применять их при возникновении определенных условий.

Теперь можно наблюдать за регистрируемыми методами управления жизненным циклом для обоих компонентов. На рис. 2.13 показано, что произойдет, когда загрузится страница, а затем изменится содержимое текстовой области.

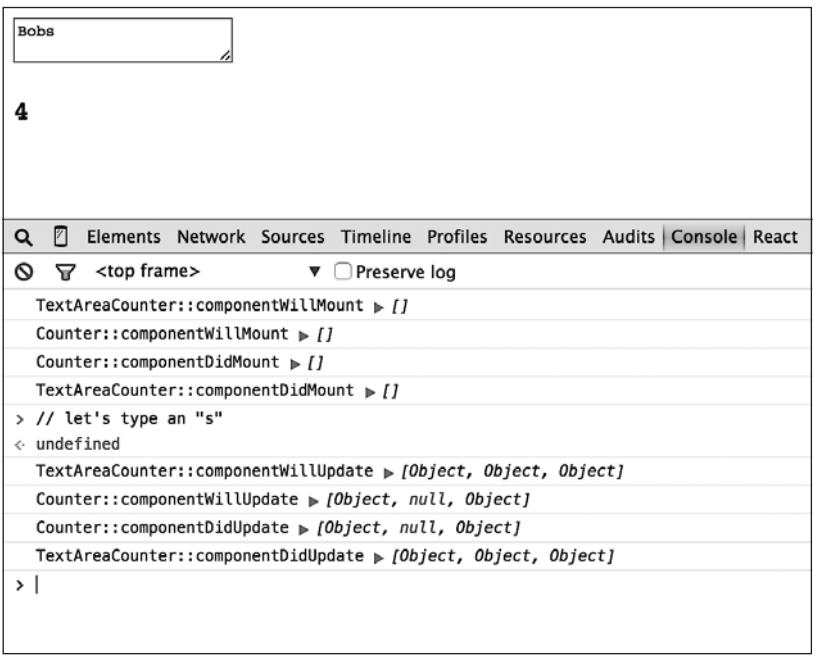


Рис. 2.13. Установка и обновление двух компонентов

Можно отследить, как дочерний компонент устанавливается и обновляется раньше своего родительского компонента.

На рис. 2.14 показано, что произойдет после удаления текста из текстовой области; значение `count` станет нулевым. В этом случае дочерний компонент `Counter` получит значение `null` и его DOM-узел будет удален из дерева DOM-модели после того, как вас уведомят с помощью функции обратного вызова `componentWillUnmount`.

Рис. 2.14. Демонтаж компонента counter

Выигрыш в производительности: предотвращение обновлений компонентов

Последний метод управления жизненным циклом, о котором вы должны знать, особенно при создании критичных к производительности частей вашего приложения, называется `shouldComponentUpdate(nextProps, nextState)`. Он вызывается перед вызовом метода `componentWillUpdate()` и предоставляет вам возможность отменить обновление (если вы посчитаете его ненужным).

Существует класс компонентов, использующих в своих методах `render()` только `this.props` и `this.state`, не прибегая к дополнительным вызовам функций. Эти компоненты называются чистыми компонентами. Они могут реализовать метод `shouldComponentUpdate()` и сравнить состояние и свойства до и после, а при отсутствии каких-либо изменений — возвращать `false`, экономя при этом долю вычислительных мощностей. Кроме того, бывают чисто статичные компоненты, не использующие ни свойства, ни состояние. Такие компоненты могут сразу же возвращать `false`.

Посмотрим, что произойдет с вызовом методов `render()` и реализацией `shouldComponentUpdate()` для получения выгоды в отношении производительности.

Сначала возьмем новый компонент `Counter`. Удалим из него примесь регистрирования и вместо этого станем отправлять регистрационную запись на консоль при каждом вызове метода `render()`:

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

Сделаем то же самое в компоненте `TextAreaCounter`:

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

  // все остальные методы ...

  render: function() {
    console.log(this.name + '::render()');
    // ... и вся остальная часть вывода на экран
  }
});
```

Когда страница будет загружена и вместо `"Bob"` будет вставлена строка `"LOL"`, вы сможете увидеть результат, показанный на рис. 2.15.

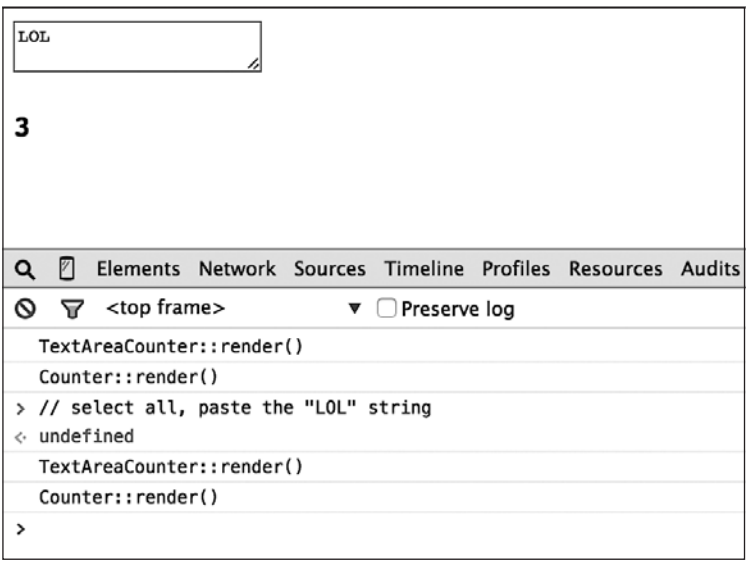


Рис. 2.15. Повторный вывод на экран обоих компонентов

Как видите, обновление текста приводит к вызову метода `render()` компонента `TextAreaCounter`, который, в свою очередь, становится причиной вызова метода `render()` компонента `Counter`. При замене `"Bob"` строкой `"LOL"` количество символов до и после обновления не меняется, следовательно, изменений в пользовательском интерфейсе счетчика не происходит и в вызове метода `render()` компонента `Counter` нет никакой необходимости. Вы можете помочь React оптимизировать этот случай, реализовав метод `shouldComponentUpdate()` и возвратив `false`, когда в последующем выводе на экран нет необходимости. Метод получает будущие значения свойств и состояния (в состоянии данный компонент не нуждается) и внутри себя сравнивает текущие и следующие значения:

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

Выполнение замены `"Bob"` на `"LOL"` не заставляет больше `Counter` заново выводиться на экран (рис. 2.16).

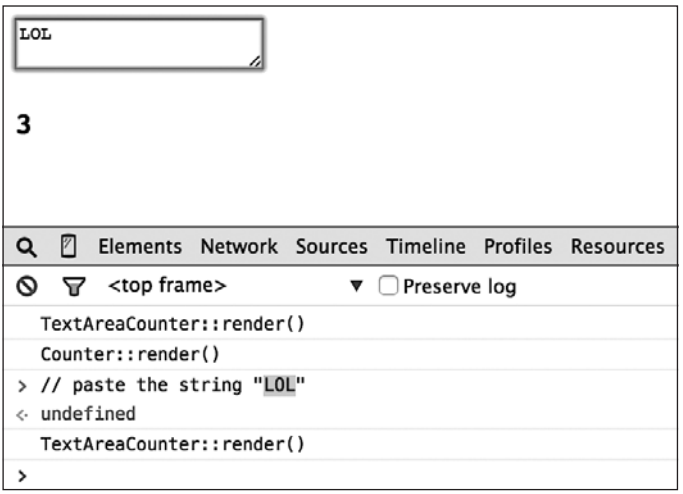


Рис. 2.16. Выигрыш в производительности: экономия одного цикла повторного вывода на экран

PureRenderMixin

Реализация `shouldComponentUpdate()` не отличается особой сложностью. И не такая уж трудная задача — превратить эту реализацию в универсальную, поскольку вы всегда сравниваете `this.props` с `nextProps` и `this.state` с `nextState`. React предоставляет одну такую универсальную реализацию в виде примеси, которую можно включать в любой компонент.

Вот как это делается:

```
<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>
```

Результат (рис. 2.17) не отличается от предыдущего — когда количество символов не меняется, метод `render()` компонента `Counter` не вызывается.

Следует отметить, что примесь `PureRenderMixin` не является частью ядра React, но входит в расширенную версию дополнений

к React. Следовательно, чтобы получить к нему доступ, вместо react/build/react.js следует включить в код react/build/react-with-addons.js. Это предоставит вам новое пространство имен `React.addons`, где наряду с другими полезными дополнениями можно найти и `PureRenderMixin`.

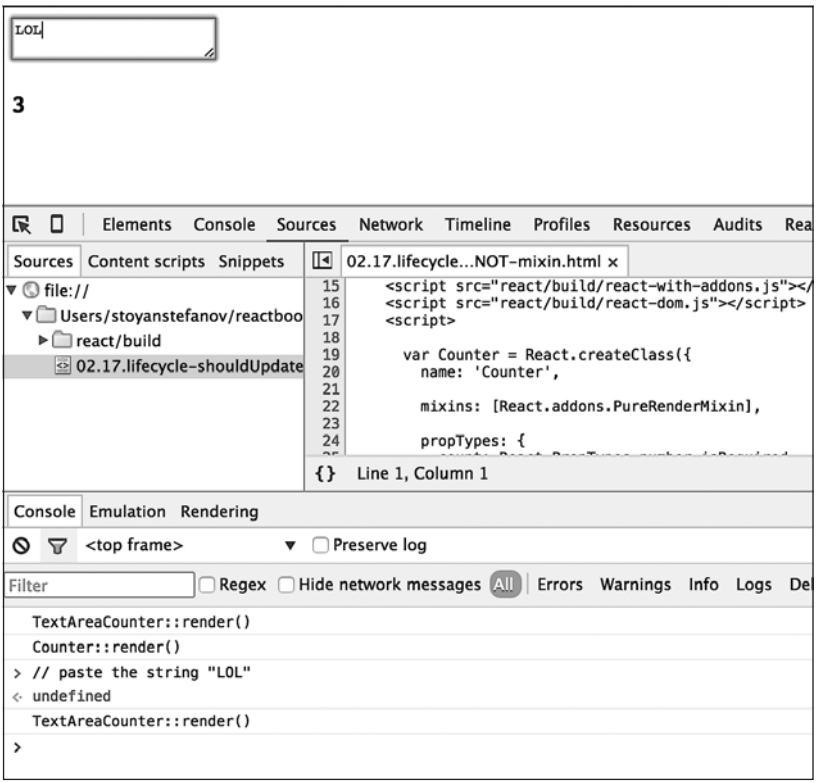


Рис. 2.17. Упрощенное получение выигрыша в производительности: подмешивание PureRenderMixin

Если не хочется включать все дополнения или нужно реализовать собственную версию примеси, не стесняйтесь заглянуть

в реализацию. Она предельно проста и понятна и служит всего лишь оболочкой для проверки равенства, представляя собой нечто подобное:

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

3 Excel: необычный табличный компонент

Вы уже знаете, как создаются пользовательские React-компоненты, как составляется (отображается) пользовательский интерфейс с применением стандартных DOM-компонентов и ваших собственных пользовательских компонентов, как задаются свойства, осуществляется работа с состоянием, производится внедрение в жизненный цикл компонента и оптимизируется производительность путем отказа от отображения при отсутствии необходимости.

Объединим все это (и в процессе приобретем новые значения о React) путем создания более интересного компонента — таблицы данных.

Это будет нечто вроде раннего прототипа Microsoft Excel v.0.1.beta, позволяющего редактировать содержимое таблицы данных, а также производить сортировку, поиск (фильтрацию) и экспорт данных в виде загружаемых файлов.

Начнем с данных

Таблицы целиком завязаны на данных, поэтому компонент необычной таблицы (почему бы не назвать его Excel?) должен получать массив данных и массив заголовков. В целях тестирования позаимствуем список книг-бестселлеров из «Википедии»:

```
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"],
];
```

Цикл создания заголовков таблицы

Первый шаг (просто чтобы было с чего начать) заключается в отображении одних лишь заголовков. Простейшая реализация должна выглядеть следующим образом:

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

Теперь, располагая работоспособным компонентом, можно посмотреть, как им пользоваться:

```
ReactDOM.render(
  React.createElement(Excel, {
    headers: headers,
    initialData: data,
  }),
  document.getElementById("app")
);
```

Результат этого начального примера показан на рис. 3.1.

Здесь появилось кое-что новое — применяемый с массивами метод `map()`, который используется для возвращения массива дочерних компонентов. Метод `map()` берет каждый элемент (в данном случае из массива `headers`) и передает его функции обратного вызова. Здесь функция обратного вызова создает и возвращает новый компонент `<th>`.

В этом и проявляется красота React: вы используете JavaScript для создания вашего пользовательского интерфейса и вам доступна

вся мощь JavaScript. Циклы и задания условий работают в обычном порядке и вам для создания пользовательского интерфейса не нужно учить еще один язык создания шаблонов или синтаксис.

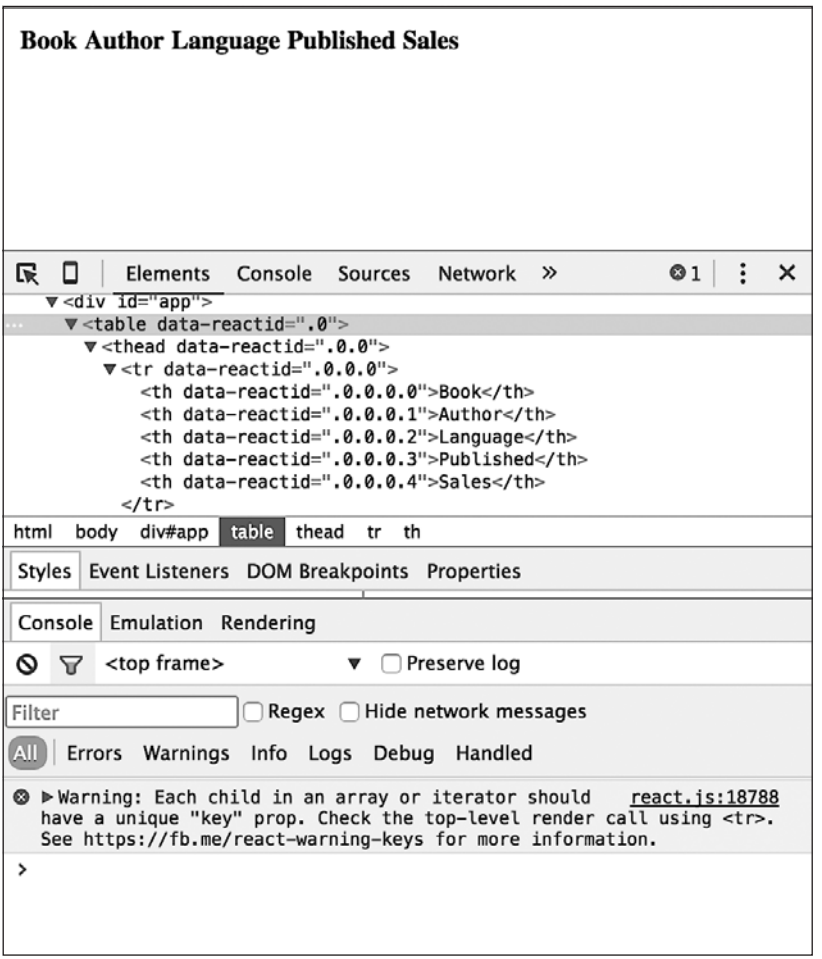


Рис. 3.1. Отображение заголовков таблицы

Вместо того, что вы видели до сих пор (как каждый дочерний компонент передавался в виде отдельных аргументов), дочерние компоненты можно передать компоненту в виде одного аргумента, представляющего собой массив. То есть работают оба варианта:

```
// отдельные аргументы
React.DOM.ul(
  null,
  React.DOM.li(null, 'one'),
  React.DOM.li(null, 'two')
);

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

Отладка для избавления от консольного предупреждения

Копия экрана на рис. 3.1 показывает предупреждение, выведенное в консоли. О чем оно сообщает и как можно исправить ситуацию? В предупреждении говорится, что каждый дочерний компонент в массиве или итераторе должен иметь уникальное свойство `key` и что нужно проверить вызов функции отображения верхнего уровня, использующий `<tr>` (Warning: Each child in an array or iterator should have a unique “key” prop. Check the top-level render call using <tr>).

Проверить вызов функции отображения, использующий `<tr>`? Поскольку в приложении имеется только один компонент, нетрудно прийти к выводу, что проблема заключается в нем, но в реальной жизни у вас может быть множество компонентов, создающих элементы `<tr>`. `Excel` является простой переменной, которой присваивается React-компонент за пределами мира React, следовательно, React не в состоянии определить имя этого компонента. Вы можете ему помочь, объявив свойство `displayName`:

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

Теперь React может идентифицировать источник проблемы и выдать вам предупреждение, сообщающее, что у каждого дочернего компонента в массиве должно быть уникальное свойство ключа `key` и что вам следует проверить метод отображения, принадлежащий компоненту `Excel`. Уже гораздо лучше. Но предупреждение все равно появляется. Чтобы исправить ситуацию теперь, когда известно, какой из методов `render()` «виноват», нужно просто сделать то, о чем говорится в предупреждении:

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

Что здесь происходит? Функции обратного вызова, передаваемые методу `Array.prototype.map()`, снабжаются тремя аргументами: значением массива, его индексом (0, 1, 2 и т. д.), а также всем массивом. Чтобы дать React свойство `key`, вы можете использовать индекс (`idx`) элемента массива и покончить с этим делом. Уникальность ключей должна соблюдаться только внутри этого массива, но не во всем React-приложении.

Теперь, когда проблема с ключами решена, можно с помощью небольшого участия CSS получить удовольствие от работы версии 0.0.1 вашего нового компонента — все имеет весьма достойный вид и дело обходится без предупреждения (рис. 3.2).

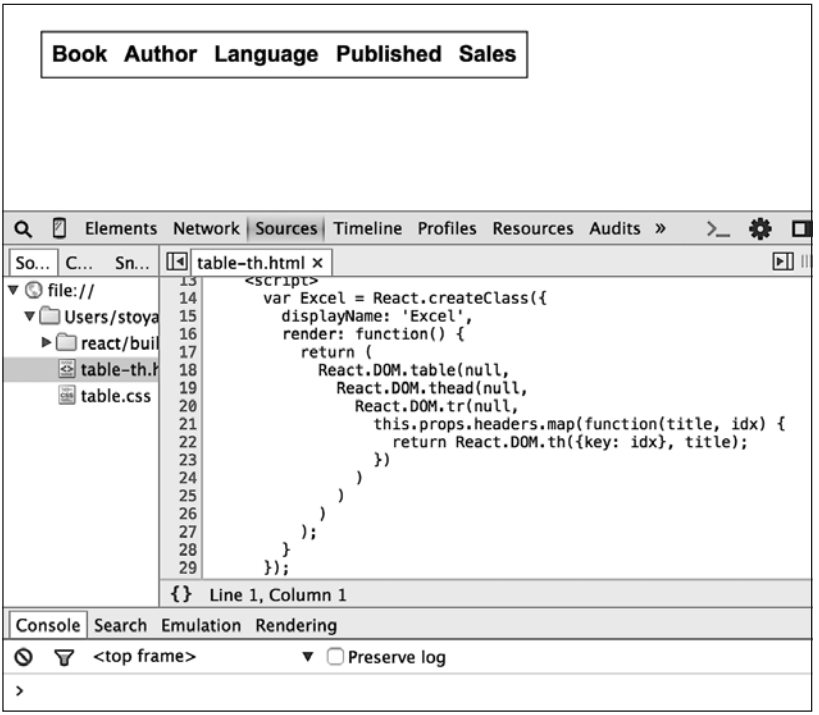


Рис. 3.2. Вывод на экран заголовков таблицы без появления предупреждения

Добавление displayName только в целях отладки может показаться лишними хлопотами, но их можно избежать: при использовании технологии JSX (рассматриваемой в главе 4) вам уже не придется определять это свойство, поскольку имя генерируется автоматически.

Добавление содержимого <td>

После получения вполне подходящего заголовка таблицы настало время добавить в нее основное наполнение. Содержимое заголовка представляет собой одномерный массив (одну строку), а вот данные наполнения имеют два измерения. Следовательно, вам нужны два цикла: один для прохода по строкам, второй — для прохода по данным (ячейкам) каждой строки. Это можно выполнить, используя те же самые циклы `.map()`, о порядке применения которых вы уже знаете:

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

Еще нужно рассмотреть содержимое переменной `data` и ответить на вопрос: откуда оно берется и как изменяется? Код, вызывающий ваш компонент `Excel`, должен иметь возможность передавать данные для инициализации таблицы. Но позже, после появления таблицы, данные будут меняться, поскольку пользователь должен иметь возможность их сортировать, редактировать и т. д. Иными словами, состояние компонента будет изменяться. Поэтому воспользуемся свойством `this.state.data` для отслеживания изменений и свойством `this.props.initialData`, чтобы позволить вызывающему коду инициализировать компонент. Теперь полноценная реализация может приобрести следующий вид (результат показан на рис. 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);
              })
            )
          );
        })
      )
    )
  );
}
```

В коде можно увидеть повторяющийся фрагмент `{key: idx}`, дающий уникальный ключ каждому элементу массива компонентов. Хотя все циклы `.map()` начинаются с индекса 0, проблемы не возникает, поскольку ключи должны быть уникальными только в текущем цикле, а не во всем приложении.

Функция render() уже приобретает трудноконтролируемый вид, особенно если дело касается отслеживания закрывающих символов } и). Но не стоит переживать — технология JSX готова исправить ситуацию!

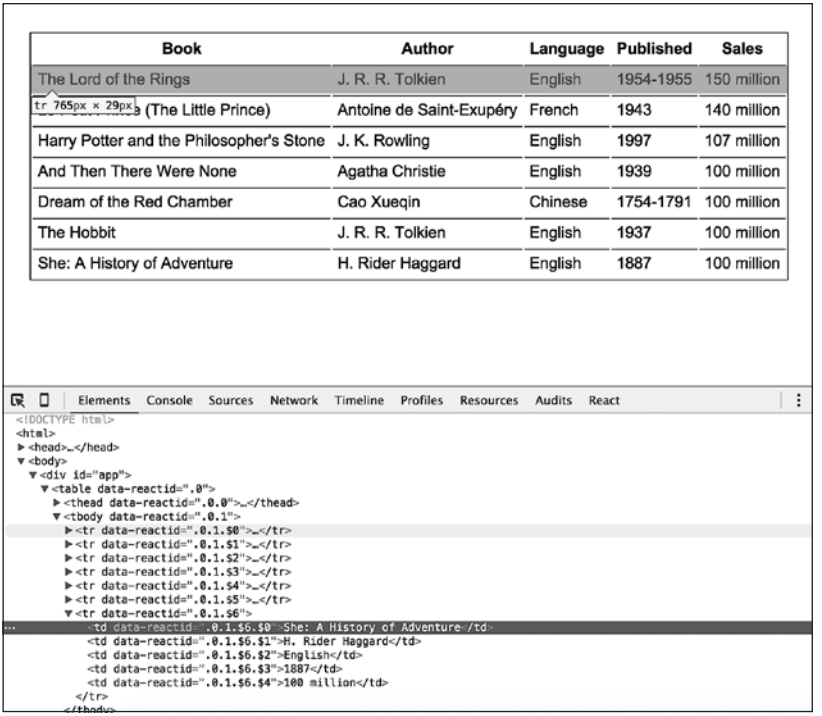


Рис. 3.3. Вывод на экран всей таблицы

В предыдущем фрагменте кода отсутствует свойство `propTypes` (чье присутствие необязательно, но зачастую желательно). Оно способствует проверке допустимости данных и документированию компонента. Применим конкретный подход и попробуем максимально сократить вероятность предоставления кем-либо нежелательных данных нашему привлекательному компоненту `Excel`. Объект `React.PropTypes` предлагает метод проверки массива, позволяющий убедиться в том, что свойство всегда является массивом. Но этим дело не заканчивается, в `arrayOf` можно также указать тип элементов массива. В данном случае зададим возмож-

ность использования для названия заголовков и для данных только строковых значений:

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

Теперь наведен полный порядок!

Как можно усовершенствовать компонент? Использование в универсальной электронной таблице Excel одних только строковых значений можно назвать излишней ограничительной мерой. В качестве упражнения вы можете разрешить применение более широкого набора типов данных (`React.PropTypes.any`) и осуществить их отображение по-разному в зависимости от типа (например, применить выравнивание чисел по правому краю).

Сортировка

Сколько раз вам доводилось смотреть на таблицу на веб-странице и испытывать острое желание отсортировать ее данные по-другому? К счастью, с React это делается довольно просто. Это станет отличным примером того, как React с блеском справляется с поставленными задачами, поскольку вам нужно лишь отсортировать массив данных, а все обновления пользовательского интерфейса будут выполнены без вашего участия.

Сначала добавим в строку заголовка обработчик щелчка:

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

Теперь реализуем функцию `_sort`. Нужно знать, по какому столбцу производить сортировку, и эти сведения удобнее всего будет извлечь, воспользовавшись свойством `cellIndex` цели события (целью события является заголовок таблицы `<th>`):

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

Использование свойства cellIndex при разработке приложений встречается довольно редко. Оно определено еще в спецификации DOM Level 1 в качестве «индекса ячейки в строке», а чуть позже, в спецификации DOM Level 2, объявлено доступным только для чтения.

Вам также понадобится копия сортируемых данных. В противном случае, если воспользоваться принадлежащим массиву методом `sort()` напрямую, он внесет в массив изменения, а значит, метод `this.state.data.sort()` изменит значение `this.state`. Как вы уже знаете, значение `this.state` не должно изменяться напрямую, это осуществляется только с помощью метода `setState()`:

```
// копирование данных
var data = this.state.data.slice();
// или 'Array.from(this.state.data)' в ES6
```

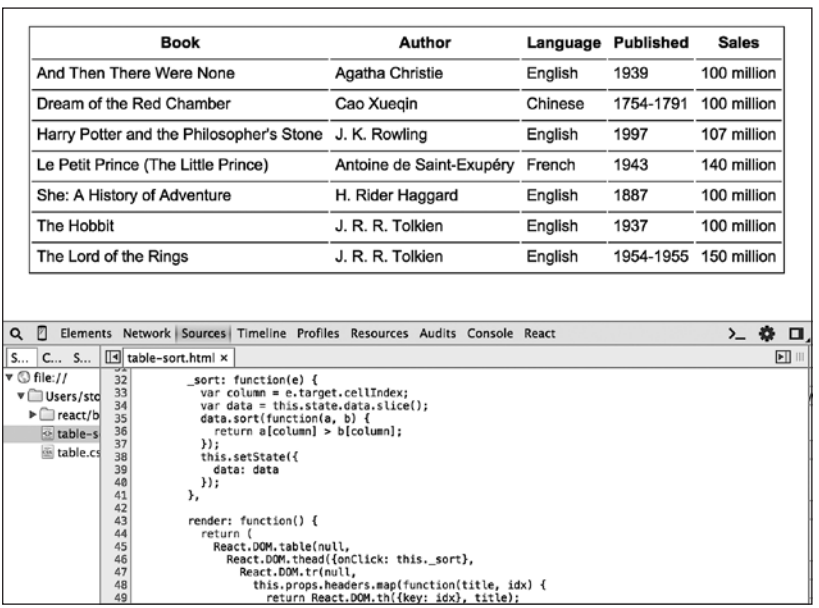
Теперь сортировка выполняется с помощью функции обратного вызова метода `sort()`:

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

И наконец, в следующей строке состояние настраивается под новые отсортированные данные:

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

Теперь при щелчке на заголовке содержимое сортируется в алфавитном порядке (рис. 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

Рис. 3.4. Сортировка книг по названию

Вот, собственно, и все — касаться отображения пользовательского интерфейса вам вообще не требуется. В методе `render()` вы уже раз и навсегда определили, как должен выглядеть компонент с получением данных. Когда данные изменяются, изменяется и пользовательский интерфейс, но это уже не ваша забота.

Как можно усовершенствовать компонент? Эта сортировка не отличается особой сложностью, но ее вполне хватает для рассмотрения возможностей React. Вы можете развивать фантазию, анализируя содержимое на предмет наличия числовых значений, с единицами измерения или без них и т. д.

Создание индикации сортировки в пользовательском интерфейсе

Таблица отсортирована подходящим образом, но по какому именно столбцу — непонятно. Обновим пользовательский интерфейс, чтобы в нем на основе отсортированного столбца были показаны стрелки. А попутно еще реализуем сортировку по убыванию.

Чтобы отслеживать новое состояние, нужны два новых свойства.

- `this.state.sortby`. Индекс столбца, подвергаемого сортировке.
- `this.state.descending`. Булево значение для определения порядка выполняемой сортировки — по возрастанию или по убыванию.

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

В функции `_sort()` нужно определить, в каком порядке будет вестись сортировка. По умолчанию она будет выполняться по возрастанию, если только индекс нового столбца не будет таким

же, как и индекс столбца, по которому уже была произведена сортировка, и если эта сортировка не была выполнена в порядке убывания:

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

Нужно также немного подправить функцию обратного вызова, используемую для сортировки:

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

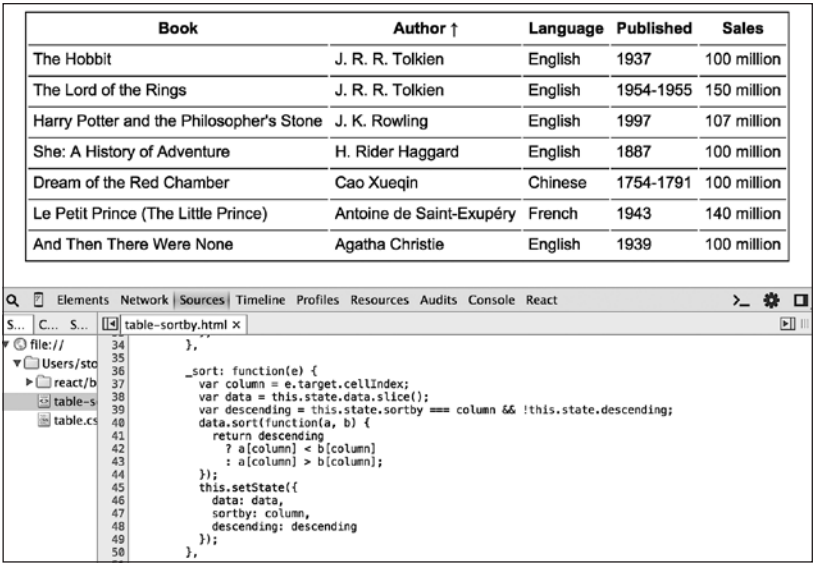
И наконец, необходимо настроить новое состояние:

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

Осталось лишь обновить функцию `render()` для обозначения направления сортировки. Просто добавим к названию текущего столбца, по которому выполняется сортировка, символ стрелки:

```
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)
```

Вот теперь сортировку можно считать функционально завершенной — ее можно проводить по любому столбцу, щелкнув один раз для сортировки в порядке возрастания и тут же еще один раз для сортировки в порядке убывания; пользовательский интерфейс обновится с предоставлением визуальной индикации (рис. 3.5).



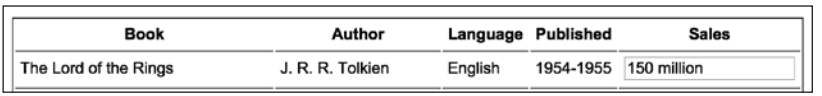
Book	Author ↑	Language	Published	Sales
The Hobbit	J. R. R. Tolkien	English	1937	100 million
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	150 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	107 million
She: A History of Adventure	H. Rider Haggard	English	1887	100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

Рис. 3.5. Сортировка в порядке возрастания и убывания

Редактирование данных

Следующий шаг в создании компонента `Excel` — предоставить пользователям возможность редактировать содержимое таблицы. Один из способов включает следующие шаги.

1. Вы дважды щелкаете кнопкой мыши на ячейке. Компонент `Excel` определяет, на какой именно ячейке был сделан двойной щелчок, и превращает ее в поле ввода с ранее введенным в него содержимым (рис. 3.6).



Book	Author	Language	Published	Sales
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	150 million

Рис. 3.6. Ячейка таблицы превращается в поле ввода после двойного щелчка

2. Редактируете содержимое (рис. 3.7).

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

Рис. 3.7. Редактирование содержимого

3. Нажимаете клавишу Enter. Поле ввода исчезает, и таблица обновляется (прежний текст заменяется новым) (рис. 3.8).

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

Рис. 3.8. Содержимое обновляется после нажатия клавиши Enter

Редактируемая ячейка

Первое, что нужно сделать, — настроить простой обработчик событий. По двойному щелчку компонент «запоминает» выбранную ячейку:

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

Обратите внимание на дружелюбную, легко читаемую форму записи onDoubleClick вместо определяемого в спецификации W3C ondblclick.

Посмотрим, как выглядит `_showEditor`:

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

Что здесь происходит?

- Функция устанавливает свойство `edit` состояния `this.state`. Этот свойство имеет значение `null`, когда редактирование не выполняется, а затем превращается в объект со свойствами `row` и `cell`, в которых содержатся индекс строки и индекс редактируемой ячейки. Итак, если вы выполните двойной щелчок на самой первой ячейке, `this.state.edit` получит значение `{row: 0, cell: 0}`.
- Чтобы определить индекс ячейки, используется, как и раньше, свойство `e.target.cellIndex`, где в качестве `e.target` фигурирует элемент `<td>`, на котором был сделан двойной щелчок.
- DOM-модель не предоставляет никакого свойства для индекса строки вроде `rowIndex`, поэтому получить этот индекс нужно самостоятельно через атрибут `data-`. У каждой ячейки должен быть атрибут `data-row` с индексом строки, который с помощью метода `parseInt()` можно проанализировать и превратить в целочисленное значение для получения обратно индекса строки.

Есть еще несколько уточнений и предварительных условий. Свойства `edit` прежде не существовало, и оно также должно быть инициализировано в методе `getInitialState()`, который теперь приобретет следующий вид:

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

Для отслеживания индексов строк понадобится свойство `data-row`. Посмотрим, на что похожа вся конструкция `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 (что нужно сделать) – превратить 'содержимое'
          // в поле ввода, если 'idx' и 'rowidx' соответствуют
          // редактируемой ячейке; в противном случае просто
          // показать содержимое

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

И наконец, нужно сделать то, что предписано комментарием `TODO`. Создадим поле ввода там, где это требуется. Из-за вызова `setState()`, устанавливающего свойство `edit`, снова вызывается вся функция `render()`. React отображает таблицу, которая предоставляет возможность обновления ячейки, на которой будет сделан двойной щелчок.

Поле ввода ячейки

Посмотрим на код, заменяющий комментарий `TODO`. Сначала нужно вспомнить о состоянии редактирования:

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

Проверяем, установлено ли свойство `edit`, и если да, то именно ли эта ячейка редактируется:

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

Если это целевая ячейка, создадим форму и поле ввода с содержимым этой ячейки:

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

Как видите, это форма с одним полем ввода, которое предварительно уже заполнено текстом из ячейки. При отправке формы управление перехватится и будет передано закрытому методу `_save()`.

Сохранение

Последний фрагмент пазла редактирования — сохранение изменений содержимого после того, как пользователь завершит набор текста и отправит форму (путем нажатия клавиши Enter):

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

После исключения возможности поведения по умолчанию (чтобы страница не перезагружалась) нужно получить ссылку на поле ввода:

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

Клонируем данные, чтобы не пришлось работать с `this.state` напрямую:

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

Обновляем часть данных, используя новое значение и индексы ячейки и строки, сохраненные в свойстве `edit` состояния `state`:

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

И наконец, устанавливаем состояние, вызывая тем самым повторное отображение пользовательского интерфейса:

```
this.setState({
    edit: null, // редактирование выполнено
    data: data,
});
```

Выводы и определение различий в виртуальной DOM-модели

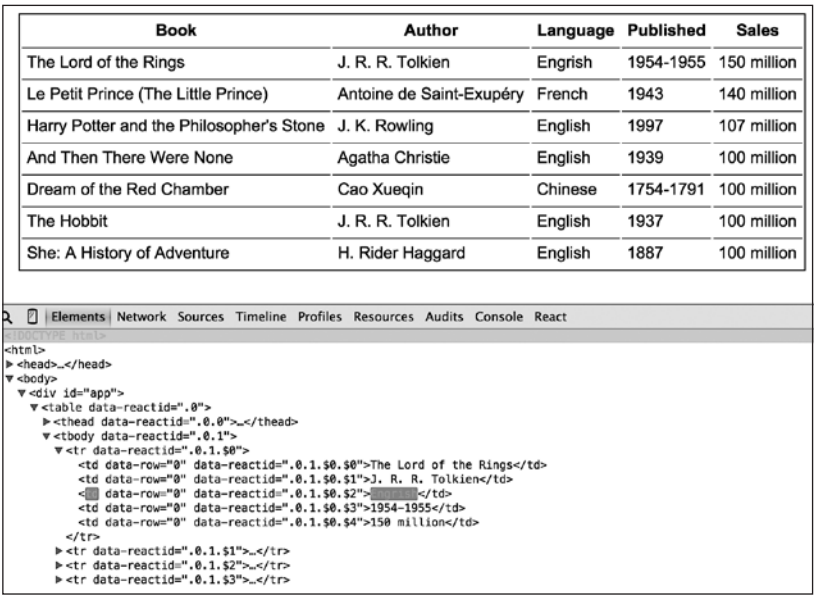
Итак, с редактированием покончено. Уложились в весьма скромный объем кода. Нам для этого потребовалось всего лишь следующее:

- отследить с помощью `this.state.edit`, какую из ячеек редактировать;
- отобразить поле ввода при показе таблицы, если индексы строки и ячейки соответствуют ячейке, на которой пользователь сделал двойной щелчок;
- обновить массив данных новым значением из поля ввода.

Как только метод `setState()` будет вызван с новыми данными, React вызовет принадлежащий компоненту метод `render()` — и пользовательский интерфейс волшебным образом обновится. Можно подумать, что нерационально было бы отображать всю

таблицу из-за изменения содержимого всего одной ячейки. Фактически React только одну ячейку и обновляет.

Если открыть инструментарий разработчика вашего браузера, можно увидеть, какая часть DOM-дерева обновлена в ходе взаимодействия с вашим приложением. На рис. 3.9 показана область инструментария разработчика, в которой выделена та часть DOM-модели, которая была изменена при внесении правки в поле языка для книги с названием The Lord of the Rings, превращающей English в Engrish.



Book	Author	Language	Published	Sales
The Lord of the Rings	J. R. R. Tolkien	Engrish	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

Рис. 3.9. Выделение изменений в DOM-модели

Закулисно React вызывает метод `render()` и создает упрощенное представление дерева желаемого результата, достигаемого в DOM-модели. Это представление известно как *виртуальное DOM-дерево*.

Когда метод `render()` вызывается снова (к примеру, после вызова `setState()`), React сравнивает виртуальное дерево до и после этого вызова и вычисляет различие. Основываясь на этом различии, React определяет для выполнения нужного изменения в DOM-модели браузера минимально требуемые DOM-операции (например, `appendChild()`, `textContent` и т. д.).

На рис. 3.9 показано, что потребовалось только одно изменение ячейки и нет необходимости повторно отображать всю таблицу. Вычисляя минимальный набор изменений и объединяя DOM-операции в пакеты, React тем самым очень бережно относится к DOM-модели в силу известной проблемы с неспешным выполнением DOM-операций (по сравнению с чистыми операциями JavaScript, вызовами функций и т. д.), поскольку зачастую узким местом солидных веб-приложений является производительность операций, связанных с отображением данных.

Короче говоря, когда дело касается производительности и обновления пользовательского интерфейса, React подставляет вам свое плечо путем:

- бережного отношения к DOM-модели;
- использования делегирования событий при взаимодействии с пользователем.

Поиск

А теперь добавим к компоненту `Excel` функцию поиска, позволяющую пользователям выполнять фильтрацию содержимого таблицы. План таков:

- добавить кнопку для включения и выключения новой функции (рис. 3.10);

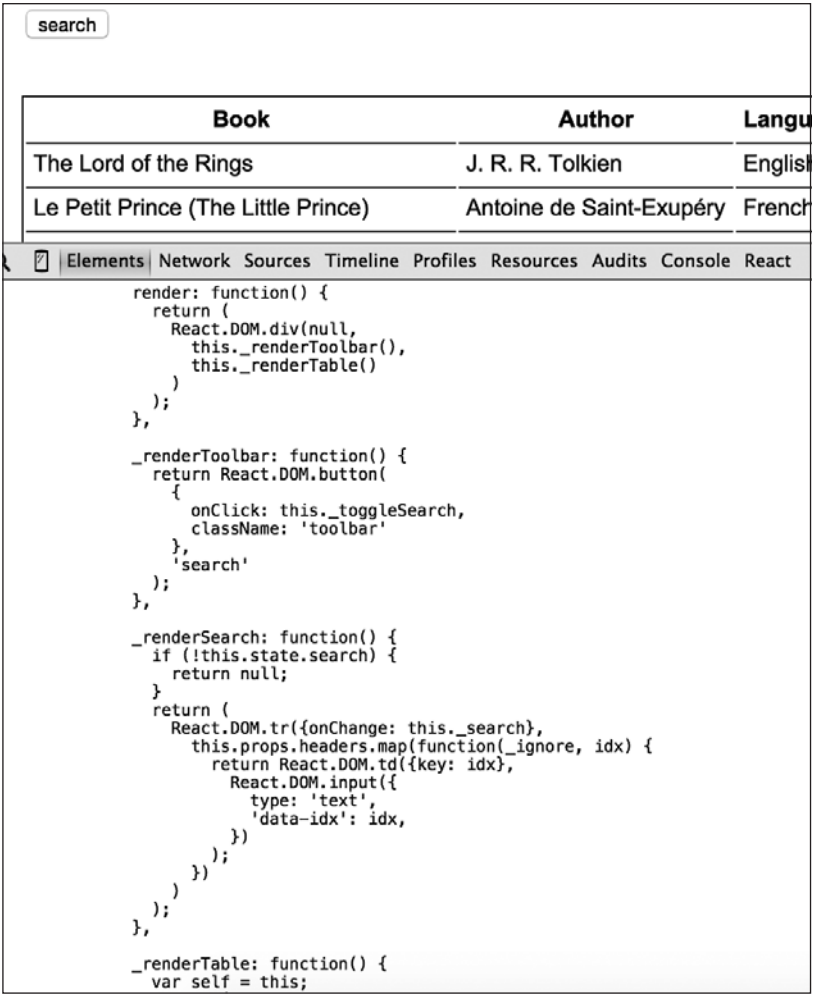


Рис. 3.10. Кнопка поиска

- если поиск включен, добавить строку полей ввода, каждое из которых предназначено для поиска в соответствующем столбце (рис. 3.11);

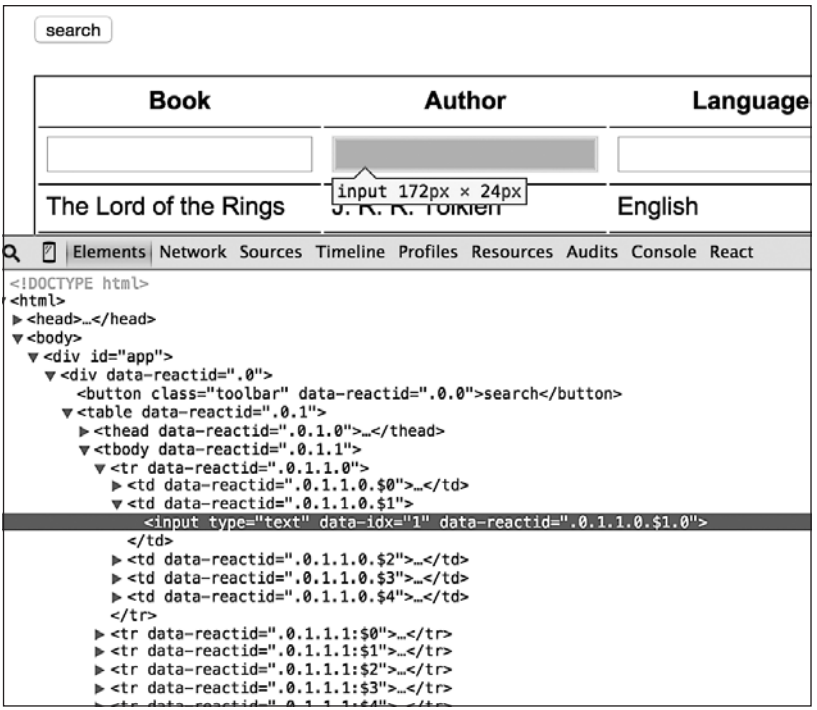


Рис. 3.11. Строка полей ввода для поиска и фильтрации

- по мере того как пользователь набирает текст в поле ввода, выполнять фильтрацию массива `state.data`, чтобы показывалось только соответствующее содержимое (рис. 3.12).

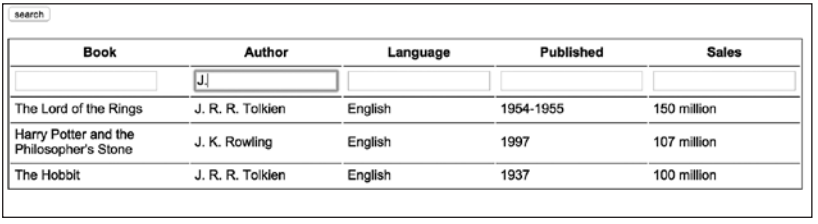


Рис. 3.12. Результаты поиска

Состояние и пользовательский интерфейс

Сначала нужно добавить к объекту `this.state` свойство `search`, чтобы отслеживать, включена или нет функция поиска:

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

Затем наступает очередь обновления пользовательского интерфейса. Чтобы упростить сопровождение кода, разобьем функцию `render()` на небольшие специализированные части. До сих пор функция `render()` занималась только отображением таблицы. Переименуем ее в `_renderTable()`. Далее кнопка поиска (Search) должна стать частью полноценной панели инструментов (вскоре к ней добавится кнопка экспорта Export), поэтому сделаем ее отображение частью функции отображения панели инструментов под названием `_renderToolbar()`.

Результат выглядит следующим образом:

```
render: function() {
  return (
    React.DOM.div(null,
      this._renderToolbar(),
      this._renderTable()
    )
  );
},

_renderToolbar: function() {
```

```
  // TODO
},

_renderTable: function() {
  // тот же код, что и у функции,
  // прежде известной как 'render()'
},
```

Как видите, новая функция `render()` возвращает контейнер `div` с двумя дочерними элементами: панелью инструментов и таблицей. Вы уже знаете, как выглядит отображение таблицы, а панель инструментов пока что представлена всего лишь одной кнопкой:

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

Если поиск включен (это означает, что у свойства `this.state.search` значение `true`), вам нужна новая строка таблицы, заполненная полями ввода. Создадим функцию `_renderSearch()`, которая позаботится о появлении этой строки:

```
_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,
          })
        );
      })
    )
  );
},
```

Как видите, если поиск не включен, функции не нужно ничего отображать — и она возвращает `null`. Другой вариант, разумеется, заключается в том, чтобы решение было принято в вызывающем эту функцию коде — и она вообще бы не вызывалась, если поиск не включен. Но предыдущий пример помогает немного упростить уже загруженную работой функцию `_renderTable()`. Вот что нужно сделать функции `_renderTable()`:

До:

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

После:

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

Поля ввода поиска всего лишь еще один дочерний узел перед основным циклом `data` (тем, который создает все строки и ячейки таблицы). Когда `_renderSearch()` возвращает `null`, React просто не отображает дополнительный дочерний узел и переходит к отображению таблицы.

Итак, все об обновлениях пользовательского интерфейса уже сказано. С вашего позволения, рассмотрим сам механизм поиска, «деловую логику», то есть фактический поиск.

Фильтрация содержимого

Функция поиска представляется довольно простой: взять массив данных, вызвать в отношении него метод `Array.prototype.filter()` и возвратить отфильтрованный массив с элементами, соответствующими строке поиска.

Пользовательский интерфейс по-прежнему использует для отображения `this.state.data`, но это значение `this.state.data` является урезанной версией предыдущего значения.

Прежде чем вести поиск, нужно скопировать данные, чтобы они не были утеряны навсегда. Это позволит пользователю вернуться к полной таблице или изменить строку поиска для получения другого соответствия. Назовем эту копию (фактически ссылку) `_preSearchData`:

```
var Excel = React.createClass({
  // содержимое...

  _preSearchData: null,

  // еще содержимое...
});
```

Когда пользователь нажимает кнопку Search, вызывается функция `_toggleSearch()`. Задача этой функции — запускать и останавливать поиск. Свою задачу она выполняет путем:

- установки для `this.state.search` значения `true` или `false` соответственно;
- запоминания прежних данных, когда поиск включается;
- возвращения к старым данным при выключении поиска.

Функция может принять следующий вид:

```
_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,
    });
  }
},
```

Остается лишь реализовать функцию `_search()`, вызываемую при каждом изменении в строке поиска, означающем, что пользователь набрал что-то в одном из полей ввода. Полная реализация с некоторыми дополнительными особенностями имеет следующий вид:

```
_search: function(e) {
  var needle = e.target.value.toLowerCase();
  if (!needle) { // строка поиска будет удалена
    this.setState({data: this._preSearchData});
    return;
  }
  var idx = e.target.dataset.idx; // в каком столбце искать
  var searchdata = this._preSearchData.filter(function(row) {
    return row[idx].toString().toLowerCase().indexOf(needle)
      > -1;
  });
  this.setState({data: searchdata});
},
```

Строка поиска берется из измененной цели события (являющейся полем ввода):

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

Если там нет строки поиска (пользователь стер набранное), функция берет исходные кэшированные данные — и они становятся новым состоянием:

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

Если строка поиска имеется, происходит фильтрация исходных данных — и отфильтрованные результаты устанавливаются в качестве нового состояния данных:

```
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});
```

И на этом функцию поиска можно считать выполненной. Для реализации данной функции вам пришлось сделать всего лишь следующее:

- добавить пользовательский интерфейс поиска;
- показывать или скрывать новый пользовательский интерфейс по мере надобности;
- создать «деловую логику», представляющую собой вызов метода `filter()` в отношении массива.

Ничего из исходного отображения таблицы менять не пришлось. Как и всегда, все заботы свелись к состоянию ваших данных и к разрешению React позаботиться об отображении (и о работе, связанной с DOM-моделью) при каждом изменении состояния данных.

Как можно усовершенствовать поиск?

Это был простой рабочий пример, использованный для иллюстрации возможностей. Можно ли усовершенствовать функцию?

Простым усовершенствованием может стать переключение надписи на кнопке поиска. Чтобы, когда поиск включен (`this.state.search === true`), она сообщала: «Выполняется поиск».

Можно еще попробовать реализовать дополнительный поиск в нескольких полях, то есть фильтровать уже отфильтрованные данные. Если пользователь в строке языка набирает `Eng`, а затем ведет поиск, используя другое поле ввода данных для поиска, почему бы не провести поиск только в результатах предыдущего поиска? Как бы вы реализовали такую функцию?

Мгновенное воспроизведение

Теперь уже известно, что ваши компоненты «заботятся» о своем состоянии и позволяют React выполнять свое отображение и повторное отображение при соответствующих обстоятельствах. Это означает, что при одних и тех же данных (состоянии и свойствах) приложение будет выглядеть абсолютно одинаково независимо от того, что изменилось до или после этого конкретного состояния данных. Это предоставляет вам отличную возможность для проведения отладки в реальных условиях.

Представим, что кто-то при использовании вашего приложения обнаружил ошибку; он может щелкнуть на кнопке для создания отчета об ошибке без объяснения того, что случилось. Этот отчет может отправить вам копию `this.state` и `this.props` — и вы сможете воссоздать точное состояние приложения и посмотреть на визуальный результат.

Откат может стать еще одной функцией на основе того, что React отображает ваше приложение одинаково при задании тех же самых свойств и состояния. Откат реализуется довольно просто: нужно всего лишь вернуться к предыдущему состоянию.

Ради интереса немного разовьем эту мысль. Мы будем записывать каждое изменение состояния в компоненте `Excel`, а затем воспроизводить его. Весьма интересно будет понаблюдать за разворачивающейся перед вами картиной всех ваших действий в обратном порядке.

В плане реализации не станем задаваться вопросом, *когда* произошло изменение, а просто проиграем изменения состояний приложения с односекундным интервалом. Для реализации данной функции вам понадобится всего лишь добавить метод `_logSetState()` и осуществить поиск-замену всех вызовов `setState()` вызовами новой функции.

Итак, все вызовы:

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

становятся:

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

Методу `_logSetState` предстоит выполнять два действия: регистрировать новое состояние, а затем передавать его по эстафете методу `setState()`. Вот как выглядит один из примеров, где делается глубокая копия состояния, добавляемая к `this._log`:

```
var Excel = React.createClass({

  _log: [],

  _logSetState: function(newState) {
  // запоминание старого состояния в клоне
  this._log.push(JSON.parse(JSON.stringify(
    this._log.length === 0 ? this.state : newState
```

```
  )));
  this.setState(newState);
},
  // ...
});
```

Теперь, после регистрации всех изменений состояния, проиграем их назад. Для запуска проигрывания добавим простой отслеживатель события, перехватывающий действия с клавиатурой и запускающий функцию `_replay()`:

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

И наконец, добавим метод `_replay()`. Он использует `setInterval()` и один раз в секунду считывает следующий объект из регистрационного журнала и передает его методу `setState()`:

```
_replay: function() {
  if (this._log.length === 0) {
    console.warn('Состояния для проигрывания отсутствуют');
    return;
  }
  var idx = -1;
  var interval = setInterval(function() {
    idx++;
    if (idx === this._log.length - 1) { // конец
      clearInterval(interval);
    }
    this.setState(this._log[idx]);
  }.bind(this), 1000);
},
```

Как можно усовершенствовать воспроизведение?

Что если реализовать функцию «откат-возвращение»? Скажем, когда нажимают сочетание клавиш Alt+Z, происходит откат на один шаг в журнале состояния, а при нажатии сочетания Alt+Shift+Z выполняется один шаг вперед по записям журнала.

А возможна ли альтернативная реализация?

Существует ли другой способ реализации функциональных возможностей типа «воспроизведение-откат» без изменения всех ваших вызовов `setState()`? Может быть, стоит для этого воспользоваться соответствующим методом управления жизненным циклом компонента (рассмотренным в главе 2)?

Скачивание данных таблицы

После всех сортировок, редактирований и поисков пользователь наконец-то доволен состоянием данных в таблице. Было бы неплохо, если бы он мог скачать результат своих усилий, чтобы с этим можно было поработать в другое время.

К счастью, в React это сделать проще простого. Нужно всего лишь забрать текущее значение `this.state.data` и вернуть его назад в JSON- или CSV-формате.

На рис. 3.13 показан конечный результат щелчка пользователем на кнопке Export CSV, скачивания файла data.csv (посмотрите в нижний левый угол окна браузера) и открытия этого файла в Microsoft Excel.

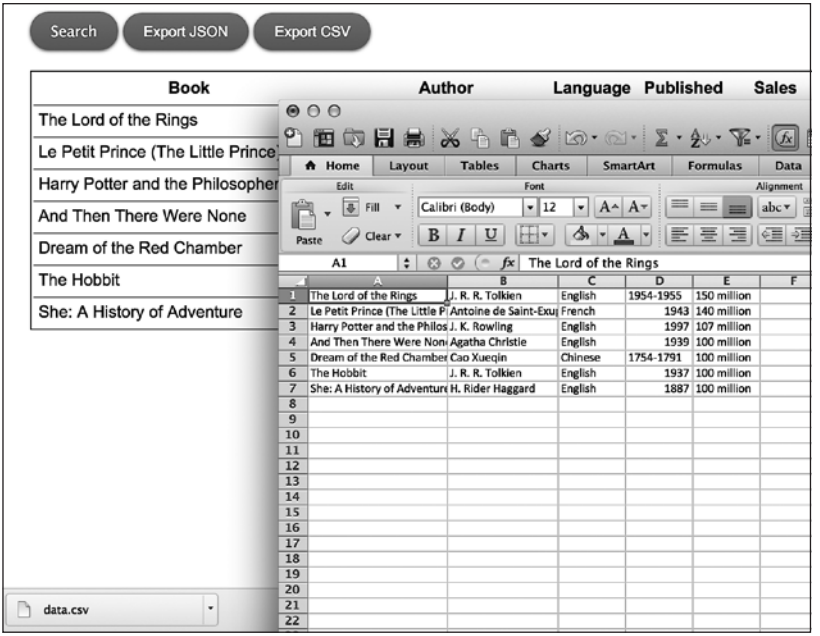


Рис. 3.13. Экспорт данных таблицы в Microsoft Excel с использованием CSV-формата

Сначала нужно добавить новые возможности на панель инструментов. Воспользуемся для этого небольшой магией HTML5, заставляющей ссылки, образуемые тегом `<a>`, запускать скачивание файлов, и сделаем так, чтобы с использованием технологии CSS новые «кнопки» становились ссылками с видом кнопок:

```
_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')
  );
},
```

Теперь обратимся к функции скачивания `_download()`. По сравнению с легкостью экспортирования в формате JSON для использования формата CSV нужно приложить немного больше усилий. Проще говоря, надо последовательно перебрать все строки и ячейки в каждой строке, создавая длинное строковое значение. Как только это будет сделано, функция инициирует скачивание с помощью атрибута `download` и большого двоичного объекта `href`, созданного с использованием `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;
},
```

4 JSX

До сих пор в книге рассматривались способы определения пользовательских интерфейсов в функциях `render()` с использованием вызовов методов `React.createElement()` и семейства методов `React.DOM.*` (например, метода `React.DOM.span()`). Среди неудобств, создаваемых множеством вызовов функций, можно отметить трудность отслеживания расстановки всех необходимых закрывающих круглых и фигурных скобок. Но есть более простой способ определения пользовательских интерфейсов — использование JSX.

JSX представляет собой отдельную от React технологию, и ее применение не носит обязательного характера. Как вы видели, в первых трех главах она так ни разу и не использовалась. Вы можете вообще отказаться от JSX. Но все же высока вероятность того, что, один раз попробовав, вы уже не захотите возвращаться к вызовам функций.

Не совсем понятно, что именно означает акроним JSX, но наиболее вероятно это сокращение от термина JavaScriptXML или JavaScript Syntax eXtension. Официальная страница проекта находится по адресу http://facebook.github.io/jsx/.

Привет, JSX

Еще раз посмотрим на окончательный вариант примера Hello World из главы 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>
```

В функции `render()` имеется довольно много вызовов других функций. Использование JSX существенно упрощает код:

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

Этот синтаксис очень похож на уже известный вам HTML. Есть только одна загвоздка: поскольку этот код не является допустимым синтаксисом JavaScript, его нельзя запустить в браузере в неизменном виде. Это код следует преобразовать *(транспилировать)* в чистый JavaScript, который может быть запущен браузером.

Транспиляция JSX

Процесс транспиляции заключается в том, что берется исходный код и переписывается с целью получения таких же результатов, но уже с использованием синтаксиса, понимаемого устаревшими браузерами. Этот процесс отличается от использования *полифиллов*.

Примером полифилла может служить добавление к `Array.prototype` метода, аналогичного методу `map()`, который был введен стандартом ECMAScript5, чтобы заставить его работать в браузерах, поддерживающих стандарт ECMAScript3:

```
if (!Array.prototype.map) {
  Array.prototype.map = function() {
    // реализация метода
  };
}

// использование
typeof [].map === 'function';
// true, теперь 'map()' можно использовать
```

Полифилл представляет собой решение из области чистого JavaScript. Он хорошо справляется с поставленной задачей при добавлении новых методов к существующим объектам или при реализации новых объектов (например, JSON). Но когда в язык вводится новый синтаксис, этого недостаточно. Такой синтаксис, например вводящий в работу ключевое слово `class`, недопустим и вызывает в браузерах без поддержки `class` ошибку анализа кода, а создать для него полифилл не представляется возможным. Поэтому для нового синтаксиса требуется проведение компиляции (транспиляции), чтобы преобразовать его *до того*, как он будет представлен для обработки браузером.

Транспиляция JavaScript проводится довольно часто, поскольку программистам хочется использовать функциональные возможности, объявленные в стандарте ECMAScript6 (известном также

как ECMAScript2015) и в последующих стандартах, не дожидаясь выхода поддерживающих эти возможности браузеров. Если у вас уже есть отработанный процесс сборки (который выполняет, к примеру, минификацию или ECMAScript6-to-5-транспиляцию), можно просто добавить к нему этап JSX-преобразования. Но предположим, что у вас еще нет процесса сборки, и пройдем все этапы настройки упрощенного процесса на стороне клиента.

Babel

Babel (ранее известный как 6to5) является транспилятором с открытым кодом, поддерживающим самые последние функциональные возможности JavaScript, а также включающим поддержку JSX. Он необходим как предварительное условие для использования JSX. В следующей главе будет показано, как настроить процесс сборки, позволяющий вам поставлять React-приложения реальным пользователям. Но в целях изучения JSX не будем ничего усложнять и проведем транспиляцию на стороне клиента.

Необходимое предупреждение: преобразование на стороне клиента предназначено исключительно для создания прототипов, обучения и проведения исследований. Из соображений производительности такое преобразование в приложениях, предназначенных для обычного применения, использоваться не должно.

Для внутрибраузерных преобразований (на стороне клиента) нужен файл browser.js. В Babel, начиная с версии 6, он больше не предоставляется, но вы всегда можете взять последнюю работоспособную копию этого файла:

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

До выхода версии v0.14 React включал выполняемый на стороне клиента сценарий JSXTransformer. Кроме этого, в предыдущих версиях NPM-пакет react-tools устанавливал утилиту командной строки jsx. Теперь они считаются устаревшими и приоритет отдается транспилятору Babel.

Клиентская сторона

Чтобы JSX заработал, на вашей странице нужно сделать две вещи:

- включить browser.js, сценарий, позволяющий транспилировать JSX;
- пометить теги `script`, использующие JSX, чтобы Babel знал, что для него в этих тегах есть работа.

До сих пор все примеры в этой книге включали библиотеку React, используя для этого следующий код:

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

Теперь нужно включить еще и преобразователь:

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

Второй шаг — добавление в теги `<script>` атрибута `type` со значением `text/babel` (которое не поддерживается браузерами) — там, где их содержимое требует преобразования.

До:

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

После:

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

При загрузке страницы запускается код browser.js, который находит все сценарии `text/babel` и преобразует их содержимое в код, который может использоваться браузером. На рис. 4.1 показано, что произойдет в браузере Chrome при попытке запуска сценария с синтаксисом JSX в его исходном виде. Как и ожидалось, будет получена ошибка синтаксиса.

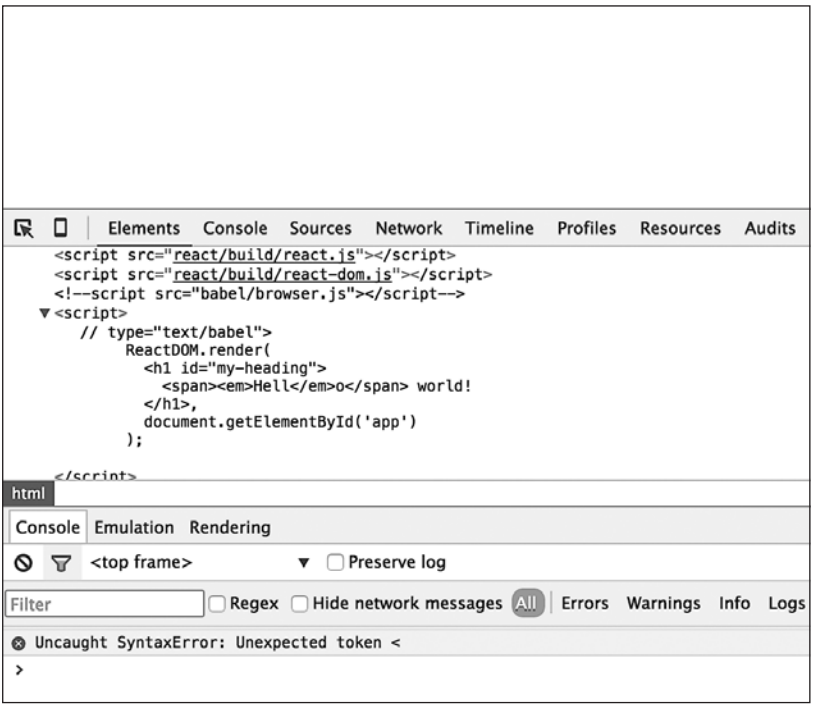


Рис. 4.1. Браузеры не понимают JSX-синтаксис

На рис. 4.2 можно увидеть, что после того, как сценарий browser.js провел транспиляцию блоков сценариев с атрибутом `type="text/babel"`, страница заработала успешно.

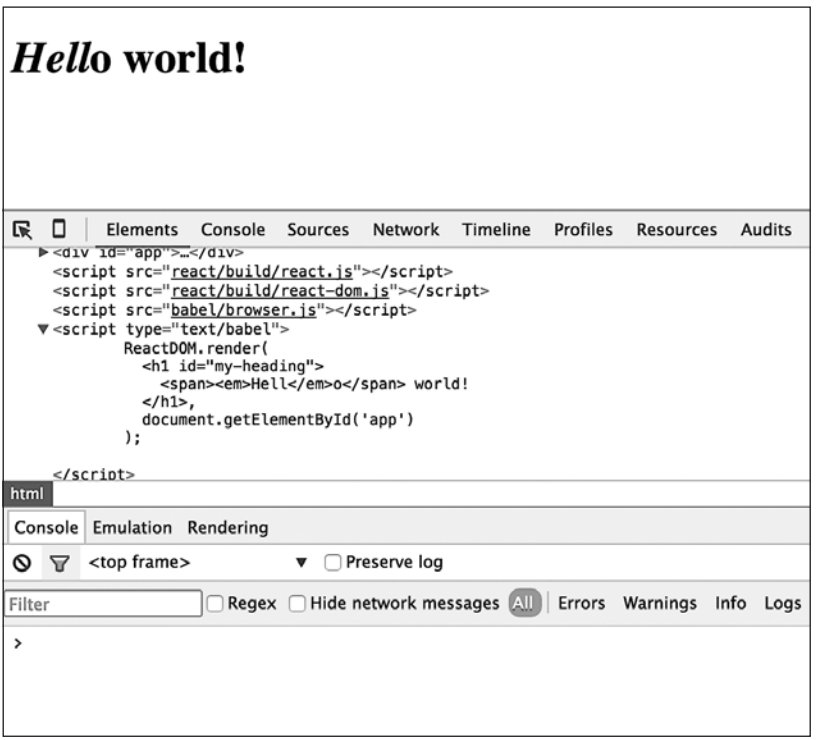


Рис. 4.2. Браузерный сценарий Babel и тип содержимого text/babel

О преобразовании JSX

Для проведения экспериментов и ознакомления с преобразованиями JSX вы можете воспользоваться интерактивным редактором, находящимся по адресу https://babeljs.io/repl/ (рис. 4.3).

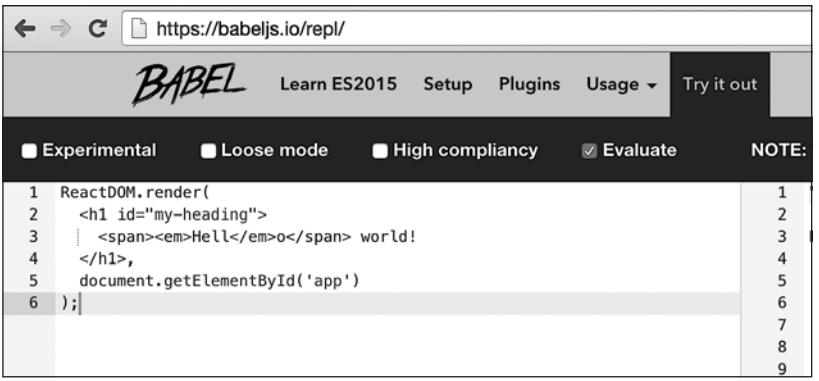


Рис. 4.3. Интерактивное средство преобразования JSX

На рис. 4.4 можно увидеть, что преобразование JSX осуществляется легко и просто: исходный JSX-код Hello World становится серией вызовов `React.createElement()` с использованием функционального синтаксиса, с которым вы уже знакомы. Это обычный код JavaScript, поэтому он легко читается и понимается.

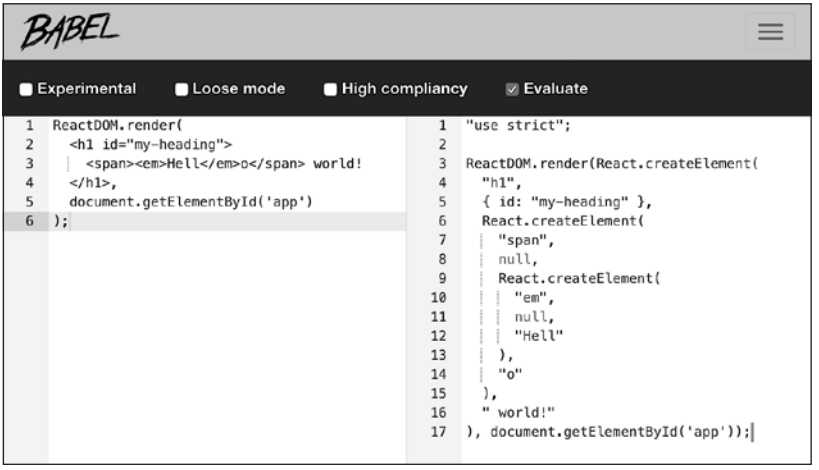


Рис. 4.4. Преобразованный пример Hello World

Есть еще одно интерактивное инструментальное средство, которое может пригодиться при изучении JSX или при переводе существующей разметки приложения из HTML, — преобразователь HTML-to-JSX (рис. 4.5).

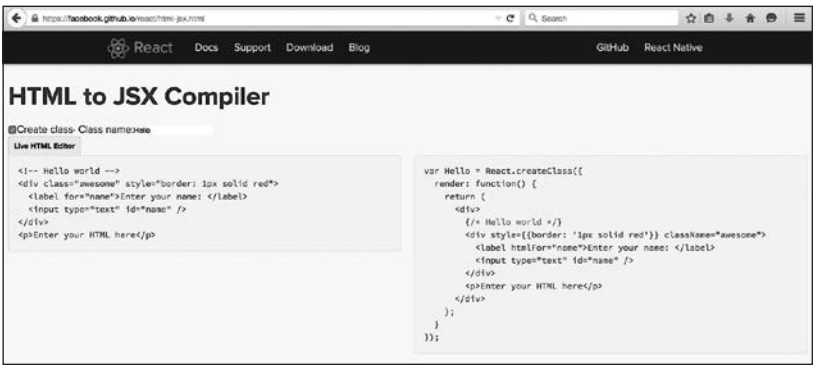


Рис. 4.5. Инструментальное средство HTML-to-JSX

JavaScript в JSX

Зачастую при создании пользовательского интерфейса нужно применять переменные, условные выражения и циклы. Вместо того чтобы выдумывать еще один шаблонный синтаксис, можно воспользоваться тем, что JSX позволяет работать с кодом JavaScript внутри разметки. Вам нужно только заключить свой код JavaScript в фигурные скобки.

Рассмотрим пример компонента `Excel` из предыдущей главы. При замене функционального синтаксиса кодом JSX в конечном итоге будет получено примерно следующее:

```
var Excel = React.createClass({

  /* фрагмент... */

  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>
    );
 }
});
```

Как видите, чтобы воспользоваться переменными, их заключают в фигурные скобки:

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

Для циклов можно заключить в фигурные скобки в том числе `map()`:

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

Код JavaScript в JSX может быть вложен на любую необходимую глубину. Код JSX можно воспринимать как JavaScript (после небольшого преобразования), но со знакомым синтаксисом HTML. С написанием кода JSX могут справиться даже те члены вашей команды, которые разбираются в JavaScript хуже вас, но при этом неплохо знают HTML. А для построения пользовательского интерфейса с живыми данными они могут изучить JavaScript в минимально необходимом объеме, всего лишь на уровне использования переменных и циклов.

В только что приведенном примере с компонентом `Excel` в функции обратного вызова `map()` есть условное выражение `if`. Хотя это условие и является вложенным, после небольшого дополнительного форматирования его можно превратить в легкочитаемый единый оператор, использующий три операнда:

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

Заметили повторяющуюся переменную title в последнем примере? От этого повторения можно избавиться:

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

```
        : null
    }</th>
);
```

Однако в таком случае придется модифицировать имеющуюся в примере функцию сортировки. В этой функции предусматривается, что пользователь будет щелкать на элементе <th>, а для определения, на каком именно элементе <th> был произведен щелчок, будет использоваться метод cellIndex. Но когда в коде JSX имеются смежные блоки {}, для их обособления применяются теги . Иными словами, фрагмент кода <th>{1}{2}</th> превращается в DOM-модели в код <th>12</th>.

Пробельные символы в JSX

Пробельные символы воспринимаются в коде JSX почти так же, как и в коде HTML, но с небольшими отличиями:

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

превращается в:

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

что отображается как `1 plus 2 is 3` — в точном соответствии с вашими ожиданиями по части HTML: несколько пробелов становятся одним пробелом.

Но в этом примере:

```
<h1>
  {1}
```

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

вы в конечном итоге получаете следующий код:

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

Как видите, все пробельные символы вырезаны, поэтому конечный результат будет `1plus2is3`. Там, где нужно, пробел можно всегда добавить с помощью конструкции `{' '}` (которая затем плодит еще больше тегов `<span>`) или же превратить строки символов в выражения и добавить пробел уже там. Иными словами, будут работать оба следующих варианта:

```
<h1>
  {/* пробельные выражения */}
  {1}
  {' '}plus{' '}
  {2}
  {' '}is{' '}
  {3}
</h1>

<h1>
  {/* пробел, прикрепленный к строковым выражениям */}
  {1}
  {' plus '}
  {2}
  {' is '}
  {3}
</h1>
```

Комментарии в JSX

В предыдущих примерах за счет добавления комментариев к разметке JSX можно увидеть, как в код внедряется новая концепция.

Поскольку выражение, заключенное в фигурные скобки `{}`, является всего лишь кодом JavaScript, можно легко добавить многострочный комментарий, воспользовавшись следующей конструкцией: `/* комментарий */`.

Можно также добавить комментарий в строку, воспользовавшись синтаксисом `// комментарий`, но при этом не стоит забывать про закрывающую фигурную скобку (`}`) выражения, которая должна быть на другой строке, чтобы не выглядеть как часть комментария:

```
<h1>
  {/* многострочный комментарий */}
  {/*
    multi
    line
    comment
  */}
  {
    // однострочный комментарий
  }
  Hello
</h1>
```

Поскольку код `{// комментарий}` не работает (`}` теперь закомментирована), от использования однострочных комментариев мало проку, и все комментарии можно привести к единому виду, выбрав во всех случаях формат многострочного комментария.

Элементы HTML

В JSX допускается использование элементов HTML:

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

Этот пример, как показано на рис. 4.6, выдает правую угловую кавычку.

Рис. 4.6. HTML-элемент в JSX

Но, если воспользоваться элементом в качестве части выражения, вы окажетесь в ситуации, когда элемент будет закодирован дважды. В этом примере:

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

HTML-элемент будет закодирован — и вы увидите результат, показанный на рис. 4.7.

Рис. 4.7. Дважды закодированный элемент HTML

Чтобы избавиться от двойного кодирования, можно воспользоваться Юникод-версией элемента HTML, в данном случае `\u00bb` (см. http://dev.w3.org/html5/html-author/charref):

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

Для удобства можно где-нибудь в верхней части модуля определить константу вместе с любым общим пробелом. Например:

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

Затем этой константой можно воспользоваться в любом нужном месте:

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

Обратили внимание на использование const вместо var? Добро пожаловать в прекрасный новый мир Babel, где можно побаловать себя всеми новшествами, предлагаемыми современным JavaScript. Подробности рассматриваются в главе 5.

Анти-XSS

Можно задаться вопросом: а зачем идти по такому сложному пути, как использование элементов HTML? Потому что здравый смысл подсказывает: они помогают бороться с XSS-атаками.

React нейтрализует все строки, противодействуя атакам класса XSS. Поэтому, когда у пользователей запрашивается какой-нибудь ввод, а они предоставляют вредоносную строку, React защищает вас от атаки. Возьмем, к примеру, следующий пользовательский ввод:

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

При определенных обстоятельствах вы можете вписать его в DOM-модель. Например:

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

Это будет иметь катастрофические последствия, поскольку на странице появится John, а тег `<script>` приведет к загрузке вредоносного кода JavaScript и откроет несанкционированный доступ к вашему приложению и конфиденциальной информации доверившихся вам пользователей.

React защищает от подобных случаев, и от вас не требуется никаких усилий. Если создать следующий код:

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

то React нейтрализует содержимое переменной `firstname` (рис. 4.8).

Hello John<script src="http://evil/co.js"></script>!

Рис. 4.8. Нейтрализация строк

Распространяемые атрибуты

В JSX у ECMAScript6 было позаимствовано весьма полезное свойство под названием *оператор распространения*, которое было внедрено в качестве удобного усовершенствования при работе с определением свойств.

Представьте, что у вас есть коллекция атрибутов, которую нужно передать компоненту `<a>`:

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

Вы можете сделать это в любой момент с помощью следующего кода:

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

Но, похоже, возникает проблема использования часто повторяющегося шаблонного кода. С помощью распространяемых атрибутов эту задачу можно выполнить всего лишь в одной строке кода:

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

В примере у вас есть объект атрибутов, который вы хотите определить (возможно, при некоторых условиях) заранее. Это полезно уже само по себе, но чаще всего применяется вариант, при котором данный объект будет получен извне, зачастую из родительского компонента. Именно такой случай и рассмотрим.

Атрибуты, распространяемые от родительского к дочернему компоненту. Представьте, что вы создаете компонент `FancyLink`, закулисно использующий обычный компонент `<a>`. Желательно, чтобы ваш компонент принял все те же атрибуты, что и `<a>` (`href`, `style`, `target` и т. д.), плюс дополнительный атрибут (скажем, `size`). Итак, кто-нибудь может воспользоваться вашим компонентом следующим образом:

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

Как ваша функция `render()` сможет воспользоваться распространяемыми атрибутами и избежать повторного определения всех свойств компонента `<a>`?

```
var FancyLink = React.createClass({
  render: function() {
    switch(this.props.size) {
      // совершение каких-либо действий на основе свойства 'size'
    }
    return <a {...this.props}>{this.props.children}</a>;
  }
});
```

Вы заметили использование метода this.props.children? Это простой и удобный метод, позволяющий переправлять вашему компоненту любое количество дочерних компонентов и обращаться к ним при составлении интерфейса.

В предыдущем фрагменте кода вы выполняете все нужные действия на основе значения свойства `size`, а затем переносите все свойства в `<a>` (включая и свойство `size`). Компонент `React.DOM.a` не имеет ни малейшего понятия о свойстве `size`, поэтому он его молча игнорирует, а все остальные свойства использует.

Можно немного улучшить код и не переправлять ненужные свойства:

```
var FancyLink = React.createClass({
  render: function() {

    switch(this.props.size) {
      // совершение каких-либо действий на основе свойства 'size'
    }

    var attribs = Object.assign({}, this.props); // поверхностное
                                                 // клонирование
```

```
    delete attribs.size;

    return <a {...attribs}>{this.props.children}</a>;
  }
});
```

Если воспользоваться синтаксисом, предлагаемым ECMAScript7 (бесплатно поставляемым вам инструментальным средством Babel!), решение задачи упрощается еще больше и обходится без всякого клонирования:

```
var FancyLink = React.createClass({
  render: function() {

    var {size, ...attribs} = this.props;

    switch (size) {
      // совершение каких-либо действий
      // на основе свойства 'size'
    }

    return <a {...attribs}>{this.props.children}</a>;
  }
});
```

Возвращение в JSX нескольких узлов

Из вашей функции `render()` всегда нужно возвращать один узел. Возвращать два узла не разрешается. Иными словами, следующий код считается ошибочным:

```
// Синтаксическая ошибка, связанная с необходимостью
// заключения смежных элементов JSX в охватывающий тег

var Example = React.createClass({
  render: function() {
```

```
    return (
      <span>
        Hello
      </span>
      <span>
        World
      </span>
    );
  }
});
```

Исправить ошибку нетрудно, нужно просто заключить все узлы в другой компонент, например в `<div>`:

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

Хотя возвратить из вашей функции `render()` массив узлов невозможно, массивы можно использовать при построении композиции — при условии, что у узлов в массиве имеются надлежащие ключевые атрибуты `key`:

```
var Example = React.createClass({
  render: function() {

    var greeting = [
      <span key="greet">Hello</span>,
      ' ',
```

```
      <span key="world">World</span>,
      '!'
    ];

    return (
      <div>
        {greeting}
      </div>
    );
  }
});
```

Обратите внимание на то, как в массив вставляются пробельные символы и другие строки, а также на то, что им не нужен ключ.

Это похоже на принятие любого количества дочерних компонентов, переданных от родительского компонента, и на их распространение в вашей функции `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')
);
```

Отличия JSX от HTML

Код JSX выглядит знакомо, поскольку похож на код HTML, но имеет преимущество — простой способ добавления динамических значений, циклов и условных выражений (нужно лишь заключить их в фигурные скобки {}). Для начала работы с JSX всегда можно воспользоваться инструментальным средством HTML-to-JSX, но чем раньше вы станете набирать свой собственный код JSX, тем лучше. Рассмотрим некоторые (возможно, несколько необычные для вас на начальном этапе обучения) различия между HTML и JSX. Кое-какие различия уже рассматривались в главе 1, но бегло повторим еще раз.

Просто class использовать нельзя, а как насчет for?

Вместо атрибутов `class` и `for` (оба они являются в ECMAScript зарезервированными словами) следует использовать названия `className` и `htmlFor`:

```
// Недопустимо!
var em = <em class="important" />;
var label = <label for="thatInput" />;

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

style является объектом

Атрибут `style` получает в качестве значения объект, а не строку с разделителями в виде точки с запятой. Имена свойств CSS набираются в «верблюжьем» регистре и без дефисов:

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

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

// Также допустима встроенная форма записи
// обратите внимание на двойные фигурные скобки { {} } –
// один комплект для динамического значения в JSX,
// а второй – для JS-объекта
var em = <em style={ {fontSize: '2em', lineHeight: '1.6'} }
/>;
```

Закрывающие теги

В HTML некоторые теги не нуждаются в закрытии, а в JSX (XML) их закрывать обязательно:

```
// Недопустимо:
// незакрытых тегов быть не должно,
// даже если в HTML это вполне допустимо
var gimmeabreak = <br>;
var list = <ul><li>item</ul>;
var meta = <meta charset="utf-8">;

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

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

Атрибуты в «верблюжьем» регистре

Вы обратили внимание, что в предыдущем фрагменте кода `charset` использована форма `charSet`? Все атрибуты в JSX должны быть указаны в «верблюжьем» регистре (camelCase). Поначалу это часто путает все карты — вы можете набрать `onclick` и заметить, что ничего не происходит, до тех пор пока не вернетесь к этому месту кода и не измените прежний код на `onClick`:

```
// Недопустимо!
var a = <a onclick="reticulateSplines()" />;

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

Исключение из этого правила составляют все атрибуты с префиксами `data-` и `aria-`, они используются в таком же виде, как и в HTML.

JSX и формы

Имеются различия между JSX и HTML и при работе с формами. Посмотрим какие.

Обработчик события onChange

При использовании элементов формы пользователи вносят изменения в их значения. В React вы можете подписаться на такие изменения с помощью атрибутов `onChange`. Это куда логичнее использования значения `checked` для переключателей и флажков, а также `selected` — в элементах выбора `<select>`. При наборе в текстовых областях и полях `<input type="text">` `onChange` активизируется по мере набора текста пользователем, что намного полезнее, чем активация при утрате элементом фокуса. Это означает, что подписываться на события мыши и клавиатуры для отслеживания набора текста при вводе изменений уже не нужно.

Сравнение value и defaultValue

Если в HTML имеется тег `<input id="i" value="hello" />`, а затем значение атрибута `value` путем набора текста изменяется на `"bye"`, то получается, что...

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

В React содержимое свойства `value` всегда имеет последнее значение вводимого текста. Если нужно указать значение по умолчанию, можно воспользоваться свойством `defaultValue`.

В следующем фрагменте кода имеются компонент `<input>` с предварительно заполненным содержимым `"hello"` и обработчик события `onChange`. Удаление последнего «о» в `"hello"` приводит к тому, что у `value` появляется значение `"hell"`, а значением свойства `defaultValue` остается `"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')
);
```

Такой же схемы нужно придерживаться и в собственных компонентах: если в них допускается применение свойства, имя которого намекает на поддержку актуальности (например, value, data), то сохраняйте актуальность его значения. Если этого не нужно, назовите свойство initialData (как делали в главе 3) или как-нибудь вроде defaultValue, чтобы было ясно, чего от него ожидать.

Значение компонента <textarea>

Для соответствия полям текстового ввода в имеющейся в React версии `<textarea>` применяются свойства `value` и `defaultValue`. В `value` поддерживается актуальность, а в `defaultValue` сохраняется исходное значение. Если следовать HTML-стилю и воспользоваться для определения значения дочерним элементом `textarea` (что не рекомендуется делать), он будет рассматриваться как имеющий такое же значение, что и свойство `defaultValue`.

В HTML поле `<textarea>` (как определено консорциумом W3C) применяет в качестве своего значения дочерний элемент, чтобы разработчики могли использовать во вводе новые строки. Но React, построенная целиком на основе JavaScript, не страдает от этого ограничения. Когда нужна новая строка, просто используется комбинация символов `\n`.

Рассмотрим следующие примеры и результаты их выполнения, показанные на рис. 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')
);
```

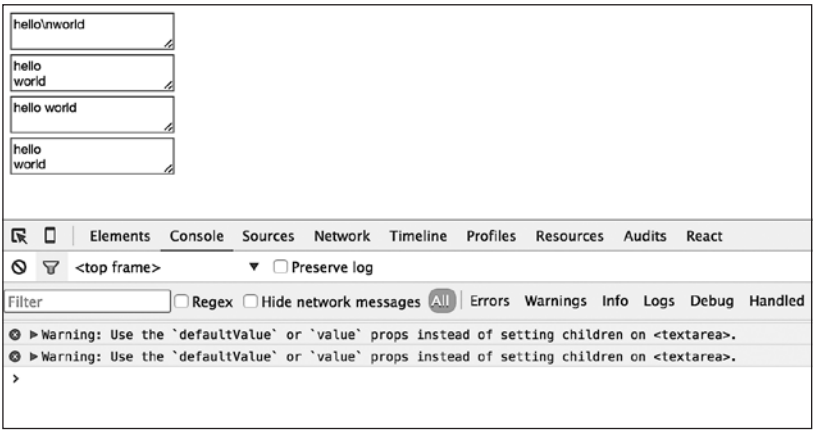


Рис. 4.9. Новые строки в текстовых областях

Обратите внимание на разницу между использованием в качестве значения свойства строки символов `"hello\nworld"` и использованием строки JavaScript `{"hello\nworld"}`.

Также возьмите на заметку, что многострочное строковое значение в JavaScript нуждается в нейтрализации с помощью обратного слеша, `\` (четвертый пример).

И наконец, посмотрите, как React выдает предупреждение об использовании для установки значения традиционного дочернего элемента, принадлежащего `<textarea>`.

Значение компонента <select>

Когда в HTML используется поле ввода `<select>`, предварительно выбранные записи указываются с помощью `<option selected>`:

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

В React для компонента `<select>` указывается `value` или же (что еще лучше) `defaultValue`:

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

То же самое действие применяется при наличии возможности выбора сразу нескольких элементов — с той лишь разницей, что предоставляется массив предварительно выбранных значений:

```
<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>
```

Если вы перепутаете код с HTML и установите для <option> атрибут selected, то React выдаст предупреждение.

Разрешается вместо записи `<select defaultValue>` использовать `<select value>`, хотя делать это не рекомендуется, поскольку вам придется позаботиться об обновлении значения, которое видит пользователь. В противном случае, когда пользователь выбирает

другой вариант, компонент `<select>` сохраняет прежний вид. Иными словами, вам нужен код, похожий на следующий:

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

Компонент Excel в JSX

В завершение воспользуемся JSX и перепишем все методы `render*()` в финальной версии компонента `Excel` из предыдущей главы. Я предлагаю вам выполнить это упражнение самостоятельно (вы всегда сможете сравнить свое решение с примером из хранилища кода, сопровождающего эту книгу).

5 Настройки для разработки приложения

Любая серьезная разработка и развертывание, кроме создания прототипа и тестирования JSX, требуют настройки процесса сборки. Если такой процесс уже отработан, нужно добавить к нему Babel-преобразование. Но предположим, что никакого процесса сборки еще нет, и начнем с нуля.

Наша цель заключается в использовании JSX и любых других современных возможностей JavaScript, не дожидаясь от браузеров их реализации. Нужно настроить преобразование, запускаемое в фоновом режиме в процессе разработки. Процесс преобразования должен выдавать код, максимально приближенный к тому коду, который ваши пользователи будут запускать на живом сайте (то есть без преобразований на стороне клиента). Процесс также должен быть как можно более незаметным, чтобы не приходилось переключаться между контекстом разработки и контекстом сборки.

Относительно процессов разработки и сборки сообществом и экосистемой JavaScript предлагается множество вариантов. Будем придерживаться простого низкоуровневого варианта сборки и не

пользоваться инструментальными средствами, выработав вместо этого собственный подход, исходя из предоставляемых возможностей, позволяющих:

- разобраться в происходящем;
- сделать после этого осознанный выбор средств для создания сборки;
- сконцентрироваться на всем, что касается React, не особо отвлекаясь на все остальное.

Типовое приложение

Начнем с определения типового образца нового приложения. Это приложение, выполняемое на стороне клиента, созданное в стиле одностраничного приложения — single-page application (SPA). В приложении используются JSX и множество нововведений, предлагаемых языком JavaScript, — предложения от стандартов ES5, ES6 (или ES2015) и будущего стандарта ES7.

Файлы и папки

В соответствии со сложившейся практикой вам понадобятся папки /css, /js и /images, а также файл index.html, позволяющий связать все эти папки вместе. Разобьем папку /js на /js/source (где будут сценарии с синтаксисом JSX) и /js/build (где будут сценарии, понятные браузерам). Кроме этого, существует еще такая категория, как /scripts, к которой относятся сценарии командной строки для выполнения сборки.

Структура каталогов для вашего типового (стандартного) приложения должна приобрести вид, показанный на рис. 5.1.

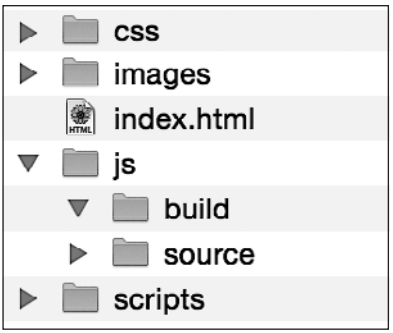


Рис. 5.1. Типовое приложение

Разобьем на части еще и каталоги /css и /js (рис. 5.2), чтобы в них были включены:

- файлы, общие для всего приложения;
- файлы, связанные с конкретными компонентами.

Это поможет сконцентрироваться на поддержке независимости компонентов, их узкой специализации и максимальной возможности многократного использования. В конечном счете хочется собрать свое собственное большое приложение, используя множество мелких компонентов, имеющих конкретное предназначение. В общем, в соответствии с принципом «разделяй и властвуй».

В довершение создадим для примера простой компонент под названием `<Logo>` (у приложений обычно бывают логотипы). По общепринятому соглашению имена компонентов начинаются с заглавной буквы, поэтому назовем его Logo, а не logo. Сохраняя согласованность всех относящихся к компонентам файлов, установим соглашение об использовании для реализации компонента путевого имени /js/source/components/*Component*.js, а для связанного с ним стилевого оформления — путевого имени

/css/components/*Component*.css. Полная структура каталогов с простым компонентом <Logo> показана на рис. 5.2.

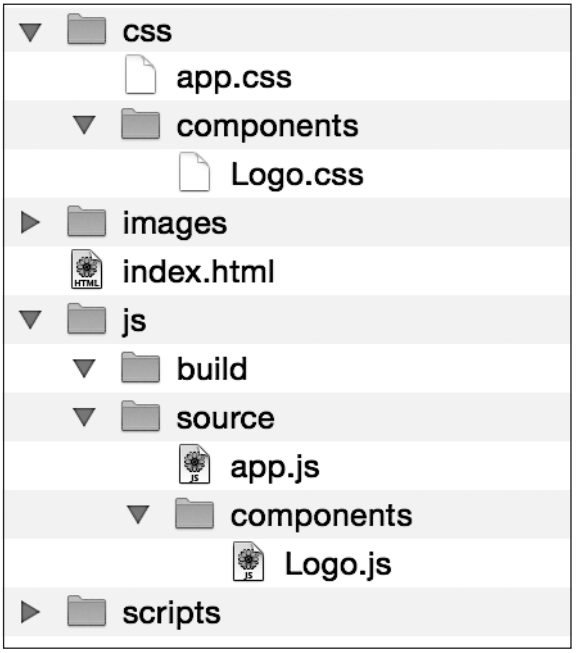


Рис. 5.2. Отдельные компоненты

index.html

Урегулировав вопрос со структурой каталогов, посмотрим, как заставить все, что в ней находится, работать в стиле Hello World. Файл index.html должен включать:

- все таблицы CSS в одном файле bundle.css;
- весь код JavaScript в одном файле bundle.js (включающем ваше приложение, а также его компоненты и их библиотечные зависимости, в том числе React);

- и, как всегда, `<div id="app">`, место, куда будет выложено ваше приложение:

```
<!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>
```

Как ни странно, но наличие единственного .css-файла и единственного .js-файла доказывает свою эффективность для широкого круга приложений. Когда ваши приложения разрастутся до размеров таких приложений, как Facebook или Twitter, эти сценарии могут стать слишком большими для первоначальной загрузки, да и пользователю в любом случае на первых порах сразу все функциональные возможности не понадобятся. Вы установите загрузчик сценариев и стилей, чтобы иметь возможность загружать дополнительный код по мере возникновения в нем потребности (решение этой задачи возлагается на вас, и в очередной раз следует заметить, что в вашем распоряжении имеется масса готовых вариантов с открытым кодом). В данном сценарии единственные файлы .css и .js становятся в некотором роде файлами начальной загрузки, тем самым минимумом, благодаря которому появляется возможность как можно скорее вывести перед глазами пользователя какое-нибудь изображение. Следовательно, схема использования единственного файла по-прежнему имеет право на существование даже при условии увеличения приложения в объеме.

Вскоре вы увидите, как файлы bundle.js и bundle.css создаются из отдельных файлов. Но сначала рассмотрим, какой CSS-код и JS-код куда попадает.

CSS

В глобальном файле /css/app.css должны содержаться общие стили, предназначенные для всего приложения, и он должен иметь примерно следующий вид:

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

Помимо стилей, предназначенных для всего приложения, вам понадобятся конкретные стили для каждого компонента. В соответствии с соглашением об использовании единственного файла CSS (и единственного файла JS) в расчете на каждый React-компонент и об их размещении в каталоге /css/components (и /js/source/components) создадим файл /css/components/Logo.css следующего содержания:

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

Еще одно простое соглашение, которое может оказаться полезным, заключается в написании имен классов CSS с заглавной буквы и в наличии у корневого элемента компонента имени

класса, совпадающего с именем компонента, следовательно, для него будет использоваться определение `className="Logo"`.

JavaScript

Точка входа в приложение, где все начинается, находится в сценарии /js/source/app.js, поэтому им и займемся:

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

И наконец, реализуем используемый в качестве примера React-компонент `<Logo>` в файле /js/source/components/Logo.js:

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

JavaScript: модернизированный код

До сих пор в примерах, приведенных в этой книге, работа велась только с простыми компонентами, также обеспечивалась доступность `React` и `ReactDOM` в качестве глобальных переменных. По мере перехода к более сложным приложениям с несколькими компонентами потребуется более продуманная организация. Распыляться глобальными переменными довольно опасно (из-за тенденции возникновения конфликта имен), и зависимость от

постоянного присутствия глобальных переменных также опасна (подумайте, что будет, если вы перейдете к другому комплектованию пакетов JS, где не соблюдается правило нахождения всего кода в единственном файле bundle.js).

Вам нужны *модули*.

Модули

Сообщество JavaScript выдвинуло несколько идей насчет модулей; одна из них — технология *CommonJS* — получила широкое распространение. Согласно этой технологии в файле содержится код, *экспортирующий* один или несколько идентификаторов (чаще всего объект, но это может быть и функция или даже отдельная переменная):

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

Одно из соглашений, которое может оказаться полезным, заключается в следующем: один модуль экспортирует что-либо одно (например, один React-компонент).

Теперь этому модулю требуется `React`, чтобы выполнить метод `React.createClass()`. Глобальных переменных больше нет, следовательно, `React` как глобальный идентификатор недоступен. Его нужно включить (или затребовать с помощью `require`) следующим образом:

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

Пусть это станет шаблоном для каждого компонента: объявление требований — в самом верху, экспорт — внизу, а между ними — сама реализация.

Модули ECMAScript

Спецификации ECMAScript предлагают развить эту идею и вводят новый синтаксис (в противоположность получаемому с помощью `require()` и `module.exports`). Он выгоден тем, что, когда дело касается транспиляции нового синтаксиса до «переваривaемого» браузером кода, Babel услужливо подставляет вам свое плечо.

При объявлении зависимостей от других модулей вместо:

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

используется:

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

А также при экспортировании из вашего модуля вместо:

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

используется:

```
export default Logo
```

Отсутствие точки с запятой в конце инструкции export — особенность, предусмотренная в ECMAScript, и в данной книге ошибкой не является.

Классы

Теперь в ECMAScript имеются классы, поэтому воспользуемся новым синтаксисом.

До:

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

После:

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

Если раньше «классы» React объявлялись с помощью объекта, то теперь с появлением реальных классов кое-что изменилось:

- в объекте больше нет произвольных свойств, только функции (методы). При надобности свойство присваивается объекту `this` внутри конструктора (будут даны дополнительные примеры и способы, которым нужно следовать);
- синтаксис метода теперь выглядит как `render(){}`, ключевые слова `function` больше не нужны;
- методы больше не разделены запятыми (`,`), как было в `var obj = {a: 1, b: 2};`.

```
class Logo extends React.Component {
  someMethod() {
  } // нет запятых

  another() { // нет ключевого слова 'function'

  }

  render() {
    return <div className="Logo" />;
  }
}
```

А теперь все вместе

По мере чтения книги будут встречаться и реализации других возможностей, предусмотренных в ECMAScript, но для типового образца, чтобы сдвинуть процесс создания нового приложения с мертвой точки как можно быстрее и с минимальными затратами, вполне достаточно упомянутых возможностей.

Теперь у вас есть index.html, а также предназначенные для приложения в целом CSS-таблицы (app.css), по одному файлу с таблицами CSS на каждый компонент (/css/components/Logo.css), точка входа в JavaScript-код приложения (app.js) и каждый

React-компонент, реализованный в конкретном модуле (например, /js/source/components/Logo.js).

Финальная версия app.js имеет следующий вид:

```
'use strict'; // эта инструкция никогда не помешает

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')
);
```

А вот как выглядит Logo.js:

```
import React from 'react';

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

export default Logo
```

Заметили разницу между импортированием React и импортированием компонента Logo: `from 'react'` и `from './components/Logo'`? Последнее выражение выглядит как путь внутри каталога (это так и есть) — вы сообщаете модулю, что нужно извлечь зависимость из файла, расположенного относительно модуля, а первое выражение касается извлечения зависимости, установленной посредством npm из общедоступного места. Посмотрим, как вы-

полняется эта работа и как волшебным образом заставить весь новый синтаксис и модули работать в браузере (включая даже старые браузеры IE!).

Настройки типового образца можно найти в хранилище кода, сопровождающем данную книгу, и воспользоваться ими для вставки в ваше приложение.

Установка обязательных инструментальных средств

Прежде чем загрузить index.html и посмотреть его в работе, необходимо сделать следующее:

- создать файл bundle.css. Это простое объединение, не требующее использования обязательных инструментальных средств;
- сделать код понятным для браузеров. Для транспиляции вам понадобится Babel;
- создать файл bundle.js. Для этого воспользуемся таким средством, как Browserify.

Средство Browserify понадобится не только для объединения сценариев, но и для:

- разрешения и включения всех зависимостей. Вы просто даете ему путевое имя файла app.js, а оно затем вычисляет все зависимости (React, Logo.js и т. д.);
- включения реализации CommonJS, чтобы работали вызовы `require()`. Babel превращает все инструкции `import` в вызовы функций `require()`.

В общем, нужно установить Babel и Browserify. Их установка выполняется с использованием npm (Node Package Manager — диспетчер пакетов Node), инструментального средства, поставляемого вместе с программной платформой Node.js.

Node.js

Для установки Node.js перейдите по адресу http://nodejs.org и получите установщик, соответствующий вашей операционной системе. Следуйте инструкциям установщика (они помогут справиться с поставленной задачей). Теперь можно воспользоваться услугами, предоставляемыми утилитой npm.

Для проверки введите в своем терминале следующую команду:

```
$ npm --version
```

Если вы не имеете опыта работы с терминалом (командной строкой), то сейчас самое время его приобрести! Если у вас Mac OS X, щелкните на поиске Spotlight (значок которого в виде увеличительного стекла находится в верхнем правом углу) и наберите `Terminal`. Если у вас Windows, найдите меню Пуск (щелкните правой кнопкой мыши на значке окна в левом нижнем углу экрана), выберите пункт Выполнить и наберите `powershell`.

В этой книге все набираемые в терминале команды предваряются символом $, используемым в качестве подсказки, позволяющей отличить их от обычного кода. При наборе команды в терминале символ $ набирать не нужно.

Browserify

Средство Browserify устанавливается с помощью npm путем набора в вашем терминале следующей команды:

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

Для проверки работоспособности средства наберите следующую команду:

```
$ browserify --version
```

Babel

Для установки интерфейса командной строки Babel наберите следующую команду:

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

Для проверки работоспособности наберите такую команду:

```
$ babel --version
```

Уловили суть?

Вообще-то лучше установить пакет Node локально без ключа --global, показанного в примерах. (Присмотримся к другой схеме, где global === bad?) При локальной установке у вас будут разные версии одних и тех же пакетов — в соответствии с потребностями каждого приложения, над которым вы работаете или которое вам нужно использовать. Но для Browserify и Babel глобальная установка пакетов предоставляет вам глобальный доступ (из любого каталога) к интерфейсу командной строки.

React и прочие

Осталось воспользоваться еще несколькими пакетами (и все будет готово):

- react, разумеется;
- react-dom (распространяется отдельно);
- babel-preset-react (предоставляет Babel поддержку для JSX и других полезных опций, связанных с React);

- babel-preset-es2015 (предоставляет вам поддержку новейших возможностей JavaScript).

Для локальной установки этих пакетов сначала перейдите в каталог своего приложения (например, с помощью команды `cd ~/reactbook/reactbook-boiler`):

```
$ 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
```

Нужно отметить, что теперь у вашего приложения есть каталог node_modules с локальными пакетами и их зависимостями. Два глобальных модуля (Babel, Browserify) находятся в каталоге node_modules, расположенном где-то в другом месте, определяемом вашей операционной системой (например, /usr/local/lib/node_modules или C:\Users{ваше_имя}\AppData\Roaming\npm\).

Займемся сборкой

В процессе сборки выполняются три действия: объединение CSS, транспиляция JS и создание пакета JS. Это не сложнее запуска трех команд.

Транспиляция JavaScript

Сначала транспилируем код JavaScript с помощью Babel:

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

Эта команда означает, что нужно взять все файлы из каталога js/source, транспилировать их, задействовав возможности React

и ES2015, и скопировать результат в js/build. Команда выведет следующую информацию:

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

И этот список будет расти по мере добавления новых компонентов.

Создание пакета JavaScript

Теперь настала очередь создания пакета:

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

Средству Browserify предписывается начать с файла app.js, учесть все зависимости и записать результат в файл bundle.js, который завершает инструкцию включения в вашем файле index.html. Чтобы проверить, что файл действительно был записан, наберите команду `less bundle.js`.

Создание пакета CSS

Создание пакета CSS выполняется (на данном этапе) настолько просто, что вам даже не нужно применять никаких специальных средств, следует просто объединить все CSS-файлы в один (используя команду `cat`). Но поскольку файл перемещается в другое место, ссылки на изображения утратят работоспособность, поэтому перепишем их с помощью вызова команды `sed`:

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

С задачей создания пакетов гораздо лучше справляется NPM, но пока нас вполне устраивает применяемый вариант.

Результаты!

На данный момент все необходимые действия завершены и можно уже посмотреть на результаты вашего упорного труда. Загрузите файл index.html в свой браузер — и увидите экран приветствия (рис. 5.3).

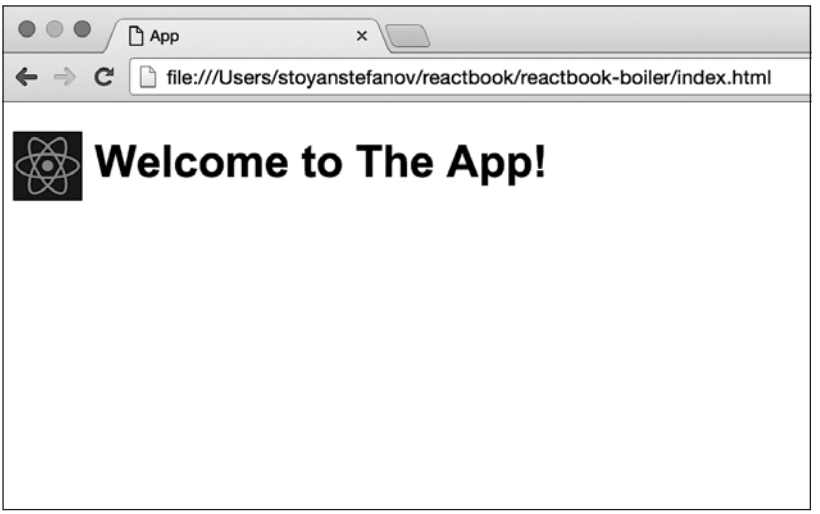


Рис. 5.3. Приглашение на вход в приложение

Версия для Windows

Предыдущие команды предназначены для использования в среде Linux или Mac OS X. Но аналогичные команды в Windows отличаются от них весьма незначительно. Первые две, за исключением разделителя каталогов, имеют идентичный вид.

То есть:

```
$ babel --presets react,es2015 js\source -d js\build
$ browserify js\build\app.js -o bundle.js
```

В Windows нет команды cat, но объединение можно выполнить следующим образом:

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

А для замены строк в файле (чтобы заставить CSS искать изображения в images, а не в ../../images) нужно применить слегка усовершенствованные возможности оболочки powershell:

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

Сборка в процессе разработки

Необходимость запуска процесса сборки при каждом внесении изменения в файл сильно докучает. К счастью, отслеживать изменения в каталоге можно из сценария, а также запускать сценарий сборки в автоматическом режиме.

Сначала поместим все три команды, составляющие процесс сборки, в файл scripts/build.sh:

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

Затем установим NPM-пакет watch:

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

Запуск команды watch похож на приказ отслеживать любые изменения в каталогах js/source/ и /css и в случае какого-либо

изменения — запускать сценарий оболочки, находящийся в файле 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
```

Разумеется, можно также поместить эту команду в файл scripts/watch.sh, чтобы всякий раз, приступая к работе над приложением, вы просто запускали команду:

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

...и пользовались готовым результатом. Вы можете вносить изменения в исходные файлы, затем обновлять страницу в браузере и просматривать новую сборку.

Развертывание

Теперь в развертывании вашего приложения нет ничего особенного, поскольку сборки создаются в ходе его разработки, поэтому особых сюрпризов ожидать не приходится. Прежде чем ваше приложение попадет к реальным пользователям, может все-таки понадобиться провести дополнительную обработку, например уменьшить размер кода и выполнить оптимизацию изображений.

В качестве примеров воспользуемся популярным средством уменьшения размера кода (минификатором) JS под названием uglify и минификатором CSS под названием cssshrink. Можно продолжить обработку, минифицировав HTML, проведя опти-

мизацию изображений, копируя файлы в сеть доставки контента content delivery network (CDN) и проведя другие нужные вам операции.

Файл scripts/deploy.sh должен приобрести следующий вид:

```
# удаление последней версии
rm -rf __deployme
mkdir __deployme

# сборка
sh scripts/build.sh

# минификация JS
uglify -s bundle.js -o __deployme/bundle.js
# минификация CSS
cssshrink bundle.css > __deployme/bundle.css
# копирование HTML и изображений
cp index.html __deployme/index.html
cp -r images/ __deployme/images/

# готово
date; echo;
```

После запуска сценария у вас должен появиться новый каталог __deployme, содержащий:

- файл index.html;
- bundle.css (минифицированный);
- bundle.js (минифицированный);
- папку images/.

Затем останется лишь скопировать этот каталог на ближайший сервер, чтобы приступить к обслуживанию ваших пользователей с применением вашего обновленного приложения.

Идем дальше

Теперь у вас есть пример конвейерной сборки и развертывания в среде оболочки. Можете его расширить в соответствии с вашими текущими потребностями или же воспользоваться более приспособленными для сборки средствами (например, Grunt или Gulp), которые способны более полно удовлетворить ваши нужды.

Имея в арсенале все приемы сборки и транспиляции, следует продолжить путь и перейти к более увлекательным темам: созданию и тестированию реального приложения с использованием самых последних многочисленных возможностей, предлагаемых современным языком JavaScript.

6 Создание приложения

Теперь, когда вы уже постигли все основы создания пользовательских React-компонентов (и использования встроенных компонентов), применения технологии JSX (или работы без нее) для определения пользовательских интерфейсов, а также сборки и развертывания результатов своей работы, настало время потрудиться над созданием более совершенного приложения.

Оно будет называться Whinepad (что-то вроде карты отзывов) и позволит пользователям делать заметки и давать оценку всем дегустируемым винам (на самом деле это не обязательно должны быть вина, можно оценивать что угодно, о чем захочется оставить отзыв). Это должно быть CRUD-приложение, умеющее делать все, что от него ожидается, то есть создавать, считывать, обновлять и удалять (create, read, update и delete — CRUD). Оно также должно быть приложением, выполняемым на стороне клиента и сохраняющим на его же стороне свои данные. Цель его создания — изучение React, поэтому информация, не относящаяся к React (например, хранение, презентация), представлена в минимальном объеме.

В процессе работы над приложением вы узнаете:

- о сборке приложения из небольших пригодных к многократному использованию компонентов;

- об обмене данными между компонентами и организации их совместной работы.

Whinepad v.0.0.1

Основываясь на стандартах, усвоенных в предыдущей главе, запустим процесс создания Whinepad — приложения выставления оценок, где оставляются заметки относительно новых вещей. Вы ведь не против, если в качестве экрана приветствия будет список уже прошедших оценку вещей в форме привлекательной таблицы? Это означает, что нужно заново воспользоваться компонентом `<Excel>` из главы 3.

Подготовка к работе

Сначала скопируйте шаблонное приложение reactbook-boiler, на основе которого будет строиться работа (копию можете получить по адресу https://github.com/stoyan/reactbook/), и переименуйте его в whinepad v0.0.1. Затем запустите сценарий watch, чтобы сборка могла производиться при внесении любых изменений:

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

Приступим к программированию

Обновите в index.html содержимое тега `title` и установите идентификатор `id="pad"`, чтобы данные элементы соответствовали новому приложению:

```
<!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>
```

Воспользуемся JSX-версией компонента `Excel` (показанной в конце главы 4) и скопируем ее в файл js/source/components/Excel.js:

```
import React from 'react';

var Excel = React.createClass({

  // Реализация...

  render: function() {
    return (
      <div className="Excel">
        {this._renderToolbar()}
        {this._renderTable()}
      </div>
    );
  },

  // продолжение реализации ...
});

export default Excel
```

Здесь можно увидеть ряд отличий от прежнего вида `Excel`:

- инструкции `import` и `export`;
- у основной части компонента теперь имеется атрибут `className="Excel"`, чтобы соответствовать недавно принятому соглашению.

По той же причине используются префиксы и во всех элементах CSS:

```
.Excel table {
  border: 1px solid black;
  margin: 20px;
}

.Excel th {
  /* и т. д. */
}

/* и т. д. */
```

Теперь осталось только включить `<Excel>`, обновив основной файл app.js. Чтобы ничего не усложнять и не выходить за рамки клиентской стороны, воспользуемся хранилищем на стороне клиента (`localStorage`). Для начала установим ряд исходных значений:

```
var headers = localStorage.getItem('headers');
var data = localStorage.getItem('data');

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

Теперь передадим данные компоненту `<Excel>`:

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

И, немного подправив Logo.css, завершим работу над версией 0.0.1 (рис. 6.1).

Рис. 6.1. Whinepad v.0.0.1

Компоненты

Повторное использование компонента `<Excel>` позволило упростить начало работы, но этот компонент перегружен задачами. К нему лучше применить принцип «разделяй и властвуй» — разбить на небольшие пригодные для многократного использования компоненты. Например, кнопки должны стать самостоятельными компонентами, чтобы их можно было использовать вне контекста таблицы `Excel`.

Кроме того, приложение нуждается в других специализированных компонентах, например в виджете рейтинга, показывающего не число, а ряд звездочек.

Настроимся на работу с новым приложением и воспользуемся еще одним вспомогательным средством — исследователем компонентов.

Его задачи:

- предоставление возможности разработки и тестирования компонентов в изолированной среде. Зачастую использование компонента в приложении привязывает его к приложению, сокращая возможности многократного использования. Обособление компонента заставляет принимать более рациональные решения о его развязке с окружающей средой;
- предоставление возможности другим разработчикам вашей команды исследовать и повторно использовать существующие компоненты. Растет объем приложения, соответственно увеличивается количество разработчиков в команде. Чтобы свести к минимуму риск того, что два разработчика будут работать над совершенно одинаковыми компонентами, и поспособствовать повторному использованию компонента (что приведет к ускорению разработки приложений), разумнее держать все компоненты в одном месте (наряду с примерами замысла их применения).

Настройка

Воспользуйтесь комбинацией клавиш Ctrl+C, чтобы остановить работу ранее запущенного сценария отслеживателя и получить возможность запуска нового отслеживателя. Скопируйте исходный минимально жизнеспособный продукт — minimum viable product (MVP) whinepad v.0.0.1 в новую папку 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
```

Исследование

Назовем средство исследования компонентов discovery.html и поместим его в корневой каталог:

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

Целиком приложение загружать не нужно, поэтому вместо файла app.js воспользуемся файлом discover.js, который содержит все примеры компонентов. Следовательно, вместо принадлежащего приложению файла bundle.js требуется включить отдельный пакет с названием discover-bundle.js:

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

Попутное создание нового пакета особого труда не представляет, нужно лишь добавить к сценарию build.sh еще одну строку:

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

И наконец, добавим к средству исследования (js/build/discover.js) пример <Logo>:

```
'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>

    {/* сюда помещаются дополнительные компоненты... */}

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

Ваше новое средство исследования компонентов (рис. 6.2) является местом запуска новых компонентов по мере их создания. Приступим к работе и поэтапно создадим нужные нам компоненты.

Рис. 6.2. Средства исследования компонентов для приложения Whinepad

Компонент <Button>

Скажу без преувеличения: кнопки нужны всем приложениям. Зачастую это красиво стилизованные обычные элементы `<button>`, но иногда в качестве кнопки должна выступать гиперссылка `<a>` (именно этот элемент был необходим в главе 3 для создания кнопок скачивания файлов). А что, если создать новую привлекательную кнопку `Button`, принимающую необязательное свойство `href`? Если оно присутствует, то в основе отображения будет фигурировать гиперссылка `<a>`.

В духе разработки на основе тестирования — test-driven development (TDD) — можно приступить к работе, выбрав обратное направление и определив пример использования компонента в средстве discovery.js.

До:

```
import Logo from './components/Logo';
{/* ... */}
{/* сюда помещаются дополнительные компоненты... */}
```

После:

```
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>

{/* сюда помещаются дополнительные компоненты... */}
```

(А нельзя ли тогда назвать это разработкой на основе исследований — discovery-driven development, DDD?)

Обратили внимание на новый шаблон () => alert('ouch')? Это пример использования функции стрелки из спецификации ES2015.

Вот другие варианты применения этой функции:

- выражение () => {} является пустой функцией (наподобие function() {});
- выражение (what, not) => console.log(what, not) является функцией с параметрами;
- выражение (a, b) => { var c = a + b; return c;} используется, когда в теле функции больше одной строки, в этом случае нужны фигурные скобки {};
- выражение let fn = arg => {} применяется при получении только одного аргумента, круглые скобки () можно не использовать.

Button.css

Согласно требованиям принятого соглашения код, определяющий стиль компонента `<Button>`, должен размещаться в файле /css/components/Button.css. В нем нет ничего необычного, там просто свойства CSS, которые придадут кнопке дополнительную привлекательность. Рассмотрим их здесь полностью и договоримся, что не станем тратить время на код CSS для других компонентов:

```
.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

Теперь рассмотрим полную версию кода в файле /js/source/components/Button.js:

```
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
```

Код этого компонента изложен кратко, но при этом полон новых идей и передового синтаксиса. Проведем исследования, начиная с самой верхней строчки!

Пакет classnames

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

Пакет `classnames` (устанавливается с помощью команды `npm i --save-dev classnames`) предоставляет полезную функцию для работы с именами классов CSS. Задача использования вашим

компонентом своих собственных классов при сохранении достаточной гибкости, позволяющей проводить настройки посредством имен классов, передаваемых родительским компонентом, ставится довольно часто. Ранее в пакете дополнительных средств React для этого была специальная утилита, но она вышла из употребления, и ее место занял этот более удачный пакет от стороннего производителя. Из данного пакета используется только одна функция:

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

Она предназначена для объединения имени класса `Button` с любыми (если таковые будут) именами классов, переданными в виде свойств при создании компонента (рис. 6.3).

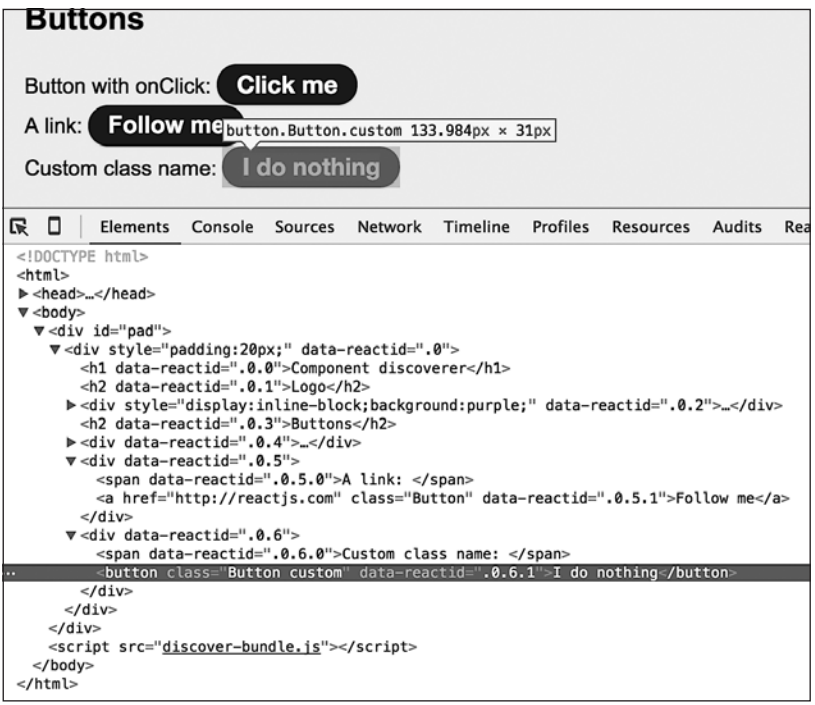


Рис. 6.3. <Button> с пользовательским именем класса

Объединить имена классов можно и самостоятельно, но classnames очень маленький пакет, который удобно применять для решения этой часто встречающейся задачи. Он также позволяет выполнять такую полезную операцию, как условное присваивание имен классам:

```
<div className={classNames({
  'mine': true, // безусловное присваивание
  'highlighted': this.state.active,
// присваивание в зависимости
// от состояния компонента…
  'hidden': this.props.hide,        // ... или его
                                    // свойств
})} />
```

Реструктуризующее присваивание

Код:

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

является просто лаконичным способом следующего объявления:

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

Функциональный компонент, не имеющий состояния

Когда компонент настолько прост (и такое бывает!) и не нуждается в поддержке состояния, для его определения можно воспользоваться функцией. Тело этой функции станет заменой вашего метода `render()`. Функция в качестве своего первого аргумента получает все свойства, поэтому в ее теле используется не `this.props.href`, как в версии класса или объекта, а `props.href`.

Используя форму определения с помощью стрелки, эту функцию можно переписать следующим образом:

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

И если есть стойкое желание выразить тело одной строкой, можно даже отказаться от использования {}, ; и return:

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

propTypes

Если используются синтаксис классов, специфицированный в ES2015, или функциональные компоненты, вы должны после определения компонента определить любые свойства (такие как propTypes) как *статичные*. Иными словами, до (ES3, ES5):

```
var PropTypes = React.PropTypes;

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

После (класс ES2015):

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

class Button extends Component {
  render() {
```

```
    /* отображение */
  }
}

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

То же самое при работе функционального компонента, не использующего состояние:

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

const Button = props => {
  /* отображение */
};

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

Формы

На данный момент компонент `<Button>` нас вполне устраивает. Перейдем к выполнению другой задачи, важной для любого приложения, в котором вводятся данные, — к работе с формами. Разработчиков приложений редко устраивает внешний вид встроенных в браузер форм ввода данных, что побуждает их создавать собственные версии. Не исключение в этом плане и приложение Whinepad.

Создадим универсальный компонент `<FormInput>` с методом `getValue()`, предоставляющим вызывающему коду доступ к записи в поле ввода. В зависимости от значения свойства `type` этот компонент должен делегировать создание поля ввода более узкоспециализированным компонентам, например компонентам ввода данных `<Suggest>`, `<Rating>` и т. д.

Начнем с компонентов более низкого уровня, которым нужны лишь методы `render()` и `getValue()`.

Компонент <Suggest>

Необычные поля ввода с автопредложением (известные как поля с упреждающим заполнением) встречаются в веб-приложениях довольно часто, но не будем ничего усложнять (рис. 6.4) и позаимствуем то, что уже предлагается браузером, а именно HTML-элемент `<datalist>`.

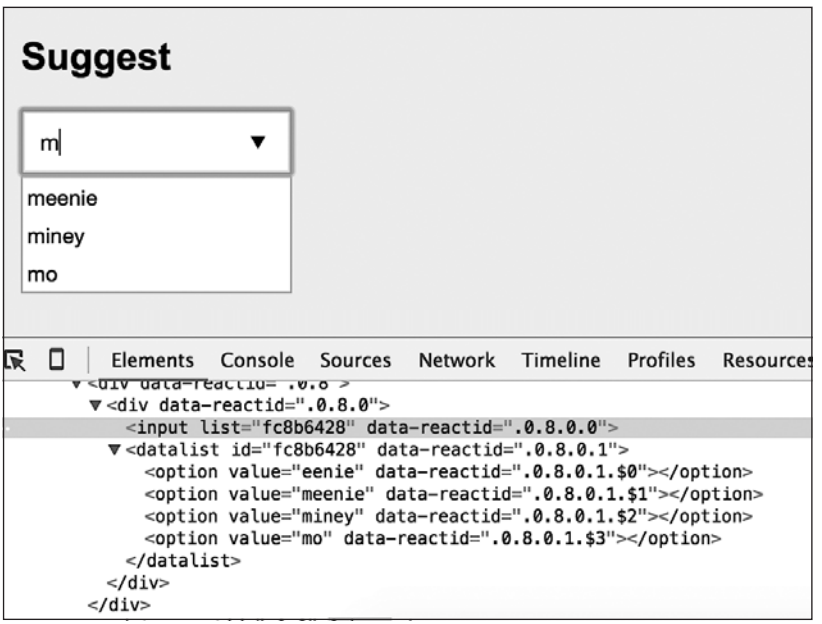


Рис. 6.4. Компонент ввода данных <Suggest> в работе

Прежде всего обновим наше исследовательское приложение:

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

Теперь приступим к реализации компонента в файле /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
```

Как видно из предыдущего кода, в этом компоненте нет абсолютно ничего особенного, он является всего лишь оболочкой вокруг прикрепленных к нему (с использованием `randomid`) элементов `<input>` и `<datalist>`.

Что касается нового ES-синтаксиса, можно увидеть появившуюся возможность использования реструктуризующего присваивания для назначения переменной более одного свойства:

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

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

А что касается нововведений, появившихся в React, можно увидеть использование атрибута `ref`.

Атрибут ref. Рассмотрим следующий код:

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

Атрибут `ref` позволяет присваивать имя конкретному экземпляру компонента React, после чего ссылаться на него по этому имени. Атрибут `ref` можно добавить к любому компоненту, но обычно он используется для ссылки на DOM-элементы, когда действительно нужно обратиться к исходной DOM-модели. Зачастую использование `ref` является обходным маневром и должны быть другие способы выполнения той же самой задачи.

В предыдущем случае нужно было получить возможность забирать при необходимости значение поля ввода `<input>`. Учитывая, что изменения в поле ввода могут считаться изменениями состояния компонента, вы можете переключиться на использование с целью отслеживания ситуации объекта `this.state`:

```
class Suggest extends Component {

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

```
    this.state = {value: props.defaultValue};
  }

  getValue() {
    return this.state.value; // 'ref' больше не используется
  }

  render() {}
}
```

Тогда полю ввода `<input>` атрибут `ref` больше не понадобится, но для обновления состояния нужен будет обработчик события `onChange`:

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

Обратите внимание на применение в методе `constructor()` выражения `this.state = {};`: это замена для метода `getInitialState()`, который использовался до появления ES6.

Компонент <Rating>

Наше приложение предназначено для оценки вещей, подвергающихся испытаниям. Проще всего давать оценку, используя показатель рейтинга в виде ряда звезд, скажем, от одной до пяти.

Этот компонент, предназначенный для многократного использования, должен быть настроен:

- на применение любого количества звезд (их исходное количество равно пяти, но почему бы не использовать, скажем, 11 звезд?);
- применение компонента только для чтения, поскольку вам не захочется, чтобы важные рейтинговые данные могли измениться из-за случайного щелчка на звездах.

Протестируйте компонент в нашем исследовательском приложении (рис. 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>
```

Рис. 6.5. Виджет для отображения рейтинга

Минимальные потребности реализации включают установки типов свойств и обслуживаемого состояния:

```
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,
    };
  }
```

```
  /* другие методы... */
}

Rating.propTypes = {
  defaultValue: PropTypes.number,
  readonly: PropTypes.bool,
  max: PropTypes.number,
};

Rating.defaultProps = {
  defaultValue: 0,
  max: 5,
};

export default Rating
```

Названия свойств говорят сами за себя: `max` представляет собой количество звезд, а `readonly` делает виджет, как нетрудно догадаться, доступным только для чтения. В состоянии фигурируют `rating`, текущее значение количества присвоенных звезд, и `tmpRating`, значение, используемое при перемещении пользователем указателя мыши над компонентом, когда еще не выражена готовность щелкнуть кнопкой мыши и отправить выбранный показатель рейтинга.

Далее следуют вспомогательные методы, занимающиеся обслуживанием актуальности состояния в ходе работы пользователя с компонентом:

```
getValue() { // предоставляется всеми нашими полями ввода
  return this.state.rating;
}

setTemp(rating) { // при проходе над элементом указателя мыши
  this.setState({tmpRating: rating});
}

setRating(rating) { // при щелчке
  this.setState({
```

```
    tmpRating: rating,
    rating: rating,
  });
}

reset() { // возвращение к реальному рейтингу
          // при уводе указателя мыши
  this.setTemp(this.state.rating);
}

componentWillReceiveProps(nextProps) {
// реагирование на внешние изменения
  this.setRating(nextProps.defaultValue);
}
```

И наконец, метод `render()`. У него имеются:

- цикл, создающий звезды от одной и до `this.props.max`. Все звезды отображаются с помощью символа с кодом `&#9734;`. Когда применяется стиль `RatingOn`, звезды становятся желтыми;
- скрытое поле ввода для работы в качестве настоящего, принадлежащего форме поля ввода (позволяет задавать значение в общем виде — подобно любому полю ввода `<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>
  );
}
```

Можно заметить еще одну особенность — использование метода привязки `bind`. В цикле отображения звезд имеет смысл запомнить текущее значение `i`, но зачем используется выражение `this.reset.bind(this)`? Его необходимо задействовать при использовании синтаксиса классов, определенного в ES. При создании привязки можно выбрать один из трех вариантов:

- как видно из предыдущего примера, выражение `this.method.bind(this)`;
- функция стрелки, выполняющей автопривязку, например `(_unused_event_) => this.method()`;
- однократная привязка в конструкторе.

Для использования третьего варианта нужно сделать следующее:

```
class Comp extents Component {
  constructor(props) {
    this.method = this.method.bind(this);
  }

  render() {
    return <button onClick={this.method}>
  }
}
```

Одно из преимуществ заключается в том, что вы, как и раньше, используете ссылку `this.method` (как с компонентами, созданными с помощью `React.createClass({})`). Еще одно преимущество заключается в том, что метод привязывается раз и навсегда, в отличие от привязки при каждом вызове `render()`. К недостаткам можно отнести наличие более объемного шаблона в контроллере.

Компонент <FormInput>

Настал черед универсального компонента `<FormInput>`, способного создавать различные компоненты ввода на основе заданных свойств. Все создаваемые компоненты ввода ведут себя согласованно (когда нужно, предоставляется метод `getValue()`).

Протестируйте компонент в нашем исследовательском приложении (рис. 6.6):

```
<h2>Form inputs</h2>
<table><tbody>
  <tr>
    <td>Vanilla input</td>  // Обычный ввод
    <td><FormInput /></td>
  </tr>
  <tr>
    <td>Prefilled</td>  // С предварительным заполнением
    <td><FormInput defaultValue="it's like a default" /></td>
  </tr>
```

```
  <tr>
    <td>Year</td>  // Год
    <td><FormInput type="year" /></td>
  </tr>
  <tr>
    <td>Rating</td> // Рейтинг
    <td><FormInput type="rating" defaultValue={4} /></td>
  </tr>
  <tr>
    <td>Suggest</td>  // Предложение
    <td><FormInput
      type="suggest"
      options={['red', 'green', 'blue']}
      defaultValue="green" />
    </td>
  </tr>
  <tr>
    <td>Vanilla textarea</td> // Обычная текстовая область
    <td><FormInput type="text" /></td>
  </tr>
</tbody></table>
```

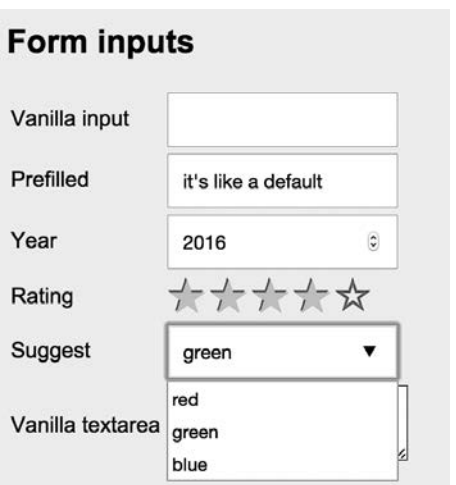


Рис. 6.6. Элементы ввода, используемые в форме

Реализация <FormInput> (js/source/components/FormInput.js) требует для корректности обычного шаблонного использования import, export и propTypes:

```
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,
  // как в вариантах автозаполнения <option>
  defaultValue: PropTypes.any,
};

export default FormInput
```

Метод render() представляет собой одну большую инструкцию switch, которая делегирует создание отдельно взятого элемента ввода узкоспециализированному компоненту или же прибегает к использованию встроенных DOM-элементов <input> и <textarea>:

```
render() {
  const common = { // свойства, применимые для всех компонентов
    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" />;
  }
}
```

Заметили использование свойства `ref`? Оно может оказаться полезным при присваивании значений элемента ввода:

```
getValue() {
  return 'value' in this.refs.input
    ? this.refs.input.value
    : this.refs.input.getValue();
}
```

Здесь `this.refs.input` является ссылкой на исходный элемент DOM-модели. Для обычных DOM-элементов вроде `<input>` и `<textarea>` вы получаете DOM-значение с помощью `this.refs.input.value`

(как при использовании традиционно применяемого в DOM выражения `document.getElementById('some-input').value`). В иных случаях для необычных пользовательских компонентов ввода (вроде `<Suggest>` и `<Rating>`) вы добираетесь до их собственных методов `getValue()`.

Компонент <Form>

Теперь у вас имеются:

- пользовательские компоненты ввода (например, `<Rating>`);
- встроенные элементы ввода (например, `<textarea>`);
- `<FormInput>` — фабрика, создающая компоненты ввода на основе значения свойства `type`.

Настало время заставить их всех работать вместе в составе компонента `<Form>` (рис. 6.7).

Компонент `<Form>` должен быть пригодным к многократному использованию, также в нем не должно быть ничего жестко заданного относительно приложения для составления рейтинга вин. (Если копнуть глубже, ничего не должно быть жестко задано относительно вина, чтобы приложение могло быть перенацелено на *оценку* любой другой категории вещей.) Компонент `<Form>` может быть настроен через массив полей, где каждое поле определяется:

- типом ввода (по умолчанию `"input"`);
- идентификатором, чтобы это поле ввода потом можно было найти;
- надписью, чтобы ее можно было поставить рядом с полем ввода;
- дополнительными вариантами для передачи их полю ввода с автопредложением.

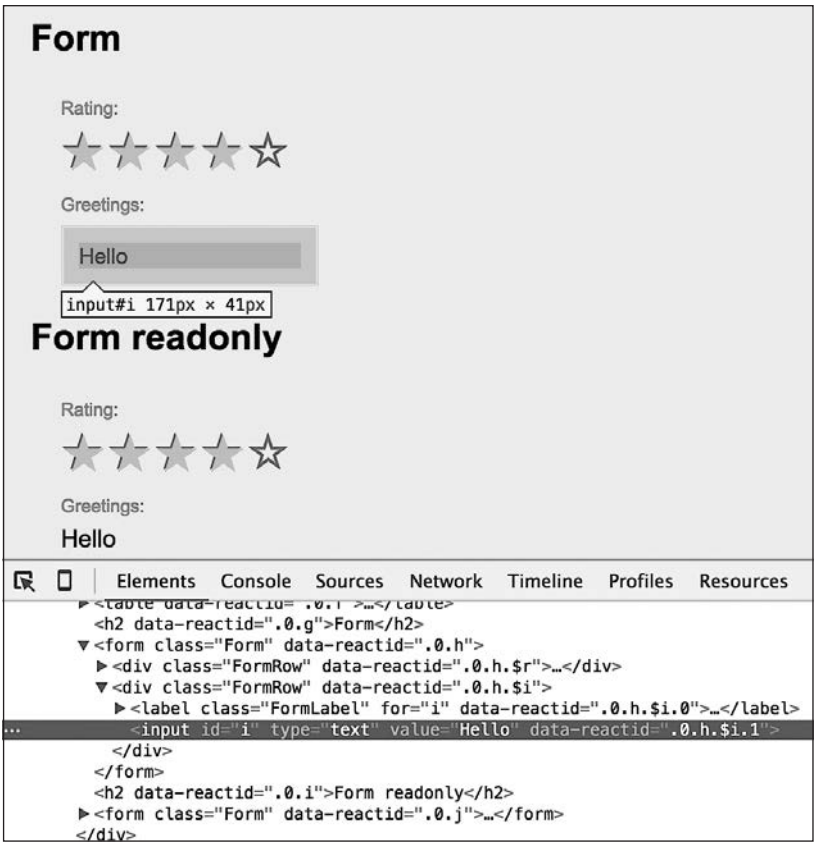


Рис. 6.7. Формы

Компонент `<Form>` также получает отображение исходных значений; он может отображаться в режиме только для чтения, чтобы пользователь не мог воспользоваться редактированием:

```
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
```

Обратите внимание на использование выражения `PropTypes.shape`. Оно позволяет вам конкретизировать свои ожидания относительно содержимого отображения. Оно более строгое, чем обобщение вроде `fields: PropTypes.arrayOf(PropTypes.object)` или `fields: PropTypes.array` и, конечно же, позволит отловить больше ошибок еще до того, как они проявятся (в самом начале использования ваших компонентов другими разработчиками).

Свойство `initialData` является отображением вида {*`имя_поля: значение`*}; это также формат данных, возвращенных принадлежащим компоненту методом `getData()`.

Пример использования компонента `<Form>` для исследовательского приложения имеет следующий вид:

```
<Form
  fields={[
    {label: 'Rating', type: 'rating', id: 'rateme'},
    {label: 'Greetings', id: 'freetext'},
  ]}
  initialData={ {rateme: 4, freetext: 'Hello'} } />
```

Теперь вернемся к реализации. Компоненту нужны методы `getData()` и `render()`:

```
getData() {
  let data = {};
  this.props.fields.forEach(field =>
    data[field.id] = this.refs[field.id].getValue()
  );
  return data;
}
```

Как видите, вам надо лишь пройтись в цикле по всем методам `getValue()`, принадлежащим компонентам ввода данных, воспользовавшись для этого свойствами `ref`, установленными в методе `render()`.

Сам метод `render()` особой сложностью не отличается, и в нем не используются какой-либо не встречавшийся вам до этого синтаксис или другие шаблоны:

```
render() {
  return (
    <form className="Form">{this.props.fields.map(field => {
      const prefilled = this.props.initialData &&
        this.props.initial
        Data[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>
  );
}
```

Компонент <Actions>

Рядом с каждой строкой в таблице данных должны быть действия (actions) (рис. 6.8), которые можно предпринять в отношении этой строки: удаление, редактирование, просмотр (когда не вся информация может поместиться в строке).

Рис. 6.8. Компонент Actions

Тестируемый в исследовательском средстве Discovery компонент `Actions` имеет следующий вид:

```
<h2>Actions</h2>
<div><Actions onAction={type => alert(type)} /></div>
```

А вот его весьма простая реализация:

```
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` — очень простой компонент, требующий лишь отображения и не поддерживающий состояния. Поэтому он может быть определен как функциональный компонент, не имеющий состояния,

с использованием функции стрелки с самым емким из всех возможных синтаксисом: без возвращаемого значения, без `{}`, без инструкции `function` (в прежние времена такую форму записи трудно было бы даже посчитать функцией!).

Код, вызывающий компонент, может подписаться на событие «действия», используя свойство `onAction`. Это простая схема для дочернего компонента, чтобы оповестить свой родительский компонент об изменении внутри компонента. Как видите, применять пользовательские события (например, на случай действия — `onAction`, на случай нападения пришельцев — `onAlienAttack` и т. д.) проще простого.

Диалоги

Следующий на очереди универсальный компонент диалога, предназначенный для сообщений или для использования диалоговых окон любого вида (вместо `alert()`) (рис. 6.9). Например, все формы добавления и редактирования данных могут быть представлены в модальном диалоговом окне в верхней части таблицы данных.

Использование:

```
<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>
```

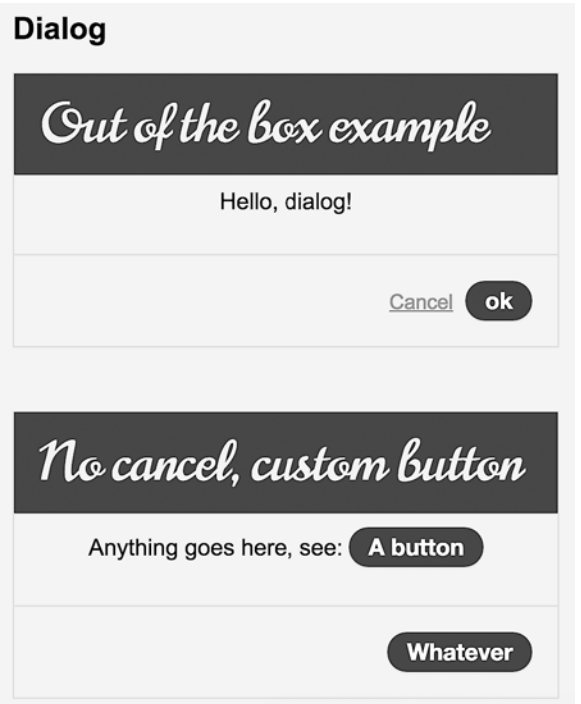


Рис. 6.9. Диалоги

Реализация очень похожа на реализацию компонента `Actions` — без состояния (нужен лишь метод `render()`) и с функцией обратного вызова `onAction`, инициируемой при нажатии пользователем кнопки в нижней части диалогового окна:

```
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
```

Но этот компонент объявляется как класс, а не как функция стрелки, поскольку ему требуется определить два дополнительных метода *управления жизненным циклом*:

```
componentWillUnmount() {
  document.body.classList.remove('DialogModalOpen');
}

componentDidMount() {
  if (this.props.modal) {
    document.body.classList.add('DialogModalOpen');
  }
}
```

Они понадобятся при создании модального диалогового окна: компонент добавляет имя класса к телу документа, следовательно, документ может получить стилевое оформление (обесцветиться).

И наконец, воспользуемся методом `render()`, чтобы составить модальную оболочку для заголовка, тела и нижней части. Тело

вмещает любой другой компонент (или простой текст); когда речь заходит о его содержимом, диалоговое окно демонстрирует свой абсолютно неприхотливый характер:

```
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>
  );
}
```

Альтернативы:

- вместо использования одного вызова `onAction` есть другой вариант: предоставление вызова `onConfirm` (по щелчку пользователя на кнопке `OK`) и вызова `onDismiss`;

- можно улучшить работу с диалоговым окном, заставив его исчезать после нажатия пользователем клавиши `Esc`. (А как бы вы смогли реализовать эту функцию?);
- у контейнера `div` имеется условное и безусловное имя класса. В компоненте можно воспользоваться модулем `classnames`, сделав это следующим образом.

До:

```
<div className={this.props.modal ? 'Dialog DialogModal' :
  'Dialog'}>
```

После:

```
<div className={classNames({
  'Dialog': true,
  'DialogModal': this.props.modal,
})}>
```

Настройка приложения

На данный момент в нашем распоряжении уже имеются все низкоуровневые компоненты, осталось получить еще два компонента: новую, усовершенствованную таблицу данных `Excel` и самый высокоуровневый родительский компонент `Whinepad`. Оба они настраиваются через объект «схемы» — описание типа данных, с которыми нужно работать в приложении. Вот как выглядит пример (js/source/schema.js), позволяющий приступить к формированию приложения, ориентированного на предоставление отзывов о дегустируемых винах:

```
import classification from './classification';

export default [
  {
    id: 'name',
    label: 'Name',
```

```
    show: true, // показать в таблице 'Excel'
    sample: '$2 chuck',
    align: 'left', // выравнивание в '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',
  },
]
```

Это пример одного из самых простых ECMAScript-модулей, которые только можно себе представить (один из тех, что экспортирует только одну переменную). Он также импортирует еще

один простой модуль, содержащий некоторые длинные варианты предварительного заполнения форм (js/source/classification.js):

```
export default {
  grapes: [
    'Baco Noir',
    'Barbera',
    'Cabernet Franc',
    'Cabernet Sauvignon',
    // ....
  ],
}
```

Теперь с помощью модуля `schema` вы можете настроить тип данных, с которыми сможете работать в приложении.

<Excel>: новый и усовершенствованный

Компонент `Excel` из главы 3 обладает избыточными для нашей новой задачи возможностями. Его усовершенствованная версия должна стать более пригодной для многократного использования. Избавимся от функции поиска (переместив ее в компонент самого верхнего уровня `Whinepad`) и от функций загрузки (эти функции вы можете добавить в `Whinepad` самостоятельно). Компонент должен заниматься только RUD-частью из набора функций, обозначаемого акронимом CRUD (то есть только чтением, обновлением и удалением) (рис. 6.10). Это редактируемая таблица, и она должна предоставить своему родительскому компоненту `Whinepad` возможность получения уведомлений при изменении данных, используя для этого метод `onDataChange`.

Компонент `Whinepad` должен позаботиться о поиске, о функции C (создании новой записи) из набора функций, обозначаемого акронимом CRUD, и о постоянном хранении данных с использованием `localStorage`. (В версии приложения, предназначенной для

реальной работы, должно быть предусмотрено хранение данных на сервере.)

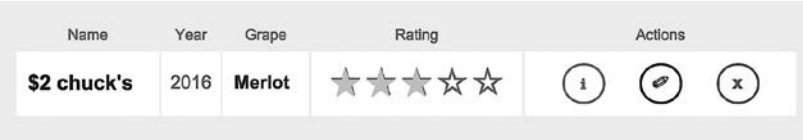


Рис. 6.10. Excel

Для настройки на типы данных оба компонента используют коллекцию из модуля `schema`.

Итак, приготовьтесь к полноценной реализации компонента `Excel` (за исключением некоторых особенностей он близок к тому компоненту, который вам знаком по главе 3):

```
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
```

Кое-что здесь требует более подробного разбора...

```
render() {
  return (
    <div className="Excel">
      {this._renderTable()}
      {this._renderDialog()}
    </div>
  );
}
```

Компонент выводит таблицу и (возможно) диалоговое окно. Диалоговое окно может быть окном подтверждения с вопросом наподобие «Вы действительно хотите выполнить удаление?», или формой для редактирования, или формой только для чтения, позволяющей лишь прочитать информацию об элементе. Или же может не быть никакого диалогового окна (исходное состояние). Когда устанавливается свойство `dialog` состояния `this.state`, данные отображаются заново, в результате чего, если это необходимо, появляется диалоговое окно.

А свойство `dialog` в состоянии устанавливается, когда пользователь выполняет щелчок на одной из кнопок компонента `<Action>`:

```
_actionClick(rowidx, action) {
  this.setState({dialog: {type: action, idx: rowidx}});
}
```

Когда данные в таблице изменяются (с использованием инструкции `this.setState({data: /**/})`), вы инициируете выдачу события изменения, уведомляющего родительский компонент, что он может обновить содержимое постоянного хранилища:

```
_fireDataChange(data) {
  this.props.onDataChange(data);
}
```

Обмен данными в обратном направлении (от родительского компонента `Whinepad` к дочернему `Excel`) осуществляется путем

изменения родительским компонентом свойства `initialData`. Компонент `Excel` готов реагировать на такие изменения, используя код:

```
componentWillReceiveProps(nextProps) {
  this.setState({data: nextProps.initialData});
}
```

А как создается форма ввода данных (рис. 6.11)? Или окно просмотра данных (рис. 6.12)? Вы открываете компонент `Dialog` с компонентом `Form` внутри него. Эти настройки для формы берутся из `schema`, а данные для полей ввода — из `this.state.data`:

```
_renderFormDialog(readonly) {
  return (
    <Dialog
```

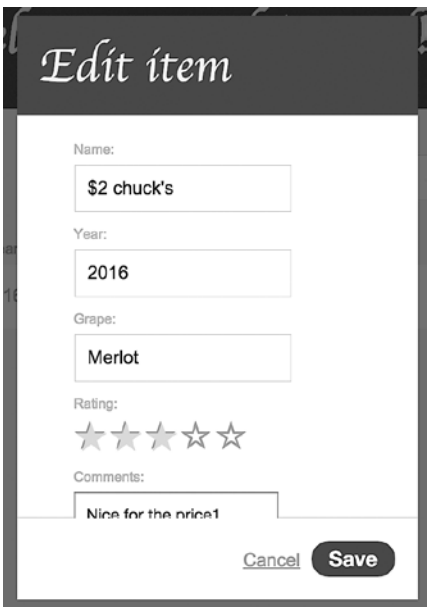


Рис. 6.11. Диалоговое окно редактирования данных (U из акронима CRUD)

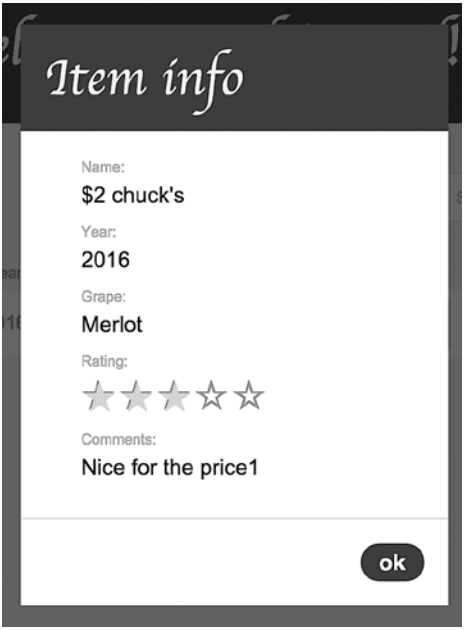


Рис. 6.12. Диалоговое окно просмотра данных (R из акронима CRUD)

```
     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>
  );
}
```

Когда пользователь выполняет редактирование, вам следует всего лишь обновить состояние и оповестить об этом подписчиков:

```
_saveDataDialog(action) {
  if (action === 'dismiss') {
    this._closeDialog();
    // всего лишь устанавливает this.state.dialog в 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);
}
```

Если применять новый ES-синтаксис, то все сводится к более широкому использованию шаблона строк:

```
// До
"Are you sure you want to delete " + nameguess + "?"

// После
{'Are you sure you want to delete "${nameguess}"?'}
```

Также обратите внимание на использование шаблонов в именах классов, поскольку приложение позволяет вам настраивать таблицу данных с помощью идентификаторов из схемы. Итак:

```
// До
<th className={"schema-" + item.id}}>

// После
<th className={'schema-${item.id}'}>
```

Возможно, самым странным синтаксисом будет тот, в котором строка шаблона заключается в квадратные скобки `[]` и используется

как имя свойства в объекте. Это не имеет отношения к React, но вы можете посчитать любопытным тот факт, что следующий код также можно использовать со строками шаблона:

```
{
  ['schema-${schema.id}']: true,
  'ExcelEditable': !isRating,
  'ExcelDataLeft': schema.align === 'left',
  'ExcelDataRight': schema.align === 'right',
  'ExcelDataCenter': schema.align !== 'left' &&
    schema.align !== 'right',
}
```

Компонент <Whinepad>

Настал черед последнего компонента, являющегося родительским для всех других (рис. 6.13). Он проще табличного компонента `Excel` и имеет меньше зависимостей:

```
import Button from './Button';
// <- для добавления новой записи: "add new item"
import Dialog from './Dialog';
// <- для вывода на экран формы добавления новой
// записи: "add new item"
import Excel from './Excel';   // <- таблица со всеми записями
import Form from './Form';     // <- форма для добавления
                               // новой записи: "add new item"
import React, {Component, PropTypes} from 'react';
```

Компонент получает только два свойства — схему данных и существующие записи:

```
Whinepad.propTypes = {
  schema: PropTypes.arrayOf(
    PropTypes.object
  ),
  initialData: PropTypes.arrayOf(
```

```
    PropTypes.object
  ),
};

export default Whinepad;
```

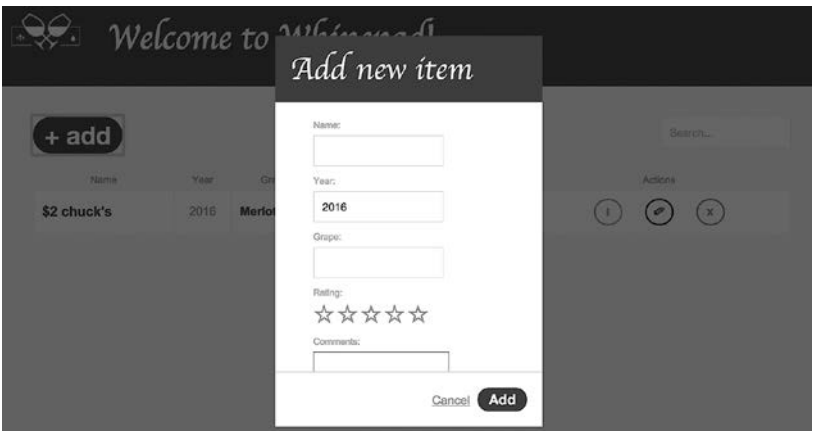


Рис. 6.13. Компонент Whinepad в работе с операцией C из акронима CRUD

После того как вы досконально изучили весь код реализации компонента `Excel`, освоение этого кода не должно вызвать у вас никаких затруднений:

```
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>
  );
 }
}
```

Обратите внимание на подписку компонента на изменение данных в `Excel` с использованием метода `onDataChange`, а также на то, что все данные сохраняются в локальном хранилище `localStorage`:

```
_commitToStorage(data) {
  localStorage.setItem('data', JSON.stringify(data));
}
```

В данном случае для сохранения данных не только на стороне клиента, но и на сервере можно было бы воспользоваться асинхронными запросами (также известными как технологии XHR, XMLHttpRequest, Ajax).

Подведение итогов

Как было показано в начале главы, основная точка входа в приложение — app.js. Сценарий app.js не является ни компонентом, ни модулем, и он ничего не экспортирует. На него возлагается лишь работа по инициализации — считывание имеющихся данных из `localStorage` и настройка компонента `<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'));

// исходные данные примера, считывание из схемы
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')
);
```

И на этом создание приложения завершается. Вы можете поработать с ним на сайте http://whinepad.com и просмотреть его код по адресу https://github.com/stoyan/reactbook/.

7 Проверка качества кода, соответствия типов, тестирование, повтор

В главе 8 будет представлена технология Flux — альтернатива управления обменом данными между компонентами (используется вместо таких средств, как метод `onDataChange`), поэтому структура кода там будет несколько изменена. А ведь неплохо было бы допускать при таком изменении как можно меньше ошибок? Рассмотрим несколько инструментальных средств, помогающих поддерживать работоспособность вашего приложения по мере его неизбежного роста и развития. К числу таких средств относятся ESLint, Flow и Jest.

Но сначала рассмотрим общее необходимое для них средство под названием `package.json`.

package.json

Как использовать npm (Node Package Manager) для установки библиотек и инструментов сторонних производителей, вы уже знаете. Кроме этого, npm также позволяет упаковывать и совмест-

но использовать ваш проект на сайте http://npmjs.com и устанавливать его другим людям. Но чтобы воспользоваться услугами, предлагаемыми npm, вам не нужно выгружать свой код на npmjs.com.

Создание пакета связано с использование файла package.json, который можно поместить в корневой каталог вашего приложения для настройки зависимостей и других дополнительных инструментов. Существует масса настроек, которые можно применить в отношении этого средства (их полный перечень можно найти на сайте https://docs.npmjs.com/files/package.json), но посмотрим, как им воспользоваться в самом скромном варианте.

Создайте в каталоге своего приложения новый файл по имени package.json:

```
$ cd ~/reactbook/whinepad2
$ touch package.json
```

И добавьте к нему следующий код:

```
{
  "name": "whinepad",
  "version": "2.0.0",
}
```

Вот и все, что нужно. После этого вы просто будете добавлять к этому файлу дополнительные настройки.

Настройка Babel

Сценарий build.sh, показанный в главе 5, запускал Babel следующим образом:

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

Эту команду можно упростить, переместив начальную настройку в package.json:

```
{
  "name": "whinepad",
  "version": "2.0.0",
```

```
  "babel": {
    "presets": [
      "es2015",
      "react"
    ]
  },
}
```

Теперь команда будет иметь следующий вид:

```
$ babel js/source -d js/build
```

Babel (как и многие другие инструментальные средства в экосистеме JavaScript) проверяет наличие файла package.json и забирает из него варианты настройки.

Сценарии

npm позволяет создавать сценарии и запускать их с помощью команды `npm run` *`имя_сценария`*. В качестве примера переместим однострочный сценарий ./scripts/watch.sh, который был показан в главе 3, в файл package.json:

```
{
  "name": "whinepad",
  "version": "2.0.0",
  "babel": {/* ... */},
  "scripts": {
    "watch": "watch \"sh scripts/build.sh\" js/source css/"
  }
}
```

Теперь для оперативной сборки кода можно воспользоваться следующей командой:

```
# до
$ sh ./scripts/watch.sh

# после
$ npm run watch
```

Если продолжить усовершенствования, можно точно так же заменить build.sh, переместив его код в package.json, или же воспользоваться специализированным средством сборки (Grunt, Gulp и т. д.), которое можно настроить в package.json. И это все (применительно к целям изучения React), что вам следует знать об этом файле.

Средство ESLint

ESLint проверяет код на наличие потенциально опасных моделей, помогает добиться единообразия вашего исходного кода, проверяя, к примеру, использование отступов и расстановку другой разрядки, а также помогает по мере создания кода отловить досадные опечатки или неиспользуемые переменные. В идеале в дополнение к запуску этого средства в качестве составной части вашего процесса сборки его нужно объединить с вашей системой проверки исходного кода и с выбранным вами текстовым редактором, чтобы вы, образно выражаясь, получали по рукам, когда будете находиться к коду ближе всего.

Установка

Кроме ESLint, вам понадобятся дополнительные модули React и Babel, чтобы помочь ESLint распознать самый передовой синтаксис ECMAScript, а также воспользоваться «правилами», свойственными JSX и React:

```
$ npm i -g eslint eslint-plugin-react eslint-plugin-babel
```

Добавьте переменную `eslintConfig` к файлу package.json:

```
{
  "name": "whinepad",
  "version": "2.0.0",
  "babel": {},
  "scripts": {},
```

```
  "eslintConfig": {
    "parser": "babel-eslint",
    "plugins": [
      "babel",
      "react"
    ],
  }
}
```

Запуск

Запустите проверку кода в отношении одного файла:

```
$ eslint js/source/app.js
```

Эта команда должна быть выполнена без ошибок, это будет означать правильное восприятие средством ESLint синтаксиса JSX и других новшеств. Но здесь есть и негативная сторона, поскольку не проводилась проверка кода на соответствие каким-либо правилам. Для каждой проверки ESLint использует правила. Сначала вам следует воспользоваться той коллекцией правил (расширением), которая рекомендована ESLint:

```
"eslintConfig": {
  "parser": "babel-eslint",
  "plugins": [],
  "extends": "eslint:recommended"
}
```

Запуск на соответствие этим правилам приводит к выявлению ошибок:

```
$ 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)
```

Второе и третье сообщения касаются переменных, не имеющих определений (исходя из правила под названием `no-undef`), но эти переменные имеют глобальный доступ в браузере, поэтому для устранения ошибки требуется дополнительная настройка:

```
"env": {
  "browser": true
}
```

Первая ошибка имеет отношение исключительно к React. С одной стороны, вам необходимо включить React, но с точки зрения ESLint в коде присутствует неиспользуемая переменная, которой здесь быть не должно. Устранить ошибку поможет добавление одного из правил, имеющихся в `eslint-plugin-react`:

```
"rules": {
  "react/jsx-uses-react": 1
}
```

При запуске проверки кода в файле сценария schema.js будет получена ошибка еще одного типа:

```
$ 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)
```

Слово `comma-dangle` означает «подвисшая запятая» (как в выражении `let a = [1,]` в противоположность выражению `let a = [1]`). Такие запятые могут считаться недопустимыми (поскольку прежде некоторыми браузерами они рассматривались как синтаксические ошибки), но выявление подобных ошибок вам на руку, поскольку упрощает проведение обновлений. Небольшие изменения

в настройках превращают практику постоянного использования запятых в поощряемое действие:

```
"rules": {
  "comma-dangle": [2, "always-multiline"],
  "react/jsx-uses-react": 1
}
```

Все правила

За полным списком правил следует обратиться к хранилищу кода, сопровождающему книгу, — этот список (как выражение лояльности к проекту) представляет собой копию собственного списка правил библиотеки React.

И наконец, добавьте проверку кода, сделав ее частью build.sh, чтобы средство ESLint контролировало ситуацию в процессе разработки, гарантируя постоянную поддержку высокого качества вашего кода:

```
# QA
eslint js/source
```

Средство Flow

Flow — статическое средство проверки соответствия типов для JavaScript. По поводу типов в целом и особенно типов в JavaScript существуют два мнения.

Кому-то нравится, что ему, образно говоря, заглядывают через плечо, чтобы убедиться, что программа работает с правильными данными. Точно так же, как проверка кода и блочное тестирование, это вселяет уверенность, что где-нибудь, где код не проверялся (или этому не придавалось особого значения), этот код не был испорчен. Ценность соблюдения типизации повышается с ростом объема приложения и неизбежно сопутствующим ему ростом количества работающих с кодом людей.

Другим же нравится динамическая, позволяющая не придерживаться строгой типизации природа JavaScript, и они полагают, что с контролем типов слишком много мороки, поскольку приведением типов приходится заниматься лишь изредка.

Разумеется, хотите вы воспользоваться этим инструментальным средством или нет, целиком зависит от вас и от вашей команды, но оно доступно для работы.

Установка

```
$ npm install -g flow-bin
$ cd ~/reactbook/whinepad2
$ flow init
```

Команда `init` создает в вашем каталоге пустой файл .flowconfig. Добавьте к нему разделы `ignore` и `include`:

```
[ignore]
.*/react/node_modules/.*

[include]
node_modules/react
node_modules/react-dom
node_modules/classnames

[libs]
[options]
```

Запуск

Для запуска нужно лишь набрать следующую команду:

```
$ flow
```

Или эту же команду — для проверки только одного файла либо каталога:

```
$ flow js/source/app.js
```

И наконец, добавьте к сценарию сборки в виде части процесса контроля качества — QA (quality assurance) — следующие команды:

```
# QA
eslint js/source
flow
```

Подписка на проверку соответствия типов

В первом комментарии файла, который нужно проверить на соответствие типов, следует воспользоваться текстом `@flow`. Однако дело это сугубо добровольное.

Начнем с подписки простого компонента `<Button>` из предыдущей главы:

```
/* @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
```

Запустим Flow:

```
$ flow js/source/components/Button.js
js/source/components/Button.js:6
  6: const Button = props =>
                    ^^^^^ parameter 'props'. Missing annotation
Found 1 error
```

Найдена ошибка, но это положительный момент — у нас появилась возможность улучшить код! Flow жалуется, что неизвестно, для чего предназначен аргумент `props`.

Flow ожидает, что такая функция:

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

должна быть аннотирована следующим образом:

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

чтобы не получалось неожиданных результатов вроде:

```
sum('1' + 2); // "12"
```

Исправление кода компонента <Button>

Аргумент `props`, получаемый функцией, является объектом. Поэтому можно сделать следующее:

```
const Button = (props: Object) =>
```

и не вызвать никаких нареканий у Flow:

```
$ flow js/source/components/Button.js
No errors!
```

Аннотация `Object` срабатывает, но можно конкретизировать происходящее, создав пользовательский тип:

```
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
```

Как видите, переключение на пользовательский тип позволяет заменить определение свойства `propTypes`, имеющееся в React. Это означает:

- прекращение проверки соответствия типов в ходе выполнения приложения. В результате выполнение кода ускоряется;
- клиенту отправляется меньше кода (меньше байтов).

Позитивно также то, что типы свойств возвращаются в верхнюю часть компонента и служат в качестве более удобной локальной документации по компоненту.

Знак вопроса в `href: ?string` означает, что это свойство может иметь значение `null`.

Теперь, когда propTypes отсутствует, жалобы на переменную PropTypes у ESLint перестанут появляться. Следовательно:

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

становится:

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

Разве плохо иметь такие средства, как ESLint, отслеживающие незначительные досадные упущения вроде этого?

Запуск Flow выявит другую ошибку:

```
$ 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
```

Проблема в том, что средство Flow не ожидало найти переменную `className` в объекте `prop`, который теперь имеет тип `Prop`. Для устранения проблемы добавьте атрибут `className` к новому типу:

```
type Props = {
  href: ?string,
  className: ?string,
};
```

app.js

При запуске Flow в отношении основного кода в app.js возникает следующая неприятность:

```
$ 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
```

Средство Flow ожидает, что методу `JSON.parse()` будут передаваться только строки, и любезно предоставляет вам сигнатуру метода `parse()`. Поскольку от `localStorage` может быть получено значение `null`, такая ситуация неприемлема. Проще всего можно справиться с данной неприятностью, просто добавив значение по умолчанию:

```
let data = JSON.parse(localStorage.getItem('data') || '');
```

Но выражение `JSON.parse('')` выдаст ошибку в браузере (хотя с точки зрения проверки соответствия типов ничего нарушено не будет), поскольку пустая строка не является приемлемыми данными, закодированными в формате JSON. Чтобы не вызывать претензий у Flow и в то же время не сталкиваться с ошибками в браузере, код нужно немного переделать.

Вы могли заметить, насколько нудной может стать работа с типами, но преимущество в том, что Flow заставляет дважды подумать о раздаваемых значениях.

Соответствующей частью кода `app.js` является:

```
let data = JSON.parse(localStorage.getItem('data'));

// исходный пример данных, считываемых из схемы
if (!data) {
  data = {};
  schema.forEach((item) => data[item.id] = item.sample);
  data = [data];
}
```

Еще одна проблема этого кода в том, что данные, которые ранее были массивом, затем превращаются в объект, после чего опять становятся массивом. У JavaScript проблем с этим нет, но то, что значение, имеющее сейчас один тип, позже превращается в зна-

чение другого типа, считается порочной практикой. Движки JavaScript, имеющиеся в браузерах, стремясь оптимизировать код, фактически выполняют назначения типов внутри себя. Следовательно, при смене типов на лету браузер может отбросить «оптимизирующий» режим, что неблагоприятно скажется на скорости его работы.

Устраним все эти проблемы.

Можно проявить особую требовательность и определить данные в качестве массива объектов:

```
let data: Array<Object>;
```

Затем попытаться считать любые сохраненные элементы в строку (или в `null`, поскольку использован символ `?`) под названием `storage`:

```
const storage: ?string = localStorage.getItem('data');
```

Если в `storage` обнаружится строка, вы просто проанализируете ее, и дело в шляпе. Или же нужно хранить данные в массиве, первый элемент которого заполнить образцом значений:

```
if (!storage) {
  data = [{}];
  schema.forEach(item => data[0][item.id] = item.sample);
} else {
  data = JSON.parse(storage);
}
```

Теперь два файла стали Flow-совместимыми. Сэкономим немного бумаги и не станем перечислять в данной главе весь набранный код, а сфокусируем внимание на более интересных особенностях Flow. Полную версию кода можно найти в хранилище, сопровождающем книгу.

Подробнее о проверке соответствия типов свойств и состояния

Когда React-компонент создается с помощью функции без поддержки состояния, свойства можно аннотировать так, как это было показано ранее:

```
type Props = {/* ... */};
const Button = (props: Props) => {/* ... */};
```

Аналогичную аннотацию можно применить с конструктором класса:

```
type Props = {/* ... */};
class Rating extends Component {
  constructor(props: Props) {/* ... */}
}
```

А если конструктор не нужен? Например, как в данном случае:

```
class Form extends Component {
  getData(): Object {}
  render() {}
}
```

На помощь придет еще одна особенность ECMAScript — свойство класса:

```
type Props = {/* ... */};
class Form extends Component {
  props: Props;
  getData(): Object {}
  render() {}
}
```

На момент написания этих строк свойства класса еще не были приняты в качестве стандарта ECMAScript, но вы можете воспользоваться ими благодаря имеющейся в Babel самой передовой предустановке stage-0. Вам нужно установить NPM-пакет

babel-preset-stage-0 и обновить раздел Babel в файле package.json следующим образом:

```
{
  "babel": {
    "presets": [
      "es2015",
      "react",
      "stage-0"
    ]
  }
}
```

Точно так же можно проаннотировать состояние вашего компонента. Кроме пользы от проверки типов, определение состояния в верхних строчках кода служит документацией для тех, кто охотится за ошибками в вашем компоненте. Рассмотрим пример:

```
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,
    };
  }
}
```

И конечно же, при каждом удобном случае вам следует применять свои собственные пользовательские типы:

```
componentWillReceiveProps(nextProps: Props) {
  this.setRating(nextProps.defaultValue);
}
```

Типы экспорта и импорта

Посмотрим на компонент `<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() {}
}
```

Здесь показано, как можно проаннотировать использование списка допустимых значений подобно React-методу `oneOf()`, применяемому для типа свойств.

Также можно увидеть, как пользовательский тип (`FormInputFieldType`) применяется в качестве части другого пользовательского типа (`FormInputField`).

И наконец, экспорт типов. Когда другой компонент использует точно такой же тип, переопределение не требуется. Он может его *импортировать* при условии, что ваш компонент окажет любезность по его *экспорту*. Посмотрим, как компонент `<Form>` использует тип из компонента `<FormInput>`:

```
import type FormInputField from './FormInput';

type Props = {
  fields: Array<FormInputField>,
  initialData?: Object,
  readonly?: boolean,
};
```

Вообще-то форме нужны оба типа из `FormInput`, и синтаксис будет иметь следующий вид:

```
import type {FormInputField, FormInputFieldValue} from './
FormInput';
```

Приведение типов

Flow позволяет указать, что конкретное значение не относится к типу, ожидаемому этим средством. Примером могут послужить обработчики событий, когда им передается объект события, а Flow воспринимает событие `target` не так, как вы рассчитывали. Рассмотрим следующий фрагмент кода из компонента `Excel`:

```
_showEditor(e: Event) {
  const target = e.target;
  this.setState({edit: {
    row: parseInt(target.dataset.row, 10),
```

```
    key: target.dataset.key,
  }});
}
```

Flow не нравится это:

```
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
```

Если посмотреть на определения, находящиеся по адресу https://github.com/facebook/flow/blob/master/lib/dom.js, то можно увидеть, что у объекта `EventTarget` нет свойства `dataset`. Но у объекта `HTMLElement` оно есть. Следовательно, на выручку придет приведение типов:

```
const target = ((e.target: any): HTMLElement);
```

Поначалу синтаксис может показаться немного странным, но, если его разобрать по частям, он обретет смысл: значение, двоеточие, тип и круглые скобки, охватывающие все три составляющие. Значение типа А становится типом Б. В данном случае объект любого типа становится таким же значением, но относится к типу `HTMLElement`.

Инварианты

Состояние в компоненте `Excel` использует два свойства, чтобы отслеживать редактирование пользователем поля и режим активного диалога:

```
this.state = {
  // ...
  edit: null, // {row index, schema.id},
  dialog: null, // {type, idx}
};
```

Эти два свойства имеют либо значения `null` (редактирование не выполняется, диалог не ведется), либо значения в виде объектов, содержащих некую информацию о редактировании или диалоге. Тип этих двух свойств может быть следующим:

```
type EditState = {
  row: number,
  key: string,
};

type DialogState = {
  idx: number,
  type: string,
};

type State = {
  data: Data,
  sortby: ?string,
  descending: boolean,
  edit: ?EditState,
  dialog: ?DialogState,
};
```

Теперь проблема в целом заключается в том, что иногда у свойств значения `null`, а иногда — нет. У Flow это вызывает вполне резонные

подозрения. При попытке использования свойства `this.state.edit.row` или `this.state.edit.key` Flow выдает ошибку:

```
Property cannot be accessed on possibly null value (Свойство
недоступно по причине возможного null-значения)
```

Вы используете эти свойства, только когда знаете об их доступности. Но Flow-то этого не знает. И никто не обещает, что по мере роста объема вашего приложения вы в конечном итоге не столкнетесь с неожиданным состоянием. А когда это произойдет, вам захочется узнать о случившемся. Чтобы не вызывать претензий у Flow и в то же время получать уведомления о неправильном поведении приложения, можно выполнить проверку на работу с пустым (`null`) значением.

До:

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

После:

```
if (!this.state.edit) {
  throw new Error('В состоянии редактирования возникла
    путаница');
}
data[this.state.edit.row][this.state.edit.key] = value;
```

Теперь все на своих местах. И когда фрагмент кода с условием на выдачу ошибки станет слишком часто повторяться, можно будет переключиться на использование функции `invariant()`. Такую функцию можно создать самостоятельно или позаимствовать из открытого исходного кода.

В этом вас поддержит NPM:

```
$ npm install --save-dev invariant
```

Добавьте в файл .flowconfig следующий код:

```
[include]
node_modules/react
node_modules/react-dom
```

```
node_modules/classnames
node_modules/invariant
```

А теперь обратитесь к вызову функции:

```
invariant(this.state.edit, 'В состоянии редактирования
  возникла путаница');
data[this.state.edit.row][this.state.edit.key] = value;
```

Тестирование

Следующая остановка на пути к беспроблемному росту объема приложения называется автоматизированным тестированием. Когда дело доходит до тестирования, перед вами опять открывается широкий выбор вариантов. Для запуска тестов в React используется инструментальное средство Jest, поэтому посмотрим на деле, что это такое и как оно может нам помочь. В качестве вспомогательного средства в React предоставляется пакет под названием react-addons-test-utils.

Итак, настало время установки дополнительных средств.

Установка

Установите интерфейс командной строки Jest:

```
$ npm i -g jest-cli
```

Вам также понадобятся babel-jest (чтобы можно было создавать тесты в стиле ES6) и пакет утилит тестирования из коллекции React:

```
$ npm i --save-dev babel-jest react-addons-test-utils
```

Далее следует обновить содержимое файла 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"
    ]
  }
}
```

Теперь можно запустить Jest с использованием следующей команды:

```
$ jest testname.js
```

Или с использованием npm:

```
$ npm test testname.js
```

Jest ищет тесты в каталоге __tests__, поэтому поместим их по путевому имени js/__tests__.

И наконец, обновим сценарий сборки, чтобы частью каждого ее процесса стали проверка кода и запуск:

```
# QA
eslint js/source js/__tests__
flow
npm test
```

Настроим также отслеживатель в файле watch.sh, чтобы он реагировал на все изменения в тестах (не забывайте, что эти функциональные возможности продублированы в package.json):

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

Первый тест

Jest является надстройкой над популярной средой Jasmine, имеющей API-интерфейс, который звучит как разговорный английский. Сначала с помощью метода `describe('набор', функция_обратного_вызова)` дается определение *набору тестов*, одной или нескольким *спецификациям тестов*, для чего используется метод `it('название теста', функция_обратного_вызова)`, а внутри каждой спецификации с помощью функции `expect()` задается утверждение.

Самый простой полноценный пример имеет следующий вид:

```
describe('Набор', () => {
  it('спецификация', () => {
    expect(1).toBe(1);
  });
});
```

Запуск теста выглядит следующим образом:

```
$ 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)
```

Если в тесте указано неверное утверждение:

```
expect(1).toBeFalsy();
```

он при выполнении дает сбой и выводит сообщение (рис. 7.1).

```
> whinepad@2.0.0 test /Users/stoyanstefanov/reactbook/whinepad2
> jest "js/__tests__/dummy-test.js"

Using Jest CLI v0.8.2, jasmine1
 FAIL  js/__tests__/dummy-test.js (3.268s)
● A suite › it is a spec
  - Expected 1 to be falsy.
        at Spec.eval (js/__tests__/dummy-test.js:3:15)
1 test failed, 0 tests passed (1 total in 1 test suite, run time 3.669s)
npm ERR! Test failed.  See above for more details.
```

Рис. 7.1. Запуск сбойного теста

Первый React-тест

Вооружившись знаниями о Jest в мире React, можно приступить к тестированию простой DOM-кнопки. Сначала выполним операции импорта:

```
import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';
```

Настроим набор тестов:

```
describe('Мы можем отобразить кнопку', () => {
  it('кнопка изменяет текст после щелчка', () => {
    // ...
  });
});
```

Покончив с шаблоном, займемся отображением и тестированием. Отобразим ряд простых элементов JSX:

```
const button = TestUtils.renderIntoDocument(
  <button
    onClick={ev => ev.target.innerHTML = 'Bye'}>
```

```
    Hello
  </button>
);
```

Здесь для вывода элемента JSX (в данном случае кнопки, изменяющей текст, когда на ней производится щелчок) мы воспользовались библиотекой утилит тестирования, имеющейся в React.

После того как что-то отображено на экране, пора проверить, соответствует ли это вашим ожиданиям:

```
expect(ReactDOM.findDOMNode(button).textContent).
toEqual('Hello');
```

Как видите, для получения доступа к DOM-узлу применяется метод `ReactDOM.findDOMNode()`. Таким образом, для проверки узла можно использовать общеизвестный API-интерфейс DOM-модели.

Зачастую нужно протестировать работу пользователя с вашим пользовательским интерфейсом. Для этого React любезно предоставляет вам метод `TestUtils.simulate`:

```
TestUtils.Simulate.click(button);
```

И последнее, что нужно проверить, — реагирует ли пользовательский интерфейс на работу с ним:

```
expect(ReactDOM.findDOMNode(button).textContent).
toEqual('Bye');
```

Далее в главе представлены дополнительные примеры и API-интерфейсы, которыми можно воспользоваться, но основным инструментарием будут такие методы:

- `TestUtils.renderIntoDocument(`*`произвольный_JSX`*`)`;
- `TestUtils.Simulate.*` для работы с интерфейсом;
- `ReactDOM.findDOMNode()` (или ряд других методов `TestUtils`) для получения ссылки на DOM-узел и проверки, имеет ли он должный вид.

Тестирование компонента <Button>

Код компонента `<Button>` имеет следующий вид:

```
/* @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
```

Протестируем его по следующим характерным особенностям:

- выводит `<a>` или `<button>` — в зависимости от наличия свойства `href` (первая спецификация);
- допускает использование пользовательских имен классов (вторая спецификация).

Начнем создание нового теста:

```
jest
  .dontMock('../source/components/Button')
  .dontMock('classnames')
;

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';
```

Инструкции `import` такие же, как и прежде, но теперь есть новые вызовы метода `jest.dontMock()`.

Имитация (mock) представляет собой замену части функционального наполнения искусственным кодом-заглушкой, якобы выполняющим работу. Заглушки часто встречаются в блочном тестировании, поскольку от него требуется протестировать блок (небольшой фрагмент кода, находящийся в изоляции) и сократить побочные эффекты относительно всего остального содержимого системы. На создание заглушек тратится немало усилий, поэтому в Jest принят противоположный подход: по умолчанию глушатся все действия. И вам предоставляется возможность отказа от глушения с помощью функции `dontMock()`, поскольку требуется протестировать не заглушку, а реальный код.

В предыдущем примере объявляется, что вы не желаете глушить `<Button>` или используемую этим компонентом библиотеку `classnames`.

Затем наступает черед включения компонента `<Button>`:

```
const Button = require('../source/components/Button');
```

На момент написания этих строк, несмотря на имеющееся в Jest-документации описание, вызов require() не работал. Вместо него требовался следующий код:

```
const Button = require('../source/components/
Button').default;
```

Также не работала инструкция import:

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

Хотя со следующим кодом было все нормально:

```
import _Button from '../source/components/Button';
const Button = _Button.default;
```

Еще один вариант предполагает использование в компоненте <Button> вместо кода export default Button кода export {Button}. А затем выполнение импорта с помощью инструкции import {Button} from '../source/component/Button'.

Надеюсь, что на момент вашего прочтения данной книги исходный импорт уже работает в соответствии с вашими ожиданиями.

Первая спецификация

Настроим набор (с помощью `describe()`) и первую спецификацию (с помощью `it()`):

```
describe('Отображение компонентов Button', () => {
  it('отображает <a> вместо <button>', () => {
    /* ... код, задающий отображение и ожидание (expect())
       ... */
  });
});
```

А теперь отобразим простую кнопку, у которой нет свойства `href`, поэтому код должен вывести элемент `<button>`:

```
const button = TestUtils.renderIntoDocument(
  <div>
    <Button>
      Hello
    </Button>
  </div>
);
```

Обратите внимание на необходимость заключения функциональных компонентов, не поддерживающих состояние, таких как `<Button>`, в еще один DOM-узел, чтобы позже их можно было найти с помощью метода `ReactDOM`.

Теперь вызов метода `ReactDOM.findDOMNode(button)` предоставит вам элемент-оболочку `<div>`; чтобы получить `<button>`, берется первый дочерний элемент и проверяется, что он действительно является кнопкой:

```
expect(ReactDOM.findDOMNode(button).children[0].nodeName).
➥ toEqual('BUTTON');
```

По аналогии в качестве составной части той же спецификации теста задается проверка, определяющая, что при наличии свойства `href` используется узел гипертекстовой ссылки:

```
const a = TestUtils.renderIntoDocument(
  <div>
```

```
    <Button href="#">
      Hello
    </Button>
  </div>
);
expect(ReactDOM.findDOMNode(a).children[0].nodeName).
➥ toEqual('A');
```

Вторая спецификация

Во второй спецификации добавляются имена пользовательских классов, а затем проверяется, можно ли их найти в нужном месте:

```
it('разрешает применять пользовательские классы CSS', () => {
  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');
});
```

Здесь важно подчеркнуть факт, касающийся Jest-глушения. Порой написанный таким образом тест работает совершенно не так, как ожидалось. Такое может произойти, если забыли снять Jest-заглушку. Следовательно, если в самом начале теста находится следующий код:

```
jest
  .dontMock('../source/components/Button')
  // .dontMock('classnames')
;
```

то Jest глушит модуль `classnames` — и тест ничего не делает. Убедиться в этом можно, написав следующий код:

```
const button = TestUtils.renderIntoDocument(
  <div><Button className="good bye">Hello</Button></div>
);
console.log(ReactDOM.findDOMNode(button).outerHTML);
```

Он запишет в консоль этот сгенерированный код HTML:

```
<div data-reactid=".2">
  <button data-reactid=".2.0">Hello</button>
</div>
```

Как видите, никаких нужных имен классов вообще не наблюдается, поскольку в заглушенном состоянии метод `classNames()` ничего не делает.

Вернем на место вызов метода `dontMock()`:

```
jest
  .dontMock('../source/components/Button')
  .dontMock('classnames')
;
```

и вы увидите, что вызов атрибута `outerHTML` показал следующий код:

```
<div data-reactid=".2">
  <button class="Button good bye"
    data-reactid=".2.0">Hello</button>
</div>
```

и тест был пройден успешно.

Когда тест ведет себя непонятно и вы интересуетесь, как выглядит созданная разметка, проще всего воспользоваться инструкцией console.log(node.outerHTML), показывающей сам код HTML.

Тестирование компонента <Actions>

`<Actions>` является еще одним компонентом, не поддерживающим состояние, следовательно, чтобы позже его можно было изучить, он нуждается в оболочке. Один из вариантов, как уже было показано на примере `<Button>`, предусматривает заключение его

в div-контейнер и получение к нему доступа при помощи следующего кода:

```
const actions = TestUtils.renderIntoDocument(
  <div><Actions /></div>
);
ReactDOM.findDOMNode(actions).children[0];
// Корневой узел <Actions>
```

Оболочка компонента

Еще один вариант — использование элемента-оболочки от React, который затем позволит вам применять множество методов TestUtils для охоты на проверяемые узлы.

Оболочка не содержит сложного кода. Определить оболочку можно в ее собственном модуле, это дает возможность многократного использования:

```
import React from 'react';
class Wrap extends React.Component {
  render() {
    return <div>{this.props.children}</div>;
  }
}
export default Wrap
```

Теперь шаблонная часть теста приобрела следующий вид:

```
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('Щелчок на значке действия', () => {
```

```
  it('вызывает определенную реакцию', () => {
    /* отображение */
    const actions = TestUtils.renderIntoDocument(
      <Wrap><Actions /></Wrap>
    );
    /* ... поиск и проверка */
  });
});
```

Mock-функции

В компоненте действий `<Actions>` нет ничего необычного. Его код выглядит следующим образом:

```
const Actions = (props: Props) =>
  <div className="Actions">
    <span
      tabIndex="0"
      className="ActionsInfo"
      title="More info"
      onClick={props.onAction.bind(null,
        'info')}>&#8505;</span>
    {/* ... еще два span-узла */}
  </div>
```

При тестировании единственное, что нужно проверить, — это надлежащий вызов при щелчке на значках действий вашей функции обратного вызова `onAction`. Jest позволяет определять функции-имитаторы (или mock-функции) и проверять факт их вызова. При использовании функций обратного вызова этого вполне достаточно.

В теле теста создается новая mock-функция, которая передается компоненту `Actions` в качестве функции обратного вызова:

```
const callback = jest.genMockFunction();
const actions = TestUtils.renderIntoDocument(
  <Wrap><Actions onAction={callback} /></Wrap>
);
```

Затем следует заняться щелчками на значках действий:

```
TestUtils
  .scryRenderedDOMComponentsWithTag(actions, 'span')
  .forEach(span => TestUtils.Simulate.click(span));
```

Обратите внимание на использование одного из методов `TestUtils` для поиска DOM-узлов. Он возвращает массив из трех `<span>`-узлов, и вы имитируете щелчок на каждом из них.

Теперь ваша mock-функция обратного вызова должна вызываться три раза. Подтвердите с помощью метода `expect()`, что именно это вам и нужно:

```
const calls = callback.mock.calls;
expect(calls.length).toEqual(3);
```

Как видите, свойство вызовов `callback.mock.calls` является массивом. У каждого вызова также имеется массив аргументов, переданных ему в процессе вызова.

Первое действие называется `'info'`, и оно вызывает `onAction`, передавая тип действия `'info'` и используя для этого код `props.onAction.bind(null, 'info')`. Следовательно, первым аргументом (0) для первой mock-функции обратного вызова (0) должен быть `'info'`:

```
expect(calls[0][0]).toEqual('info');
```

Аналогично этому ожидаемые результаты двух других действий задаются следующим кодом:

```
expect(calls[1][0]).toEqual('edit');
expect(calls[2][0]).toEqual('delete');
```

Методы find и scry

Библиотека `TestUtils` предоставляет вам ряд функций для поиска DOM-узлов в дереве отображения React. Например, поиск узла по имени тега или имени класса. Один пример вы уже видели:

```
TestUtils.scryRenderedDOMComponentsWithTag(actions, 'span')
```

А вот еще один:

```
TestUtils.scryRenderedDOMComponentsWithClass(actions,
  'ActionsInfo')
```

Наряду с методами `scry*` в вашем распоряжении есть соответствующие методы `find*`. Например:

```
TestUtils.findRenderedDOMComponentWithClass(actions,
  'ActionsInfo')
```

Обратите внимание на использование в имени составляющей слова `Component`, а не `Components`. В отличие от методов `scry*`, которые дают массив совпадений (даже если совпадение всего одно или их вовсе нет), методы `find*` возвращают только одно совпадение. Если совпадений нет или их сразу несколько, возникает ошибка. Поэтому поиски с помощью методов `find*` всегда ведутся с полной уверенностью, что в дереве имеется всего лишь один искомый DOM-узел.

Дополнительные имитируемые взаимодействия

Протестируем виджет `Rating`. Он изменяет состояние при наступлении событий прохождения указателя мыши над элементом (mouseover), увода указателя с элемента (mouseout) и щелчка на элементе (click). Используемый шаблон имеет следующий вид:

```
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('работы', () => {
```

```
  it('обрабатывает пользовательские действия', () => {
    const input = TestUtils.renderIntoDocument(<Rating />);
    /* изложите здесь в методе expect() ваши ожидания */
  });
});
```

Обратите внимание, что заключать `<Rating>` при его отображении в какую-либо оболочку не нужно. Это не функциональный компонент, поддерживающий состояние, поэтому он вполне работоспособен и без оболочки.

У виджета имеется несколько звезд (по умолчанию пять), каждая из которых заключена в `span`-контейнер. Найдем их:

```
const stars = TestUtils.scryRenderedDOMComponentsWithTag(input,
'span');
```

Теперь тест имитирует действия, вызывающие наступление события `mouseOver`, затем `mouseOut` и следом `click` на четвертой звезде (`span[3]`). Когда это произойдет, звезды с первой по четвертую должны перейти в состояние «включено», иными словами, получить имя класса `RatingOn`, а пятая звезда должна оставаться «выключенной»:

```
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);
```

Обратите также внимание на то, как тест добирается до состояния компонента, чтобы проверить корректность значений `state.rating` и `state.tmpRating`. Возможно, это несколько бесцеремонно, но все же, если ожидаются «открытые» результаты, какая разница, какое внутреннее состояние компонент выбирает для управления? Но выяснить это, конечно же, возможно.

Тестирование полного взаимодействия

Напишем несколько тестов для компонента `Excel`. Все же он достаточно большой и способен серьезно нарушить поведение приложения, если что-нибудь пойдет не так. Для начала создадим следующий код:

```
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('Редактирование данных', () => {
  it('сохраняет новые данные', () => {
    /* ... отображение, взаимодействие, проверка */
  });
});
```

В первую очередь обратите внимание на вызов метода `jest.autoMockOff();` в самом начале кода. Вместо перечисления всех компонентов, используемых компонентом `Excel` (и компонентов, которые те, в свою очередь, применяют), можно одним махом выключить полностью все глушение.

Затем вам, очевидно, понадобятся схема и образцовые данные для инициализации компонента (аналогично app.js).

Теперь перейдем к отображению:

```
const callback = jest.genMockFunction();
const table = TestUtils.renderIntoDocument(
  <Excel
    schema={schema}
    initialData={data}
    onDataChange={callback} />
);
```

Все это, конечно, хорошо, но теперь изменим значение в первой ячейке первой строки. Зададим новое значение:

```
const newname = '$2.99 chuck';
```

Нас интересует следующая ячейка:

```
const cell = TestUtils.scryRenderedDOMComponentsWithTag(table,
  'td')[0];
```

На момент написания книги для предоставления поддержки dataset, отсутствующей в используемой Jest реализации DOM-модели, требовался обходной вариант:

```
cell.dataset = { // обход недостатков поддержки DOM,
                 // имеющихся в Jest
  row: cell.getAttribute('data-row'),
  key: cell.getAttribute('data-key'),
};
```

Двойной щелчок на ячейке превращает ее содержимое в форму с полем ввода данных:

```
TestUtils.Simulate.doubleClick(cell);
```

Изменение значения поля ввода данных и отправка формы:

```
cell.getElementsByTagName('input')[0].value = newname;
TestUtils.Simulate.submit(cell.getElementsByTagName('form')[0]);
```

Теперь содержимое ячейки уже является не формой, а простым текстом:

```
expect(cell.textContent).toBe(newname);
```

И функция обратного вызова `onDataChange` была вызвана с массивом, содержащим объекты, состоящие из данных таблицы в виде пар «ключ — значение». Можно проверить, что mock-функция обратного вызова получает новые данные надлежащим образом:

```
expect(callback.mock.calls[0][0][0].name).toBe(newname);
```

Здесь `[0][0][0]` означает, что первым аргументом mock-функции выступает массив, в котором первый элемент является объектом (соответствующим записи в таблице) со свойством `name`, равным `"$2.99 chuck"`.

Вместо использования TestUtils.Simulate.submit можно выбрать TestUtils.Simulate.keyDown и выдать событие нажатия клавиши Enter, при котором также осуществляется отправка данных формы.

В качестве второй спецификации теста удалим одну строку образцовых данных:

```
it('deletes data', () => {
  // То же, что и раньше
```

```
  const callback = jest.genMockFunction();
  const table = TestUtils.renderIntoDocument(
    <Excel
      schema={schema}
      initialData={data}
      onDataChange={callback} />
  );

  TestUtils.Simulate.click( // значок x
    TestUtils.findRenderedDOMComponentWithClass(table,
    'ActionsDelete')
  );

  TestUtils.Simulate.click( // диалог подтверждения
    TestUtils.findRenderedDOMComponentWithClass(table,
    'Button')
  );

  expect(callback.mock.calls[0][0].length).toBe(0);
});
```

Как и в предыдущем примере, `callback.mock.calls[0][0]` является новым массивом данных после взаимодействия. Только на этот раз в нем ничего не осталось, поскольку одну запись тест удалил.

Полнота охвата

После того как вы изучите эти темы, ситуация упростится и может стать повторяющейся. Обеспечить охват тестированием всех возможных сценариев развития событий можете только вы. К примеру, щелкните на кнопке открытия информационного окна, отмените действие, щелкните на кнопке открытия окна удаления записи, отмените действие, щелкните еще раз и только после этого удалите запись.

Использование тестов — разумное решение, поскольку они помогают ускорить разработку, приобрести уверенность в своих действиях и проводить реорганизацию кода без лишних опасений за конечный результат. Тесты помогают поставить на место ваших соисполнителей, которые думают, что вносимые ими изменения носят изолированный характер, а на самом деле последствия распространяются гораздо шире. Один из способов придать процессу написания тестов форму некой игры заключается в использовании возможности *охвата кода*.

Можно применить следующую команду:

```
$ jest --coverage
```

запускающую все тесты, которые только могут быть найдены, и выводящую отчет о том, сколько строк, функций и тому подобного было протестировано (или *охвачено тестами*). Пример показан на рис. 7.2.

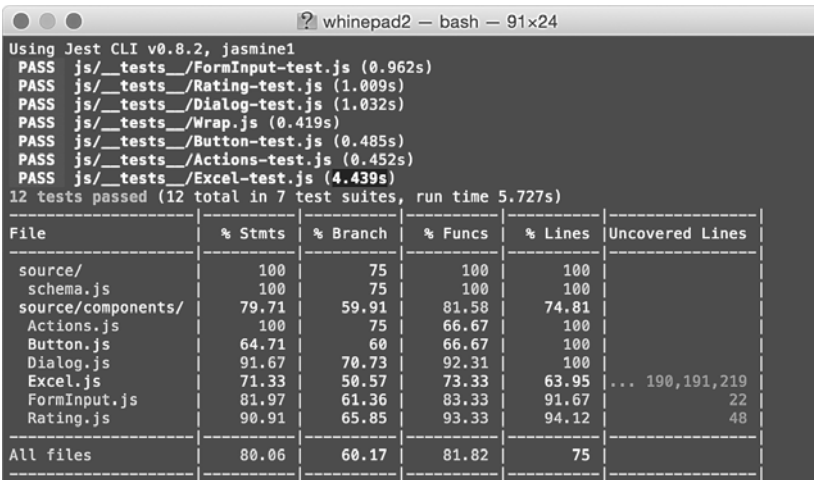


Рис. 7.2. Отчет об охвате тестированием всего кода

Как видите, не все прошло идеально и, несомненно, есть потенциальная возможность для написания дополнительных тестов.

Одной из полезных особенностей отчета об охватывающем тестировании является возможность выявления не прошедших тестирование строк кода. То есть, даже если вы протестировали компонент `FormInput`, строка 22 не была протестирована. Номера строк, вызывающих вопросы, возвращаются инструкцией `return`:

```
getValue(): FormInputFieldValue {
  return 'value' in this.refs.input
    ? this.refs.input.value
    : this.refs.input.getValue();
}
```

Оказывается, что эта функция никогда тестами не проверялась. Настало время исправить ситуацию, оперативно задав тестовую спецификацию:

```
it('возвращает введенное значение', () => {
  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);
});
```

Первым вызовом `expect()` тестируется поле ввода, встроенное в DOM-модель, а вторым вызовом тестируется специализированное поле ввода. Теперь должны быть выполнены оба итога применения тернарного оператора в методе `getValue()`.

Отчет об охвате кода вознаграждает вас результатом, показывающим, что теперь строка 22 также охвачена тестированием (рис. 7.3).

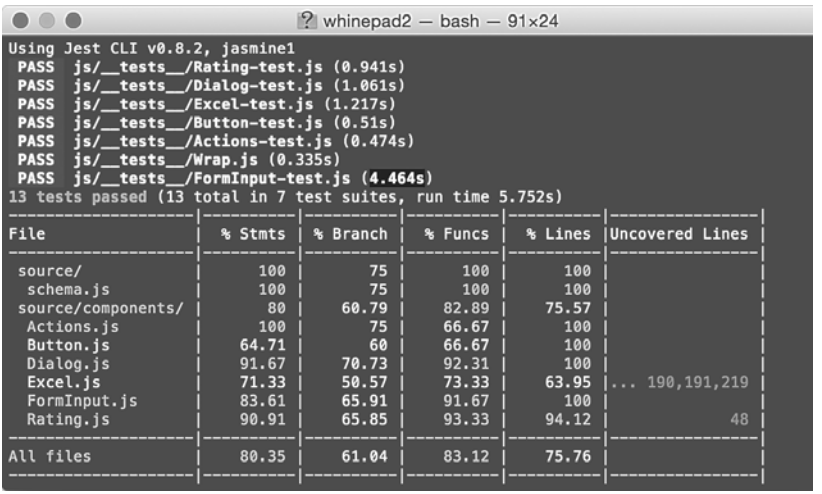


Рис. 7.3. Обновленный отчет об охвате тестированием всего кода

8 Flux

Последняя глава этой книги посвящена Flux — технологии, являющейся альтернативным способом управления обменом данными между компонентами и средством управления всеми потоками данных в вашем приложении. До сих пор вы видели, как обмен данными выполняется путем передачи свойств от родительского компонента дочернему с последующим отслеживанием изменений, происходящих в дочернем компоненте (например, с помощью метода `onDataChange`). Но, передавая свойства таким способом, иногда в конечном итоге можно получить компонент, принимающий слишком много свойств. Это затрудняет тестирование такого компонента и проверку того, что все возможные комбинации и перестановки свойств работают в соответствии с вашими ожиданиями.

Кроме того, временами возникают сценарии, где для свойств нужен «сквозной канал» от родительского компонента к дочернему, а затем и к внучатому, правнучатому и т. д. Появляется тенденция повторяемости (что плохо само по себе), которая вызывает путаницу и повышает психическую нагрузку на человека, читающего код (слишком многое приходится одновременно держать в памяти).

Flux предлагает способ преодоления этих трудностей и сохранения комфортного психического состояния, не утрачивая при этом возможности управления потоком данных в вашем приложении. Flux не является библиотекой кодов, скорее это замысел, касающийся способа организации (создания архитектуры) данных приложения. Все же в большинстве случаев важны именно данные. Пользователи заходят в ваше приложение для работы со своими деньгами, электронной почтой, фотографиями или чем-нибудь еще. Даже если пользовательский интерфейс не отличается особой элегантностью, они могут с этим смириться. Но они никогда не должны попадать в непонятные ситуации относительно состояния данных («Я отправил или не отправил эти 30 долларов?»).

Идеи, присущие Flux-технологии, воплощены во многих реализациях с открытым кодом. Вместо всеобъемлющего охвата имеющихся вариантов в этой главе рассматривается подход типа «сделай сам». Как только вы уясните замысел (и приобретете уверенность в преимуществах его реализации), можно будет продолжить исследование доступных вариантов или приступить к разработке собственного решения.

Основной замысел

Замысел основан на том, что самое важное в вашем приложении — это данные. Они содержатся в *хранилище* (store). React-компоненты, или *представление* (view), считывают данные из хранилища и выводят их на экран. Затем дает о себе знать пользователь приложения, выполняющий *действие* (action), щелкая, к примеру, на кнопке. Действие заставляет обновлять данные в хранилище, что оказывает влияние на представление. И этот цикл повторяется раз за разом (рис. 8.1). Поток данных

идет в одном направлении (однонаправленно), что существенно упрощает отслеживание, осмысление происходящего и отладку.

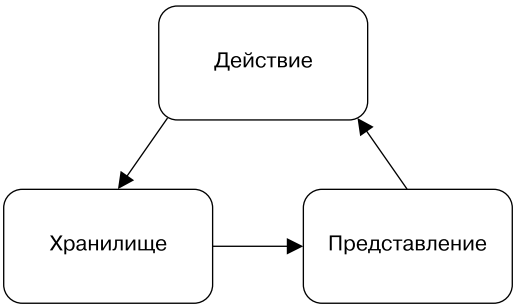


Рис. 8.1. Однонаправленный поток данных

Существует множество вариантов и развитий этой идеи, где используется большее количество действий, несколько хранилищ и работа *диспетчера* (dispatcher), но прежде чем переходить к пространным объяснениям, посмотрим на программный код.

Иной взгляд на Whinepad

У приложения Whinepad есть высокоуровневый React-компонент `<Whinepad>`, созданный с применением следующего кода:

```
<Whinepad
  schema={schema}
  initialData={data} />
```

В свою очередь, компонент `<Whinepad>` формирует компонент `<Excel>`:

```
<Excel
  schema={this.props.schema}
```

```
initialData={this.state.data}
onDataChange={this._onExcelDataChange.bind(this)} />
```

Сначала от `<Whinepad>` к `<Excel>` в неизменном виде передается (проходит по каналу) `schema`, то есть описание данных, с которыми работает приложение. (А затем точно так же осуществляется ее передача в адрес компонента `<Form>`.) Здесь явно прослеживаются некая повторяемость и шаблонность. А что, если потребуется прогнать по каналу несколько подобных свойств? Вскоре *внешняя сторона* ваших компонентов станет слишком большой, что явно не принесет особой пользы.

Понятие «внешняя сторона» в данном контексте означает свойства, получаемые компонентом. Оно используется в качестве синонима API-интерфейса или сигнатуры функции. Как и всегда в программировании, внешнюю сторону лучше сводить к минимуму. Функцию, получающую десять аргументов, намного труднее использовать, доводить до нужного состояния и тестировать, чем функцию, получающую всего два аргумента или вообще не получающую никаких аргументов.

Схема `schema` просто передается в неизменном виде, но, похоже, на данные это не распространяется. `<Whinepad>` получает свойство `initialData`, но затем передает компоненту `<Excel>` его версию (`this.state.data`, а не `this.props.initialData`). И тут возникают вопросы: чем новые данные отличаются от оригинала? И где находится единственный источник истины, когда речь заходит о самых последних данных?

В реализации, которая была показана в предыдущей главе, самые актуальные данные содержались в `<Whinepad>` и все работало без нареканий. Но не вполне понятно, почему компонент пользовательского интерфейса (а React занимается созданием

пользовательского интерфейса) должен быть хранителем источника истины.

Для выполнения этой миссии введем хранилище.

Хранилище

Сначала скопируем весь созданный до сих пор код:

```
$ cd ~/reactbook
$ cp -r whinepad2 whinepad3
$ cd whinepad3
$ npm run watch
```

Затем создадим новый каталог для хранения Flux-модулей (чтобы отделить их от компонентов пользовательского интерфейса React), которых будет всего два — хранилище (store) и действия (actions):

```
$ mkdir js/source/flux
$ touch js/source/flux/CRUDStore.js
$ touch js/source/flux/CRUDActions.js
```

В архитектуре Flux может быть несколько хранилищ (например, одно для пользовательских данных, другое — для настроек приложения и т. д.), но наше внимание будет приковано только к одному — CRUD-хранилищу, целиком предназначенному для списка записей; в данном примере речь пойдет о записях марок вина и вашего мнения о них.

Для `CRUDStore` технология React не пригодится, и код хранилища может быть реализован в виде простого объекта JavaScript:

```
/* @flow */

let data;
```

```
let schema;

const CRUDStore = {

getData(): Array<Object> {
  return data;
},

getSchema(): Array<Object> {
  return schema;
},
};

export default CRUDStore
```

Как видите, хранилище обслуживает единый источник истины в виде локального модуля данных переменных и схемы и охотно возвращает эти данные всем, кто ими интересуется. Хранилище также допускает обновление данных (за исключением схемы, остающейся неизменной в течение всего жизненного цикла приложения):

```
setData(newData: Array<Object>, commit: boolean = true) {
  data = newData;
  if (commit && 'localStorage' in window) {
    localStorage.setItem('data', JSON.stringify(newData));
  }
  emitter.emit('change');
},
```

Здесь, кроме обновления локальной переменной `data`, обновляется и постоянное хранилище, которое в данном случае представлено объектом `localStorage`, но оно может быть также размещено на сервере, получающем XHR-запрос. Это обновление происходит только при отправке данных, поскольку обновлять постоянное хранилище при каждом изменении не нужно. Например, при поиске хочется, чтобы его результатом были самые

последние данные, но не нужно, чтобы результаты попадали на постоянное хранение. Что если после вызова `setData()` случится сбой электропитания и будут утеряны все данные, кроме результатов поиска?

И наконец, здесь видно, что выдается событие `'change'` (изменение). (На этом моменте мы еще остановимся.)

Другими полезными методами, которые могут быть предоставлены хранилищем, являются общий подсчет строк данных и подчсет объема данных в одной строке:

```
getCount(): number {
  return data.length;
},

getRecord(recordId: number): ?Object {
  return recordId in data ? data[recordId] : null;
},
```

Чтобы запустить приложение, нужно инициализировать хранилище. Прежде эта работа выполнялась в app.js, но теперь она вполне резонно возлагается на хранилище, чтобы вся работа с данными велась в одном месте:

```
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);
  }
},
```

А в app.js теперь осуществляется начальная загрузка приложения:

```
// ...
import CRUDStore from './flux/CRUDStore';
import Whinepad from './components/Whinepad';
import schema from './schema';

CRUDStore.init(schema);

ReactDOM.render(
  <div>
    {/* код JSX */}
    <Whinepad />
  {/* ... */}
);
```

Как видите, после инициализации хранилища компоненту `<Whinepad>` не нужно получать никаких свойств. Необходимые ему данные доступны с помощью вызова метода `CRUDStore.getData()`, а описание данных берется из вызова метода `CRUDStore.getSchema()`.

Возможно, вас удивляет, почему хранилище считывает данные отдельно, а затем полагается на схему, передаваемую извне. Разумеется, можно заставить хранилище импортировать модуль schema. Но, наверное, вполне разумно позволить приложению решать, откуда будет браться схема. Будет ли она определяться пользователем, или поставляться в виде модуля, или присутствовать в виде жестко заданного кода?

События хранилища

Вы помните часть кода `emitter.emit('change');`, выполняемую при обновлении хранилищем своих данных? Это способ информирования хранилищем любых заинтересованных модулей пользовательского интерфейса о том, что данные изменились

и они теперь могут выполнить собственное обновление, считывая свежие данные из хранилища. А как выполняется эта выдача события?

Реализовать модель подписки на событие можно множеством способов, по сути, все сводится к составлению списка заинтересованных сторон (подписчиков) и к необходимости «публикации» о наступлении события путем запуска на выполнение функции обратного вызова каждого подписчика (той функции, которую подписчик предоставляет при оформлении подписки).

Чтобы не утруждать себя созданием собственного кода, для формирования частей кода, занимающихся подпиской на события, воспользуемся небольшой открытой библиотекой под названием fbemitter:

```
$ npm i --save-dev fbemitter
```

Обновим файл .flowconfig:

```
[ignore]
.*/fbemitter/node_modules/.*
# и так далее ...

[include]
node_modules/classnames
node_modules/fbemitter
# и так далее ...
```

Импортирование и инициализация источника событий происходят в верхней части модуля хранилища:

```
/* @flow */

import {EventEmitter} from 'fbemitter';

let data;
```

```
let schema;
const emitter = new EventEmitter();

const CRUDStore = {
  // ...
};
export default CRUDStore
```

У источника событий две задачи:

- сбор подписок;
- уведомление подписчиков (как уже было показано при использовании `emitter.emit('change')` в `setData()`).

Предоставить коллекции подписчиков можно с использованием метода в хранилище, поэтому вызывающему коду вдаваться в какие-либо подробности не нужно:

```
const CRUDStore = {
  // ...
  addListener(eventType: string, fn: Function) {
      emitter.addListener(eventType, fn);
  },
  // ...
};
```

Вот теперь хранилище `CRUDStore` стало функционально завершенным.

Использование хранилища в <Whinepad>

Компонент `<Whinepad>` при использовании Flux-технологии существенно упростился. Главным образом это достигнуто за счет перекладывания функциональных обязанностей на модуль `CRUDActions` (который вскоре будет показан), но помощь оказана и со стороны модуля `CRUDStore`. Работать со свойством `this.state.data` больше

нет необходимости. Оно было востребовано только лишь для его передачи компоненту `<Excel>`. Но теперь `<Excel>` может обратиться за данными в хранилище. Фактически компоненту `<Whinepad>` вообще не нужно работать с хранилищем. Но добавим еще одну функцию, для которой требуется обращение к хранилищу. Эта функция предназначена для отображения общего количества записей в поле поиска (рис. 8.2).

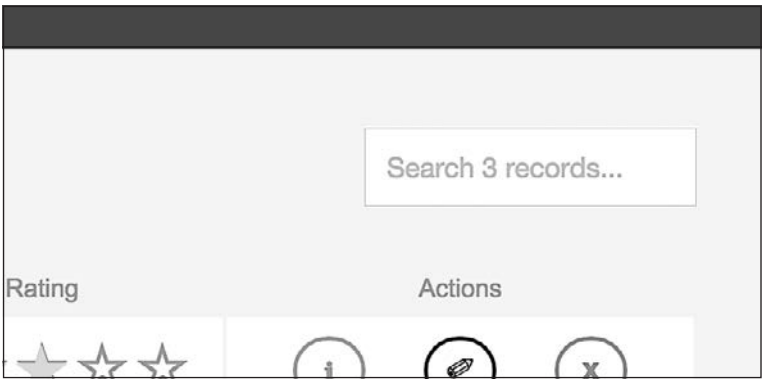


Рис. 8.2. Количество записей в поле поиска

Ранее метод `constructor()` компонента `<Whinepad>` устанавливал состояние следующим образом:

```
this.state = {
  data: props.initialData,
  addnew: false,
};
```

Теперь вам не требуется свойство `data`, но возникла потребность в свойстве `count`, которое следует инициализировать путем считывания данных из хранилища:

```
/* @flow */

// ...
```

```
import CRUDStore from '../flux/CRUDStore';
// ...

class Whinepad extends Component {
  constructor() {
    super();
    this.state = {
      addnew: false,
      count: CRUDStore.getCount(),
    };
  }
  /* ... */
}

export default Whinepad
```

Дополнительно в конструктор нужно добавить подписку на изменения в хранилище, чтобы получить возможность обновления общего количества в `this.state`:

```
constructor() {
  super();
  this.state = {
    addnew: false,
    count: CRUDStore.getCount(),
  };

  CRUDStore.addListener('change', () => {
    this.setState({
      count: CRUDStore.getCount(),
    })
  });
}
```

И на этом все обязательное взаимодействие с хранилищем заканчивается. При любом производимом каким-либо образом обновлении данных в хранилище (и вызове в `CRUDStore` метода `setData()`) хранилище выдает событие `'change'` (изменение).

Компонент <Whinepad> отслеживает выдачу этого события и обновляет свое состояние. Как вам уже известно, установка состояния приводит к повторному отображению компонента, то есть снова вызывается метод render(). Его задача, как и обычно, заключается в создании пользовательского интерфейса на основе состояния и значений свойств:

```
render() {
  return (
    {/* ... */}
    <input
      placeholder={this.state.count === 1
        ? 'Search 1 record...'
        : `Search ${this.state.count} records...`
      }
    />
    {/* ... */}
  );
}
```

Есть смысл также реализовать в <Whinepad> метод shouldComponentUpdate(). В данных могут осуществляться изменения, не оказывающие влияния на общее количество записей (например, редактирование записи или редактирование отдельного поля в записи). В таком случае компонент не требует повторного отображения:

```
shouldComponentUpdate(newProps: Object, newState: State):
boolean {
  return (
    newState.addnew !== this.state.addnew ||
    newState.count !== this.state.count
  );
}
```

И наконец, компоненту <Whinepad> больше не нужно передавать свойства данных и схемы компоненту <Excel>. Не нужно ему

и подписываться на метод onDataChange, поскольку все изменения поступают в виде события 'change' от хранилища. Теперь соответствующие части метода render() в компоненте <Whinepad> получили следующий вид:

```
render() {
  return (
    {/* ... */}
    <div className="WhinepadDatagrid">
        <Excel />
    </div>
    {/* ... */}
  );
}
```

Использование хранилища в <Excel>

Компонент <Excel> точно так же, как и <Whinepad>, больше не нуждается в свойствах. Конструктор может считать из хранилища схему schema и сохранить ее в this.schema. Разница между хранением схемы в this.state.schema и в this.schema только в том, что состояние предполагает некую степень изменений, а схема является константой.

Что касается данных, исходное значение this.state.data считывается из хранилища и его получение в виде свойства больше уже не осуществляется.

И наконец, конструктор подписывается на событие хранилища 'change', поэтому состояние может быть обновлено самыми свежими данными (что вызовет запуск повторного отображения):

```
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(),
    })
  });
}
```

И это все, что нужно сделать в компоненте `<Excel>`, чтобы воспользоваться хранилищем. Метод `render()` по-прежнему считывает данные из `this.state` для точно такого же их представления, что и раньше.

Необходимость копирования данных из хранилища в `this.state` может вызвать удивление. А нельзя ли сделать так, чтобы метод `render()` получал доступ к хранилищу и выполнял чтение непосредственно из него? Конечно, можно. Но тогда компонент утратит свою «чистоту». Следует помнить, что чистый компонент визуализации выполняет отображение только на основе имеющихся у него свойств и состояния. Любые вызовы функций в `render()` выглядят подозрительно, ведь никогда не известно, какого рода значения будут получены из внешнего вызова. Возникают сложности в отладке, и приложение становится менее предсказуемым («Почему показано число 2, когда в состоянии содержится 1? А, вот в чем дело, оказывается, в `render()` есть вызов функции»).

Использование хранилища в <Form>

Компонент формы также получает схему (в виде свойства полей `fields`) и свойство исходных значений `defaultValues` для предварительного заполнения формы или отображения версии,

предназначенной только для чтения. И то и другое отныне находится в хранилище. Теперь форма может взять свойство `recordId` и найти нужные данные в хранилище:

```
/* @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
```

Форма не подписывается на событие хранилища `'change'`, поскольку не ожидает изменений в ходе редактирования в ней данных. Но такое развитие событий нельзя исключать; скажем, другой пользователь редактирует данные в то же самое время или

тот же самый пользователь открыл то же самое приложение в двух вкладках и редактирует данные в обоих экземплярах приложения. В таком случае можно отслеживать изменения данных и предупреждать пользователя, работающего в другом экземпляре приложения.

Где провести границу?

Где провести границу между использованием Flux-хранилища и применением свойств в реализации, предшествовавшей Flux? Хранилище представляет собой удобный универсальный магазин, удовлетворяющий все потребности в данных. Оно избавляет вас от необходимости раздачи свойств. Но оно снижает возможность многократного использования компонентов. Теперь уже нельзя повторно использовать компонент `Excel` в совершенно другом контексте, поскольку в нем жестко закодирован поиск данных в хранилище `CRUDStore`. Но при условии, что новый контекст похож на эту используемую CRUD-технологию (что весьма вероятно, иначе зачем бы вам понадобилась редактируемая таблица данных?), вы также можете воспользоваться и хранилищем. Однако не забудьте, что приложение может применять столько хранилищ, сколько ему угодно.

Низкоуровневые компоненты вроде кнопок и форм ввода лучше не связывать с хранилищем. Они вполне могут справиться со своей задачей, пользуясь исключительно свойствами. Любые типы компонентов, находящиеся между двумя экстремальными позициями, — простые кнопки (такие как `<Button>`) и общие родительские компоненты (такие как `<Whinepad>`) — попадают в зону неопределенности, и решение остается за вами. Должна ли форма прикрепляться к хранилищу типа `CRUDstore`, как показано ранее, или должна обходиться без него и сохранять возможность повсеместного повторного использования? Выберите наиболее

рациональные решения с точки зрения поставленной задачи и перспектив повторного использования создаваемого в данный момент компонента.

Действия

Действия (actions) — способ изменения данных в хранилище (store). Когда пользователи взаимодействуют с представлением (view), они совершают действия, обновляющие данные в хранилище, которое отправляет событие заинтересованным в нем представлениям.

Для реализации действий `CRUDActions`, обновляющих хранилище `CRUDStore`, можно ничего не усложнять и воспользоваться простым объектом JavaScript:

```
/* @flow */

import CRUDStore from './CRUDStore';

const CRUDActions = {
  /* методы */
};

export default CRUDActions
```

CRUD-действия

Какие методы должны быть реализованы в модуле `CRUDActions`? Обычно предполагается, что создание — `create()`, удаление — `delete()`, обновление — `update()`... Вот только в этом приложении можно обновить всю запись или обновить отдельное поле, поэтому реализуем методы `updateRecord()` и `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);
  },

  /* ... */
};
```

Во всем этом нет ничего необычного: текущие данные считываются из хранилища, с ними производятся определенные действия (обновление, удаление, добавление или создание), а затем данные записываются обратно в хранилище.

Составляющая R акронима CRUD вам не нужна, поскольку соответствующее действие предоставляется хранилищем.

Поиск и сортировка

В предыдущей реализации компонент `<Whinepad>` отвечал за поиск данных. Причина заключалась в том, что поле поиска находилось в принадлежащем компоненту методе `render()`. Но вообще-то поиск должен быть где-нибудь ближе к данным.

Точно так же сортировка была частью компонента `<Excel>`, поскольку для ее выполнения использовались обработчики события `onclick`, выдаваемого по щелчку на заголовках таблицы. Но, опять-таки, сортировку лучше выполнять ближе к тому месту, где содержатся данные.

Можно порассуждать, где именно должны осуществляться поиск и сортировка — в действиях или хранилище? Похоже, что подходят оба места. Но в данной реализации оставим хранилище в покое, чтобы оно могло только получать и устанавливать данные, а также отвечать за выдачу событий. Манипуляция данными осуществляется в действиях, поэтому перенесем сортировку и поиск из компонентов пользовательского интерфейса в модуль `CRUDActions`:

```
/* @flow */
/* ... */
const CRUDActions = {

  /* ... CRUD-методы ... */

  _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)
    ));
  },
};
```

И с этим кодом модуль `CRUDActions` можно считать функционально законченным. Посмотрим, как он используется компонентами `<Whinepad>` и `<Excel>`.

Можно не согласиться с тем, что данная часть функции sort() принадлежит CRUDActions:

```
search(e: Event) {
  const target = ((e.target: any): HTMLInputElement);
  const needle: string = target.value.toLowerCase();
  /* ... */
}
```

Возможно, модуль действия не должен ничего знать о пользовательском интерфейсе, и «правильная» сигнатура должна быть больше похожа на следующую:

```
search(needle: string) {
  /* ... */
}
```

Это вполне разумно, можно пойти и по этому пути. Только вот компоненту <Whinepad> хлопот прибавится, к тому же потребуется более объемный код, чем <input onChange="CRUDActions.search">.

Использование действий в <Whinepad>

Посмотрим, как теперь выглядит компонент `<Whinepad>` после перехода на Flux-действия. Во-первых, в него, конечно же, должен быть включен модуль действий:

```
/* @flow */

/* ... */
```

```
import CRUDActions from '../flux/CRUDActions';
/* ... */

class Whinepad extends Component {/* ... */}

export default Whinepad
```

Вспомним, что класс `Whinepad` отвечает за добавление новых записей и за поиск существующих записей (рис. 8.3).

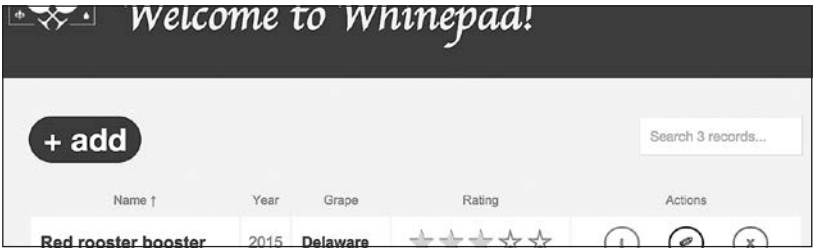


Рис. 8.3. Область ответственности Whinepad за работу с данными

Что касается добавления новых записей, то ранее `Whinepad` отвечал за работу со своим собственным свойством `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);
  }
}
```

Но теперь эта обязанность по обновлению хранилища (оно же единственный источник истинных данных) перешла к модулю действий:

```
_addNew(action: string) {
  this.setState({addnew: false});
  if (action === 'confirm') {
    CRUDActions.create(this.refs.form.getData());
  }
}
```

Нет ни состояния, которое нужно обслуживать, ни данных, с которыми нужно работать. При каком-либо действии пользователя его надо просто делегировать — и оно пойдет по однонаправленному потоку данных.

Аналогично обстоят дела и с поиском. Если прежде он выполнялся в отношении собственного свойства компонента `this.state.data`, то теперь все сводится к наличию следующего кода:

```
<input
  placeholder={this.state.count === 1
    ? 'Search 1 record...'
    : 'Search ${this.state.count} records...'
  }
  onChange={CRUDActions.search.bind(CRUDActions)}
  onFocus={CRUDActions.startSearching.bind(CRUDActions)} />
```

Использование действий в компоненте <Excel>

Компонент `Excel` является инициатором и получателем сортировки, удаления и обновления, предоставляемых модулем `CRUDActions`. Если помните, раньше удаление выглядело следующим образом:

```
_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);
}
```

А сейчас оно превратилось в следующий код:

```
_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);
}
```

Теперь уже не выдается событие изменения данных, поскольку отслеживание событий, происходящих в компоненте `Excel`, никем не ведется, всеобщие интересы нацелились на хранилище. И больше не нужно работать со свойством `this.state.data`. Вместо этого вся работа возложена на модуль действий, а обновление происходит, когда модуль хранилища выдает событие.

Аналогично обстоят дела с сортировкой и обновлением записей. Вся работа с данными превратилась в отдельные вызовы методов модуля `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
```

Полностью преобразованная версия приложения Whinepad, использующего Flux-технологию, доступна в хранилище кода, сопровождающем книгу.

И еще немного о Flux

Вот и все. Приложение теперь перешло на использование архитектуры Flux (некой разновидности ее самодельной версии). Есть представление (view), отправляющее данные о действиях пользователя модулю действий (actions), который обновляет данные в единственном хранилище (store), выдающем события. Затем представление отслеживает события хранилища и обновляется. Получается замкнутый цикл.

Но существуют и другие варианты развития этой идеи, которые могут оказаться полезными по мере роста объема приложения.

Действия могут отправляться не только представлением (view) (рис. 8.4), но и с сервера. Возможно, какие-то данные устарели. Может быть, другие пользователи меняли данные и приложение определило это после синхронизации с сервером. Или прошло

время — и должны быть предприняты какие-то действия (настал срок выкупить забронированные билеты, сессия просрочена, и нужно начинать все сначала!).

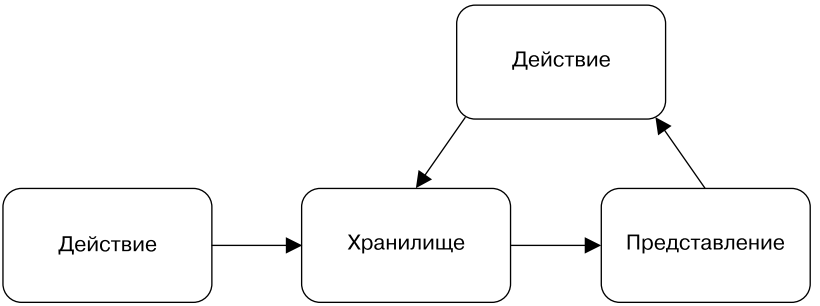


Рис. 8.4. Дополнительные действия

Когда возникает ситуация использования нескольких источников действий, на первый план выходит весьма конструктивная идея единого диспетчера (dispatcher) (рис. 8.5). Этот диспетчер отвечает за передачу всех этих действий в хранилище (store) или хранилища.

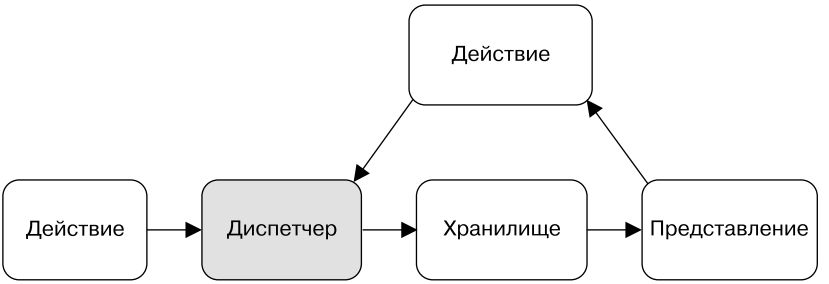


Рис. 8.5. Диспетчер

А в более интересных приложениях приходится иметь дело с различными действиями, поступающими из пользовательского интерфейса, от сервера или еще откуда-нибудь, и с несколькими

хранилищами, каждое из которых отвечает за свои собственные данные (рис. 8.6).

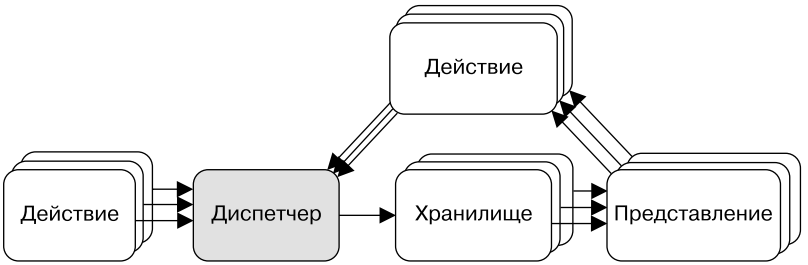


Рис. 8.6. Усложненный, но по-прежнему однонаправленный поток данных

Существует множество решений с открытым кодом, позволяющих реализовать Flux-архитектуру. Но вы всегда можете для начала выбрать небольшое по объему приложение и наращивать его по мере увеличения объема данных либо путем развития своего собственного решения, либо выбрав одно из предлагаемых решений с открытым кодом, о возможностях расширения которого вам уже известно.

Библиотека immutable

Завершим книгу небольшим изменением в двух частях Flux: в хранилище (store) и в действиях (actions), переключившись на неизменяемую (immutable) структуру данных для записей о вине. Когда речь заходит о React-приложениях, *неизменяемость* встречается довольно часто, даже если она не имеет ничего общего с библиотекой React.

Неизменяемый объект создается один раз и, соответственно, не может быть изменен. Неизменяемые объекты обычно проще понять и обосновать их применение. Например, строки зачастую закулисно реализуются как неизменяемые объекты.

В JavaScript, чтобы использовать идею неизменяемости, можно воспользоваться npm-пакетом `immutable`:

```
$ npm i --save-dev immutable
```

Нужно также дополнить содержимое файла .flowconfig:

```
# ....

[include]
# ...
node_modules/immutable

# ...
```

Полную документацию по библиотеке можно найти в Интернете.

Поскольку вся обработка данных теперь производится в модулях хранилища и действий, то обновлять, по сути, придется только эти два модуля.

Хранилище данных при использовании библиотеки immutable

Библиотека immutable среди прочих предлагает такие структуры данных, как список — List, стек — Stack и отображение — Map. Выберем List, поскольку он ближе всех к массиву, который прежде использовался приложением:

```
/* @flow */

import {EventEmitter} from 'fbemitter';
```

```
import {List} from 'immutable';

let data: List<Object>;
let schema;
const emitter = new EventEmitter();
```

Обратите внимание на новый тип данных — неизменяемый `List`.

Новый список создается с помощью выражения `let list = List()` и передачи исходных значений. Посмотрим, как теперь хранилище инициализирует список:

```
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));
    }
  },

  /* .. */
};
```

Как видите, список инициализируется с помощью массива. Далее для работы с данными используется относящийся к спискам API-интерфейс. После создания список становится неизменяемым, то есть он не может быть изменен. (Но, как вы вскоре увидите, все манипуляции происходят в модуле `CRUDActions`.)

Кроме инициализации и сигнатуры типа, изменения в хранилище незначительны — все, чем оно занимается, сводится к установке и получению данных.

Следует внести одно небольшое изменение в `getCount()`, поскольку неизменяемый список не содержит свойства длины `length`:

```
// До
getCount(): number {
  return data.length;
},

// После
getCount(): number {
  return data.count(); // также работает 'data.size'
},
```

И наконец, нужно выполнить обновление метода `getRecord()`, необходимость этого обусловлена тем, что библиотека immutable не предлагает обращения по индексам (подобно встроенным массивам):

```
// До
getRecord(recordId: number): ?Object {
  return recordId in data ? data[recordId] : null;
},

// После
getRecord(recordId: number): ?Object {
  return data.get(recordId);
},
```

Работа с данными при использовании библиотеки immutable

Вспомним, как в JavaScript действуют методы работы со строками:

```
let hi = 'Hello';
let ho = hi.toLowerCase();
```

```
hi; // "Hello"
ho; // "hello"
```

Строка, присвоенная переменной `hi`, не изменяется. Вместо нее создается новая строка.

То же самое происходит и с неизменяемым списком:

```
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 ]
```

Обратили внимание на метод push()? Неизменяемые списки ведут себя во многом подобно массивам, поэтому для работы с ними доступны такие методы, как map(), forEach() и т. д. Отчасти именно поэтому компоненты пользовательского интерфейса, по сути, не нуждаются в изменениях. (Если говорить начистоту, то одно изменение, касающееся синтаксиса квадратных скобок для доступа к массиву, все же понадобилось.) Причина еще и в том, что, как уже упоминалось, теперь данные обрабатываются главным образом в модулях хранилища (store) и действий (actions).

Как же изменение структуры данных влияет на модуль действий (actions)? На самом деле весьма незначительно. Поскольку неизменяемый список предлагает методы `sort()` и `filter()`, по части сортировки и поиска ничего менять не нужно. Изменения касаются только методов `create()`, `delete()` и двух методов `update*()`.

Рассмотрим метод `delete()`:

```
/* @flow */

import CRUDStore from './CRUDStore';
```

```
import {List} from 'immutable';

const CRUDActions = {

  /* ... */

  delete(recordId: number) {
    // До:
    // let data = CRUDStore.getData();
    // data.splice(recordId, 1);
    // CRUDStore.setData(data);

    // После:
    let data: List<Object> = CRUDStore.getData();
    CRUDStore.setData(data.remove(recordId));
  },

  * ... */
};

export default CRUDActions;
```

Возможно, JavaScript-метод `splice()` имеет слегка странное название, но он возвращает извлеченную часть массива, модифицируя при этом исходный массив. Все это создает небольшую путаницу при его использовании в одной строке кода. А вот неизменяемый список позволяет все уместить в одной строке. Если бы не славная сигнатура типа, то все свелось бы к простой однострочной форме:

```
delete(recordId: number) {
  CRUDStore.setData(CRUDStore.getData().remove(recordId));
},
```

В мире библиотеки immutable имеющий однозначное толкование своего названия метод `remove()` не оказывает на исходный список никакого влияния, оставляя его в неизмененном виде. Метод `remove()` дает вам новый список с одной удаленной записью. Затем

новый список назначается новыми данными для их сохранения в хранилище.

На него в этом отношении похожи и другие методы для работы с данными (их проще использовать, чем методы для работы с массивами):

```
/* ... */
create(newRecord: Object) {
// unshift() – как и для работы с массивами
  CRUDStore.setData(CRUDStore.getData().unshift(newRecord));
},

updateRecord(recordId: number, newRecord: Object) {
// set(), так как нет []
  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));
},
/* ... */
```

Вот и все! Теперь у вас есть приложение, которое использует:

- React-компоненты для определения пользовательского интерфейса;
- JSX для создания компонентов;
- Flux для организации потока данных;
- неизменяемые (благодаря использованию библиотеки immutable) данные;
- Babel, чтобы воспользоваться самыми последними возможностями ECMAScript;

- Flow для проверки соответствия типов и выявления синтаксических ошибок;
- ESLint для дополнительной проверки на наличие ошибок и на соответствие соглашениям;
- Jest для проведения блочного тестирования.

Как и всегда, вы можете ознакомиться с полноценной рабочей версией № 3 приложения Whinepad (его «immutable-редакцией») в хранилище кода книги. А поработать с готовым приложением можно по адресу http://whinepad.com.

Стоян Стефанов

React.js. Быстрый старт

Перевел с английского Н. Вильчинский

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*О. Андросик*
Художник	*С. Заматевская*
Корректоры	*Т. Курьянович, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Питер Пресс».

Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург, улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 03.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 22.02.17. Формат 60×90/16. Бумага офсетная. Усл. п. л. 19,000. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор». 142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**